

Foreword by Peter Hill, CEO, International Software
Benchmarking Standards Group

Software Engineering Best Practices

Lessons from Successful Projects
in the Top Companies



CAPERS JONES

Chief Scientist Emeritus, Software Productivity Research

Software Engineering Best Practices

ABOUT THE AUTHOR

CAPERS JONES is currently the president and CEO of Capers Jones & Associates LLC. He is also the founder and former chairman of Software Productivity Research LLC (SPR). He holds the title of Chief Scientist Emeritus at SPR. Capers Jones founded SPR in 1984. Before founding SPR, Capers was assistant director of programming technology for the ITT Corporation at the Programming Technology Center in Stratford, Connecticut. He was also a manager and researcher at IBM in California.

Capers is a well-known author and international public speaker. Some of his books have been translated into six languages. All of his books are translated into Japanese, and his newest books are available in Chinese editions as well. Among his book titles are *Patterns of Software Systems Failure and Success* (International Thomson Computer Press, 1995); *Applied Software Measurement*, Third Edition (McGraw-Hill, 2008); *Software Quality: Analysis and Guidelines for Success* (International Thomson, 1997); *Estimating Software Costs*, Second Edition (McGraw-Hill, 2007); and *Software Assessments, Benchmarks, and Best Practices* (Addison Wesley Longman, 2000). The third edition of his book *Applied Software Measurement* was published by McGraw-Hill in 2008.

Capers and his colleagues from SPR have collected historical data from more than 600 corporations and more than 30 government organizations. This historical data is a key resource for judging the effectiveness of software process improvement methods. More than 13,000 projects have been reviewed. This data is also widely cited in software litigation in cases where quality, productivity, and schedules are part of the proceedings. In addition to his technical books, Mr. Jones has also received recognition as an historian after the publication of *The History and Future of Narragansett Bay* in 2006 by Universal Publishers. Mr. Jones was the keynote speaker at the annual Japanese Symposium on Software Testing in Tokyo in 2008. He was also keynote speaker at the World Congress of Quality, and the keynote speaker at the 2008 conference of the International Function Point Users Group (IFPUG). His research studies include quality estimating, quality measurement, software cost and schedule estimation, and software metrics. He has been awarded a life-time membership in IFPUG. He was also named as a Distinguished Advisor to the Consortium for Information Technology Software Quality (CISQ).

Software Engineering Best Practices

Lessons from Successful Projects
in the Top Companies

Capers Jones



New York Chicago San Francisco Lisbon London Madrid
Mexico City Milan New Delhi San Juan Seoul
Singapore Sydney Toronto

Copyright © 2010 by The McGraw-Hill Companies. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN: 978-0-07-162162-5

MHID: 0-07-162162-8

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-162161-8, MHID: 0-07-162161-X.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please e-mail us at bulksales@mcgraw-hill.com.

Information has been obtained by McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill, or others, McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. ("McGraw-Hill") and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

This book is dedicated to many colleagues who have studied and advanced the field of software engineering. Some of these include Allan Albrecht, Barry Boehm, Fred Brooks, Tom DeMarco, the late Jim Frame, Peter Hill, Watts Humphrey, Steve Kan, Steve McConnell, Roger Pressman, Tony Salvaggio, Paul Strassmann, Jerry Weinberg, and Ed Yourdon.

This page intentionally left blank

Contents at a Glance

Chapter 1.	Introduction and Definitions of Software Best Practices	1
Chapter 2.	Overview of 50 Software Best Practices	39
Chapter 3.	A Preview of Software Development and Maintenance in 2049	177
Chapter 4.	How Software Personnel Learn New Skills	227
Chapter 5.	Software Team Organization and Specialization	275
Chapter 6.	Project Management and Software Engineering	351
Chapter 7.	Requirements, Business Analysis, Architecture, Enterprise Architecture, and Design	437
Chapter 8.	Programming and Code Development	489
Chapter 9.	Software Quality: The Key to Successful Software Engineering	555
Index		645

This page intentionally left blank

Contents

Foreword	xv
Acknowledgments	xvii
Introduction	xix

Chapter 1. Introduction and Definitions of Software Best Practices	1
What Are “Best Practices” and How Can They Be Evaluated?	7
Multiple Paths for Software Development, Deployment, and Maintenance	10
Paths for Software Deployment	12
Paths for Maintenance and Enhancements	14
Quantifying Software Development, Deployment, and Maintenance	16
Critical Topics in Software Engineering	19
Overall Ranking of Methods, Practices, and Sociological Factors	23
Summary and Conclusions	36
Readings and References	36
 Chapter 2. Overview of 50 Software Best Practices	 39
1. Best Practices for Minimizing Harm from Layoffs and Downsizing	41
2. Best Practices for Motivation and Morale of Technical Staff	45
3. Best Practices for Motivation and Morale of Managers and Executives	47
4. Best Practices for Selection and Hiring of Software Personnel	50
5. Best Practices for Appraisals and Career Planning for Software Personnel	50
6. Best Practices for Early Sizing and Scope Control of Software Applications	51
7. Best Practices for Outsourcing Software Applications	53
8. Best Practices for Using Contractors and Management Consultants	58
9. Best Practices for Selecting Software Methods, Tools, and Practices	59
10. Best Practices for Certifying Methods, Tools, and Practices	64
11. Best Practices for Requirements of Software Applications	70
12. Best Practices for User Involvement in Software Projects	72
13. Best Practices for Executive Management Support of Software Applications	74
14. Best Practices for Software Architecture and Design	75
15. Best Practices for Software Project Planning	77

16. Best Practices for Software Project Cost Estimating	79
17. Best Practices for Software Project Risk Analysis	81
18. Best Practices for Software Project Value Analysis	83
19. Best Practices for Canceling or Turning Around Troubled Projects	84
20. Best Practices for Software Project Organization Structures	87
21. Best Practices for Training Managers of Software Projects	89
22. Best Practices for Training Software Technical Personnel	91
23. Best Practices for Use of Software Specialists	92
24. Best Practices for Certifying Software Engineers, Specialists, and Managers	94
25. Best Practices for Communication During Software Projects	97
26. Best Practices for Software Reusability	99
27. Best Practices for Certification of Reusable Materials	101
28. Best Practices for Programming or Coding	107
29. Best Practices for Software Project Governance	109
30. Best Practices for Software Project Measurements and Metrics	110
31. Best Practices for Software Benchmarks and Baselines	112
32. Best Practices for Software Project Milestone and Cost Tracking	115
33. Best Practices for Software Change Control Before Release	117
34. Best Practices for Configuration Control	119
35. Best Practices for Software Quality Assurance (SQA)	120
36. Best Practices for Inspections and Static Analysis	124
37. Best Practices for Testing and Test Library Control	128
38. Best Practices for Software Security Analysis and Control	132
39. Best Practices for Software Performance Analysis	134
40. Best Practices for International Software Standards	136
41. Best Practices for Protecting Intellectual Property in Software	136
42. Best Practices for Protecting Against Viruses, Spyware, and Hacking	139
43. Best Practices for Software Deployment and Customization	154
44. Best Practices for Training Clients or Users of Software Applications	156
45. Best Practices for Customer Support of Software Applications	157
46. Best Practices for Software Warranties and Recalls	158
47. Best Practices for Software Change Management After Release	160
48. Best Practices for Software Maintenance and Enhancement	161
49. Best Practices for Updates and Releases of Software Applications	164
50. Best Practices for Terminating or Withdrawing Legacy Applications	166
Summary and Conclusions	167
Readings and References	167

Chapter 3. A Preview of Software Development and Maintenance in 2049	177
Introduction	177
Requirements Analysis Circa 2049	179
Design Circa 2049	182
Software Development Circa 2049	184
User Documentation Circa 2049	186
Customer Support in 2049	188
Deployment and Customer Training in 2049	190

Maintenance and Enhancement in 2049	191
Software Outsourcing in 2049	195
Software Package Evaluation and Acquisition in 2049	204
Technology Selection and Technology Transfer in 2049	207
Enterprise Architecture and Portfolio Analysis in 2049	210
A Preview of Software Learning in 2049	213
Due Diligence in 2049	216
Certification and Licensing in 2049	218
Software Litigation in 2049	221
Summary and Conclusions	225
Readings and References	225
 Chapter 4. How Software Personnel Learn New Skills	 227
Introduction	227
The Evolution of Software Learning Channels	228
What Topics Do Software Engineers Need to Learn Circa 2009?	230
Software Engineering Specialists Circa 2009	233
Varieties of Software Specialization Circa 2009	236
Approximate Ratios of Specialists to General Software Personnel	241
Evaluating Software Learning Channels Used by Software Engineers	243
Software Areas Where Additional Education Is Needed	266
New Directions in Software Learning	267
Summary and Conclusions	268
Curricula of Software Management and Technical Topics	268
Readings and References	273
 Chapter 5. Software Team Organization and Specialization	 275
Introduction	275
Quantifying Organizational Results	276
The Separate Worlds of Information Technology and Systems Software	277
Colocation vs. Distributed Development	278
The Challenge of Organizing Software Specialists	281
Software Organization Structures from Small to Large	284
One-Person Software Projects	284
Pair programming for software development and maintenance	286
Self-Organizing Agile Teams	289
Team Software Process (TSP) Teams	293
Conventional Departments with Hierarchical Organization Structures	298
Conventional Departments with Matrix Organization Structures	304
Specialist Organizations in Large Companies	308
Software Maintenance Organizations	309
Customer Support Organizations	322
Software Test Organizations	328
Software Quality Assurance (SQA) Organizations	342
Summary and Conclusions	348
Readings and References	349

Chapter 6. Project Management and Software Engineering	351
Introduction	351
Software Sizing	359
Software Progress and Problem Tracking	403
Software Benchmarking	408
Summary and Conclusions	433
Readings and References	434
 Chapter 7. Requirements, Business Analysis, Architecture, Enterprise Architecture, and Design	 437
Introduction	437
Software Requirements	439
Statistical Analysis of Software Requirements	442
Business Analysis	468
Software Architecture	470
Enterprise Architecture	475
Software Design	479
Summary and Conclusions	484
Readings and References	485
 Chapter 8. Programming and Code Development	 489
Introduction	489
A Short History of Programming and Language Development	490
Why Do We Have More than 2500 Programming Languages?	492
Exploring the Popularity of Programming Languages	495
How Many Programming Languages Are Really Needed?	499
Creating a National Programming Language Translation Center	501
Why Do Most Applications Use Between 2 and 15 Programming Languages	504
How Many Programmers Use Various Programming Languages?	506
What Kinds of Bugs or Defects Occur in Source Code?	509
Logistics of Software Code Defects	512
Preventing and Removing Defects from Application Source Code	518
Forms of Programming Defect Prevention	520
Forms of Programming Defect Removal	529
Economic Problems of the “Lines of Code” Metric	537
Summary and Conclusions	552
Readings and References	552
 Chapter 9. Software Quality: The Key to Successful Software Engineering	 555
Introduction	555
Defining Software Quality	558
Measuring Software Quality	585
Defect Prevention	600

Software Defect Removal	613
Software Quality Specialists	619
Summary and Conclusions on Software Specialization	632
The Economic Value of Software Quality	633
Summary and Conclusions	642
Readings and References	643

Index	645
-------	-----

This page intentionally left blank

Foreword

Software Engineering is about people—the people who have a need for and use software, the people who build software, the people who test software, the people who manage software projects, and the people who support and maintain software. So why is it that the emphasis of software engineering still concentrates on the technology?

How old is the business of software development? Let's settle on 55 years. That means a full generation has moved through the industry. These people were trained, spent their lives working on software, and are now retired or close to retirement. In their time, they have seen innumerable promises of “a better way”—silver bullets by the score. There have been thousands of new programming languages, all manner of methodologies, and a plethora of new tools. It sometimes seems as if the software industry is as strongly driven by fads and fashion as the garment industry. Many practitioners become apostolic in their worship of a particular approach, practice, or tool—for a while at least. But when the metrics are collected and analyzed, the sad truth is revealed—as an industry we have not made a great deal of progress. There have been no major breakthroughs that have led to the painless production of quality software delivered on time and within a predicted cost. And the skeptics ask, “Is software engineering an oxymoron?”

Our fixation on technology blinds us. We don't want to, or can't, see the need to embrace basic, sound engineering practices. In this book, Capers Jones contends that if the software industry wants to be recognized as an engineering discipline, rather than a craft, then it must employ solid engineering practices to deliver an acceptable result, economically. Technology is fascinating, but it is not the most important factor when it comes to doing good work and delivering a good result. The way people work, the choices they make, and the disciplines they choose to apply, will have more impact on the success of a software project than the choice of technology.

There must be times when Capers feels like a prophet alone in a desert. He has been tirelessly providing the software industry with guidance

and instruction on how to become a profession and an engineering discipline. His efforts span 16 books over 28 years. There are times when I wonder if he ever sleeps. Draft chapters of this book would arrive in my inbox at all times of the night and day. When you read something and think to yourself: *Well, that makes sense; it's obvious, really*, you realize the author has done a good job. Capers is such a writer. What he writes is engaging, understandable, and practical. My copies of his books are well thumbed, and festooned with Post-it notes. A sure sign of the books' practical usefulness.

Capers has an ability to stand back and observe the essence of the problems that still plague our industry. He is fearless in attacking practices that he sees as dangerous—in this book his targets are the use of the measurements *Cost per Defect* and *Lines of Code*. His views will, no doubt, be controversial, despite his well-reasoned dismantling of these dangerous economic measures. Debate and discussion will rage—these are long overdue. It takes a professional of Capers' standing to light the touch paper to ignite these debates.

Software quality also comes under the microscope in this book. He describes software quality as the key factor to successful software engineering—the driving factor that has more influence on software costs, schedules, and success than any other. There will be controversy here too as Capers challenges some common definitions of software quality.

Throughout the book, there is an emphasis on the need for measurement and metrics. Capers is critical of the lack of measurement, and the use of metrics that he describes as *hazardous*. The software industry deserves to be critically questioned while it makes little use of measurement and metrics. As Capers asserts, terms like “best practices” are an embarrassment when we cannot present statistical evidence.

Software engineering is 55 years old; the time has come for it to mature. In this book, Capers Jones's emphasis on the people and management issues of software engineering point the way toward achieving that maturity, and with it the prospect of the software industry being recognized as an engineering discipline.

—Peter R. Hill
CEO

International Software Benchmarking Standards Group
(ISBSG) Limited

Acknowledgments

Thanks as always to my wife, Eileen, who has put up with the writing of 16 books over 28 years.

Thanks also to the many colleagues who provided insights and helpful information for this book, and also for past books: Michael Bragen and Doug Brindley, the CTO and president of my former company, Software Productivity Research (SPR); Tom Cagley, the president of the International Function Point Users Group (IFPUG); Bob Charrette, the president of ITABHI; Ben Chelf, of Coverity Systems; Steven Cohen, from Microsoft; Dr. Taz Doughtry, from James Madison University; Chas Douglass, a former president of Software Productivity Research; Dr. Larry Driben, the president of Pearl Street software; Gary Gack, president of Process Fusion; Jonathan Glazer, the president of PowerBees; Scott Goldfarb, the president of the Quality and Productivity Management Group; Steve Heffner, CEO of Pennington Systems; Peter Hill, the CEO of International Software Benchmarking Standards Group (ISBSG); Watts Humphrey, from the Software Engineering Institute (SEI); Ken Hamer-Hodges, the president of Sipantic; Dr. Steve Kan, from IBM Rochester; Dr. Leon Kappleman, from the University of North Texas; Ravindra Karanam, the CTO of Unisys software operations; Dov Levy, the president of Dovél Systems; Dr. Tom Love, the president of Shoulders Corporation; Steve McConnell, the president of Construx; Michael Milutis, the director of the Information Technology Metrics and Productivity Institute (ITMPI); Peter Mollins, the chief of marketing of Relativity Technology; Prasanna Moses, from Unisys; Dr. Walker Royce, the head of IBM's Rational group; Dr. Akira Sakakibara, a distinguished scientist from IBM Tokyo; Tony Salvaggio, the president of Computer Aid Inc. (CAI); Paul Strassmann, president of the Information Economics Press (and the former CIO of the DoD); and Cem Tanyel, a vice president and general manager of Unisys Application Development Services.

A special tribute should be given to two executives who did a great deal to professionalize software. The late James H. Frame was director of IBM's Santa Teresa lab and then VP of the ITT Programming Development

Center at Stratford, Connecticut. The late Ted Climis was an IBM vice president who recognized the critical role of software at IBM. Both men clearly understood that software was vital to corporate business operations and that it needed to be designed and built to the highest standards of excellence.

Introduction

Between the time this book was started and the time it was completed, the global recession began. As a result, this book moved in a somewhat different direction than older books dealing with software engineering.

Due to the recession, the book now includes material on dealing with layoffs and downsizing; on the changing economic balance between the United States and other countries; and on the economics of software during a recessionary period.

Software engineering was not immediately affected by the financial crisis and the recession when it started in 2008, but as time passed, venture funds began to run out. Other forms of financing for software companies became difficult, so by the middle of 2009, software engineering positions were starting to be affected by layoffs. Specialty positions such as quality assurance and technical writing have been affected even more severely, since such positions are often the first to go during economic downturns.

One unexpected byproduct of the recession is that the availability of software engineers combined with a reduction in compensation has made the United States a candidate for becoming an outsource country.

As of 2009, the cost differentials between the United States, India, and China are being lowered, and the convenience of domestic contracts versus international contracts may prove to be beneficial for the software engineering community of the United States.

As the recession continues, the high costs of software are being examined more seriously than in the past. The recession also highlights the fact that software has always been insecure. Due to the recession, cyber-crime such as the theft of valuable information, identity theft, and even embezzlement are increasing rapidly. There are also sharp increases in “phishing” or attempting to use false messages to gain access to personal bank accounts.

From the author's viewpoint, the recession is highlighting four critical areas where software engineering needs to improve to become a solid engineering discipline rather than a craft:

1. Software security needs to be improved organically by raising the immunity levels of applications and including better security features in programming languages themselves. Access control and permissions are weak links in software engineering.
2. Software quality needs to be improved in terms of both defect prevention methods and defect removal methods. Poor quality has damaged the economics of software for 50 years, and this situation cannot continue. Every major application needs to use effective combinations of inspections, static analysis, and testing. Testing alone is inadequate to achieve high quality levels.
3. Software measurements need to be improved in order to gain better understanding of the true economic picture of software development and software maintenance. This implies moving to activity-based costs. Better measurement also implies analyzing the flaws of traditional metrics such as "cost per defect" and "lines of code," which violate the rules of standard economics.
4. Due to the recession, new development is slowing down, so the economics of software maintenance and renovation need to be better understood. Methods of preserving and renovating legacy applications are increasingly important, as are methods of mining legacy applications for "lost" requirements and business rules.

As of 2009, the great majority of companies and the great majority of software engineers have no effective measurements of either productivity or quality. This is not a suitable situation for a true engineering discipline. Lack of measurements combined with hazardous metrics mean that evaluating the effectiveness of methods such as Agile, Rational Unified Process (RUP), and the Team Software Process (TSP) is harder than it should be.

While the lack of measurements and the ability to judge the effectiveness of software engineering methods and practices was inconvenient when the economy was growing, it is a serious problem during a recession. Poor measurements have made phrases such as "best practices" embarrassing for software, because a majority of the best-practice claims have not been based on solid measurements using valid metrics.

This book attempts to show a combination of metrics and measurements that can demonstrate effectiveness and hopefully place software engineering on a sound economic basis. The "best practices" described

herein are those where quantitative data provides at least a provisional ability to judge effectiveness.

The book is divided into nine chapters, each of which deals with a set of technical and social issues.

Chapter 1: Introduction and Definitions of Software Best Practices Because many software applications may last for 25 years or more after they are first delivered, software engineering is not just concerned with development. Software engineering needs to consider the many years of maintenance and enhancement after delivery. Software engineering also needs to include effective methods for extracting or “mining” legacy applications to recover lost business rules and requirements.

There are more software engineers working on maintenance than on new development. Many of the maintenance software engineers are tasked with maintaining applications they did not develop, which may be coded in “dead” languages, and where there are neither specifications nor effective comments in the code itself.

Software engineering “best practices” are not a “one size fits all” technology. Evaluating best practices requires that the practices be understood for small applications of 100 function points or below, medium applications of 1000 function points, and large systems that may top 100,000 function points in size.

Further, the best practices that are effective for web applications and information technology are not necessarily the same as those with good results on embedded applications, systems software, and weapons systems.

As a result of these two wide sets of variations, this book evaluates best practices in terms of both application size and application type.

Chapter 2: Overview of 50 Software Best Practices This chapter discusses 50 software engineering best practices. Not all of the practices are purely technical. For example, during recessionary periods, companies have layoffs that if done poorly will damage operational effectiveness for many years.

This chapter deals with development best practices, maintenance best practices, management best practices, and sociological best practices such as those dealing with layoffs, which are often handled poorly and eliminate too few managers and executives and too many software engineers and specialists.

Methods other than layoffs such as reducing the work periods and compensation of all employees are usually preferable to actual staff reductions.

Other best-practice areas include security control, quality control, risk analysis, governance, development, maintenance, and renovation of legacy applications.

Portions of this chapter have served in software litigation where failure to conform to software engineering best practices were part of the plaintiff's claims.

Chapter 3: A Preview of Software Development and Maintenance in 2049

When software engineering is viewed up close as it is practiced in 2009, it is difficult to envision changes and improvements. Chapter 3 skips ahead to 2049 and explores what software engineering might be like in a world where all of us are connected via social networks, where the work of software engineering can be divided easily among software engineers who may live in many different countries.

The chapter also looks ahead to specific technical topics such as the role of data mining in gathering requirements and the possible availability of substantial libraries of certified reusable material. Also possible are intelligent agents and search-bots that will accumulate and even analyze information on critical topics.

Given the rapid rate of technological progress, it can be anticipated that computing devices, networks, and communication channels will be extremely sophisticated by 2049. But software engineering tends to lag hardware. Significant improvements are needed in security, quality, and reusability to keep pace with hardware and network evolution.

Chapter 4: How Software Personnel Learn New Skills Due in part to the recession, publishers of paper books and also software journals are experiencing severe economic problems and many are reducing staffs. Online publishing and electronic books such as the Amazon Kindle and the Sony PR-505 are rapidly expanding. Web publication, blogs, and Twitter are also expanding in terms of both providers and users.

Chapter 4 evaluates 17 channels that are available for transmitting and learning new software engineering information. Each channel is ranked in terms of learning effectiveness and cost-effectiveness. Long-range predictions are made as to the future of each channel.

Some of the learning channels evaluated include conventional paper books, electronic books, software journals, electronic journals and blogs, wiki sites, commercial education, in-house education, academic education, live conferences, and webinars, or online conferences. Electronic media have surpassed print media in terms of cost-effectiveness and are challenging older media in terms of learning effectiveness.

Chapter 4 also suggests curricula for software engineers, software quality assurance personnel, software test personnel, software project office personnel, and software managers. While today's academic curricula

are sufficient for mainstream software engineering, there is a shortage of solid education on topics such as sizing, estimating, planning, security, quality control, maintenance, renovation, and software engineering economic analysis.

Software metrics are poorly served by the academic community, with very little critical analysis of the flaws of common metrics such as cost per defect and lines of code.

While functional metrics are taught in a number of universities, there is little in the way of critical economic analysis of older metrics that behave erratically or that violate the canons of manufacturing economics. In particular, the impact of fixed costs on productivity is not dealt with, and this is the main reason why both lines of code and cost per defect are invalid for economic analysis.

Chapter 5: Software Team Organization and Specialization Software engineering organizations range from one person independent companies that produce small applications up through massive organizations with more than 1000 personnel who are part of companies that may employ more than 50,000 software personnel.

Large software engineering organizations employ more than 90 kinds of specialists in addition to software engineers themselves: quality assurance, technical writers, database administration, security specialists, webmasters, and metrics specialists are a few examples.

Chapter 5 shows the results of many different kinds of organization structures, including pair programming, small Agile teams, hierarchical organizations, matrix organizations, and virtual organizations that are geographically dispersed. It also shows the most effective ways of organizing specialists such as software quality assurance, testing, technical documentation, and project offices.

For example, for large projects in large companies, separate maintenance teams and separate test groups tend to be more effective than having maintenance and testing performed by the development team itself. Specialists and generalists must work together, and organization structures have a strong impact on overall results.

Chapter 6: Project Management and Software Engineering It is common knowledge that many software projects are sized incorrectly, estimated incorrectly, and have schedules committed that are too short for the capabilities of the development team. These lapses in project management can cause the affected software projects to either fail completely or to have serious cost and schedule overruns.

Chapter 6 deals with the critical management functions that can cause software engineering failures if they are not done well: sizing,

planning, estimating, progress tracking, resource tracking, benchmarks, and change management.

Chapter 6 suggests that every software project larger than trivial should collect both quality and productivity data that can be used for baselines and benchmarks. Collecting data on productivity and quality should be universal and not rare exceptions.

Careful measurement of methods utilized and results achieved is a sign of professionalism. Failure to measure is a sign that “software engineering” is not yet a true engineering discipline.

Chapter 7: Requirements, Business Analysis, Architecture, Enterprise Architecture, and Design Long before any code can be written, it is necessary to understand user requirements. These requirements need to be mapped onto effective software architectures and then translated into detailed designs. In addition, new applications need to be placed in the context of the overall enterprise portfolio of applications. With more than 20 forms of requirements methods and 40 kinds of design methods, software engineers have a great many choices.

This chapter discusses the most widely used methods of dealing with requirements and design issues and shows the classes and types of applications for which they are best suited. Agile methods, the unified modeling language (UML), and many other techniques are discussed.

The full portfolio for a large corporation circa 2009 can include more than 5000 applications totaling more than 10 million function points. The portfolio can include in-house applications, outsourced applications, commercial applications, and open-source applications.

The portfolio can include web applications, information technology applications, embedded software, and systems software. Due to the recession, it is increasingly important for corporate executives to know the size, contents, value, security levels, and quality levels of corporate portfolios.

Portfolio analysis has been hampered in the past by the difficulty of quantifying the sizes of various applications. New high-speed sizing methods that operate on commercial applications and on open-source applications as well as on in-house applications are beginning to eliminate these historical problems. It is now possible to size thousands of applications in a matter of a few days to a few weeks.

Chapter 8: Programming and Code Development This chapter deals with programming and code development from an unusual viewpoint. As of 2009, there are about 2500 programming languages and dialects. This chapter asks questions about why software engineering has so many languages.

Chapter 8 also asks whether the plethora of languages is helpful or harmful to the software engineering profession. In addition, it discusses the reasons many applications use between 2 and 15 different programming languages. The general conclusion is that while some programming languages do benefit software development, the existence of 2500 languages is a maintenance nightmare.

The chapter suggests the need for a National Programming Translation Center that would record the syntax of all known languages and assist in converting applications written in dead languages into modern languages.

The chapter also includes information on the kinds of bugs found in source code, and the most effective “personal” methods of defect prevention and defect removal that are carried out by software engineers prior to public activities such as function and regression testing.

Personal software methods such as desk checking and unit testing are normally not measured. However, volunteers do record information on defects found via “private” defect removal activities, so some data is available.

This chapter also discusses methods of measuring programming productivity and quality levels. The chapter is controversial due to challenging the traditional “lines of code” (LOC) metric as being economically invalid. The LOC metric penalizes high-level languages and distorts economic analysis.

Already in 2009, the lines of code metric cannot deal with requirements, design, screens, or documentations. Collectively, the costs of these noncode activities constitute more than 60 percent of total development expenses.

The alternative is functional metrics, which can handle all known software engineering activities. However, software functional metrics have been slow and expensive. New high-speed functional metrics are starting to appear circa 2009 that promise to expand the usage of such metrics.

Chapter 9: Software Quality: The Key to Successful Software Engineering

Quality has long been one of the weakest links in the chain of technologies associated with software engineering. This chapter attempts to cover all major factors that influence software quality, including both defect prevention methods and defect removal methods.

The chapter discusses the strengths and weaknesses of formal inspections, static analysis, and 17 different kinds of testing. In addition, the chapter deals with various troublesome metrics that degrade understanding software quality. For example, the popular “cost per defect” metric actually penalizes quality and achieves the lowest cost for the buggyest applications! In addition, quality has economic value far in

excess of the mere cost of removing defects, and this value cannot be shown using cost per defect metrics.

The main theme of the chapter is that quality is the driving factor that has more influence on software costs, schedules, and success than any other. But poor measurement practices have made it difficult to carry out valid software engineering economic studies.

This chapter is controversial because it challenges two common definitions of quality. The definition that quality means “conformance to requirements” is challenged on the grounds that many requirements are harmful or “toxic” and should not be implemented. The definition that quality means conformance to a list of words ending in “ility,” such as “portability,” is also challenged on the grounds that some of these terms can be neither predicted nor measured. Quality needs a definition that can be predicted before applications begin and measured when they are complete.

Quality is the key to successful software engineering. But before the key can unlock a door to real professionalism, it is necessary to know how to measure software quality and also software economics. The chapter concludes that an activity that cannot measure its own results is not a true engineering discipline. It is time for software engineering to study critical topics such as defect potentials and defect removal efficiency levels.

As of 2009, most projects have far too many bugs or defects, and remove less than 85 percent of these prior to delivery. Every software engineer and every software project manager should know what combination of inspections, static analysis, and test stages is needed to achieve defect removal efficiency levels that approach 99 percent. Without such knowledge based on measurements, software engineering is a misnomer, and software development is only a craft and not a true profession.

Overall Goals of *Software Engineering Best Practices* One of the inspirations for this book was an older book from 1982. The older book was Paul Starr’s Pulitzer Prize winner, *The Social Transformation of American Medicine*.

Until I read Paul Starr’s book, I did not realize that 150 years ago, medical degrees were granted after two years of study, without any internships or residency requirements. In fact, most physicians-in-training never entered a hospital while in medical school. Even more surprising, medical schools did not require either a college degree or a high-school diploma for admission. More than 50 percent of U.S. physicians never went to college.

Paul Starr’s book detailed the attempts of the American Medical Association to improve academic training of physicians, establish a canon of professional malpractice to weed out quacks, and to improve

the professional status of physicians. There are many lessons in Paul Starr's book that would be valuable for software engineering.

The primary goal of this book on software engineering best practices is to provide incentive for putting software engineering on a solid basis of facts derived from accurate measurement of quality and productivity.

As the recession continues, there is an increasing need to minimize software failures, speed up software delivery, and reduce software maintenance expenses. These needs cannot be accomplished without careful measurements of the effectiveness of tools, methods, languages, and software organization structures.

Accurate measurement is the key that will unlock better software quality and security. Better software quality and security are the keys that will allow software engineering to become a true profession that is equal to older engineering fields in achieving successful results.

Measurement of software engineering results will also lead to more and better benchmarks, which in turn will provide solid proofs of software engineering methods that have proven to be effective. The overall themes of the book are the need for better measurements, better benchmarks, better quality control, and better security as precursors to successful software engineering.

This page intentionally left blank

Introduction and Definitions of Software Best Practices

As this book was being written, the worst recession of the 21st century abruptly started on September 15, 2008, with the bankruptcy filing of Lehman Brothers. All evidence to date indicates a deep and prolonged recession that may last for more than a year. In spite of signs of partial recovery in mid 2009, job losses continue to rise as do foreclosures and bankruptcies. Even the most optimistic projections of recovery are pointing to late 2010, while pessimistic projections are pointing towards 2011 or 2012. Indeed, this recession may cause permanent changes in the financial industry, and it is unclear when lost jobs will return. So long as unemployment rates top 10 percent in many states, the economy cannot be healthy.

Software is not immune to the failing economy. Many software companies will close, and thousands of layoffs will occur as companies contract and try to save money.

Historically, software costs have been a major component of corporate expense. Software costs have also been difficult to control, and have been heavily impacted by poor quality, marginal security, and other chronic problems.

Poor software engineering, which gave rise to seriously flawed economic models, helped cause the recession. As the recession deepens, it is urgent that those concerned with software engineering take a hard look at fundamental issues: quality, security, measurement of results, and development best practices. This book will discuss the following topics that are critical during a major recession:

- Minimizing harm from layoffs and downsizing
- Optimizing software quality control
- Optimizing software security control

- Migration from custom development to certified reusable components
- Substituting legacy renovation for new development
- Measuring software economic value and risk
- Planning and estimating to reduce unplanned overruns

This book does not offer panaceas, but it does discuss a number of important technical areas that need improvement if *software engineering* is to become a legitimate term for an occupation that has been a craft or art form rather than a true engineering field.

So long as software applications are hand-coded on a line-by-line basis, “software engineering” will be a misnomer. Switching from custom-development to construction from certified reusable components has the best prospect of making really significant improvements in both software engineering disciplines and in software cost structures.

More than a dozen excellent books are in print in 2009 on the topic of software engineering. Readers might well ask why another book on software engineering is needed. The main reason can be seen by considering the major cost drivers of large software applications. As of 2009, the results are distressing.

From working as an expert witness in software litigation, and from examining the software engineering results of more than 600 companies and government organizations, the author has found that the software industry spends more money on finding bugs and on cancelled projects than on anything else! As of 2009, the 15 major cost drivers of the software industry in descending order are shown in Table 1-1.

(Note that topic #3, “Producing English words,” refers to the 90 documents associated with large software projects. Many large software applications spend more time and money creating text documents than they do creating source code.)

These 15 major cost drivers are not what they should be for a true engineering field. Ideally, we in the field should be spending much more

TABLE 1-1 Major Cost Drivers for Software Applications Circa 2009

1. Finding and fixing bugs	9. Project management
2. Cancelled projects	10. Renovation and migration
3. Producing English words	11. Innovation (new kinds of software)
4. Security flaws and attacks	12. Litigation for failures and disasters
5. Requirements changes	13. Training and learning software
6. Programming or coding	14. Avoiding security flaws
7. Customer support	15. Assembling reusable components
8. Meetings and communication	

money on innovation and programming, and much less money on fixing bugs, cancelled projects, and problems of various kinds, such as combating security flaws. In a true engineering field, we should also be able to use far greater quantities of zero-defect reusable components than today's norms.

One goal of this book is to place software engineering excellence and best practices on a sound quantitative basis. If software engineering can become a true engineering discipline in which successful projects outnumber failures, cost drivers will be transformed. A goal of this book is to help transform software cost drivers, hopefully within ten years, so that they follow a pattern illustrated by Table 1-2.

Under this revised set of cost drivers, defect repairs, failures, and cancelled projects drop from the top of the list to the bottom. Recovery from security attacks would also shift toward the bottom due to better security controls during development.

Heading up the revised list would be innovation and the creation of new forms of software. Programming is only in 11th place, because a true engineering discipline would be able to utilize far more zero-defect reusable components than is possible in 2009. The revised list of cost drivers shows what expenditure patterns might look like if software engineering becomes a true profession instead of a craft that uses only marginal methods that frequently lead to failure instead of to success.

Since the software industry is now more than 60 years old, renovation, migration, and maintenance of legacy applications would still remain near the top of the list of cost drivers, even if software engineering were to become a true engineering discipline instead of a craft as it is today. In every industry older than 50 years, maintenance and enhancement work are major cost elements.

That brings up another purpose of this book. This book examines best practices for the entire life cycle of software applications, from early requirements through deployment and then through maintenance. Since some large applications are used for 30 years or more, this book

TABLE 1-2 Revised Sequence of Cost Drivers Circa 2019

1. Innovation (new kinds of software)	9. Requirements changes
2. Renovation and migration	10. Producing English words
3. Customer support	11. Programming or coding
4. Assembling reusable components	12. Finding and fixing bugs
5. Meetings and communications	13. Security flaws and attacks
6. Avoiding security flaws	14. Cancelled projects
7. Training and learning software	15. Litigation for failures and disasters
8. Project management	

covers a very wide range of topics. It deals not only with development best practices, but also with deployment best practices, maintenance and renovation best practices, and, eventually, best practices for withdrawal of applications when they finally end their useful lives.

Since many large projects fail and are never completed or delivered at all, this book also deals with best practices for attempting to turn around and salvage projects that are in trouble. If the project's value has turned negative so that salvage is not a viable option, this book will also consider best practices for termination of flawed applications.

For software, the software personnel in 2009 working on maintenance and enhancements of legacy applications outnumber the workers on new applications, yet the topics of maintenance and enhancement are underreported in the software engineering literature.

In spite of many excellent books on software engineering, we still need to improve quality control and security control in order to free up resources for innovation and for improved forms of software applications. We also need to pay more attention to maintenance and to enhancements of legacy applications.

As of 2009, the software industry spends more than 50 cents out of every dollar expended on software to fix bugs and deal with security flaws or disasters such as cancelled projects. Actual innovation and new forms of software get less than 10 cents out of every dollar.

If we can professionalize our development practices, quality practices, and security practices, it is hoped that disasters, bug repairs, and security repairs can drop below 15 cents out of every dollar. If this occurs, then the freed-up funds should allow as much as 40 cents out of every dollar to go to innovative new kinds of software.

Software applications of 10,000 function points (unit of measure of the business functionality an information system provides) cost around \$2,000 per function point from the start of requirements until delivery. Of this cost, more than \$800 per function point will be spent on finding and fixing bugs that probably should not be there in the first place. Such large applications, if delivered at all, take between 48 and 60 months. The overall costs are far too high, and the distribution of those costs indicates very poor engineering practices.

By means of better defect prevention methods and utilization of zero-defect reusable material, we would greatly improve the economic position of software engineering if we could develop 10,000-function point applications for less than \$500 per function point, and could spend less than \$100 per function point on finding and fixing bugs. Development schedules of between 12 and 18 months for 10,000 function points would also be valuable, since shorter schedules allow quicker responses to changing market conditions. These goals are theoretically possible using state-of-the-art software methods and practices. But moving from

theory to practical reality will require major transformation in quality control and also migration from line-by-line coding to construction of applications from zero-defect standard components. An open question is whether this transformation can be accomplished in ten years. It is not certain if ten years are sufficient, but it is certain that such profound changes won't occur in less than ten years.

As of 2009, large software projects are almost always over budget, usually delivered late, and are filled with bugs when they're finally delivered. Even worse, as many as 35 percent of large applications in the 10,000–function point or more size range will be cancelled and never delivered at all.

Since cancelled projects are more expensive than successfully completed projects, the waste associated with large software applications is enormous. Completed software applications in the range of 10,000 function points cost about \$2,000 per function point to build. But cancelled projects in the 10,000–function point range cost about \$2,300 per function point since they are usually late and over budget at the point of cancellation!

The software industry has the highest failure rate of any so-called engineering field. An occupation that runs late on more than 75 percent of projects and cancels as many as 35 percent of large projects is not a true engineering discipline.

Once deployed and delivered to users, software applications in the 10,000–function point range have annual maintenance and enhancement costs of between \$200 and \$400 per function point per calendar year. Of these costs, about 50 percent goes to fixing bugs, and the other 50 percent goes to enhancements or adding new features.

Here, too, cost improvements are needed. Ideally, defect repair costs should come down to less than \$25 per function point per year. Use of maintenance workbenches and renovation tools should drop enhancement costs down below \$75 per function point per year. A weak link in maintenance and enhancement is that of customer support, which remains highly labor intensive and generally unsatisfactory.

Testimony and depositions noted during litigation in which the author worked as an expert witness revealed that many software projects that end up in court due to cancellation or excessive overruns did not follow sound engineering practices. Five common problems occurred with cancelled or disastrous projects:

- Estimates prior to starting the project were inaccurate and excessively optimistic.
- Quality control during the project was inadequate.
- Change control during the project was inadequate.

- Tracking of progress during development was severely inadequate or even misleading.
- Problems were ignored or concealed rather than dealt with rapidly and effectively when they first were noted.

When successful projects are examined after completion and delivery, the differences between success and failure become clear. Successful software projects are good at planning and estimating, good at quality control, good at change management, good at tracking progress, and good at resolving problems rather than ignoring them. Successful software projects tend to follow sound engineering practices, but failing projects don't.

Depositions and court testimony reveal more subtle and deeper issues. As of 2009, an increasing amount of quantitative data can provide convincing proof that certain methods and activities are valuable and that others are harmful. For example, when schedules start to slip or run late, managers often try to recover by taking unwise actions such as bypassing inspections or trying to shorten testing. Such actions always backfire and make the problems worse. Why don't software project managers know that effective quality control shortens schedules and that careless quality control lengthens them?

One reason for making such mistakes is that although many books on software engineering and quality tell how to go about effective quality control, they don't provide quantitative results. In other words, what the software engineering community needs is not more "how to do it" information, but rather information on "what will be the results of using this method?" For example, information such as the following would be very useful:

"A sample of 50 projects of 10,000 function points was examined. Those using design and code inspections averaged 36 months in development schedules and achieved 96 percent defect removal efficiency levels."

"A sample of 125 similar projects of 10,000 function points that did not use design and code inspections was examined. Of this sample, 50 were cancelled without completion, and the average schedule for the 75 completed applications was 60 months. Defect removal efficiency averaged only 83 percent."

There is a major need to quantify the results of software development methods and approaches such as Agile development, waterfall development, Six Sigma for software, the Capability Maturity Model Integrated (CMMI), inspections, the Rational Unified Process (RUP), Team Software Process (TSP), and many more. This book will attempt to provide quantitative information for many common development methods. Note, however, that hybrid approaches are also common, such as using the Team Software Process (TSP) in conjunction with the Capability Maturity

Model Integrated (CMMI). Common hybrid forms will be discussed, but there are too many variations to deal with all of them.

What Are “Best Practices” and How Can They Be Evaluated?

A book entitled *Software Engineering Best Practices* should start by defining exactly what is meant by the phrase “best practice” and then explain where the data came from in order to include each practice in the set. A book on best practices should also provide quantitative data that demonstrates the results of best practices.

Because practices vary by application size and type, evaluating them is difficult. For example, the Agile methods are quite effective for projects below about 2,500 function points, but they lose effectiveness rapidly above 10,000 function points. Agile has not yet even been attempted for applications in the 100,000–function point range and may even be harmful at that size.

To deal with this situation, an approximate scoring method has been developed that includes both size and type. Methods are scored using a scale that runs from +10 to –10 using the criteria shown in Table 1-3. Both the approximate impact on productivity and the approximate impact on quality are included. The scoring method can be applied to specific ranges such as 1000 function points or 10,000 function points. It can also be applied to specific types of software such as information technology, web application, commercial software, military software, and several others. The scoring method runs from a maximum of +10 to a minimum of –10, as shown in Table 1-3.

The midpoint or “average” against which improvements are measured are traditional methods such as waterfall development performed by organizations either that don’t use the Software Engineering Institute’s Capability Maturity Model or that are at level 1. This fairly primitive combination remains more or less the most widely used development method even in 2009.

One important topic needs to be understood. Quality needs to be improved faster and to a higher level than productivity in order for productivity to improve at all. The reason for this is that finding and fixing bugs is overall the most expensive activity in software development. Quality leads and productivity follows. Attempts to improve productivity without improving quality first are ineffective.

For software engineering, a historically serious problem has been that measurement practices are so poor that quantified results are scarce. There are many claims for tools, languages, and methodologies that assert each should be viewed as a best practice. But empirical data on their actual effectiveness in terms of quality or productivity has been scarce.

TABLE 1-3 Scoring Ranges for Software Methodologies and Practices

Score	Productivity Improvement	Quality Improvement
10	25%	35%
9	20%	30%
8	17%	25%
7	15%	20%
6	12%	17%
5	10%	15%
4	7%	10%
3	3%	5%
2	1%	2%
1	0%	0%
0	0%	0%
-1	0%	0%
-2	-1%	-2%
-3	-3%	-5%
-4	-7%	-10%
-5	-10%	-15%
-6	-12%	-17%
-7	-15%	-20%
-8	-17%	-25%
-9	-20%	-30%
-10	-25%	-35%

This book attempts a different approach. To be described as a *best practice*, a language, tool, or method needs to be associated with software projects in the top 15 percent of the applications measured and studied by the author and his colleagues. To be included in the set of best practices, a specific method or tool has to demonstrate by using quantitative data that it improves schedules, effort, costs, quality, customer satisfaction, or some combination of these factors. Furthermore, enough data needs to exist to apply the scoring method shown in Table 1-3.

This criterion brings up three important points:

Point 1: Software applications vary in size by many orders of magnitude. Methods that might be ranked as best practices for small programs of 1000 function points may not be equally effective for large systems of 100,000 function points. Therefore this book and the scoring method use size as a criterion for judging “best in class” status.

Point 2: Software engineering is not a “one size fits all” kind of occupation. There are many different forms of software, such as embedded applications, commercial software packages, information technology projects, games, military applications, outsourced applications, open source applications, and several others. These various kinds of software applications do not necessarily use the same languages, tools, or development methods. Therefore this book considers the approaches that yield the best results for each type of software application.

Point 3: Tools, languages, and methods are not equally effective or important for all activities. For example, a powerful programming language such as Objective C will obviously have beneficial effects on coding speed and code quality. But which programming language is used has no effect on requirements creep, user documentation, or project management. Therefore the phrase “best practice” also has to identify which specific activities are improved. This is complicated because activities include development, deployment, and post-deployment maintenance and enhancements. Indeed, for large applications, development can take up to five years, installation can take up to one year, and usage can last as long as 25 years before the application is finally retired. Over the course of more than 30 years, hundreds of activities will occur.

The result of the preceding factors is that selecting a set of best practices for software engineering is a fairly complicated undertaking. Each method, tool, or language needs to be evaluated in terms of its effectiveness by size, by application type, and by activity. This book will discuss best practices in a variety of contexts:

- Best practices by size of the application
- Best practices by type of software (embedded, web, military, etc.)
- Best practices by activity (development, deployment, and maintenance)

In 2009, software engineering is not yet a true profession with state certification, licensing, board examinations, formal specialties, and a solid body of empirical facts about technologies and methods that have proven to be effective. There are, of course, many international standards. Also, various kinds of certification are possible on a voluntary basis. Currently, neither standards nor certification have demonstrated much in the way of tangible improvements in software success rates.

This is not to say that certification or standards have no value, but rather that *proving* their value by quantification of quality and productivity is a difficult task. Several forms of test certification seem to result in higher levels of defect removal efficiency than observed when uncertified

testers work on similar applications. Certified function-point counters have been shown experimentally to produce more accurate results than uncertified counters when counting trial examples. However, much better data is needed to make a convincing case that would prove the value of certification.

As to standards, the results are very ambiguous. No solid empirical data indicates, for example, that following ISO quality standards results in either lower levels of potential defects or higher levels of defect removal efficiency. Some of the security standards seem to show improvements in reduced numbers of security flaws, but the data is sparse and unverified by controlled studies.

Multiple Paths for Software Development, Deployment, and Maintenance

One purpose of this book is to illustrate a set of “paths” that can be followed from the very beginning of a software project all the way through development and that lead to a successful delivery. After delivery, the paths will continue to lead through many years of maintenance and enhancements.

Because many paths are based on application size and type, a network of possible paths exists. The key to successful software engineering is to find the specific path that will yield the best results for a specific project. Some of the paths will include Agile development, and some will include the Team Software Process (TSP). Some paths will include the Rational Unified Process (RUP), and a few might even include traditional waterfall development methods.

No matter which specific path is used, the destination must include fundamental goals for the application to reach a successful conclusion:

- Project planning and estimating must be excellent and accurate.
- Quality control must be excellent.
- Change control must be excellent.
- Progress and cost tracking must be excellent.
- Measurement of results must be excellent and accurate.

Examples of typical development paths are shown in Figure 1-1. This figure illustrates the development methods and quality practices used for three different size ranges of software applications.

To interpret the paths illustrated by Figure 1-1, the Methods boxes near the top indicate the methods that have the best success rates. For example, at fewer than 1000 function points, Agile has the most success. But for larger applications, the Team Software Process (TSP) and

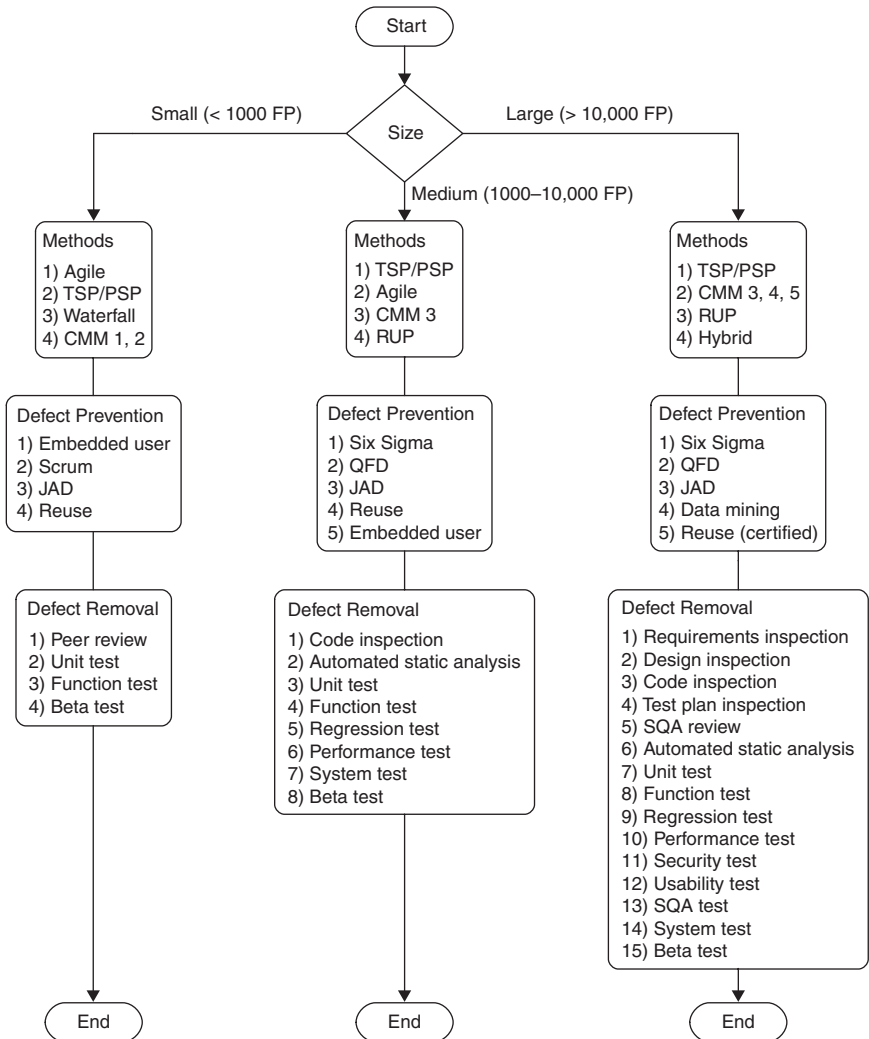


Figure 1-1 Development practices by size of application

Personal Software Process (PSP) have the greatest success. However, all of the methods in the boxes have been used for applications of the sizes shown, with reasonable success.

Moving down, the Defect Prevention and Defect Removal boxes show the best combinations of reviews, inspections, and tests. As you can see, larger applications require much more sophistication and many more kinds of defect removal than small applications of fewer than 1000 function points.

Continuing with the analogy of paths, there are hundreds of paths that can lead to delays and disasters, while only a few paths lead to successful outcomes that combine high quality, short schedules, and low costs. In fact, traversing the paths of a major software project resembles going through a maze. Most of the paths will be dead ends. But examining measurement and quantitative data is like looking at a maze from a tall ladder: they reveal the paths that lead to success and show the paths that should be avoided.

Paths for Software Deployment

Best practices are not limited to development. A major gap in the literature is that of best practices for installing or deploying large applications. Readers who use only personal computer software such as Windows Vista, Microsoft Office, Apple OS X, Intuit Quicken, and the like may wonder why deployment even matters. For many applications, installation via download, CD, or DVD may require only a few minutes. In fact, for Software as a Service (SaaS) applications such as the Google word processing and spreadsheet applications, downloads do not even occur. These applications are run on the Google servers and are not in the users' computers at all.

However, for large mainframe applications such as telephone switching systems, large mainframe operating systems, and enterprise resource planning (ERP) packages, deployment or installation can take a year or more. This is because the applications are not just installed, but require substantial customization to match local business and technical needs.

Also, training of the users of large applications is an important and time-consuming activity that might take several courses and several weeks of class time. In addition, substantial customized documentation may be created for users, maintenance personnel, customer support personnel, and other ancillary users. Best practices for installation of large applications are seldom covered in the literature, but they need to be considered, too.

Not only are paths through software development important, but also paths for delivery of software to customers, and then paths for maintenance and enhancements during the active life of software applications. Figure 1-2 shows typical installation paths for three very different situations: Software as a Service, self-installed applications, and those requiring consultants and installation specialists.

Software as a Service (SaaS) requires no installation. For self-installed applications, either downloads from the Web or physical installation via CD or DVD are common and usually accomplished with moderate ease. However, occasionally there can be problems, such as the release of a

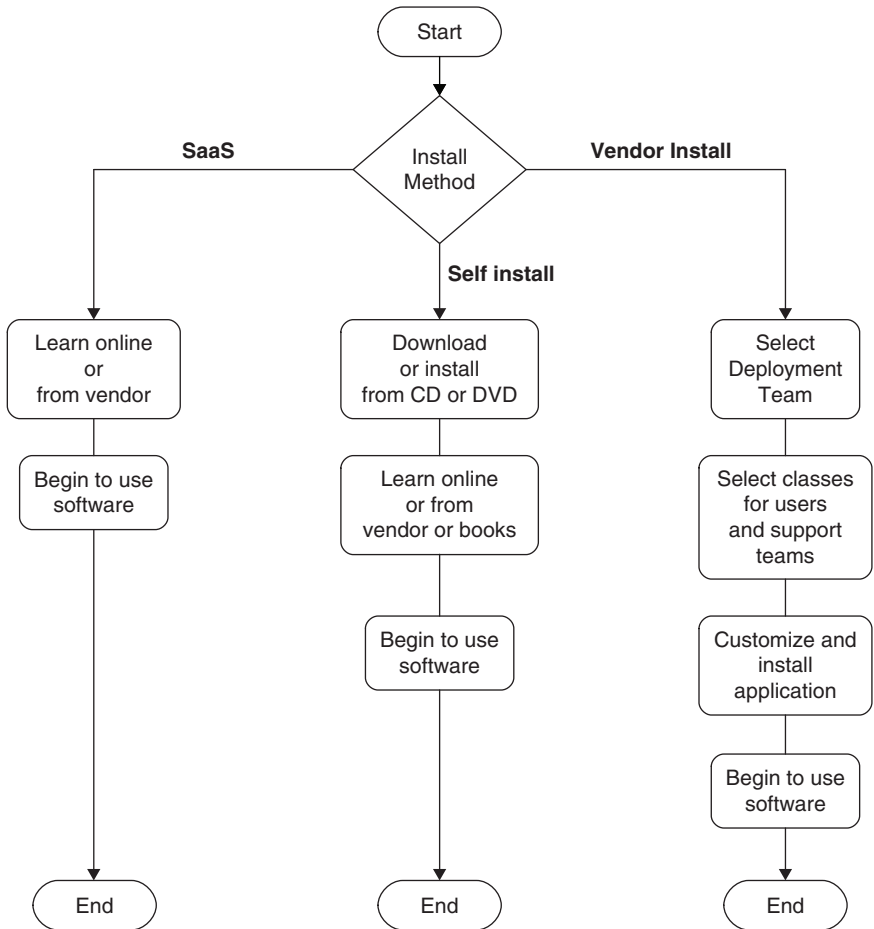


Figure 1-2 Deployment practices by form of deployment

Norton AntiVirus package that could not be installed until the previous version was uninstalled. However, the previous version was so convoluted that the normal Windows uninstall procedure could not remove it. Eventually, Symantec had to provide a special uninstall tool (which should have been done in the first place).

However, the really complex installation procedures are those associated with large mainframe applications that need customization as well as installation. Some large applications such as ERP packages are so complicated that sometimes it takes install teams of 25 consultants and 25 in-house personnel a year to complete installation.

Because usage of these large applications spans dozens of different kinds of users in various organizations (accounting, marketing, customer

support, manufacturing, etc.), a wide variety of custom user manuals and custom classes need to be created.

From the day large software packages are delivered until they are cut-over and begin large-scale usage by all classes of users, as long as a year can go by. Make no mistake: installation, deployment, and training users of large software applications is not a trivial undertaking.

Paths for Maintenance and Enhancements

Once software applications are installed and start being used, several kinds of changes will occur over time:

- All software applications have bugs or defects, and as these are found, they will need to be repaired.
- As businesses evolve, new features and new requirements will surface, so existing applications must be updated to keep them current with user needs.
- Government mandates or new laws such as changes in tax structures must be implemented as they occur, sometimes on very short notice.
- As software ages, structural decay always occurs, which may slow down performance or cause an increase in bugs or defects. Therefore if the software continues to have business value, it may be necessary to “renovate” legacy applications. *Renovation* consists of topics such as restructuring or refactoring to lower complexity, identification and removal of error-prone modules, and perhaps adding features at the same time. Renovation is a special form of maintenance that needs to be better covered in the literature.
- After some years of usage, aging legacy applications may outlive their utility and need replacement. However, redeveloping an existing application is not the same as starting a brand-new application. Existing business rules can be extracted from the code using data-mining techniques, since the original requirements and specifications usually lag and are not kept current.

Therefore, this book will attempt to show the optimal paths not only for development, but also for deployment, maintenance, and enhancements. Figure 1-3 illustrates three of the more common and important paths that are followed during the maintenance period.

As can be seen from Figure 1-3, maintenance is not a “one size fits all” form of modification. Unfortunately, the literature on software maintenance is very sparse compared with the literature on software development. Defect repairs, enhancements, and renovations are very different kinds of activities and need different skill sets and sometimes different tools.

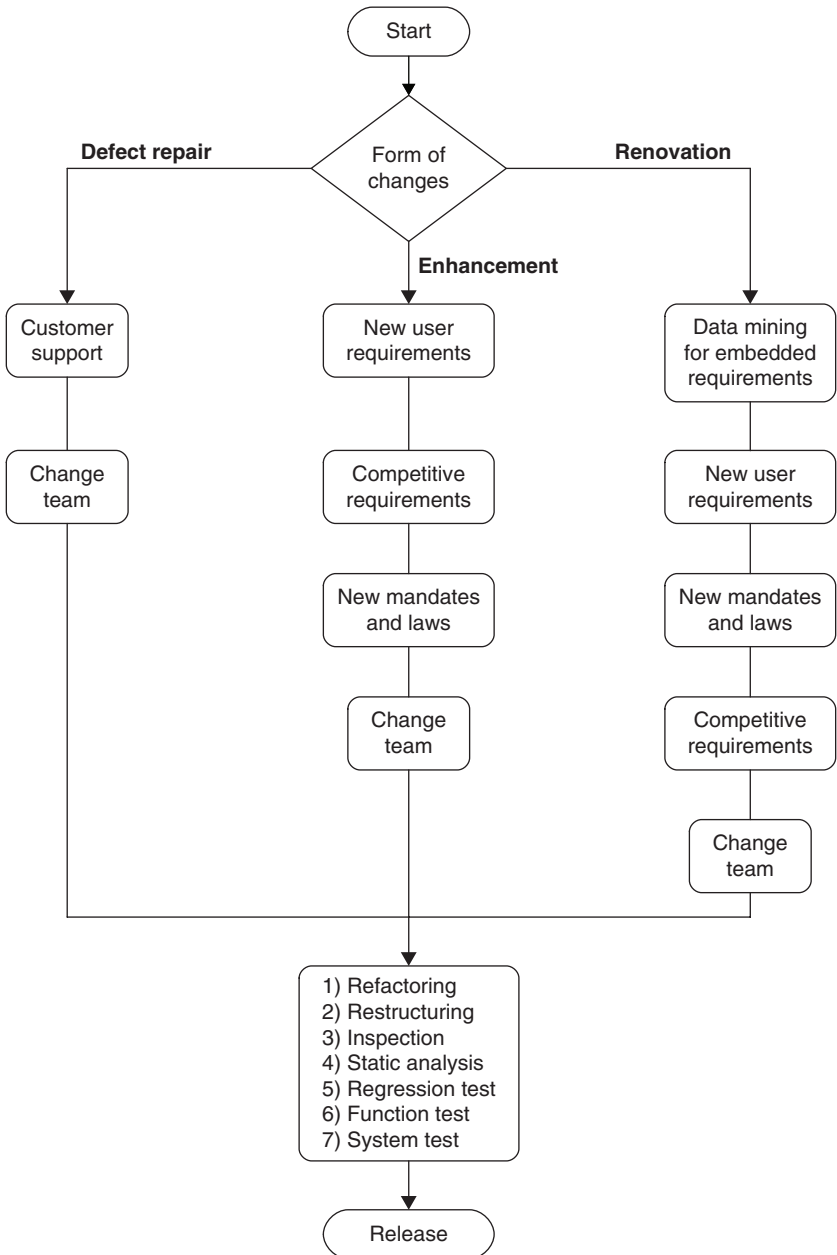


Figure 1-3 Major forms of maintenance and enhancement

Developing a major application in the 10,000 to 100,000 function-point size range is a multiyear undertaking that can easily last five years. Deploying such an application can take from 6 months to 12 months. Once installed, large software applications can continue to be used for 25 years or more. During usage, enhancements and defect repairs will be continuous. At some point, renovation or restoring the application to reduce complexity and perhaps migrate to new file structures or new programming languages might occur. Therefore, analysis of best practices needs to span at least a 30-year period. Development alone is only a fraction of the total cost of ownership of major software applications. This book will take a long view and attempt to encompass all best practices from the first day a project starts until the last user signs off, perhaps 30 years later.

Quantifying Software Development, Deployment, and Maintenance

This book will include productivity benchmarks, quality benchmarks, and data on the effectiveness of a number of tools, methodologies, and programming practices. It will also include quantitative data on the costs of training and deployment of methodologies. The data itself comes from several sources. The largest amount of data comes from the author's own studies with hundreds of clients between 1973 and 2009.

Other key sources of data include benchmarks gathered by Software Productivity Research LLC (SPR) and data collected by the nonprofit International Software Benchmarking Standards Group (ISBSG). In addition, selected data will be brought in from other sources. Among these other sources are the David Consulting Group, the Quality/Productivity (Q/P) consulting group, and David Longstreet of Longstreet consulting. Other information sources on best practices will include the current literature on software engineering and various portals into the software engineering domain such as the excellent portal provided by the Information Technology Metrics and Productivity Institute (ITMPI). Information from the Software Engineering Institute (SEI) will also be included. Other professional associations such as the Project Management Institute (PMI) and the American Society for Quality (ASQ) will be cited, although they do not publish very much quantitative data.

All of these sources provide benchmark data primarily using function points as defined by the International Function Point Users Group (IFPUG). This book uses IFPUG function points for all quantitative data dealing with quality and productivity.

There are several other forms of function point, including COSMIC (Common Software Measurement International Consortium) function

points and Finnish function points. While data in these alternative metrics will not be discussed at length in this book, citations to sources of benchmark data will be included. Other metrics such as use case points, story points, and goal-question metrics will be mentioned and references provided.

(It is not possible to provide accurate benchmarks using either *lines of code* metrics or *cost per defect* metrics. As will be illustrated later, both of these common metrics violate the assumptions of standard economics, and both distort historical data so that real trends are concealed rather than revealed.)

On the opposite end of the spectrum from best practices are *worst practices*. The author has been an expert witness in a number of breach-of-contract lawsuits where depositions and trial documents revealed the major failings that constitute worst practices. These will be discussed from time to time, to demonstrate the differences between the best and worst practices.

In between the sets of best practices and worst practices are many methods and practices that might be called *neutral practices*. These may provide some benefits for certain kinds of applications, or they may be slightly harmful for others. But in neither case does use of the method cause much variation in productivity or quality.

This book attempts to replace unsupported claims with empirical data derived from careful measurement of results. When the software industry can measure performance consistently and accurately, can estimate the results of projects with good accuracy, can build large applications without excessive schedule and cost overruns, and can achieve excellence in quality and customer satisfaction, then we can call ourselves “software engineers” without that phrase being a misnomer. Until our successes far outnumber our failures, software engineering really cannot be considered to be a serious and legitimate engineering profession.

Yet another major weakness of software engineering is a widespread lack of measurements. Many software projects measure neither productivity nor quality. When measurements are attempted, many projects use metrics and measurement approaches that have serious flaws. For example, the most common metric in the software world for more than 50 years has been *lines of code* (LOC). As will be discussed in Chapter 6 later in this book, LOC metrics penalize high-level languages and can’t measure noncode activities at all. In the author’s view, usage of lines of code for economic studies constitutes professional malpractice.

Another flawed metric is that of *cost per defect* for measuring quality. This metric actually penalizes quality and achieves its lowest values for the buggiest applications. Cost per defect cannot be used to measure zero-defect applications. Here, too, the author views cost per defect as professional malpractice if used for economic study.

Mathematical problems with the cost per defect metric have led to the urban legend that “it costs 100 times as much to fix a bug after delivery as during development.” This claim is not based on time and motion studies, but is merely due to the fact that cost per defect goes up as numbers of defects go down. Defect repairs before and after deployment take about the same amount of time. Bug repairs at both times range from 15 minutes to more than eight hours. Fixing a few subtle bugs can take much longer, but they occur both before and after deployment.

Neither lines of code nor cost per defect can be used for economic analysis or to demonstrate software best practices. Therefore this book will use function point metrics for economic study and best-practice analysis. As mentioned, the specific form of function point used is that defined by the International Function Point Users Group (IFPUG).

There are other metrics in use such as COSMIC function points, use case points, story points, web object points, Mark II function points, Finnish function points, feature points, and perhaps 35 other function point variants. However, as of 2008, only IFPUG function points have enough measured historical data to be useful for economic and best-practice analysis on a global basis. Finnish function points have several thousand projects, but most of these are from Finland where the work practices are somewhat different from the United States. COSMIC function points are used in many countries, but still lack substantial quantities of benchmark data as of 2009 although this situation is improving.

This book will offer some suggested conversion rules between other metrics and IFPUG function points, but the actual data will be expressed in terms of IFPUG function points using the 4.2 version of the counting rules.

As of this writing (late 2008 and early 2009), the function point community has discussed segmenting function points and using a separate metric for the technical work of putting software onto various platforms, or getting it to work on various operating systems. There is also discussion of using a separate metric for the work associated with quality, such as inspections, testing, portability, reliability, and so on. In the author’s view, both of these possible changes in counting practices are likely to conceal useful information rather than reveal it. These measurement issues will be discussed at length later in this book in Chapter 6.

IFPUG function point metrics are far from perfect, but they offer a number of advantages for economic analysis and identification of best practices. Function points match the assumptions of standard economics. They can measure information technology, embedded applications, commercial software, and all other types of software. IFPUG function points can be used to measure noncode activities as well as to measure coding work. Function points can be used to measure defects in requirements

and design as well as to measure code defects. Function points can handle every activity during both development and maintenance. In addition, benchmark data from more than 20,000 projects is available using IFPUG function points. No other metric is as stable and versatile as function point metrics.

One key fact should be obvious, but unfortunately it is not. To demonstrate high quality levels, high productivity levels, and to identify best practices, it is necessary to have accurate measurements in place. For more than 50 years, the software engineering domain has utilized measurement practices and metrics that are seriously flawed. An occupation that cannot measure its own performance with accuracy is not qualified to be called an engineering discipline. Therefore another purpose of this book is to demonstrate how economic analysis can be applied to software engineering projects. This book will demonstrate methods for measuring productivity and quality with high precision.

Critical Topics in Software Engineering

As of 2009, several important points about software engineering have been proven beyond a doubt. Successful software projects use state-of-the-art quality control methods, change control methods, and project management methods. Without excellence in quality control, there is almost no chance of a successful outcome. Without excellence in change control, creeping requirements will lead to unexpected delays and cost overruns. Without excellent project management, estimates will be inaccurate, plans will be defective, and tracking will miss serious problems that can cause either outright failure or significant overruns. Quality control, change control, and project management are the three critical topics that can lead to either success or failure. The major forms of best practices that will be discussed in this book include the following:

1. Introduction, Definitions, and Ranking of Software Practices

Definitions and rankings of:

- Best practices
- Very good practices
- Good practices
- Fair practices
- Neutral practices
- Harmful practices
- Worst practices

Definitions of professional malpractice

2. Overview of 50 Best Practices

Overview of social and morale best practices

Overview of best practices for:

- Organization
- Development
- Quality and security
- Deployment
- Maintenance

3. A Preview of Software Development and Maintenance in 2049

Requirements analysis circa 2049

Design in 2049

Software development in 2049

User documentation circa 2049

Customer support in 2049

Maintenance and enhancement in 2049

Deployment and training in 2049

Software outsourcing in 2049

Technology selection and technology transfer in 2049

Software package evaluation and acquisition in 2049

Enterprise architecture and portfolio analysis in 2049

Due diligence in 2049

Software litigation in 2049

4. How Software Personnel Learn New Skills

Evolution of software learning channels

Varieties of software specialization

Evaluation of software learning channels in descending order:

Number 1: Web browsing

Number 2: Webinars, podcasts, and e-learning

Number 3: Electronic books (e-books)

Number 4: In-house education

Number 5: Self-study using CDs and DVDs

Number 6: Commercial education

Number 7: Vendor education

Number 8: Live conferences

Number 9: Wiki sites

Number 10: Simulation web sites

Number 11: Software journals

Number 12: Self-study using books and training materials

Number 13: On-the-job training

Number 14: Mentoring

Number 15: Professional books, monographs, and technical reports

Number 16: Undergraduate university education

Number 17: Graduate university education

5. **Team Organization and Specialization**

Large teams and small teams

Finding optimal organization structures

Matrix versus hierarchical organizations

Using project offices

Specialists and generalists

Pair programming

Use of Scrum sessions for local development

Communications for distributed development

In-house development, outsource development, or both

6. **Project Management**

Measurement and metrics

Sizing applications

Risk analysis of applications

Planning and estimating

Governance of applications

Tracking costs and progress

Benchmarks for comparison against industry norms

Baselines to determine process improvements

Cancelled projects and disaster recovery

Minimizing the odds of litigation in outsource agreements

7. **Architecture, Business Analysis, Requirements, and Design**

Alignment of software and business needs

Gathering requirements for new applications

Mining legacy applications for requirements
Requirements change or “creeping requirements”
Requirements churn or subtle changes
The role of architecture in software
Design methods for software
Requirements change and multiple releases

8. Code Development

Development methodology selection
Choice of programming languages
Multiple languages in the same application
Coding techniques
Reusable code
Code change control

9. Quality Control, Inspections, and Testing

Six Sigma for software
Defect estimation
Defect and quality measurements
Design and code inspections
Static analysis
Manual testing
Automated testing
Configuration control

10. Security, Virus Protection, Spyware, and Hacking

Prevention methods for security threats
Defenses against active security threats
Recovery from security attacks

11. Deployment and Customization of Large Applications

Selecting deployment teams
Customizing large and complex applications
Developing customized training materials
Cut-over and parallel runs of new and old applications

12. Maintenance and Enhancements

Maintenance (defect repairs)

- Enhancements (new features)
- Mandatory changes (government regulations)
- Customer support
- Renovation of legacy applications
- Maintenance outsourcing

13. **Companies That Utilize Best Practices**

- Advanced Bionics
- Aetna Insurance
- Amazon
- Apple Computers
- Computer Aid Inc.
- Coverity
- Dovel Technologies
- Google
- IBM
- Microsoft
- Relativity Technologies
- Shoulders Corporation
- Unisys

These topics are of course not the only factors that need to be excellent or where best practices are beneficial. But these topics are the core issues that can eventually change the term “software engineering” from an oxymoron into a valid description of an occupation that has at last matured enough to be taken seriously by other and older forms of engineering.

Overall Ranking of Methods, Practices, and Sociological Factors

To be considered a best practice, a method or tool has to have some quantitative proof that it actually provides value in terms of quality improvement, productivity improvement, maintainability improvement, or some other tangible factors.

Although more than about 200 topics can have an impact on software, only 200 are shown here. Solid empirical data exists for about 50 out of the 200. For the rest, the data is anecdotal or inconsistent. The data has been gathered from observations of about 13,000 projects in 600 companies.

However, that data spans more than 20 years of observation, so the data is of inconsistent ages. It is easily possible that some of the practices are out of place on the list, or will change places as more data becomes available. Even so, methods and practices in the top 50 have proven to be beneficial in scores or hundreds of projects. Those in the bottom 50 have proven to be harmful.

Between the “good” and “bad” ends of this spectrum are a significant number of practices that range from intermittently helpful to occasionally harmful. These are termed *neutral*. They are sometimes marginally helpful and sometimes not. But in neither case do they seem to have much impact.

Although this book will deal with methods and practices by size and by type, it might be of interest to show the complete range of factors ranked in descending order, with the ones having the widest and most convincing proof of usefulness at the top of the list. Table 1-4 lists a total of 200 methodologies, practices, and social issues that have an impact on software applications and projects.

Recall that the scores are the aggregated results of specific scores for applications of fewer than 1000 function points to more than 10,000 function points. In the full table, systems and embedded applications, commercial applications, information technology, web applications, and other types are also scored separately. Table 1-4 shows the overall average scores.

TABLE 1-4 Evaluation of Software Methods, Practices, and Results

	Methodology, Practice, Result	Average
	Best Practices	
1.	Reusability (> 85% zero-defect materials)	9.65
2.	Defect potentials < 3.00 per function point	9.35
3.	Defect removal efficiency > 95%	9.32
4.	Personal Software Process (PSP)	9.25
5.	Team Software Process (TSP)	9.18
6.	Automated static analysis	9.17
7.	Inspections (code)	9.15
8.	Measurement of defect removal efficiency	9.08
9.	Hybrid (CMM + TSP/PSP + others)	9.06
10.	Reusable feature certification	9.00
11.	Reusable feature change controls	9.00
12.	Reusable feature recall method	9.00
13.	Reusable feature warranties	9.00
14.	Reusable source code (zero defect)	9.00

TABLE 1-4 Evaluation of Software Methods, Practices, and Results (*continued*)

	Methodology, Practice, Result	Average
	Very Good Practices	
15.	Early estimates of defect potentials	8.83
16.	Object-oriented (OO) development	8.83
17.	Automated security testing	8.58
18.	Measurement of bad-fix injections	8.50
19.	Reusable test cases (zero defect)	8.50
20.	Formal security analysis	8.43
21.	Agile development	8.41
22.	Inspections (requirements)	8.40
23.	Time boxing	8.38
24.	Activity-based productivity measures	8.33
25.	Reusable designs (scalable)	8.33
26.	Formal risk management	8.27
27.	Automated defect tracking tools	8.17
28.	Measurement of defect origins	8.17
29.	Benchmarks against industry data	8.15
30.	Function point analysis (high speed)	8.15
31.	Formal progress reports (weekly)	8.06
32.	Formal measurement programs	8.00
33.	Reusable architecture (scalable)	8.00
34.	Inspections (design)	7.94
35.	Lean Six Sigma	7.94
36.	Six Sigma for software	7.94
37.	Automated cost-estimating tools	7.92
38.	Automated maintenance workbenches	7.90
39.	Formal cost-tracking reports	7.89
40.	Formal test plans	7.81
41.	Automated unit testing	7.75
42.	Automated sizing tools (function points)	7.73
43.	Scrum session (daily)	7.70
44.	Automated configuration control	7.69
45.	Reusable requirements (scalable)	7.67
46.	Automated project management tools	7.63
47.	Formal requirements analysis	7.63
48.	Data mining for business rule extraction	7.60
49.	Function point analysis (pattern matches)	7.58
50.	High-level languages (current)	7.53
51.	Automated quality and risk prediction	7.53
52.	Reusable tutorial materials	7.50

(Continued)

TABLE 1-4 Evaluation of Software Methods, Practices, and Results (*continued*)

	Methodology, Practice, Result	Average
	Very Good Practices	
53.	Function point analysis (IFPUG)	7.37
54.	Measurement of requirements changes	7.37
55.	Formal architecture for large applications	7.36
56.	Best-practice analysis before start	7.33
57.	Reusable feature catalog	7.33
58.	Quality function deployment (QFD)	7.32
59.	Specialists for key skills	7.29
60.	Joint application design (JAD)	7.27
61.	Automated test coverage analysis	7.23
62.	Re-estimating for requirements changes	7.17
63.	Measurement of defect severity levels	7.13
64.	Formal SQA team	7.10
65.	Inspections (test materials)	7.04
66.	Automated requirements analysis	7.00
67.	DMAIC (design, measure, analyze, improve, control)	7.00
68.	Reusable construction plans	7.00
69.	Reusable HELP information	7.00
70.	Reusable test scripts	7.00
	Good Practices	
71.	Rational Unified Process (RUP)	6.98
72.	Automated deployment support	6.87
73.	Automated cyclomatic complexity analysis	6.83
74.	Forensic analysis of cancelled projects	6.83
75.	Reusable reference manuals	6.83
76.	Automated documentation tools	6.79
77.	Capability Maturity Model (CMMI Level 5)	6.79
78.	Annual training (technical staff)	6.67
79.	Metrics conversion (automated)	6.67
80.	Change review boards	6.62
81.	Formal governance	6.58
82.	Automated test library control	6.50
83.	Formal scope management	6.50
84.	Annual training (managers)	6.33
85.	Dashboard-style status reports	6.33
86.	Extreme programming (XP)	6.28
87.	Service-oriented architecture (SOA)	6.26
88.	Automated requirements tracing	6.25
89.	Total cost of ownership (TCO) measures	6.18

TABLE 1-4 Evaluation of Software Methods, Practices, and Results (*continued*)

	Methodology, Practice, Result	Average
	Good Practices	
90.	Automated performance analysis	6.17
91.	Baselines for process improvement	6.17
92.	Use cases	6.17
93.	Automated test case generation	6.00
94.	User satisfaction surveys	6.00
95.	Formal project office	5.88
96.	Automated modeling/simulation	5.83
97.	Certification (Six Sigma)	5.83
98.	Outsourcing (maintenance => CMMI Level 3)	5.83
99.	Capability Maturity Model (CMMI Level 4)	5.79
100.	Certification (software quality assurance)	5.67
101.	Outsourcing (development => CMM 3)	5.67
102.	Value analysis (intangible value)	5.67
103.	Root-cause analysis	5.50
104.	Total cost of learning (TCL) measures	5.50
105.	Cost of quality (COQ)	5.42
106.	Embedded users in team	5.33
107.	Normal structured design	5.17
108.	Capability Maturity Model (CMMI Level 3)	5.06
109.	Earned-value measures	5.00
110.	Unified modeling language (UML)	5.00
111.	Value analysis (tangible value)	5.00
	Fair Practices	
112.	Normal maintenance activities	4.54
113.	Rapid application development (RAD)	4.54
114.	Certification (function points)	4.50
115.	Function point analysis (Finnish)	4.50
116.	Function point analysis (Netherlands)	4.50
117.	Partial code reviews	4.42
118.	Automated restructuring	4.33
119.	Function point analysis (COSMIC)	4.33
120.	Partial design reviews	4.33
121.	Team Wiki communications	4.33
122.	Function point analysis (unadjusted)	4.33
123.	Function points (micro 0.001 to 10)	4.17
124.	Automated daily progress reports	4.08
125.	User stories	3.83
126.	Outsourcing (offshore => CMM 3)	3.67

(Continued)

TABLE 1-4 Evaluation of Software Methods, Practices, and Results (continued)

	Methodology, Practice, Result	Average
	Fair Practices	
127.	Goal-question metrics	3.50
128.	Certification (project managers)	3.33
129.	Refactoring	3.33
130.	Manual document production	3.17
131.	Capability Maturity Model (CMMI Level 2)	3.00
132.	Certification (test personnel)	2.83
133.	Pair programming	2.83
134.	Clean-room development	2.50
135.	Formal design languages	2.50
136.	ISO quality standards	2.00
	Neutral Practices	
137.	Function point analysis (backfiring)	1.83
138.	Use case points	1.67
139.	Normal customer support	1.50
140.	Partial governance (low-risk projects)	1.00
141.	Object-oriented metrics	0.33
142.	Manual testing	0.17
143.	Outsourcing (development < CMM 3)	0.17
144.	Story points	0.17
145.	Low-level languages (current)	0.00
146.	Outsourcing (maintenance < CMM 3)	0.00
147.	Waterfall development	-0.33
148.	Manual change control	-0.50
149.	Manual test library control	-0.50
150.	Reusability (average quality materials)	-0.67
151.	Capability Maturity Model (CMMI Level 1)	-1.50
152.	Informal progress tracking	-1.50
153.	Outsourcing (offshore < CMM 3)	-1.67
	Unsafe Practices	
154.	Inadequate test library control	-2.00
155.	Generalists instead of specialists	-2.50
156.	Manual cost estimating methods	-2.50
157.	Inadequate measurement of productivity	-2.67
158.	Cost per defect metrics	-2.83
159.	Inadequate customer support	-2.83
160.	Friction between stakeholders and team	-3.50
161.	Informal requirements gathering	-3.67
162.	Lines of code metrics (logical LOC)	-4.00
163.	Inadequate governance	-4.17

TABLE 1-4 Evaluation of Software Methods, Practices, and Results (*continued*)

	Methodology, Practice, Result	Average
	Unsafe Practices	
164.	Lines of code metrics (physical LOC)	-4.50
165.	Partial productivity measures (coding)	-4.50
166.	Inadequate sizing	-4.67
167.	High-level languages (obsolete)	-5.00
168.	Inadequate communications among team	-5.33
169.	Inadequate change control	-5.42
170.	Inadequate value analysis	-5.50
	Worst Practices	
171.	Friction/antagonism among team members	-6.00
172.	Inadequate cost estimating methods	-6.04
173.	Inadequate risk analysis	-6.17
174.	Low-level languages (obsolete)	-6.25
175.	Government mandates (short lead times)	-6.33
176.	Inadequate testing	-6.38
177.	Friction/antagonism among management	-6.50
178.	Inadequate communications with stakeholders	-6.50
179.	Inadequate measurement of quality	-6.50
180.	Inadequate problem reports	-6.67
181.	Error-prone modules in applications	-6.83
182.	Friction/antagonism among stakeholders	-6.83
183.	Failure to estimate requirements changes	-6.85
184.	Inadequate defect tracking methods	-7.17
185.	Rejection of estimates for business reasons	-7.33
186.	Layoffs/loss of key personnel	-7.33
187.	Inadequate inspections	-7.42
188.	Inadequate security controls	-7.48
189.	Excessive schedule pressure	-7.50
190.	Inadequate progress tracking	-7.50
191.	Litigation (noncompete violation)	-7.50
192.	Inadequate cost tracking	-7.75
193.	Litigation (breach of contract)	-8.00
194.	Defect potentials > 6.00 per function point	-9.00
195.	Reusability (high defect volumes)	-9.17
196.	Defect removal efficiency < 85%	-9.18
197.	Litigation (poor quality/damages)	-9.50
198.	Litigation (security flaw damages)	-9.50
199.	Litigation (patent violation)	-10.00
200.	Litigation (intellectual property theft)	-10.00

The candidates for best practices will be discussed and evaluated later in this book in Chapters 7, 8, and 9. Here in Chapter 1 they are only introduced to show what the overall set looks like.

Note that the factors are a mixture. They include full development methods such as Team Software Process (TSP) and partial methods such as quality function deployment (QFD). They include specific practices such as “inspections” of various kinds, and also include social issues such as friction between stakeholders and developers. They include metrics such as “lines of code,” which is ranked as a harmful factor because this metric penalizes high-level languages and distorts both quality and productivity data. What all these things have in common is that they either improve or degrade quality and productivity.

Since programming languages are also significant, you might ask why specific languages such as Java, Ruby, or Objective C are not included. Because, as of 2009, more than 700 programming languages exist; a new language is created about every month.

In addition, a majority of large software applications utilize several languages at the same time, such as Java and HTML, or use combinations that may top a dozen languages in the same applications. Later in Chapter 8 this book will discuss the impact of languages and their virtues or weaknesses, but there are far too many languages, and they change far too rapidly, for an evaluation to be useful for more than a few months. Therefore in Table 1-4, languages are covered only in a general way: whether they are high level or low level, and whether they are current languages or “dead” languages no longer used for new development.

This book is not a marketing tool for any specific products or methods, including the tools and methods developed by the author. This book attempts to be objective and to base conclusions on quantitative data rather than on subjective opinions.

To show how methods and practices differ by *size* of project, Table 1-5 illustrates the top 30 best practices for small projects of 1000 function points and for large systems of 10,000 or more function points. As can be seen, the two lists are very different.

For small projects, Agile, extreme programming, and high-level programming languages are key practices because coding is the dominant activity for small applications. When large applications are analyzed, quality control ascends to the top. Also, careful requirements, design, and architecture are important for large applications.

There are also differences in best practices by *type* of application. Table 1-6 shows the top 30 best practices for information technology (IT) projects compared with embedded and systems software projects.

Although high-quality reusable components are the top factor for both, the rest of the two lists are quite different. For information technology

TABLE 1-5 Best Practices for 1000– and 10,000–Function Point Software Projects

Small (1000 function points)	Large (10,000 function points)
1. Agile development	1. Reusability (> 85% zero-defect materials)
2. High-level languages (current)	2. Defect potentials < 3.00 per function point
3. Extreme programming (XP)	3. Formal cost tracking reports
4. Personal Software Process (PSP)	4. Inspections (requirements)
5. Reusability (> 85% zero-defect materials)	5. Formal security analysis
6. Automated static analysis	6. Measurement of defect removal efficiency
7. Time boxing	7. Team Software Process (TSP)
8. Reusable source code (zero defect)	8. Function point analysis (high speed)
9. Reusable feature warranties	9. Capability Maturity Model (CMMI Level 5)
10. Reusable feature certification	10. Automated security testing
11. Defect potentials < 3.00 per function point	11. Inspections (design)
12. Reusable feature change controls	12. Defect removal efficiency > 95%
13. Reusable feature recall method	13. Inspections (code)
14. Object-oriented (OO) development	14. Automated sizing tools (function points)
15. Inspections (code)	15. Hybrid (CMM + TSP/PSP + others)
16. Defect removal efficiency > 95%	16. Automated static analysis
17. Hybrid (CMM + TSP/PSP + others)	17. Personal Software Process (PSP)
18. Scrum session (daily)	18. Automated cost estimating tools
19. Measurement of defect removal efficiency	19. Measurement of requirements changes
20. Function point analysis (IFPUG)	20. Service-oriented architecture (SOA)
21. Automated maintenance workbenches	21. Automated quality and risk prediction
22. Early estimates of defect potentials	22. Benchmarks against industry data
23. Team Software Process (TSP)	23. Quality function deployment (QFD)
24. Embedded users in team	24. Formal architecture for large applications
25. Benchmarks against industry data	25. Automated defect tracking tools
26. Measurement of defect severity levels	26. Reusable architecture (scalable)
27. Use cases	27. Formal risk management
28. Reusable test cases (zero defects)	28. Activity-based productivity measures
29. Automated security testing	29. Formal progress reports (weekly)
30. Measurement of bad-fix injections	30. Function point analysis (pattern matches)

TABLE 1-6 Best Practices for IT Projects and Embedded/Systems Projects

Information Technology (IT) Projects	Embedded and Systems Projects
1. Reusability (> 85% zero-defect materials)	1. Reusability (> 85% zero-defect materials)
2. Formal governance	2. Defect potentials < 3.00 per function point
3. Team Software Process (TSP)	3. Defect removal efficiency > 95%
4. Personal Software Process (PSP)	4. Team Software Process (TSP)
5. Agile development	5. Measurement of defect severity levels
6. Defect removal efficiency > 95%	6. Inspections (code)
7. Formal security analysis	7. Lean Six Sigma
8. Formal cost tracking reports	8. Six Sigma for software
9. Defect potentials < 3.00 per function point	9. Automated static analysis
10. Automated static analysis	10. Measurement of defect removal efficiency
11. Measurement of defect removal efficiency	11. Hybrid (CMM + TSP/PSP + others)
12. Function point analysis (IFPUG)	12. Personal Software Process (PSP)
13. Service-oriented architecture (SOA)	13. Formal security analysis
14. Joint application design (JAD)	14. Formal cost tracking reports
15. Function point analysis (high speed)	15. Function point analysis (high speed)
16. Automated sizing tools (function points)	16. Inspections (design)
17. Data mining for business rule extraction	17. Automated project management tools
18. Benchmarks against industry data	18. Formal test plans
19. Hybrid (CMM + TSP/PSP + others)	19. Quality function deployment (QFD)
20. Reusable feature certification	20. Automated cost estimating tools
21. Reusable feature change controls	21. Automated security testing
22. Reusable feature recall method	22. Object-oriented (OO) development
23. Reusable feature warranties	23. Inspections (test materials)
24. Reusable source code (zero defect)	24. Agile development
25. Early estimates of defect potentials	25. Automated sizing tools (function points)
26. Measurement of bad-fix injections	26. Reusable feature certification
27. Reusable test cases (zero defect)	27. Reusable feature change controls
28. Inspections (requirements)	28. Reusable feature recall method
29. Activity-based productivity measures	29. Reusable feature warranties
30. Reusable designs (scalable)	30. Reusable source code (zero defect)

projects, at least for those developed by Fortune 500 companies, governance is in the number 2 spot for best practices. This is because inadequate or incompetent governance can now lead to criminal charges against corporate officers as a result of the Sarbanes-Oxley Act of 2002.

For systems and embedded software, quality control measures of various kinds are the top-ranked best practices. Historically, systems and embedded software have had the best and most sophisticated software quality control in the history of software. This is because the main products of the systems and embedded domain are complex physical devices that might cause catastrophic damages or death if quality control is deficient. Thus manufacturers of medical devices, aircraft control systems, fuel injection, and other forms of systems and embedded applications have long had sophisticated quality control, even before software was used for physical devices.

The main point is that software development is not a “one size fits all” kind of work. Best practices must be carefully selected to match both the size and the type of the software under development.

A few basic principles are true across all sizes and all types: quality control, change control, good estimating, and good measurement are critical activities. Reuse is also critical, with the caveat that only zero-defect reusable objects provide solid value.

Although this book is primarily about software engineering best practices, it is useful to discuss polar opposites and to show worst practices, too. The definition of a *worst practice* as used in this book is a method or approach that has been proven to cause harm to a significant number of projects that used it. The word “harm” means degradation of quality, reduction of productivity, or concealing the true status of projects. In addition, “harm” includes data that is so inaccurate that it leads to false conclusions about economic value.

Each of the harmful methods and approaches individually has been proven to cause harm in a significant number of applications that used them. This is not to say that they always fail. Sometimes, rarely, they may even be useful. But in a majority of situations, they do more harm than good in repeated trials.

A distressing aspect of the software industry is that bad practices seldom occur in isolation. From examining the depositions and court documents of lawsuits for projects that were cancelled or never operated effectively, it usually happens that multiple worst practices are used concurrently.

From data and observations on the usage patterns of software methods and practices, it is distressing to note that practices in the harmful or worst set are actually found on about 65 percent of U.S. software projects. Conversely, best practices that score 9 or higher have only been noted on about 14 percent of U.S. software projects. It is no wonder that failures far outnumber successes for large software applications!

From working as an expert witness in a number of breach-of-contract lawsuits, the author has found that many harmful practices tend to occur repeatedly. These collectively are viewed by the author as candidates for

being deemed “professional malpractice.” The definition of *professional malpractice* is something that causes harm that a trained practitioner should know is harmful and therefore avoid using it. Table 1-7 shows 30 of these common but harmful practices.

Not all of these 30 occur at the same time. In fact, some of them, such as the use of generalists, are only harmful for large applications in the 10,000–function point range. However, this collection of harmful practices has been a drain on the software industry and has led to many lawsuits.

TABLE 1-7 Software Methods and Practices Considered “Professional Malpractice”

Rank	Methods and Practices	Scores
1.	Defect removal efficiency < 85%	–9.18
2.	Defect potentials > 6.00 per function point	–9.00
3.	Reusability (high defect volumes)	–7.83
4.	Inadequate cost tracking	–7.75
5.	Excessive schedule pressure	–7.50
6.	Inadequate progress tracking	–7.50
7.	Inadequate security controls	–7.48
8.	Inadequate inspections	–7.42
9.	Inadequate defect tracking methods	–7.17
10.	Failure to estimate requirements changes	–6.85
11.	Error-prone modules in applications	–6.83
12.	Inadequate problem reports	–6.67
13.	Inadequate measurement of quality	–6.50
14.	Rejection of estimates for business reasons	–6.50
15.	Inadequate testing	–6.38
16.	Inadequate risk analysis	–6.17
17.	Inadequate cost estimating methods	–6.04
18.	Inadequate value analysis	–5.50
19.	Inadequate change control	–5.42
20.	Inadequate sizing	–4.67
21.	Partial productivity measures (coding)	–4.50
22.	Lines of code (LOC) metrics	–4.50
23.	Inadequate governance	–4.17
24.	Inadequate requirements gathering	–3.67
25.	Cost per defect metrics	–2.83
26.	Inadequate customer support	–2.83
27.	Inadequate measurement of productivity	–2.67
28.	Generalists instead of specialists for large systems	–2.50
29.	Manual cost estimating methods for large systems	–2.50
30.	Inadequate test library control	–2.00

Note that two common metrics are ranked as professional malpractice: “lines of code” and “cost per defect.” They are viewed as malpractice because both violate the tenets of standard economics and distort data so that economic results are impossible to see. The lines of code metric penalizes high-level languages. The cost per defect metric penalizes quality and achieves its best results for the buggiest artifacts. These problems will be explained in more detail later.

With hundreds of methods and techniques available for developing and maintaining software, not all of them can be classified as either best practices or worst practices. In fact, for many practices and methods, the results are so mixed or ambiguous that they can be called “neutral practices.”

The definition of a *neutral practice* is a method or tool where there is little statistical data that indicates either help or hindrance in software development. In other words, there are no quantified changes in either a positive or negative direction from using the method.

Perhaps the most interesting observation about neutral practices is that they occur most often on small projects of 1500 function points and below. Many years of data indicate that small projects can be developed in a fairly informal manner and still turn out all right. This should not be surprising, because the same observation can be made about scores of products. For example, a rowboat does not need the same rigor of development as does a cruise ship.

Because small projects outnumber large applications by more than 50 to 1, it is not easy to even perform best-practice analysis if small programs are the only projects available for analysis. Many paths lead to success for small projects. As application size goes up, the number of successful paths goes down in direct proportion. This is one of the reasons why university studies seldom reach the same conclusions as industrial studies when it comes to software engineering methods and results. Universities seldom have access to data from large software projects in the 10,000– to 100,000–function point size range.

Serious analysis of best and worst practices requires access to data from software applications that are in the size range of 10,000 or more function points. For large applications, failures outnumber successes. At the large end of the size spectrum, the effects of both best and worst practices are magnified. As size increases, quality control, change control, and excellence in project management become more and more important on the paths to successful projects.

Smaller applications have many advantages. Due to shorter development schedules, the number of changing requirements is low. Smaller applications can be built successfully using a variety of methods and processes. Topics such as estimates and plans are much easier for small projects with limited team size.

Large software applications in the range of 10,000 or more function points are much more difficult to create, have significant volumes of requirements changes, and will not be successful without topnotch quality control, change control, and project management. Many kinds of specialists are needed for large software applications, and also special organizations are needed such as project offices, formal quality assurance teams, technical writing groups, and testing organizations.

Summary and Conclusions

There are hundreds of ways to make large software systems fail. There are only a few ways of making them successful. However, the ways or “paths” that lead to success are not the same for small projects below 1000 function points and large systems above 10,000 function points. Neither are the ways or paths the same for embedded applications, web applications, commercial software, and the many other types of software applications in the modern world.

Among the most important software development practices are those dealing with planning and estimating before the project starts, with absorbing changing requirements during the project, and with successfully handling bugs or defects. Another key element of success is being proactive with problems and solving them quickly, rather than ignoring them and hoping they will go away.

Successful projects using state-of-the-art methods are always excellent in the critical activities: estimating, change control, quality control, progress tracking, and problem resolution. By contrast, projects that run late or fail usually had optimistic estimates, did not anticipate changes, failed to control quality, tracked progress poorly, and ignored problems until too late.

Software engineering is not yet a true and recognized engineering field. It will never become one so long as our failures outnumber our successes. It would benefit the global economy and our professional status to move software engineering into the ranks of true engineering fields. But accomplishing this goal requires better quality control, better change control, better measurements, and much better quantification of our results than we have today.

Readings and References

These selections include some of the historic books on software best practices, and also on broader topics such as quality in general. One book on medical practice is included: Paul Starr’s book *The Social Transformation of American Medicine*, which won a Pulitzer Prize in 1982. This book discusses the improvements in medical education and

certification achieved by the American Medical Association (AMA). The path followed by the AMA has considerable relevance to improving software engineering education, best practices, and certification. Thomas Kuhn's book on *The Structure of Scientific Revolutions* is also included, because software engineering needs a revolution if it is to shift from custom development of unique applications to construction of generic applications from certified reusable components.

- Boehm, Barry. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.
- Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1974, rev. 1995.
- Bundschuh, Manfred, and Carol Dekkers. *The IT Measurement Compendium*. Berlin: Springer-Verlag, 2008.
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Crosby, Philip B. *Quality Is Free*. New York: New American Library, Mentor Books, 1979.
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- DeMarco, Tom. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1999.
- Garmus, David, and David Herron. *Function Point Analysis—Measurement Practices for Successful Software Projects*. Boston: Addison Wesley Longman, 2001.
- Gilb, Tom, and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.
- Glass, Robert L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Glass, Robert L. *Software Creativity*, Second Edition. Atlanta, GA: developer.*books, 2006.
- Hamer-Hodges, Ken. *Authorization Oriented Architecture—Open Application Networking and Security in the 21st Century*. Philadelphia: Auerbach Publications, to be published in December 2009.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989.
- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Humphrey, Watts. *TSP—Leading a Development Team*. Boston: Addison Wesley, 2006.
- Humphrey, Watts. *Winning with Software: An Executive Strategy*. Boston: Addison Wesley, 2002.
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York: McGraw-Hill, 2008.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston: Addison Wesley Longman, 2003.
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press, 1996.
- Love, Tom. *Object Lessons*. New York: SIGS Books, 1993.
- McConnell, Steve. *Code Complete*. Redmond, WA: Microsoft Press, 1993.
- Myers, Glenford. *The Art of Software Testing*. New York: John Wiley & Sons, 1979.
- Pressman, Roger. *Software Engineering—A Practitioner's Approach*, Sixth Edition. New York: McGraw-Hill, 2005.
- Starr, Paul. *The Social Transformation of American Medicine*. New York: Basic Books, 1982.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.

Weinberg, Gerald M. *Becoming a Technical Leader*. New York: Dorset House, 1986.

Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

Yourdon, Ed. *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.

Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

Overview of 50 Software Best Practices

Since not everyone reads books from cover to cover, it seems useful to provide a concise overview of software engineering best practices before expanding the topics later in the book. As it happens, this section was originally created for a lawsuit, which was later settled. That material on best practices has been updated here to include recent changes in software engineering technologies.

These best-practice discussions focus on projects in the 10,000–function point range. The reason for this is pragmatic. This is the size range where delays and cancellations begin to outnumber successful completions of projects.

The best practices discussed in this book cover a timeline that can span 30 years or more. Software development of large applications can take five years. Deploying and customizing such applications can take another year. Once deployed, large applications have extremely long lives and can be used for 25 years or more.

Over the 25-year usage period, numerous enhancements and defect repairs will occur. There may also be occasional “renovation” or restructuring of the application, changing file formats, and perhaps converting the source code to a newer language or languages.

The set of best practices discussed here spans the entire life cycle from the day a project starts until the day that the application is withdrawn from service. The topics include, but are not limited to, the best practices for the 50 subjects discussed here:

1. Minimizing harm from layoffs and downsizing
2. Motivation and morale of technical staff
3. Motivation and morale of managers and executives

4. Selection and hiring of software personnel
5. Appraisals and career planning for software personnel
6. Early sizing and scope control of software applications
7. Outsourcing software applications
8. Using contractors and management consultants
9. Selecting software methods, tools, and practices
10. Certifying methods, tools, and practices
11. Requirements of software applications
12. User involvement in software projects
13. Executive management support of software applications
14. Software architecture and design
15. Software project planning
16. Software project cost estimating
17. Software project risk analysis
18. Software project value analysis
19. Canceling or turning around troubled projects
20. Software project organization structures
21. Training managers of software projects
22. Training software technical personnel
23. Use of software specialists
24. Certification of software engineers, specialists, and managers
25. Communication during software projects
26. Software reusability
27. Certification of reusable materials
28. Programming or coding
29. Software project governance
30. Software project measurements and metrics
31. Software benchmarks and baselines
32. Software project milestone and cost tracking
33. Software change control before release
34. Configuration control
35. Software quality assurance
36. Inspections and static analysis

37. Testing and test library control
38. Software security analysis and control
39. Software performance analysis
40. International software standards
41. Protecting intellectual property in software
42. Protection against viruses, spyware, and hacking
43. Software deployment and customization
44. Training clients or users of software applications
45. Customer support after deployment of software applications
46. Software warranties and recalls
47. Software change management after release
48. Software maintenance and enhancement
49. Updates and releases of software applications
50. Terminating or withdrawing legacy applications

Following are summary discussions of current best practices for these 50 managerial and technical areas.

1. Best Practices for Minimizing Harm from Layoffs and Downsizing

As this book is written, the global economy is rapidly descending into the worst recession since the Great Depression. As a result, unprecedented numbers of layoffs are occurring. Even worse, a number of technology companies will probably run out of funds and declare bankruptcy.

Observations during previous economic downturns show that companies often make serious mistakes when handling layoffs and downsizing operations. First, since the selection of personnel to be removed is usually made by managers and executives, technical personnel are let go in larger numbers than managerial personnel, which degrades operational performance.

Second, administrative and support personnel such as quality assurance, technical writers, metrics and measurement specialists, secretarial support, program librarians, and the like are usually let go before software engineers and technical personnel. As a result, the remaining technical personnel must take on a variety of administrative tasks for which they are neither trained nor qualified, which also degrades operational performance.

The results of severe layoffs and downsizing usually show up in reduced productivity and quality for several years. While there are no

perfect methods for dealing with large-scale reductions in personnel, some approaches can minimize the harm that usually follows:

Bring in outplacement services to help employees create résumés and also to find other jobs, if available.

For large corporations with multiple locations, be sure to post available job openings throughout the company. The author once observed a large company with two divisions co-located in the same building where one division was having layoffs and the other was hiring, but neither side attempted any coordination.

If yours is a U.S. company that employs offshore workers brought into the United States on temporary visas, it would be extremely unwise during the recession to lay off employees who are U.S. citizens at higher rates than overseas employees. It is even worse to lobby for or to bring in more overseas employees while laying off U.S. citizens. This has been done by several major companies such as Microsoft and Intel, and it results in severe employee morale loss, to say nothing of very bad publicity. It may also result in possible investigations by state and federal officials.

Analyze and prioritize the applications that are under development and in the backlog, and attempt to cut those applications whose ROIs are marginal.

Analyze maintenance of existing legacy applications and consider ways of reducing maintenance staff without degrading security or operational performance. It may be that renovation, restructuring, removal of error-prone modules, and other quality improvements can reduce maintenance staffing but not degrade operational performance.

Calculate the staffing patterns needed to handle the applications in the backlog and under development after low-ROI applications have been purged.

As cuts occur, consider raising the span of control or the number of technical personnel reporting to one manager. Raising the span of control from an average of about 8 technical employees per manager to 12 technical employees per manager is often feasible. In fact, fairly large spans of control may even improve performance by reducing contention and disputes among the managers of large projects.

Do not attempt to skimp on inspections, static analysis, testing, and quality control activities. High quality yields better performance and smaller teams, while low quality results in schedule delays, cost overruns, and other problems that absorb effort with little positive return.

Carefully analyze ratios of technical personnel to specialists such as technical writing, quality assurance, configuration control, and other personnel. Eliminating specialists in significantly larger numbers than software engineers will degrade the operational performance of the software engineers.

In a severe recession, some of the departing personnel may be key employees with substantial information on products, inventions, and intellectual property. While most companies have nondisclosure agreements in place for protection, very few attempt to create an inventory of the knowledge that might be departing with key personnel. If layoffs are handled in a courteous and professional manner, most employees would be glad to leave behind information on key topics. This can be done using questionnaires or “knowledge” interviews. But if the layoffs are unprofessional or callous to employees, don’t expect employees to leave much useful information behind.

In a few cases where there is a complete closure of a research facility, some corporations allow departing employees to acquire rights to intellectual properties such as copyrights and even to patents filed by the employees. The idea is that some employees may form startup companies and thereby continue to make progress on useful ideas that otherwise would drop from view.

As cuts in employment are being made, consider the typical work patterns of software organizations. For a staff that totals 1000 personnel, usually about half are in technical work such as software engineering, 30 percent are specialists of various kinds, and 20 percent are management and staff personnel. However, more time and effort are usually spent finding and fixing bugs than on any other measurable activity.

After downsizing, it could be advantageous to adopt technologies that improve quality, which should allow more productive work from smaller staffs. Therefore topics such as inspections of requirements and design, code inspections, Six Sigma, static analysis, automated testing, and methods that emphasize quality control such as the Team Software Process (TSP) may allow the reduced staffing available to actually have higher productivity than before.

A study of work patterns by the author in 2005 showed that in the course of a normal 220-day working year, only about 47 days were actually spent on developing the planned features of new applications by software engineering technical personnel. About 70 days were spent on testing and bug repairs. (The rest of the year was spent on meetings, administrative tasks, and dealing with changing requirements.)

Therefore improving quality via a combination of defect prevention and more effective defect removal (i.e., inspections and static analysis before testing, automated testing, etc.) could allow smaller staffs to perform the same work as larger staffs. If it were possible to cut down defect removal days to 20 days per year instead of 70 days, that would have the effect of doubling the time available for new development efforts.

Usually one of the first big cuts during a recession is to reduce customer support, with severe consequences in terms of customer satisfaction. Here, too, higher quality prior to delivery would allow smaller

customer support teams to handle more customers. Since customer support tends to be heavily focused on defect issues, it can be hypothesized that every reduction of 220 defects in a delivered application could reduce the number of customer support personnel by one, but would not degrade response time or time to achieve satisfactory solutions. This is based on the assumption that customer support personnel speak to about 30 customers per day, and each released defect is encountered by 30 customers. Therefore each released defect occupies one day for one customer support staff member, and there are 220 working days per year.

Another possible solution would be to renovate legacy applications rather than build new replacements. Renovation and the removal of error-prone modules, plus running static analysis tools and restructuring highly complex code and perhaps converting the code to a newer language, might stretch out the useful lives of legacy applications by more than five years and reduce maintenance staffing by about one person for every additional 120 bugs removed prior to deployment. This is based on the assumption that maintenance programmers typically fix about 10 bugs per month (severity 1 and 2 bugs, that is).

The bottom line is that if U.S. quality control were better than it is today, smaller staffs could actually accomplish more new development than current staffs. Too many days are being wasted on bug removal for defects that could either be prevented or removed prior to testing.

A combination of defect prevention and effective defect removal via inspections, static analysis, and automated and conventional testing could probably reduce development staffing by 25 percent, maintenance staffing by 50 percent, and customer support staffing by about 20 percent without any reduction in operational efficiency, customer satisfaction, or productivity. Indeed development schedules would improve because they usually slip more during testing than at any other time, due to excessive defects. As the economy sinks into recession, it is important to remember not only that “quality is free,” as stated by Phil Crosby, but that it also offers significant economic benefits for software.

One problem that has existed for many years is that few solid economic studies have been performed and published that convincingly demonstrate the value of software quality. A key reason for this is that the two most common metrics for quality, lines of code and cost per defect, are flawed and cannot deal with economics topics. Using defect removal costs per function point is a better choice, but these metrics need to be deployed in organizations that actually accumulate effort, cost, and quality data simultaneously. From studies performed by the author, combinations of defect prevention and defect removal methods that lower defect potentials and raise removal efficiency greater than 95 percent, simultaneously benefit development costs, development schedules, maintenance costs, and customer support costs.

Over and above downsizing, many companies are starting to enforce reduced work months or to require unpaid time off on the part of employees in order to keep cash flow positive. Reduced work periods and reduced compensation for all employees is probably less harmful than cutting staff and keeping compensation constant for the remainder. However, caution is needed because if the number of required days off exceeds certain thresholds, employees may switch from being legally recognized as full-time workers to becoming part-time workers. If this occurs, then their medical benefits, pensions, and other corporate perks might be terminated. Since policies vary from company to company and state to state, there is no general rule, but it is a problem to be reckoned with.

The information technology employees of many state governments, and some municipal governments, have long had benefits that are no longer offered by corporations. These include payment for sick days not used, defined pension programs, accumulating vacation days from year to year, payment for unused vacation days at retirement, and zero-payment health benefits. As state governments face mounting deficits, these extraordinary benefits are likely to disappear in the future.

There are no perfect solutions for downsizing and laying off personnel, but cutting specialists and administrative personnel in large numbers may cause unexpected problems. Also, better quality control and better maintenance or renovation can allow smaller remaining staffs to handle larger workloads without excessive overtime, loss of operational efficiency, or degradation of customer satisfaction.

2. Best Practices for Motivation and Morale of Technical Staff

Many software engineers and other specialists such as quality assurance and technical writers are often high-energy, self-motivated individuals. Psychological studies of software personnel do indicate some interesting phenomena, such as high divorce rates and a tendency toward introversion.

The nature of software development and maintenance work tends to result in long hours and sometimes interruptions even in the middle of the night. That being said, a number of factors are useful in keeping technical staff morale at high levels.

Studies of exit interviews of software engineers at major corporations indicate two distressing problems: (1) the best personnel leave in the largest numbers and (2) the most common reason stated for voluntary attrition is "I don't like working for bad management."

Thus, some sophisticated companies such as IBM have experimented with *reverse appraisals*, where employees evaluate management

performance, as well as *normal appraisals*, where employee performance is evaluated.

Following are some topics noted in a number of leading companies where morale is fairly high, such as IBM, Microsoft, Google, and the like:

Emphasize doing things right, rather than just working long hours to make artificial and probably impossible schedules.

Allow and support some personal projects if the individuals feel that the projects are valuable.

Ensure that marginal or poor managers are weeded out, because poor management drives out good software engineers in larger numbers than any other factor.

Ensure that appraisals are fair, honest, and can be appealed if employees believe that they were incorrectly downgraded for some reason.

Have occasional breakfast or lunch meetings between executives and technical staff members, so that topics of mutual interest can be discussed in an open and nonthreatening fashion.

Have a formal appeal or “open door” program so that technical employees who feel that they have not been treated fairly can appeal to higher-level management. An important corollary of such a program is “no reprisals.” That is, no punishments will be levied against personnel who file complaints.

Have occasional awards for outstanding work. But recall that many small awards such as “dinners for two” or days off are likely to be more effective than a few large awards. But don’t reward productivity or schedules achieved at the expense of quality.

As business or economic situations change, keep all technical personnel apprised of what is happening. They will know if a company is in financial distress or about to merge, so formal meetings to keep personnel up to date are valuable.

Suggestion programs that actually evaluate suggestions and take actions are often useful. But suggestion programs that result in no actions are harmful.

Surprisingly, some overtime tends to raise morale for psychological reasons. Overtime makes projects seem to be valuable, or else they would not require overtime. But excessive amounts of overtime (i.e., 60-hour weeks) are harmful for periods longer than a couple of weeks.

One complex issue is that software engineers in most companies are viewed as members of “professional staffs” rather than hourly workers. Unless software engineers and technical workers are members of unions, they normally do not receive any overtime pay regardless of the hours worked. This issue has legal implications that are outside the scope of this book.

Training and educational opportunities pay off in better morale and also in better performance. Therefore setting aside at least ten days a year

for education either internally or at external events would be beneficial. It is interesting that companies with ten or more days of annual training have higher productivity rates than companies with no training.

Other factors besides these can affect employee morale, but these give the general idea. Fairness, communication, and a chance to do innovative work are all factors that raise the morale of software engineering personnel.

As the global economy slides into a serious recession, job opportunities will become scarce even for top performers. No doubt benefits will erode as well, as companies scramble to stay solvent. The recession and economic crisis may well introduce new factors not yet understood.

3. Best Practices for Motivation and Morale of Managers and Executives

The hundred-year period between 1908 and the financial crisis and recession of 2008 may later be viewed by economic historians as the “golden age” of executive compensation and benefits.

The global financial crisis and the recession followed by attempts to bail out industries and companies that are in severe jeopardy have thrown a spotlight on a troubling topic: the extraordinary salaries, bonuses, and retirement packages for top executives.

Not only do top executives in many industries have salaries of several million dollars per year, but they also have bonuses of millions of dollars, stock options worth millions of dollars, pension plans worth millions of dollars, and “golden parachutes” with lifetime benefits and health-care packages worth millions of dollars.

Other benefits include use of corporate jets and limos, use of corporate boxes at major sports stadiums, health-club memberships, golf club memberships, and scores of other “perks.”

Theoretically these benefits are paid because top executives are supposed to maximize the values of companies for shareholders, expand business opportunities, and guide corporations to successful business opportunities.

But the combination of the financial meltdown and the global recession coupled with numerous instances of executive fraud and malpractice (as shown by Enron) will probably put an end to unlimited compensation and benefits packages. In the United States at least companies receiving federal “bail out” money will have limits on executive compensation. Other companies are also reconsidering executive compensation packages in light of the global recession, where thousands of companies are losing money and drifting toward bankruptcy.

From 2008 onward, executive compensation packages have been under a public spotlight and probably will be based much more closely on corporate profitability and business success than in the past. Hopefully, in

the wake of the financial meltdown, business decisions will be more carefully thought out, and long-range consequences analyzed much more carefully than has been the practice in the past.

Below the levels of the chief executives and the senior vice presidents are thousands of first-, second-, and third-line managers, directors of groups, and other members of corporate management.

At these lower levels of management, compensation packages are similar to those of the software engineers and technical staff. In fact, in some companies the top-ranked software engineers have compensation packages that pay more than first- and some second-line managers receive, which is as it should be.

The skill sets of successful managers in software applications are a combination of management capabilities and technical capabilities. Many software managers started as software engineers, but moved into management due to problem-solving and leadership abilities.

A delicate problem should be discussed. If the span of control or number of technical workers reporting to a manager is close to the national average of eight employees per manager, then it is hard to find qualified managers for every available job. In other words, the ordinary span of control puts about 12.5 percent of workers into management positions, but less than 8 percent are going to be really good at it.

Raising the span of control and converting the less-qualified managers into staff or technical workers might have merit. A frequent objection to this policy is how can managers know the performance of so many employees. However, under the current span of control levels, managers actually spend more time in meetings with other managers than they do with their own people.

As of 2009, software project management is one of the toughest kinds of management work. Software project managers are charged with meeting imposed schedules that may be impossible, with containing costs that may be low, and with managing personnel who are often high-energy and innovative.

When software projects fail or run late, the managers receive the bulk of the blame for the situation, even though some of the problems were due to higher-level schedule constraints or to impossible client demands. It is an unfortunate fact that software project managers have more failures and fewer successes than hardware engineering managers, marketing managers, or other managers.

The main issues facing software project management include schedule constraints, cost constraints, creeping requirements, quality control, progress tracking, and personnel issues.

Scheduling software projects with accuracy is notoriously difficult, and indeed a majority of software projects run late, with the magnitude of the delays correlating with application size. Therefore management

morale tends to suffer because of constant schedule pressures. One way of minimizing this issue is to examine the schedules of similar projects by using historical data. If in-house historical data is not available, then data can be acquired from external sources such as the International Software Benchmarking Standards Group (ISBSG) in Australia. Careful work breakdown structures are also beneficial. The point is, matching project schedules with reality affects management morale. Since costs and schedules are closely linked, the same is true for matching costs to reality.

One reason costs and schedules for software projects tend to exceed initial estimates and budgets is creeping requirements. Measurements using function points derived from requirements and specifications find the average rate of creep is about 2 percent per calendar month. This fact can be factored into initial estimates once it is understood. In any case, significant changes in requirements need to trigger fresh schedule and cost estimates. Failure to do this leads to severe overruns, damages management credibility, and of course low credibility damages management morale.

Most software projects run into schedule problems during testing due to excessive defects. Therefore upstream defect prevention and pretest defects removal activities such as inspections and static analysis are effective therapies against schedule and cost overruns. Unfortunately, not many managers know this, and far too many tend to skimp on quality control. However, if quality control is good, morale is also likely to be good, and the project will have a good chance of staying on target. Therefore excellence in quality control tends to benefit both managerial and professional staff morale.

Tracking software progress and reporting on problems is perhaps the weakest link in software project management. In many lawsuits for breach of contract, depositions reveal that serious problems were known to exist by the technical staff and first-line managers, but were not revealed to higher-level management or to clients until it was too late to fix them. The basic rule of project tracking should be: "no surprises." Problems seldom go away by themselves, so once they are known, report them and try and solve them. This will benefit both employee and management morale much more than sweeping problems under the rug.

Personnel issues are also important for software projects. Since many software engineers are self-motivated, have high energy levels, and are fairly innovative, management by example is better than management by decree. Managers need to be fair and consistent with appraisals and to ensure that personnel are kept informed of all issues arriving from higher up in the company, such as possible layoffs or sales of business units.

Unfortunately, software management morale is closely linked to software project successes, and as of 2009, far too many projects fail.

Basing plans and estimates on historical data and benchmarks rather than on client demands would also improve management morale. Historical data is harder to overturn than estimates.

4. Best Practices for Selection and Hiring of Software Personnel

As the global economy slides into a severe recession, many companies are downsizing or even going out of business. As a result, it is a buyers' market for those companies that are doing well and expanding. At no time in history have so many qualified software personnel been on the job market at the same time as at the end of 2008 and during 2009.

It is still important for companies to do background checks of all applicants, since false résumés are not uncommon and are likely to increase due to the recession. Also, multiple interviews with both management and technical staff are beneficial to see how applicants might fit into teams and handle upcoming projects.

If entry-level personnel are being considered for their first jobs out of school, some form of aptitude testing is often used. Some companies also use psychological interviews with industrial psychologists. However, these methods have ambiguous results.

What seem to give the best results are multiple interviews combined with a startup evaluation period of perhaps six months. Successful performance during the evaluation period is a requirement for joining the group on a full-time regular basis.

5. Best Practices for Appraisals and Career Planning for Software Personnel

After about five years on the job, software engineers tend to reach a major decision on their career path. Either the software engineer wants to stay in technical work, or he or she wants to move into management.

Technical career paths can be intellectually satisfying and also have good compensation plans in many leading companies. Positions such as "senior software engineer" or "corporate fellow" or "advisory architect" are not uncommon and are well respected. This is especially true for corporations such as IBM that have research divisions where top-gun engineers can do very innovative projects of their own choosing.

While some managers do continue to perform technical work, their increasing responsibilities in the areas of schedule management, cost management, quality management, and personnel management obviously reduce the amount of time available for technical work.

Software engineering has several different career paths, with development programming, maintenance programming, business analysis,

systems analysis, quality assurance, architecture, and testing all moving in somewhat different directions.

These various specialist occupations bring up the fact that software engineering is not yet a full profession with specialization that is recognized by state licensing boards. Many kinds of voluntary specialization are available in topics such as testing and quality assurance, but these have no legal standing.

Large corporations can employ as many as 90 different kinds of specialists in their software organizations, including technical writers, software quality assurance specialists, metrics specialists, integration specialists, configuration control specialists, database administrators, program librarians, and many more. However, these specialist occupations vary from company to company and have no standard training or even standard definitions.

Not only are there no standard job titles, but also many companies use a generic title such as “member of the technical staff,” which can encompass a dozen specialties or more.

In a study of software specialties in large companies, it was common to find that the human resource groups had no idea of what specialties were employed. It was necessary to go on site and interview managers and technical workers to find out this basic information.

In the past, one aspect of career planning for the best technical personnel and managers included “job hopping” from one company to another. Internal policies within many companies limited pay raises, but switching to another company could bypass those limits. However, as the economy retracts, this method is becoming difficult. Many companies now have hiring freezes and are reducing staffs rather than expanding. Indeed, some may enter bankruptcy.

6. Best Practices for Early Sizing and Scope Control of Software Applications

For many years, predicting the size of software applications was difficult and very inaccurate. Calculating size by using function-point metrics had to be delayed until requirements were known, but by then it was too late for the initial software cost estimates and schedule plans. Size in terms of source code could only be guessed at by considering the sizes of similar applications, if any existed and their sizes were known.

However, in 2008 and 2009, new forms of size analysis became available. Now that the International Software Benchmarking Standards Group (ISBSG) has reached a critical mass with perhaps 5,000 software applications, it is possible to acquire reliable size data for many kinds of software applications from the ISBSG.

Since many applications are quite similar to existing applications, acquiring size data from ISBSG is becoming a standard early-phase activity. This data also includes schedule and cost information, so it is even more valuable than size alone. However, the ISBSG data supports function point metrics rather than lines of code. Since function points are a best practice and the lines of code approach is malpractice, this is not a bad situation, but it will reduce the use of ISBSG benchmarks by companies still locked into LOC metrics.

For novel software or for applications without representation in the ISBSG data, several forms of high-speed sizing are now available. A new method based on pattern matching can provide fairly good approximations of size in terms of function points, source code, and even for other items such as pages of specifications. This method also predicts the rate at which requirements are likely to grow during development, which has long been a weak link in software sizing.

Other forms of sizing include new kinds of function point approximations or “light” function point analysis, which can predict function point size in a matter of a few minutes, as opposed to normal counting speeds of only about 400 function points per day.

Early sizing is a necessary precursor to accurate estimation and also a precursor to risk analysis. Many kinds of risks are directly proportional to application size, so the earlier the size is known, the more complete the risk analysis.

For small applications in the 1000–function point range, all features are usually developed in a single release. However, for major applications in the 10,000– to 100,000–function point range, multiple releases are the norm.

(For small projects using the Agile approach, individual features or functions are developed in short intervals called *sprints*. These are usually in the 100– to 200–function point range.)

Because schedules and costs are directly proportional to application size, major systems are usually segmented into multiple releases at 12- to 18-month intervals. Knowing the overall size, and then the sizes of individual functions and features, it is possible to plan an effective release strategy that may span three to four consecutive releases. By knowing the size of each release, accurate schedule and cost estimating becomes easier to perform.

Early sizing using pattern matching can be done before requirements are known because this method is based on external descriptions of a software application and then by matching the description against the “patterns” of other similar applications.

The high-speed function point methods are offset in time and need at least partial requirements to operate successfully.

The best practice for early sizing is to use one or more (or all) of the high-speed sizing approaches before committing serious funds to a software application. If the size is large enough so that risks are likely to be severe, then corrective actions can be applied before starting development, when there is adequate time available.

Two innovative methods for software scope control have recently surfaced and seem to be effective. One is called *Northern Scope* because it originated in Finland. The other is called *Southern Scope* because it originated in Australia. The two are similar in that they attempt to size applications early and to appoint a formal scope manager to monitor growth of possible new features. By constantly focusing on scope and growth issues, software projects using these methods have more success in their initial releases because, rather than stuffing too many late features into the first release, several follow-on releases are identified and populated early.

These new methods of scope control have actually led to the creation of a new position called *scope manager*. This new position joins several other new jobs that have emerged within the past few years, such as web master and scrum master.

Sizing has been improving in recent years, and the combination of ISBSG benchmarks plus new high-speed sizing methods shows promise of greater improvements in the future.

7. Best Practices for Outsourcing Software Applications

For the past 20 years, U.S. corporations have been dealing with a major business issue: should software applications be built internally, or turned over to a contractor or outsourcer for development. Indeed the issue is bigger than individual applications and can encompass all software development operations, all software maintenance operations, all customer support operations, or the entire software organization lock, stock, and barrel.

The need for best practices in outsource agreements is demonstrated by the fact that within about two years, perhaps 25 percent of outsource agreements will have developed some friction between the clients and the outsource vendors. Although results vary from client to client and contractor to contractor, the overall prognosis of outsourcing within the United States approximates the following distribution, shown in Table 2-1, is derived from observations among the author's clients.

Software development and maintenance are expensive operations and have become major cost components of corporate budgets. It is not uncommon for software personnel to exceed 5 percent of total corporate employment, and for the software and computing budgets to exceed 10 percent of annual corporate expenditures.

TABLE 2-1 Approximate Distribution of U.S. Outsource Results After 24 Months

Results	Percent of Outsource Arrangements
Both parties generally satisfied	70%
Some dissatisfaction by client or vendor	15%
Dissolution of agreement planned	10%
Litigation between client and contractor probable	4%
Litigation between client and contractor in progress	1%

Using the function point metric as the basis of comparison, most large companies now own more than 2.5 million function points as the total volume of software in their mainframe portfolios, and some very large companies such as AT&T and IBM each own well over 10 million function points.

As an example of the more or less unplanned growth of software and software personnel in modern business, some of the larger banks and insurance companies now have software staffs that number in the thousands. In fact, software and computing technical personnel may compose the largest single occupation group within many companies whose core business is far removed from software.

As software operations become larger, more expensive, and more widespread, the executives of many large corporations are asking a fundamental question: *Should software be part of our core business or not?*

This is not a simple question to answer, and the exploration of some of the possibilities is the purpose of this section. You would probably want to make software a key component of your core business operations under these conditions:

- You sell products that depend upon your own proprietary software.
- Your software is currently giving your company significant competitive advantage.
- Your company's software development and maintenance effectiveness are far better than your competitors'.

You might do well to consider outsourcing of software if its relationship to your core business is along the following lines:

- Software is primarily used for corporate operations, not as a product.
- Your software is not particularly advantageous compared with your competitors'.
- Your development and maintenance effectiveness are marginal.

Once you determine that outsourcing either specific applications or portions of your software operations is a good match to your business plans, some of the topics that need to be included in outsource agreements include

- The sizes of software contract deliverables must be determined during negotiations, preferably using function points.
- Cost and schedule estimation for applications must be formal and complete.
- Creeping user requirements must be dealt with in the contract in a way that is satisfactory to both parties.
- Some form of independent assessment of terms and progress should be included.
- Anticipated quality levels should be included in the contract.
- Effective software quality control steps must be utilized by the vendor.
- If the contract requires that productivity and quality improvements be based on an initial baseline, then great care must be utilized in creating a baseline that is accurate and fair to both parties.
- Tracking of progress and problems during development must be complete and not overlook or deliberately conceal problems.

Fortunately, all eight of these topics are amenable to control once they are understood to be troublesome if left to chance. An interesting sign that an outsource vendor is capable of handling large applications is if they utilize state-of-the-art quality control methods.

The state-of-the-art for large software applications includes sophisticated defect prediction methods, measurements of defect removal efficiency, utilization of defect prevention methods, utilization of formal design and code inspections, presence of a Software Quality Assurance (SQA) department, use of testing specialists, and usage of a variety of quality-related tools such as defect tracking tools, complexity analysis tools, debugging tools, and test library control tools.

Another important best practice for software outsource contracts involves dealing with changing requirements, which always occur. For software development contracts, an effective way of dealing with changing user requirements is to include a sliding scale of costs in the contract itself. For example, suppose a hypothetical contract is based on an initial agreement of \$1000 per function point to develop an application of 1000 function points in size, so that the total value of the agreement is \$1 million.

The contract might contain the following kind of escalating cost scale for new requirements added downstream:

Initial 1000 function points	=	\$1000 per function point
Features added more than 3 months after contract signing	=	\$1100 per function point
Features added more than 6 months after contract signing	=	\$1250 per function point
Features added more than 9 months after contract signing	=	\$1500 per function point
Features added more than 12 months after contract signing	=	\$1750 per function point
Features deleted or delayed at user request	=	\$250 per function point

Similar clauses can be utilized with maintenance and enhancement outsource agreements, on an annual or specific basis, such as:

Normal maintenance and defect repairs	=	\$250 per function point per year
Mainframe to client-server conversion	=	\$500 per function point per system
Special “mass update” search and repair	=	\$75 per function point per system

(Note that the actual cost per function point for software produced in the United States runs from a low of less than \$300 per function point for small end-user projects to a high of more than \$5,000 per function point for large military software projects. The data shown here is for illustrative purposes and should not actually be used in contracts as it stands.)

The advantage of the use of function point metrics for development and maintenance contracts is that they are determined from the user requirements and cannot be unilaterally added or subtracted by the contractor.

In summary form, successful software outsourced projects in the 10,000–function point class usually are characterized by these attributes:

- Less than 1 percent monthly requirements changes after the requirements phase
- Less than 1 percent total volume of requirements “churn”
- Fewer than 5.0 defects per function point in total volume
- More than 65 percent defect removal efficiency before testing begins
- More than 96 percent defect removal efficiency before delivery

Also in summary form, unsuccessful outsource software projects in the 10,000–function point class usually are characterized by these attributes:

- More than 2 percent monthly requirements changes after the requirements phase
- More than 5 percent total volume of requirements churn
- More than 6.0 defects per function point in total volume
- Less than 35 percent defect removal efficiency before testing begins
- Less than 85 percent defect removal efficiency before delivery

In performing “autopsies” of cancelled or failed projects, it is fairly easy to isolate the attributes that distinguish disasters from successes. Experienced project managers know that false optimism in estimates, failure to plan for changing requirements, inadequate quality approaches, and deceptive progress tracking lead to failures and disasters. Conversely, accurate estimates, careful change control, truthful progress tracking, and topnotch quality control are stepping-stones to success.

Another complex topic is what happens to the employees whose work is outsourced. The best practice is that they will be reassigned within their own company and will be used to handle software applications and tasks that are not outsourced. However, it may be that the outsourcing company will take over the personnel, which is usually a very good to fair practice based on the specifics of the companies involved. The worst case is that the personnel whose work is outsourced will be laid off.

In addition to outsourcing entire applications or even portfolios, there are also *partial* outsource agreements for specialized topics such as testing, static analysis, quality assurance, and technical writing. However, these partial assignments may also be done in-house by contractors who work on-site, so it is hard to separate outsourcing from contract work for these special topics.

Whether to outsource is an important business decision. Using best practices for the contract between the outsource vendor and the client can optimize the odds of success, and minimize the odds of expensive litigation.

In general, maintenance outsource agreements are less troublesome and less likely to end up in court than development outsource agreements. In fact, if maintenance is outsourced, that often frees up enough personnel so that application backlogs can be reduced and major new applications developed.

As the economy worsens, there is uncertainty about the future of outsourcing. Software will remain an important commodity, so outsourcing

will no doubt stay as an important industry. However, the economic crisis and the changes in inflation rates and currency values may shift the balance of offshore outsourcing from country to country. In fact, if deflation occurs, even the United States could find itself with expanding capabilities for outsourcing.

8. Best Practices for Using Contractors and Management Consultants

As this book is written in 2009, roughly 10 percent to 12 percent of the U.S. software population are not full-time employees of the companies that they work for. They are contractors or consultants.

On any given business day in any given Fortune 500 company, roughly ten management consultants will be working with executives and managers on topics that include benchmarks, baselines, strategic planning, competitive analysis, and a number of other specialized topics.

Both of these situations can be viewed as being helpful practices and are often helpful enough to move into the best practice category.

All companies have peaks and valleys in their software workloads. If full-time professional staff is on board for the peaks, then they won't have any work during the valley periods. Conversely, if full-time professional staffing is set up to match the valleys, when important new projects appear, there will be a shortage of available technical staff members.

What works best is to staff closer to the valley or midpoint of average annual workloads. Then when projects occur that need additional resources, bring in contractors either for the new projects themselves, or to take over standard activities such as maintenance and thereby free up the company's own technical staff. In other words, having full-time staffing levels 5 percent to 10 percent below peak demand is a cost-effective strategy.

The primary use of management consultants is to gain access to special skills and knowledge that may not be readily available in-house. Examples of some of the topics where management consultants have skills that are often lacking among full-time staff include

- Benchmarks and comparisons to industry norms
- Baselines prior to starting process improvements
- Teaching new or useful technologies such as Agile, Six Sigma, and others
- Measurement and metrics such as function point analysis
- Selecting international outsource vendors
- Strategic and market planning for new products

- Preparing for litigation or defending against litigation
- Assisting in process improvement startups
- Attempting to turn around troubled projects
- Offering advice about IPOs, mergers, acquisitions, and venture financing

Management consultants serve as a useful conduit for special studies and information derived from similar companies. Because management consultants are paid for expertise rather than for hours worked, many successful management consultants are in fact top experts in their selected fields.

Management consultants have both a strategic and a tactical role. Their strategic work deals with long-range topics such as market positions and optimizing software organization structures. Their tactical role is in areas such as Six Sigma, starting measurement programs, and aiding in collecting function point data.

In general, usage both of hourly contractors for software development and maintenance, and of management consultants for special topics benefits many large corporations and government agencies. If not always best practices, the use of contractors and management consultants are usually at least good practices.

9. Best Practices for Selecting Software Methods, Tools, and Practices

Unfortunately, careful selection of methods, tools, and practices seldom occurs in the software industry. Either applications are developed using methods already in place, or there is rush to adopt the latest fad such as CASE, I-CASE, RAD, and today, Agile in several varieties.

A wiser method of selecting software development methods would be to start by examining benchmarks for applications that used various methods, and then to select the method or methods that yield the best results for specific sizes and types of software projects.

As this book is written, thousands of benchmarks are now available from the nonprofit International Software Benchmarking Standards Group (ISBSG), and most common methods are represented. Other benchmark sources are also available, such as Software Productivity Research, the David Consulting Group, and others. However, ISBSG is available on the open market to the public and is therefore easiest to access.

Among the current choices for software development methods can be found (in alphabetical order) Agile development, clean-room development, Crystal development, Dynamic Systems Development Method (DSDM), extreme programming (XP), hybrid development, iterative development, object-oriented development, pattern-based development,

Personal Software Process (PSP), rapid application development (RAD), Rational Unified Process (RUP), spiral development, structured development, Team Software Process (TSP), V-model development, and waterfall development.

In addition to the preceding, a number of partial development methods deal with specific phases or activities. Included in the set of partial methods are (in alphabetical order) code inspections, data-state design, design inspections, flow-based programming, joint application design (JAD), Lean Six Sigma, pair programming, quality function deployment (QFD), requirements inspections, and Six Sigma for software. While the partial methods are not full development methods, they do have a measurable impact on quality and productivity.

It would be useful to have a checklist of topics that need to be evaluated when selecting methods and practices. Among these would be

Suitability by application size How well the method works for applications ranging from 10 function points to 100,000 function points. The Agile methods seem to work well for smaller applications, while Team Software Process (TSP) seems to work well for large systems, as does the Rational Unified Process (RUP). Hybrid methods also need to be included.

Suitability by application type How well the method works for embedded software, systems software, web applications, information technology applications, commercial applications, military software, games, and the like.

Suitability by application nature How well the method works for new development, for enhancements, for warranty repairs, and for renovation of legacy applications. There are dozens of development methodologies, but very few of these also include maintenance and enhancement. As of 2009, the majority of “new” software applications are really replacements for aging legacy applications. Therefore data mining of legacy software for hidden requirements, enhancements, and renovation should be standard features in software methodologies.

Suitability by attribute How well the method supports important attributes of software applications including but not limited to defect prevention, defect removal efficiency, minimizing security vulnerabilities, achieving optimum performance, and achieving optimum user interfaces. A development method that does not include both quality control and measurement of quality is really unsuitable for critical software applications.

Suitability by activity How well the method supports requirements; architecture; design; code development; reusability; pretest inspections; static analysis; testing; configuration control; quality assurance; user information; and postrelease maintenance, enhancement, and customer support.

The bottom line is that methodologies should be deliberately selected to match the needs of specific projects, not used merely because they are a current fad or because no one knows of any other approach.

As this book is written, formal technology selection seems to occur for less than 10 percent of software applications. About 60 percent use whatever methods are the local custom, while about 30 percent adopt the most recent popular method such as Agile, whether or not that method is a good match for the application under development.

Development process refers to a standard set of activities that are performed in order to build a software application. (*Development process* and *development methodology* are essentially synonyms.)

For conventional software development projects, about 25 activities and perhaps 150 tasks are usually included in the work breakdown structure (WBS). For Agile projects, about 15 activities and 75 tasks are usually included in the work breakdown structure.

The work breakdown structure of large systems will vary based on whether the application is to be developed from scratch, or it involves modifying a package or modifying a legacy application. In today's world circa 2009, projects that are modifications are actually more numerous than complete new development projects.

An effective development process for projects in the nominal 10,000–function point range that include acquisition and modification of commercial software packages would resemble the following:

1. Requirements gathering
2. Requirements analysis
3. Requirements inspections
4. Data mining of existing similar applications to extract business rules
5. Architecture
6. External design
7. Internal design
8. Design inspections
9. Security vulnerability analysis
10. Formal risk analysis
11. Formal value analysis
12. Commercial-off-the-shelf (COTS) package analysis
13. Requirements/package mapping
14. Contacting package user association

15. Package licensing and acquisition
16. Training of development team in selected package
17. Design of package modifications
18. Development of package modifications
19. Development of unique features
20. Acquisition of certified reusable materials
21. Inspection of package modifications
22. Documentation of package modifications
23. Inspections of documentation and HELP screens
24. Static analysis of package modifications
25. General testing of package modifications
26. Specialized testing of package modifications (performance, security)
27. Quality assurance review of package modifications
28. Training of user personnel in package and modifications
29. Training of customer support and maintenance personnel
30. Deployment of package modifications

These high-level activities are usually decomposed into a full work breakdown structure with between 150 and more than 1000 tasks and lower-level activities. Doing a full work breakdown structure is too difficult for manual approaches on large applications. Therefore, project management tools such as Artemis Views, Microsoft Project, Primavera, or similar tools are always used in leading companies.

Because requirements change at about 2 percent per calendar month, each of these activities must be performed in such a manner that changes are easy to accommodate during development; that is, some form of iterative development is necessary for each major deliverable.

However, due to fixed delivery schedules that may be contractually set, it is also mandatory that large applications be developed with multiple releases in mind. At a certain point, all features for the initial release must be frozen, and changes occurring after that point must be added to follow-on releases. This expands the concept of iterative development to a multiyear, multirelease philosophy.

A number of sophisticated companies such as IBM and AT&T have long recognized that change is continuous with software applications. These companies tend to have fixed release intervals, and formal planning for releases spreads over at least the next two releases after the current release.

Formal risk analysis and value analysis are also indicators of software sophistication. As noted in litigation, failing projects don't perform risk analyses, so they tend to be surprised by factors that delay schedules or cause cost overruns.

Sophisticated companies always perform formal risk analysis for major topics such as possible loss of personnel, changing requirements, quality, and other key topics. However, one form of risk analysis is not done very well, even by most sophisticated companies: security vulnerabilities. Security analysis, if it is done at all, is often an afterthought.

A number of approaches have proven track records for large software projects. Among these are the capability maturity model (CMM) by the Software Engineering Institute (SEI) and the newer Team Software Process (TSP) and Personal Software Process (PSP) created by Watts Humphrey and also supported by the SEI. The Rational Unified Process (RUP) also has had some successes on large software projects. For smaller applications, various flavors of Agile development and extreme programming (XP) have proven track records of success. Additional approaches such as object-oriented development, pattern matching, Six Sigma, formal inspections, prototypes, and reuse have also demonstrated value for large applications.

Over and above "pure" methods such as the Team Software Process (TSP), hybrid approaches are also successful. The hybrid methods use parts of several different approaches and blend them together to meet the needs of specific projects. As of 2009, hybrid or blended development approaches seem to outnumber pure methods in terms of usage.

Overall, hybrid methods that use features of Six Sigma, the capability maturity model, Agile, and other methods have some significant advantages. The reason is that each of these methods in "pure" form has a rather narrow band of project sizes and types for which they are most effective. Combinations and hybrids are more flexible and can match the characteristics of any size and any type. However, care and expertise are required in putting together hybrid methods to be sure that the best combinations are chosen. It is a job for experts and not for novices.

There are many software process improvement network (SPIN) chapters in major cities throughout the United States. These organizations have frequent meetings and serve a useful purpose in disseminating information about the successes and failures of various methods and tools.

It should be obvious that any method selected should offer improvements over former methods. For example, current U.S. averages for software defects total about 5.00 per function point. Defect removal efficiency averages about 85 percent, so delivered defects amount to about 0.75 per function point.

Any new process should lower defect potentials, raise defect removal efficiency, and reduce delivered defects. Suggested values for an improved

process would be no more than 3.00 defects per function point, 95 percent removal efficiency, and delivered defects of no more than 0.15 defect per function point.

Also, any really effective process should raise productivity and increase the volume of certified reusable materials used for software construction.

10. Best Practices for Certifying Methods, Tools, and Practices

The software industry tends to move from fad to fad with each methodology du jour making unsupported claims for achieving new levels of productivity and quality. What would be valuable for the software industry is a nonprofit organization that can assess the effectiveness of methods, tools, and practices in an objective fashion.

What would also be useful are standardized measurement practices for collecting productivity and quality data for all significant software projects.

This is not an easy task. It is unfeasible for an evaluation group to actually try out or use every development method, because such usage in real life may last for several years, and there are dozens of them.

What probably would be effective is careful analysis of empirical results from projects that used various methods, tools, and practices. Data can be acquired from benchmark sources such as the International Software Benchmarking Standards Group (ISBSG), or from other sources such as the Finnish Software Metrics Association.

To do this well requires two taxonomies: (1) a taxonomy of software applications that will provide a structure for evaluating methods by size and type of project and (2) a taxonomy of software methods and tools themselves.

A third taxonomy, of software feature sets, would also be useful, but as of 2009, does not exist in enough detail to be useful. The basic idea of all three taxonomies is to support pattern matching. In other words, applications, their feature sets, and development methods all deal with common issues, and it would be useful if the patterns associated with these issues could become visible. That would begin to move the industry toward construction of software from standard reusable components.

The two current taxonomies deal with what kinds of software might use the method, and what features the method itself contains.

It would not be fair to compare the results of a large project of greater than 10,000 function points with a small project of less than 1000 function points. Nor would it be fair to compare an embedded military application against a web project. Therefore a standard taxonomy for placing software projects is a precursor for evaluating and selecting methods.

From performing assessment and benchmark studies with my colleagues over the years, a four-layer taxonomy seems to provide a suitable structure for software applications:

Nature The term *nature* refers to whether the project is a new development, an enhancement, a renovation, or something else. Examples of the nature parameter include new development, enhancement of legacy software, defect repairs, and conversion to a new platform.

Scope The term *scope* identifies the size range of the project running from a module through an enterprise resource planning (ERP) package. Sizes are expressed in terms of function point metrics as well as source code. Size ranges cover a span that runs from less than 1 function point to more than 100,000 function points. To simplify analysis, sizes can be discrete, that is, 1, 10, 100, 1000, 10,000, and 100,000 function points. Examples of the scope parameter include prototype, evolutionary prototype, module, reusable module, component, stand-alone program, system, and enterprise system.

Class The term *class* identifies whether the project is for external use within the developing organization, or whether it is to be released externally either on the Web or in some other form. Examples of the class parameter include internal applications for a single location, internal applications for multiple locations, external applications for the public domain (open source), external applications to be marketed commercially, external applications for the Web, and external applications embedded in hardware devices.

Type The term *type* refers to whether the application is embedded software, information technology, a military application, an expert system, a telecommunications application, a computer game, or something else. Examples of the type parameter include batch applications, interactive applications, web applications, expert systems, robotics applications, process-control applications, scientific software, neural net applications, and hybrid applications that contain multiple types concurrently.

This four-part taxonomy can be used to define and compare software projects to ensure that similar applications are being compared. It is also interesting that applications that share the same patterns on this taxonomy are also about the same size when measured using function point metrics.

The second taxonomy would define the features of the development methodology itself. There are 25 topics that should be included:

Proposed Taxonomy for Software Methodology Analysis

1. Team organization
2. Specialization of team members

3. Project management—planning and estimating
4. Project management—tracking and control
5. Change control
6. Architecture
7. Business analysis
8. Requirements
9. Design
10. Reusability
11. Code development
12. Configuration control
13. Quality assurance
14. Inspections
15. Static analysis
16. Testing
17. Security
18. Performance
19. Deployment and customization of large applications
20. Documentation and training
21. Nationalization
22. Customer support
23. Maintenance (defect repairs)
24. Enhancement (new features)
25. Renovation

These 25 topics are portions of an approximate 30-year life cycle that starts with initial requirements and concludes with final withdrawal of the application many years later. When evaluating methods, this checklist can be used to show which portions of the timeline and which topics the methodology supports.

Agile development, for example, deals with 8 of these 25 factors:

1. Team organization
2. Project management—planning and estimating
3. Change control
4. Requirements
5. Design

6. Code development
7. Configuration control
8. Testing

In other words, Agile is primarily used for new development of applications rather than for maintenance and enhancements of legacy applications.

The Team Software Process (TSP) deals with 16 of the 25 factors:

1. Team organization
2. Specialization of team members
3. Project management—planning and estimating
4. Project management—tracking and control
5. Change control
6. Requirements
7. Design
8. Reusability
9. Code development
10. Configuration control
11. Quality assurance
12. Inspections
13. Static analysis
14. Testing
15. Security
16. Documentation and training

TSP is also primarily a development method, but one that concentrates on software quality control and also that includes project management components for planning and estimating.

Another critical aspect of evaluating software methods, tools, and practices is to measure the resulting productivity and quality levels. Measurement is a weak link for the software industry. To evaluate the effectiveness of methods and tools, great care must be exercised. Function point metrics are best for evaluation and economic purposes. Harmful and erratic metrics such as lines of code and cost per defect should be avoided.

However, to ensure apples-to-apples comparison between projects using specific methods, the measures need to granular-down to the level of specific activities. If only project- or phase-level data is used, it will be too inaccurate to use for evaluations.

Although not every project uses every activity, the author makes use of a generalized activity chart of accounts for collecting benchmark data at the activity level:

Chart of Accounts for Activity-Level Software Benchmarks

1. Requirements (initial)
2. Requirements (changed and added)
3. Team education
4. Prototyping
5. Architecture
6. Project planning
7. Initial design
8. Detail design
9. Design inspections
10. Coding
11. Reusable material acquisition
12. Package acquisition
13. Code inspections
14. Static analysis
15. Independent verification and validation
16. Configuration control
17. Integration
18. User documentation
19. Unit testing
20. Function testing
21. Regression testing
22. Integration testing
23. Performance testing
24. Security testing
25. System testing
26. Field testing
27. Software quality assurance
28. Installation
29. User training
30. Project management

Unless specific activities are identified, it is essentially impossible to perform valid comparisons between projects.

Software quality also needs to be evaluated. While cost of quality (COQ) would be preferred, two important supplemental measures should always be included. These are defect potentials and defect removal efficiency.

The *defect potential* of a software application is the total number of bugs found in requirements, design, code, user documents, and bad fixes. The *defect removal efficiency* is the percentage of bugs found prior to delivery of software to clients.

As of 2009, average values for defect potentials are

Defect Origins	Defects per Function Point
Requirements bugs	1.00
Design bugs	1.25
Coding bugs	1.75
Documentation bugs	0.60
Bad fixes (secondary bugs)	0.40
TOTAL	5.00

Cumulative defect removal efficiency before delivery is only about 85 percent. Therefore methods should be evaluated in terms of how much they reduce defect potentials and increase defect removal efficiency levels. Methods such as the Team Software Process (TSP) that lower potentials below 3.0 bugs per function point and that raise defect removal efficiency levels above 95 percent are generally viewed as best practices.

Productivity also needs to be evaluated. The method used by the author is to select an average or midpoint approach such as Level 1 on the capability maturity model integration (CMMI) as a starting point. For example, average productivity for CMMI 1 applications in the 10,000–function point range is only about 3 function points per staff month. Alternative methods that improve on these results, such as Team Software Process (TSP) or the Rational Unified Process (RUP), can then be compared with the starting value. Of course some methods may degrade productivity, too.

The bottom line is the evaluating software methodologies, tools, and practices scarcely performed at all circa 2009. A combination of activity-level benchmark data from completed projects, a formal taxonomy for pinning down specific types of software applications, and a formal taxonomy for identifying features of the methodology are all needed. Accurate quality data in terms of defect potentials and defect removal efficiency levels is also needed.

One other important topic for certification would be to show improvements versus current U.S. averages. Because averages vary by size and type of application, a sliding scale is needed. For example, current average schedules from requirements to delivery can be approximated by raising the function point total of the application to the 0.4 power. Ideally, an optimal development process would reduce the exponent to the 0.3 power.

Current defect removal efficiency for software applications in the United States is only about 85 percent. An improved process should yield results in excess of 95 percent.

Defect potentials or total numbers of bugs likely to be encountered can be approximated by raising the function point total of the application to the 1.2 power, which results in alarmingly high numbers of defects for large systems. An improved development process should lower defect potentials below about the 1.1 power.

The volume of certified reusable material in current software applications runs from close to 0 percent up to perhaps 50 percent, but the average value is less than 25 percent. The software industry would be in much better shape economically if the volume of certified reusable materials could top 85 percent on average, and reach 95 percent for relatively common kinds of applications.

The bottom line is that certification needs to look at quantitative results and include information on benefits from adopting new methods. One additional aspect of certification is to scan the available reports and benchmarks from the International Software Benchmarking Standards Group (ISBSG). As their collection of historical benchmarks rises above 5,000 projects, more and more methods are represented in enough detail to carry out multiple-regression studies and to evaluate their impacts.

11. Best Practices for Requirements of Software Applications

As of 2009, more than 80 percent of software applications are not new in the sense that such applications are being developed for the very first time. Most applications today are replacements for older and obsolete applications.

Because these applications are obsolete, it usually happens that their written specifications have been neglected and are out of date. Yet in spite of the lack of current documents, the older applications contain hundreds or thousands of business rules and algorithms that need to be transferred to the new application.

Therefore, as of 2009, requirements analysis should not deal only with new requirements but should also include data mining of the legacy code to extract the hidden business rules and algorithms. Some tools

are available to do this, and also many maintenance workbenches can display code and help in the extraction of latent business rules.

Although clear requirements are a laudable goal, they almost never occur for nominal 10,000–function point software applications. The only projects the author has observed where the initial requirements were both clear and unchanging were for specialized small applications below 500 function points in size.

Businesses are too dynamic for requirements to be completely unchanged for large applications. Many external events such as changes in tax laws, changes in corporate structure, business process reengineering, or mergers and acquisitions can trigger changes in software application requirements. The situation is compounded by the fact that large applications take several years to develop. It is unrealistic to expect that a corporation can freeze all of its business rules for several years merely to accommodate the needs of a software project.

The most typical scenario for dealing with the requirements of a nominal 10,000–function point application would be to spend several months in gathering and analyzing the initial requirements. Then as design proceeds, new and changed requirements will arrive at a rate of roughly 2 percent per calendar month. The total volume of requirements surfacing after the initial requirements exercise will probably approach or even exceed 50 percent. These new and changing requirements will eventually need to be stopped for the first release of the application, and requirements surfacing after about 9 to 12 months will be aimed at follow-on releases of the application.

The state of the art for gathering and analyzing the requirements for 10,000–function point projects includes the following:

- Utilization of joint application design (JAD) for initial requirements gathering
- Utilization of quality function deployment (QFD) for quality requirements
- Utilization of security experts for security analysis and vulnerability prevention
- Utilization of prototypes for key features of new applications
- Mining legacy applications for requirements and business rules for new projects
- Full-time user involvement for Agile projects
- Ensuring that requirements are clearly expressed and can be understood
- Utilization of formal requirement inspections with both users and vendors

- Creation of a joint client/vendor change control board
- Selection of domain experts for changes to specific features
- Ensuring that requirements traceability is present
- Multirelease segmentation of requirements changes
- Utilization of automated requirements analysis tools
- Careful analysis of the features of packages that will be part of the application

The lowest rates of requirements changes observed on 10,000–function point projects are a little below 0.5 percent a month, with an accumulated total of less than 10 percent compared with the initial requirements. However, the maximum amount of growth has topped 200 percent. Average rates of requirements change run between 1 percent and 3 percent per calendar month during the design and coding phases, after which changes are deferred to future releases.

The concurrent use of JAD sessions, careful analysis of requirements, requirements inspections, and prototypes can go far to bring the requirements process under technical and management control.

Although the results will not become visible for many months or sometimes for several years, the success or failure of a large software project is determined during the requirements phase. Successful projects will be more complete and thorough in gathering and analyzing requirements than failures. As a result, successful projects will have fewer changes and lower volumes of requirements creep than failing projects.

However, due to the fact that most new applications are partial replicas of existing legacy software, requirements should include data mining to extract latent business rules and algorithms.

12. Best Practices for User Involvement in Software Projects

It is not possible to design and build nominal 10,000–function point business applications without understanding the requirements of the users. Further, when the application is under development, users normally participate in reviews and also assist in trials of specific deliverables such as screens and documents. Users may also review or even participate in the development of prototypes for key inputs, outputs, and functions. User participation is a major feature of the new Agile development methodology, where user representatives are embedded in the project team. For any major application, the state of the art of user involvement includes participation in:

1. Joint application design (JAD) sessions
2. Quality function deployment (QFD)
3. Reviewing business rules and algorithms mined from legacy applications
4. Agile projects on a full-time basis
5. Requirements reviews
6. Change control boards
7. Reviewing documents produced by the contractors
8. Design reviews
9. Using prototypes and sample screens produced by the contractors
10. Training classes to learn the new application
11. Defect reporting from design through testing
12. Acceptance testing

User involvement is time-consuming but valuable. On average, user effort totals about 20 percent of the effort put in by the software technical team. The range of user involvement can top 50 percent at the high end and be less than 5 percent at the low end. However, for large and complex projects, if the user involvement totals to less than about 10 percent of the effort expended by the development team, the project will be at some risk of having poor user satisfaction when it is finally finished.

The Agile methodology includes a full-time user representative as part of the project team. This method does work well for small projects and small numbers of users. It becomes difficult or impossible when the number of users is large, such as the millions of users of Microsoft Office or Microsoft Vista. For applications with millions of users, no one user can possibly understand the entire range of possible uses.

For these high-usage applications, surveys of hundreds of users or focus groups where perhaps a dozen users offer opinions are preferred. Also, usability labs where users can try out features and prototypes are helpful.

As can be seen, there is no “one size fits all” method for software applications that can possibly be successful for sizes of 1, 10, 100, 1000, 10,000, and 100,000 function points. Each size plateau and each type of software needs its own optimal methods and practices.

This same situation occurs with medicine. There is no antibiotic or therapeutic agent that is successful against all diseases including bacterial and viral illness. Each condition needs a unique prescription. Also as with medicine, some conditions may be incurable.

13. Best Practices for Executive Management Support of Software Applications

The topic of executive management support of new applications varies with the overall size of the application. For projects below about 500 function points, executive involvement may hover around zero, because these projects are so low in cost and low in risk as to be well below the level of executive interest.

However, for large applications in the 10,000–function point range, executive scrutiny is the norm. It is an interesting phenomenon that the frequent failure of large software projects has caused a great deal of distrust of software managers by corporate executives. In fact, the software organizations of large companies are uniformly regarded as the most troublesome organizations in the company, due to high failure rates, frequent overruns, and mediocre quality levels.

In the software industry overall, the state of the art of executive management support indicates the following roles:

- Approving the return on investment (ROI) calculations for software projects
- Providing funding for software development projects
- Assigning key executives to oversight, governance, and project director roles
- Reviewing milestone, cost, and risk status reports
- Determining if overruns or delays have reduced the ROI below corporate targets

Even if executives perform all of the roles that normally occur, problems and failures can still arise. A key failing of software projects is that executives cannot reach good business decisions if they are provided with disinformation rather than accurate status reports. If software project status reports and risk assessments gloss over problems and technical issues, then executives cannot control the project with the precision that they would like. Thus, inadequate reporting and less-than-candid risk assessments will delay the eventual and prudent executive decision to try and limit further expenses by terminating projects that are out of control.

It is a normal corporate executive responsibility to ascertain why projects are running out of control. One of the reasons why executives at many large corporations distrust software is because software projects have a tendency to run out of control and often fail to provide accurate status reports. As a result, top executives at the levels of senior vice presidents, chief operating officers, and chief executive officers find

software to be extremely frustrating and unprofessional compared with other operating units.

As a class, the corporate executives that the author has met are more distrustful of software organizations than almost any other corporate group under their management control. Unfortunately, corporate executives appear to have many reasons for being distrustful of software managers after so many delays and cost overruns.

All of us in the software industry share a joint responsibility for raising the professional competence of software managers and software engineers to such a level that we receive (and deserve) the trust of corporate client executives.

14. Best Practices for Software Architecture and Design

For small stand-alone applications in the 1000–function point range, both architecture and design are often informal activities. However, as application sizes increase to 10,000 and then 100,000 function points, both architecture and design become increasingly important. They also become increasingly complicated and expensive.

Enterprise architecture is even larger in scope, and it attempts to match total corporate portfolios against total business needs including sales, marketing, finance, manufacturing, R&D, and other operating units. At the largest scale, enterprise architecture may deal with more than 5,000 applications that total more than 10 million function points.

The architecture of a large software application concerns its overall structure and the nature of its connections to other applications and indeed to the outside world. As of 2009, many alternative architectures are in use, and a specific architecture needs to be selected for new applications. Some of these include monolithic applications, service-oriented architecture (SOA), event-driven architecture, peer-to-peer, pipes and filters, client-server, distributed, and many others, including some specialized architectures for defense and government applications.

A colleague from IBM, John Zachman, developed an interesting and useful schema that shows some of the topics that need to be included in the architectural decisions for large software applications. The overall Zachman schema is shown in Table 2-2.

In the Zachman schema, the columns show the essential activities, and the rows show the essential personnel involved with the software. The intersections of the columns and rows detail tasks and decisions for each join of the rows and columns. A quick review of Table 2-2 reveals the rather daunting number of variables that need to be dealt with to develop the architecture for a major software application.

TABLE 2-2 Example of the Zachman Architectural Schema

	What	How	Where	Who	When	Why
Planner						
Owner						
Designer						
Builder						
Contractor						
Enterprise						

The design of software applications is related to architecture, but deals with many additional factors. As of 2009, the selection of design methods is unsettled, and there are more than 40 possibilities. The unified modeling language (UML) and use-cases are currently the hottest of the design methods, but scores of others are also in use. Some of the other possibilities include old-fashioned flowcharts, HIPO diagrams, Warnier-Orr diagrams, Jackson diagrams, Nassi-Schneiderman charts, entity-relationship diagrams, state-transition diagrams, action diagrams, decision tables, data-flow diagrams, object-oriented design, pattern-based design, and many others, including hybrids and combinations.

The large number of software design methods and diagramming techniques is a sign that no single best practice has yet emerged. The fundamental topics of software design include descriptions of the functions and features available to users and how users will access them. At the level of internal design, the documents must describe how those functions and features will be linked and share information internally. Other key elements of software design include security methods and performance issues. In addition, what other applications will provide data to or take data from the application under development must be discussed. Obviously, the design must also deal with hardware platforms and also with software platforms such as the operating systems under which the application will operate.

Because many software applications are quite similar, and have been for more than 50 years, it is possible to record the basic features, functions, and structural elements of common applications into *patterns* that can be reused over and over. Reusable design patterns will become a best practice once a standard method for describing those patterns emerges from the many competing design languages and graphic approaches that are in current use.

It is possible to visualize some of these architectural patterns by examining the structures of existing applications using automated tools. In fact, mining existing software for business rules, algorithms,

and architectural information is a good first step toward creating libraries of reusable components and a workable taxonomy of software features.

Enterprise architecture also lends itself to pattern analysis. Any consultant who visits large numbers of companies in the same industries cannot help but notice that software portfolios are about 80 percent similar for all insurance companies, banks, manufacturing companies, pharmaceuticals, and so forth. In fact, the New Zealand government requires that all banks use the same software, in part to make audits and security control easier for regulators (although perhaps increasing the risk of malware and denial of service attacks).

What the industry needs as of 2009 are effective methods for visualizing and using these architectural patterns. A passive display of information will be insufficient. There is a deeper need to link costs, value, numbers of users, strategic directions, and other kinds of business information to the architectural structures. In addition, it is necessary to illustrate the data that the software applications use, and also the flows of information and data from one operating unit to another and from one system to another; that is, dynamic models rather than static models would be the best representation approach. Given the complexity and kinds of information, what would probably be most effective for visualization of patterns would be dynamic holographic images.

15. Best Practices for Software Project Planning

Project planning for large software projects in large corporations often involves both planning specialists and automated planning tools. The state of the art for planning software projects circa 2009 for large projects in the nominal 10,000–function point range involves

- Development of complete work breakdown structures
- Collecting and analyzing historical benchmark data from similar projects
- Planning aid provided by formal project offices
- Consideration to staff hiring and turnover during the project
- Usage of automated planning tools such as Artemis Views or Microsoft Project
- Factoring in time for requirements gathering and analysis
- Factoring in time for handling changing requirements

- Consideration given to multiple releases if requirements creep is extreme
- Consideration given to transferring software if outsourcing is used
- Consideration given to supply chains if multiple companies are involved
- Factoring in time for a full suite of quality control activities
- Factoring in risk analysis of major issues that are likely to occur

Successful projects do planning very well indeed. Delayed or cancelled projects, however, almost always have planning failures. The most common planning failures include (1) not dealing effectively with changing requirements, (2) not anticipating staff hiring and turnover during the project, (3) not allotting time for detailed requirements analysis, (4) not allotting sufficient time for formal inspections, testing, and defect repairs, and (5) essentially ignoring risks until they actually occur.

Large projects in sophisticated companies will usually have planning support provided by a *project office*. The project office will typically be staffed by between 6 and 10 personnel and will be well equipped with planning tools, estimating tools, benchmark data, tracking tools, and other forms of data analysis tools such as statistical processors.

Because project planning tools and software cost-estimating tools are usually provided by different vendors, although they share data, planning and estimating are different topics. As used by most managers, the term *planning* concerns the network of activities and the critical path required to complete a project. The term *estimating* concerns cost and resource predictions, and also quality predictions. The two terms are related but not identical. The two kinds of tools are similar, but not identical.

Planning and estimating are both more credible if they are supported by benchmark data collected from similar projects. Therefore all major projects should include analysis of benchmarks from public sources such as the International Software Benchmarking Standards Group (ISBSG) as well as internal benchmarks. One of the major problems of the software industry, as noted during litigation, is that accurate plans and estimates are often replaced by impossible plans and estimates based on business needs rather than on team capabilities. Usually these impossible demands come from clients or senior executives, rather than from the project managers. However, without empirical data from similar projects, it is difficult to defend plans and estimates no matter how accurate they are. This is a subtle risk factor that is not always recognized during risk analysis studies.

16. Best Practices for Software Project Cost Estimating

For small applications of 1000 or fewer function points, manual estimates and automated estimates are about equal in terms of accuracy. However, as application sizes grow to 10,000 or more function points, automated estimates continue to be fairly accurate, but manual estimates become dangerously inaccurate by leaving out key activities, failing to deal with changing requirements, and underestimating test and quality control. Above 10,000 function points in size, automated estimating tools are the best practice, while manual estimation is close to professional malpractice.

Estimating software projects in the nominal 10,000–function point range is a critical activity. The current state of the art for estimating large systems involves the use of:

- Formal sizing approaches for major deliverables based on function points
- Secondary sizing approaches for code based on lines of code metrics
- Tertiary sizing approaches using information such as screens, reports, and so on
- Inclusion of reusable materials in the estimates
- Inclusion of supply chains in the estimate if multiple companies are involved
- Inclusion of travel costs if international or distributed teams are involved
- Comparison of estimates to historical benchmark data from similar projects
- Trained estimating specialists
- Software estimating tools (CHECKPOINT, COCOMO, KnowledgePlan, Price-S, SEER, SLIM, SoftCost, etc.)
- Inclusion of new and changing requirements in the estimate
- Quality estimation as well as schedule and cost estimation
- Risk prediction and analysis
- Estimation of all project management tasks
- Estimation of plans, specifications, and tracking costs
- Sufficient historical benchmark data to defend an estimate against arbitrary changes

There is some debate in the software literature about the merits of estimating tools versus manual estimates by experts. However, above

10,000 function points, there are hardly any experts in the United States, and most of them work for the commercial software estimating companies.

The reason for this is that in an entire career, a project manager might deal only with one or two really large systems in the 10,000–function point range. Estimating companies, on the other hand, typically collect data from dozens of large applications.

The most common failing of manual estimates for large applications is that they are excessively optimistic due to lack of experience. While coding effort is usually estimated fairly well, manual estimates tend to understate paperwork effort, test effort, and the impacts of changing requirements. Even if manual estimates were accurate for large applications, which they are not, the cost of updating manual estimates every few weeks to include changing requirements is prohibitively expensive.

A surprising observation from litigation is that sometimes accurate estimates are overruled and rejected precisely because they are accurate! Clients or top managers reject the original and accurate estimate, and replace it with an artificial estimate made up out of thin air. This is because the original estimate showed longer schedules and higher costs than the clients wanted, so they rejected it. When this happens, the project has more than an 80 percent chance of failure, and about a 99 percent chance of severe cost and schedule overruns.

A solution to this problem is to support the estimate by historical benchmarks from similar applications. These can be acquired from the International Software Benchmarking Standards Group (ISBSG) or from other sources. Benchmarks are perceived as being more real than estimates, and therefore supporting estimates with historical benchmarks is a recommended best practice. One problem with this approach is that historical benchmarks above 10,000 function points are rare, and above 100,000 function points almost nonexistent.

Failing projects often understate the size of the work to be accomplished. Failing projects often omit to perform quality estimates at all. Overestimating productivity rates is another common reason for cost and schedule overruns. Underestimating paperwork costs is also a common failing.

Surprisingly, both successful and failing projects are similar when estimating coding schedules and costs. But failing projects are excessively optimistic in estimating testing schedules and costs. Failing projects also tend to omit requirements changes during development, which can increase the size of the project significantly.

Because estimating is complex, trained estimating specialists are the best, although such specialists are few. These specialists always utilize one or more of the leading commercial software estimating tools or sometimes use proprietary estimating tools. About half of our leading

clients utilize two commercial software estimating tools frequently and may own as many as half a dozen. Manual estimates are never adequate for major systems in the 10,000–function point range.

Manual estimates using standard templates are difficult to modify when assumptions change. As a result, they often fall behind the reality of ongoing projects with substantial rates of change. My observations of the overall results of using manual estimates for projects of more than about 1000 function points is that they tend to be incomplete and err on the side of excessive optimism.

For large projects of more than 10,000 function points, manual estimates are optimistic for testing, defect removal schedules, and costs more than 95 percent of the time. Manual estimating is hazardous for large projects.

For many large projects in large companies, estimating specialists employed by the project offices will do the bulk of the cost estimating using a variety of automated estimating tools. Often project offices are equipped with several estimating tools such as COCOMO, KnowledgePlan, Price-S, SEER, SoftCost, SLIM, and so on, and will use them all and look for convergence of results.

As previously discussed, even accurate estimates may be rejected unless they are supported by historical data from similar projects. In fact, even historical data may sometimes be rejected and replaced by impossible demands, although historical data is more credible than unsupported estimates.

For small projects of fewer than 1000 function points, coding remains the dominant activity. For these smaller applications, automated and manual cost estimates are roughly equal in accuracy, although of course the automated estimates are much quicker and easier to change.

17. Best Practices for Software Project Risk Analysis

Make no mistake about it, large software projects in the 10,000–function point range are among the most risky business endeavors in human history. The major risks for large software projects include

- Outright cancellation due to excessive cost and schedule overruns
- Outright termination due to downsizing or bankruptcy due to the poor economy
- Cost overruns in excess of 50 percent compared with initial estimates
- Schedule overruns in excess of 12 months compared with initial estimates
- Quality control so inept that the software does not work effectively
- Requirements changes in excess of 2 percent per calendar month

- Executive or client interference that disrupts the project
- Failure of clients to review requirements and plans effectively
- Security flaws and vulnerabilities
- Performance or speed too slow to be effective
- Loss of key personnel from the project during development
- The presence of error-prone modules in legacy applications
- Patent violations or theft of intellectual property
- External risks (fire, earthquakes, hurricanes, etc.)
- Sale or acquisition of a business unit with similar software

From analysis of depositions and court documents in breach of contract litigation, most failing projects did not even perform a formal risk analysis. In addition, quality control and change management were inadequate. Worse, project tracking was so inept that major problems were concealed rather than being dealt with as they occurred. Another ugly risk is that sometimes fairly accurate estimates were rejected and replaced by impossible schedule and cost targets based on business needs rather than team capabilities.

The state of the art of software risk management is improving. Traditionally, formal risk analysis by trained risk experts provided the best defense. However, risk estimation tools and software risk models were increasing in numbers and sophistication circa 2008. The new Application Insight tool from Computer Aid Inc. and the Software Risk Master prototype of the author are examples of predictive tools that can quantify the probabilities and effects of various forms of risk.

As of 2009, the best practices for software risk management include

- Early risk assessment even prior to development of full requirements
- Early prediction of defect potentials and removal efficiency levels
- Comparison of project risk patterns to similar projects
- Acquisition of benchmarks from the ISBSG database
- Early review of contracts and inclusion of quality criteria
- Early analysis of change control methods
- Early analysis of the value of the application due to the poor economy

The importance of formal risk management rises with application size. Below 1000 function points, risk management is usually optional. Above 10,000 function points, risk assessments are mandatory. Above 100,000 function points, failure to perform careful risk assessments is evidence of professional malpractice.

From repeated observations during litigation for breach of contract, effective risk assessment is almost never practiced on applications that later end up in court. Instead false optimism and unrealistic schedules and cost estimates get the project started in a bad direction from the first day.

Unfortunately, most serious risks involve a great many variable factors. As a result, combinatorial complexity increases the difficulty of thorough risk analysis. The unaided human mind has trouble dealing with problems that have more than two variables. Even automated risk models may stumble if the number of variables is too great, such as more than ten. As seen by the failure of economic risk models to predict the financial crisis of 2008, risk analysis is not a perfect field and may miss serious risks. There are also false positives, or risk factors that do not actually exist, although these are fairly rare.

18. Best Practices for Software Project Value Analysis

Software value analysis is not very sophisticated as this book is written in 2009. The value of software applications prior to development may not even be quantified, and if it is quantified, then the value may be suspect.

Software applications have both financial and intangible value aspects. The financial value can be subdivided into cost reductions and revenue increases. The intangible value is more difficult to characterize, but deals with topics such as customer satisfaction, employee morale, and the more important topics of improving human life and safety or improving national defense.

Some of the topics that need to be included in value analysis studies include

Tangible Financial Value

- Cost reductions from new application
- Direct revenue from new application
- Indirect revenue from new application due to factors such as hardware sales
- “Drag along” or revenue increases in companion applications
- Domestic market share increases from new application
- International market share increases from new application
- Competitive market share decreases from new application
- Increases in numbers of users due to new features
- User performance increases
- User error reductions

Intangible Value

- Potential harm if competitors instead of you build application
- Potential harm if competitors build similar application
- Potential gain if your application is first to market
- Synergy with existing applications already released
- Benefits to national security
- Benefits to human health or safety
- Benefits to corporate prestige
- Benefits to employee morale
- Benefits to customer satisfaction

What the author has proposed is the possibility of constructing a *value point* metric that would resemble function point metrics in structure. The idea is to have a metric that can integrate both financial and intangible value topics and therefore be used for return-on-investment calculations.

In general, the financial value points would be equal to \$1000. The intangible value points would have to be mapped to a scale that provided approximate equivalence, such as each customer added or lost would be worth 10 value points. Obviously, value associated with saving human lives or national defense would require a logarithmic scale since those values may be priceless.

Value points could be compared with cost per function point for economic studies such as return on investment and total cost of ownership (TCO).

19. Best Practices for Canceling or Turning Around Troubled Projects

Given the fact that a majority of large software projects run late or are cancelled without ever being completed, it is surprising that the literature on this topic is very sparse. A few interesting technical papers exist, but no full-scale books. Of course there are many books on software disasters and outright failures, but they are hardly best practice discussions of trying to rescue troubled projects.

Unfortunately, only a small percentage of troubled projects can be rescued and turned into successful projects. The reasons for this are twofold: First, troubled projects usually have such bad tracking of progress that it is too late to rescue the project by the time the problems surface to higher management or to clients. Second, troubled projects with

schedule delays and cost overruns steadily lose value. Although such projects may have had a positive value when first initiated, by the time of the second or third cost overrun, the value has probably degraded so much that it is no longer cost-effective to complete the application. An example will clarify the situation.

The example shows an original estimate and then three follow-on estimates produced when senior management was alerted to the fact that the previous estimate was no longer valid. The application in question is an order entry system for a large manufacturing company. The initial planned size was 10,000 function points.

The original cost estimate was for \$20 million, and the original value estimate was for \$50 million. However, the value was partly based upon the application going into production in 36 months. Every month of delay would lower the value.

Estimate 1: January 2009

Original size (function points)	10,000
Original budget (dollars)	\$20,000,000
Original schedule (months)	36
Original value (dollars)	\$50,000,000
Original ROI	\$2.50

Estimate 2: June 2010

Predicted size (function points)	12,000
Predicted costs (dollars)	\$25,000,000
Predicted schedule (months)	42
Predicted value (dollars)	\$45,000,000
Predicted ROI	\$1.80
Recovery possible	

Estimate 3: June 2011

Predicted size (function points)	15,000
Predicted costs (dollars)	\$30,000,000
Predicted schedule (months)	48
Predicted value (dollars)	\$40,000,000
Predicted ROI	\$1.33
Recovery unlikely	

Estimate 4: June 2012

Predicted size (function points)	17,000
Predicted costs (dollars)	\$35,000,000
Predicted schedule (months)	54
Predicted value (dollars)	\$35,000,000
Predicted ROI	\$1.00
Recovery impossible	

As can be seen, the steady increase in creeping requirements triggered a steady increase in development costs and a steady increase in development schedules. Since the original value was based in part on completion in 36 months, the value eroded so that the project was no longer viable. By the fourth estimate, recovery was unfeasible and termination was the only choice.

The truly best practice, of course, would be to avoid the situation by means of a careful risk analysis and sizing study before the application started. Once the project is under way, best practices for turnarounds include

- Careful and accurate status tracking
- Re-estimation of schedules and costs due to requirements changes
- Re-estimation of value at frequent intervals
- Considering intangible value as well as internal rate of return and financial value
- Using internal turnaround specialists (if available)
- Hiring external turnaround consultants
- Threatening litigation if the application is under contract

If the application has negative value and trying to turn it around is unfeasible, then best practices for cancellation would include

- Mining the application for useful algorithms and business rules
- Extracting potentially useful reusable code segments
- Holding a formal “postmortem” to document what went wrong
- Assembling data for litigation if the application was under contract

Unfortunately, cancelled projects are common, but usually don’t generate much in the way of useful data to avoid similar problems in the future. Postmortems should definitely be viewed as best practices for cancelled projects.

One difficulty in studying cancelled projects is that no one wants to spend the money to measure application size in function points. However, the advent of new high-speed, low-cost function point methods means that the cost of counting function points is declining from perhaps \$6.00 per function point counted down to perhaps \$0.01 per function point counted. At a cost of a penny per function point, even a 100,000–function point disaster can now be quantified. Knowing the sizes of cancelled projects will provide new insights into software economics and aid in forensic analysis.

20. Best Practices for Software Project Organization Structures

Software project organization structures and software specialization are topics that have more opinions than facts associated with them. Many adherents of the “small team” philosophy claim that software applications developed by teams of six or fewer are superior in terms of quality and productivity. However, such small teams cannot develop really large applications.

As software projects grow in size, the number and kinds of specialists that are normally employed goes up rapidly. With increases in personnel, organization structures become more complex, and communication channels increase geometrically. These larger groups eventually become so numerous and diverse that some form of project office is required to keep track of progress, problems, costs, and issues.

A study performed by the author and his colleagues of software occupation groups in large corporations and government agencies identified more than 75 different specialties. Because software engineering is not a licensed profession with formal specialties, these specialists are seldom clearly identified in personnel records. Therefore on-site visits and discussions with local managers are needed to ascertain the occupations that are really used.

The situation is made more complex because some companies do not identify specialists by job title or form of work, but use a generic title such as “member of the technical staff” to encompass scores of different occupations.

Also adding to the difficulty of exploring software specialization is the fact that some personnel who develop embedded software are not software engineers, but rather electrical engineers, automotive engineers, telecommunications engineers, or some other type of engineer. In many cases, these engineers refuse to be called “software engineers” because software engineering is lower in professional status and not a recognized professional engineering occupation.

Consider the differences in the number and kind of personnel who are likely to be used for applications of 1000 function points, 10,000 function points, and 100,000 function points. For small projects of 1000 function points, generalists are the norm and specialists are few. But as applications reach 10,000 and 100,000 function points, specialists become more important and more numerous. Table 2-3 illustrates typical staffing patterns for applications of three sizes an order of magnitude apart.

As can easily be seen from Table 2-3, the diversity of occupations rises rapidly as application size increases. For small applications, generalists predominate, but for large systems, various kinds of specialists can top one third of the total team size.

TABLE 2-3 Personnel Staffing Patterns for Software Projects

Occupation Group	1000 Function Points	10,000 Function Points	100,000 Function Points
Architect		1	5
Configuration control		2	8
Database administration		2	10
Estimating specialist		1	3
Function point counters		2	5
Measurement specialist		1	5
Planning specialist		1	3
Project librarian		2	6
Project manager	1	6	75
Quality assurance		2	12
Scrum master		3	8
Security specialist		1	5
Software engineers	5	50	600
Technical writer	1	3	12
Testers		5	125
Web designer		1	5
TOTAL STAFF	7	83	887
Function points per staff member	142.86	120.48	112.74

Table 2-3 also illustrates why some methods such as Agile development do very well for small projects, but may not be a perfect match for large projects. As project sizes grow larger, it is hard to accommodate all of the various specialists into the flexible and cohesive team organizations that are the hallmark of the Agile approach.

For example, large software projects benefit from specialized organization such as project offices, formal software quality assurance (SQA) organizations, formal testing groups, measurement groups, change management boards, and others as well. Specialized occupations that benefit large projects include architecture, security, database administration, configuration control, testing, and function point analysis.

Melding these diverse occupations into a cohesive and cooperating team for large software projects is not easy. Multiple departments and multiple specialists bring about a geometric increase in communication channels. As a result, a best practice for large software projects above 10,000 function points is a project office whose main role is

coordination of the various skills and activities that are a necessary part of large-system development. The simplistic Agile approach of small self-organizing teams is not effective above about 2,500 function points.

Another issue that needs examination is the *span of control*, or the number of employees reporting to a manager. For reasons of corporate policy, the average number of software employees who report to a manager in the United States is about eight. However, the observed range of employees per manager runs from 3 to more than 20.

Studies carried out by the author within IBM noted that having eight employees per manager tended to put more people in management than were really qualified to do managerial work well. As a result, planning, estimating, and other managerial functions were sometimes poorly performed. My study concluded that changing the average span of control from 8 to 11 would allow marginal managers to be reassigned to staff or technical positions. Cutting down on the number of departments would also reduce communication channels and allow managers to have more time with their own teams, rather than spending far too much time with other managers.

Even worse, personality clashes between managers and technical workers sometimes led to the voluntary attrition of good technologists. In fact, when exit interviews are examined, two distressing problems tend to occur: (1) The best people leave in the largest numbers; and (2) The most common reason cited for departure was “I don’t want to work under bad management.”

Later in this book the pros and cons of small teams, large departments, and various spans of control will be discussed at more length, as will special topics such as pair programming.

21. Best Practices for Training Managers of Software Projects

When major delays or cost overruns for projects occur in the nominal 10,000–function point range, project management problems are always present. Conversely, when projects have high productivity and quality levels, good project management is always observed. The state of the art for project management on large projects includes knowledge of:

1. Sizing techniques such as function points
2. Formal estimating tools and techniques
3. Project planning tools and techniques
4. Benchmark techniques and sources of industry benchmarks
5. Risk analysis methods

6. Security issues and security vulnerabilities
7. Value analysis methods
8. Project measurement techniques
9. Milestone and cost tracking techniques
10. Change management techniques
11. All forms of software quality control
12. Personnel management techniques
13. The domain of the applications being developed

For the global software industry, it appears that project management was a weak link and possibly the weakest link of all. For example, for failing projects, sizing by means of function points is seldom utilized. Formal estimating tools are not utilized. Although project planning tools may be used, projects often run late and over budget anyway. This indicates that the plans were deficient and omitted key assumptions such as the normal rate of requirements change, staff turnover, and delays due to high defect volumes found during testing.

The roles of management in outsource projects are more complex than the roles of management for projects developed internally. It is important to understand the differences between *client management* and *vendor project management*.

The active work of managing the project is that of the vendor project managers. It is their job to create plans and schedules, to create cost estimates, to track costs, to produce milestone reports, and to alert the client directors and senior client executives to the existence of potential problems.

The responsibility of the client director or senior client executive centers around facilitation, funding, and approval or rejection of plans and estimates produced by the vendor's project manager.

Facilitation means that the client director will provide access for the vendor to business and technical personnel for answering questions and gathering requirements. The client director may also provide to the vendor technical documents, office space, and sometimes tools and computer time.

Funding means that the client director, after approval by corporate executives, will provide the money to pay for the project.

Approval means that the client director will consider proposals, plans, and estimates created by the vendor and either accept them, reject them, or request that they be modified and resubmitted.

The main problems with failing projects seem to center around the approval role. Unfortunately clients may be presented with a stream of optimistic estimates and schedule commitments by vendor project

management and asked to approve them. This tends to lead to cumulative overruns, and the reason for this deserves comment.

Once a project is under way, the money already spent on it will have no value unless the project is completed. Thus if a project is supposed to cost \$1 million, but has a cost overrun that needs an additional \$100,000 for completion, the client is faced with a dilemma. Either cancel the project and risk losing the accrued cost of a million dollars, or provide an additional 10 percent and bring the project to completion so that it returns positive value and results in a working application.

If this scenario is repeated several times, the choices become more difficult. If a project has accrued \$5 million in costs and seems to need another 10 percent, both sides of the dilemma are more expensive. This is a key problem with projects that fail. Each time a revised estimate is presented, the vendor asserts that the project is nearing completion and needs only a small amount of time and some additional funds to bring it to full completion. This can happen repeatedly.

All corporations have funding criteria for major investments. Projects are supposed to return positive value in order to be funded. The value can consist of either revenue increases, cost reductions, or competitive advantage. A typical return on investment (ROI) for a software project in the United States would be about 3 to 1. That is, the project should return \$3.00 in positive value for every \$1.00 that is spent on the project.

During the course of development the accrued costs are monitored. If the costs begin to exceed planned budgets, then the ROI for the project will be diminished. Unfortunately for failing projects, the ability of client executives to predict the ROI can be damaged by inaccurate vendor estimating methods and cost control methods.

The root problem, of course, is that poor estimating methods are never realistic nor are the schedules: they are always optimistic. Unfortunately, it can take several iterations before the reality of this pattern emerges.

Each time a vendor presents revised estimates and schedules, there may be no disclosure to clients of internal problems and risks that the vendor is aware of. Sometimes this kind of problem does not surface until litigation occurs, when all of the vendor records have to be disclosed and vendor personnel are deposed.

The bottom line is that training of software managers needs to be improved in the key tasks of planning, estimating, status reporting, cost tracking, and problem reporting.

22. Best Practices for Training Software Technical Personnel

The software development and maintenance domains are characterized by workers who usually have a fairly strong work ethic and reasonable

competence in core activities such as detail design, programming, and unit testing. Many software personnel put in long hours and are fairly good in basic programming tasks. However, to be successful on specific, 10,000–function point applications, some additional skill sets are needed—knowledge of the following:

1. Application domains
2. The database packages, forms, tools, and products
3. The skill sets of the subcontract companies
4. Joint application design (JAD) principles
5. Formal design inspections
6. Complexity analysis
7. All programming languages utilized
8. Security issues and security vulnerabilities (a weak area)
9. Performance issues and bottlenecks (a weak area)
10. Formal code inspections
11. Static analysis methods
12. Complexity analysis methods
13. Change control methods and tools
14. Performance measurement and optimization techniques
15. Testing methods and tools

When software technical problems occur, they are more often related to the lack of specialized knowledge about the application domain or about specific technical topics such as performance optimization rather than to lack of knowledge of basic software development approaches.

There may also be lack of knowledge of key quality control activities such as inspections, JAD, and specialized testing approaches. In general, common programming tasks are not usually problems. The problems occur in areas where special knowledge may be needed, which brings up the next topic.

23. Best Practices for Use of Software Specialists

In many human activities, specialization is a sign of technological maturity. For example, the practice of medicine, law, and engineering all encompass dozens of specialists. Software is not yet as sophisticated as the more mature professions, but specialization is now occurring in increasing numbers. After analyzing the demographics of large software

production companies in a study commissioned by AT&T, from 20 to more than 90 specialized occupations now exist in the software industry.

What is significant about specialization in the context of 10,000–function point projects is that projects with a full complement of a dozen or more specialists have a better chance of success than those relying upon generalists.

The state of the art of specialization for nominal 10,000–function point projects would include the following specialist occupation groups:

1. Configuration control specialists
2. Cost estimating specialists
3. Customer liaison specialists
4. Customer support specialists
5. Database administration specialists
6. Data quality specialists
7. Decision support specialists
8. Development specialists
9. Domain knowledge specialists
10. Security specialists
11. Performance specialists
12. Education specialists
13. Function point specialists (certified)
14. Graphical user interface (GUI) specialists
15. Human factors specialists
16. Integration specialists
17. Joint application design (JAD) specialists
18. SCRUM masters (for Agile projects)
19. Measurement specialists
20. Maintenance specialists for postrelease defect repairs
21. Maintenance specialists for small enhancements
22. Outsource evaluation specialists
23. Package evaluation specialists
24. Performance specialists
25. Project cost estimating specialists
26. Project planning specialists
27. Quality assurance specialists

28. Quality measurement specialists
29. Reusability specialists
30. Risk management specialists
31. Standards specialists
32. Systems analysis specialists
33. Systems support specialists
34. Technical writing specialists
35. Testing specialists
36. Tool specialists for development and maintenance workbenches

Senior project managers need to know what specialists are required and should take active and energetic steps to find them. The domains where specialists usually outperform generalists include technical writing, testing, quality assurance, database design, maintenance, and performance optimization. For some tasks such as function point analysis, certification examinations are a prerequisite to doing the work at all. Really large projects also benefit from using planning and estimating specialists.

Both software development and software project management are now too large and complex for generalists to know everything needed in sufficient depth. The increasing use of specialists is a sign that the body of knowledge of software engineering and software management is expanding, which is a beneficial situation.

For the past 30 years, U.S. and European companies have been outsourcing software development, maintenance, and help-desk activities to countries with lower labor costs such as India, China, Russia, and a number of others. In general it is important that outsource vendors utilize the same kinds of methods as in-house development, and in particular that they achieve excellence in quality control.

Interestingly a recent study of outsource practices by the International Software Benchmarking Standards Group (ISBSG) found that outsource projects tend to use more tools and somewhat more sophisticated planning and estimating methods than similar projects produced in-house. This is congruent with the author's own observations.

24. Best Practices for Certifying Software Engineers, Specialists, and Managers

As this book is written in 2008 and 2009, software engineering itself and its many associated specialties are not fully defined. Of the 90 or so occupations noted in the overall software domain, certification is

possible for only about a dozen topics. For these topics, certification is voluntary and has no legal standing.

What would benefit the industry would be to establish a joint certification board that would include representatives from the major professional associations such as the ASQ, IEEE (Institute of Electrical & Electronics Engineers), IIBA (International Institute of Business Analysis), IFPUG (International Function Point Users Group), PMI (Project Management Institute), SEI (Software Engineering Institute), and several others. The joint certification board would identify the specialist categories and create certification criteria. Among these criteria might be examinations or certification boards, similar to those used for medical specialties.

As this book is written, voluntary certification is possible for these topics:

- Function point analysis (IFPUG)
- Function point analysis (COSMIC)
- Function point analysis (Finnish)
- Function point analysis (Netherlands)
- Microsoft certification (various topics)
- Six Sigma green belt
- Six Sigma black belt
- Certified software project manager (CSPM)
- Certified software quality engineer (CSQE)
- Certified software test manager (CSTM)
- Certified software test professional (CSTP)
- Certified software tester (CSTE)
- Certified scope manager (CSM)

These forms of certification are offered by different organizations that in general do not recognize certification other than their own. There is no central registry for all forms of certification, nor are their standard examinations.

As a result of the lack of demographic data about those who are registered, there is no solid information as to what percentage of various specialists and managers are actually certified. For technical skills such as function point analysis, probably 80 percent of consultants and employees who count function points are certified. The same or even higher is true for Six Sigma. However, for testing, for project management, and for quality assurance, it would be surprising if the percentage of those certified were higher than about 20 percent to 25 percent.

It is interesting that there is overall certification neither for “software engineering” nor for “software maintenance engineering” as a profession.

Some of the technical topics that might be certified if the industry moves to a central certification board would include

- Software architecture
- Software development engineering
- Software maintenance engineering
- Software quality assurance
- Software security assurance
- Software performance analysis
- Software testing
- Software project management
- Software scope management

Specialized skills would also need certification, including but not limited to:

- Six Sigma for software
- Quality function deployment (QFD)
- Function point analysis (various forms)
- Software quality measurements
- Software productivity measurements
- Software economic analysis
- Software inspections
- SEI assessments
- Vendor certifications (Microsoft, Oracle, SAP, etc.)

The bottom line as this book is written is that software certification is voluntary, fragmented, and of unknown value to either practitioners or to the industry. Observations indicate that for technical skills such as function point analysis, certified counters are superior in accuracy to self-taught practitioners. However, more study is needed on the benefits of software quality and test certification.

What is really needed though is coordination of certification and the establishment of a joint certification board that would consider all forms of software specialization. The software engineering field would do well to consider how specialties are created and governed in medicine and law, and to adopt similar policies and practices.

For many forms of certification, no quantitative data is available that indicates that certification improves job performance. However, for some forms of certification, enough data is available to show tangible improvements:

1. Testing performed by certified testers is about 5 percent higher in defect removal efficiency than testing performed by uncertified testers.
2. Function point analysis performed by certified function point counters seldom varies by more than 5 percent when counting trial applications. Counts of the same trials by uncertified counters vary by more than 50 percent.
3. Applications developed where certified Six Sigma black belts are part of the development team tend to have lower defect potentials by about 1 defect per function point, and higher defect removal efficiency levels by about 7 percent. (Compared against U.S. averages of 5.0 defects per function point and 85 percent defect removal efficiency.)

Unfortunately, as this book is written, other forms of certification are ambiguous in terms of quantitative results. Obviously, those who care enough about their work to study and successfully pass written examinations tend to be somewhat better than those who don't, but this is difficult to show using quantified data due to the very sparse sets of data available.

What would benefit the industry would be for software to follow the pattern of the American Medical Association and have a single organization responsible for identifying and certifying specialists, rather than independent and sometimes competing organizations.

25. Best Practices for Communication During Software Projects

Large software applications in the nominal 10,000–function point domain are always developed by teams that number from at least 50 to more than 100 personnel. In addition, large applications are always built for dozens or even hundreds of users, many of whom will be using the application in specialized ways.

It is not possible to build any large and complex product where dozens of personnel and dozens of users need to share information unless communication channels are very well planned and utilized. Communication needs are even greater when projects involve multiple subcontractors.

As of 2009, new kinds of virtual environments where participants interact using avatars in a virtual-reality world are starting to enter

the business domain. Although such uses are experimental in 2009, they are rapidly approaching the mainstream. As air travel costs soar and the economy sours, methods such as virtual communication are likely to expand rapidly. Within ten years, such methods might well outnumber live meetings and live conferences.

Also increasing in use for interproject communication are “wiki sites,” which are collaborative networks that allow colleagues to share ideas, documents, and work products.

The state of the art for communication on a nominal 10,000–function point project includes all of the following:

- Monthly status reports to corporate executives from project management
- Weekly progress reports to clients by vendor project management
- Daily communication between clients and the prime contractor
- Daily communication between the prime contractor and all subcontractors
- Daily communication between developers and development management
- Use of virtual reality for communication across geographic boundaries
- Use of “wiki” sites for communication across geographic boundaries
- Daily “scrum” sessions among the development team to discuss issues
- Full e-mail support among all participants
- Full voice support among all participants
- Video conference communication among remote locations
- Automated distribution of documents and source code among developers
- Automated distribution of change requests to developers
- Automated distribution of defect reports to developers
- Emergency or “red flag” communication for problems with a material impact

For failing projects, many of these communication channels were either not fully available or had gaps that tended to interfere with both communication and progress. For example, cross-vendor communications may be inadequate to highlight problems. In addition, the status reports to executives may gloss over problems and conceal them, rather than highlight causes for projected cost and schedule delays.

The fundamental purpose of good communications was encapsulated in a single phrase by Harold Geneen, the former chairman of ITT Corporation: “NO SURPRISES.”

From reviewing the depositions and court documents of breach of contract litigation, it is alarming that so many projects drift along with inadequate status tracking and problem reporting. Usually projects that are cancelled or that have massive overruns do not even start to deal with the issues until it is far too late to correct them. By contrast, successful projects have fewer serious issues to deal with, more effective tracking, and much more effective risk abatement programs. When problems are first reported, successful projects immediately start task forces or risk-recovery activities.

26. Best Practices for Software Reusability

At least 15 different software artifacts lend themselves to reusability. Unfortunately, much of the literature on software reuse has concentrated only on reusing source code, with a few sparse and intermittent articles devoted to other topics such as reusable design.

The state of the art of developing nominal 10,000–function point projects includes substantial volumes of reusable materials. Following are the 15 artifacts for software projects that are potentially reusable:

1. Architecture
2. Requirements
3. Source code (zero defects)
4. Designs
5. Help information
6. Data
7. Training materials
8. Cost estimates
9. Screens
10. Project plans
11. Test plans
12. Test cases
13. Test scripts
14. User documents
15. Human interfaces

Not only are there many reusable artifacts, but also for reuse to be both a technical and business success, quite a lot of information needs to be recorded:

- All customers or users in case of a recall
- All bugs or defects in reusable artifacts
- All releases of reusable artifacts
- Results of certification of reusable materials
- All updates or changes

Also, buggy materials cannot safely be reused. Therefore extensive quality control measures are needed for successful reuse, including but not limited to:

- Inspections of reusable text documents
- Inspections of reusable code segments
- Static analysis of reusable code segments
- Testing of reusable code segments
- Publication of certification certificates for reusable materials

Successful software reuse involves much more than simply copying a code segment and plugging it into a new application.

One of the common advantages of using an outsource vendor is that these vendors are often very sophisticated in reuse and have many reusable artifacts available. However, reuse is most often encountered in areas where the outsource vendor is a recognized expert. For example, an outsource vendor that specializes in insurance applications and has worked with a dozen property and casualty insurance companies probably has accumulated enough reusable materials to build any insurance application with at least 50 percent reusable components.

Software reuse is a key factor in reducing costs and schedules and in improving quality. However, reuse is a two-edged sword. If the quality levels of the reusable materials are good, then reusability has one of the highest returns on investment of any known software technology. But if the reused materials are filled with bugs or errors, the ROI can become very negative. In fact, reuse of high quality or poor quality materials tends to produce the greatest extreme in the range of ROI of any known technology: plus or minus 300 percent ROIs have been observed.

Software reusability is often cited as a panacea that will remedy software's sluggish schedules and high costs. This may be true theoretically, but reuse will have no practical value unless the quality levels of the reusable materials approach zero defects.

A newer form of reuse termed *service-oriented architecture* (SOA) has appeared within the past few years. The SOA approach deals with reuse by attempting to link fairly large independent functions or “services” into a cohesive application. The functions themselves can also operate in a stand-alone mode and do not require modification. SOA is an intriguing concept that shows great promise, but as of 2009, the concepts are more theoretical than real. In any event, empirical data on SOA costs, quality, and effectiveness have not yet become available.

Software reusability to date has not yet truly fulfilled the high expectations and claims made for it. Neither object-oriented class libraries nor other forms of reuse such as commercial enterprise resource planning (ERP) packages have been totally successful.

To advance reuse to the status of really favorable economics, better quality for reusable materials and better security control for reusable materials need to be more widely achieved. The technologies for accomplishing this appear to be ready, so perhaps within a few years, reuse will finally achieve its past promise of success.

To put software on a sound economic basis, the paradigm for software needs to switch from software development using custom code to software construction using standard reusable components. In 2009, very few applications are constructed from standard reusable components. Part of the reason is that software quality control is not good enough for many components to be used safely. Another part of the reason is the lack of standard architectures for common application types and the lack of standard interfaces for connecting components. The average volume of high-quality reusable material in typical applications today is less than 25 percent. What is needed is a step-by-step plan that will raise the volume of high-quality reusable material to more than 85 percent on average and to more than 95 percent for common applications types.

27. Best Practices for Certification of Reusable Materials

Reuse of code, specifications, and other material is also a two-edged sword. If the materials approach zero-defect levels and are well developed, then they offer the best ROI of any known technology. But if the reused pieces are buggy and poorly developed, they only propagate bugs through dozens or hundreds of applications. In this case, software reuse has the worst negative ROI of any known technology.

Since reusable material is available from hundreds of sources of unknown reliability, it is not yet safe to make software reuse a mainstream development practice. Further, reusable material, or at least source code, may have security flaws or even deliberate “back doors”

inserted by hackers, who then offer the materials as a temptation to the unwary.

This brings up an important point: what must happen for software reuse to become safe, cost-effective, and valuable to the industry?

The first need is a central certification facility or multiple certification facilities that can demonstrate convincingly that candidates for software reuse are substantially bug free and also free from viruses, spyware, and keystroke loggers. Probably an industry-funded nonprofit organization would be a good choice for handling certification. An organization similar to Underwriters Laboratories or Consumer Reports comes to mind.

But more than just certification of source code is needed. Among the other topics that are precursors to successful reuse would be

- A formal taxonomy of reusable objects and their purposes
- Standard interface definitions for linking reusable objects
- User information and HELP text for all reusable objects
- Test cases and test scripts associated with all reusable objects
- A repository of all bug reports against reusable objects
- Identification of the sources of reusable objects
- Records of all changes made to reusable objects
- Records of all variations of reusable objects
- Records of all distributions of reusable objects in case of recalls
- A charging method for reusable material that is not distributed for free
- Warranties for reusable material against copyright and patent violations

In other words, if reuse is going to become a major factor for software, it needs to be elevated from informal and uncontrolled status to formal and controlled status. Until this can occur, reuse will be of some value, but hazardous in the long run. It would benefit the industry to have some form of nonprofit organization serve as a central repository and source of reusable material.

Table 2-4 shows the approximate development economic value of high-quality reusable materials that have been certified and that approach zero-defect levels. The table assumes reuse not only of code, but also of architecture, requirements, design, test materials, and documentation. The example in Table 2-4 is a fairly large system of 10,000 function points. This is the size where normally failures top 50 percent, productivity sags, and quality is embarrassingly bad. As can be seen, as

TABLE 2-4 Development Value of High-Quality Reusable Materials

Application size (function points) = 10,000							
Reuse Percent	Staff	Effort (months)	Prod. (FP/Mon.)	Schedule (months)	Defect Potential	Removal Efficiency	Delivered Defects
0.00%	67	2,654	3.77	39.81	63,096	80.00%	12,619
10.00%	60	2,290	4.37	38.17	55,602	83.00%	9,452
20.00%	53	1,942	5.15	36.41	48,273	87.00%	6,276
30.00%	47	1,611	6.21	34.52	41,126	90.00%	4,113
40.00%	40	1,298	7.70	32.45	34,181	93.00%	2,393
50.00%	33	1,006	9.94	30.17	27,464	95.00%	1,373
60.00%	27	736	13.59	27.59	21,012	97.00%	630
70.00%	20	492	20.33	24.60	14,878	98.00%	298
80.00%	13	279	35.86	20.91	9,146	98.50%	137
90.00%	7	106	94.64	15.85	3,981	99.00%	40
100.00%	4	48	208.33	12.00	577	99.50%	3

the percentage of reuse increases, both productivity and quality levels improve rapidly, as do development schedules.

No other known development technology can achieve such a profound change in software economics as can high-quality reusable materials. This is the goal of object-oriented development and service-oriented architecture. So long as software applications are custom-coded and unique, improvement in productivity and quality will be limited to gains of perhaps 25 percent to 30 percent. For really major gains of several hundred percent, high-quality reuse appears to be the only viable option.

Not only would high-quality reusable material benefit development, but maintenance and enhancement work would also improve. However, there is a caveat with maintenance. Once a reusable component is installed in hundreds or thousands of applications, it is mandatory to be able to recall it, update it, or fix any latent bugs that are reported. Therefore both certification and sophisticated usage records are needed to achieve maximum economic value. In this book *maintenance* refers to defect repairs. Adding new features is called *enhancement*.

Table 2-5 illustrates the maintenance value of reusable materials.

Both development staffing and maintenance staffing have strong correlations to delivered defects, and therefore would be reduced as the volume of certified reusable materials goes up.

Customer support is also affected by delivered defects, but other factors also impact support ratios. Over and above delivered defects, customer support is affected by numbers of users and by numbers of installations of the application.

TABLE 2-5 Maintenance Value of High-Quality Reusable Materials

Application size (function points) = 10,000							
Reuse Percent	Staff	Effort (months)	Prod. (FP/Mon.)	Schedule (months)	Defect Potential	Removal Efficiency	Latent Defects
0.00%	13	160	62.50	12.00	12,619	80.00%	2,524
10.00%	12	144	69.44	12.00	9,452	83.00%	1,607
20.00%	11	128	78.13	12.00	6,276	87.00%	816
30.00%	9	112	89.29	12.00	4,113	90.00%	411
40.00%	8	96	104.17	12.00	2,393	93.00%	167
50.00%	7	80	125.00	12.00	1,373	95.00%	69
60.00%	5	64	156.25	12.00	630	97.00%	19
70.00%	4	48	208.33	12.00	298	98.00%	6
80.00%	3	32	312.50	12.00	137	98.50%	2
90.00%	1	16	625.00	12.00	40	99.00%	0
100.00%	1	12	833.33	12.00	3	99.50%	0

In general, one customer support person is assigned for about every 1000 customers. (This is not an optimum ratio and explains why it is so difficult to reach customer support without long holds on telephones.) A ratio of one support person for about every 150 customers would reduce wait time, but of course raise costs. Because customer support is usually outsourced to countries with low labor costs, the monthly cost is assumed to be only \$4,000 instead of \$10,000.

Small companies with few customers tend to be better in customer support than large companies with thousands of customers, because the support staffs are not saturated for small companies.

Table 2-6 shows approximate values for customer support as reuse goes up. Table 2-6 assumes 500 install sites and 25,000 users.

Because most customer support calls deal with quality issues, improving quality would actually have very significant impact on support costs, and would probably improve customer satisfaction and reduce wait times as well.

Enhancements would also benefit from certified reusable materials. In general, enhancements average about 8 percent per year; that is, if an application is 10,000 function points at delivery, then about 800 function points would be added the next year. This is not a constant value, and enhancements vary, but 8 percent is a useful approximation. Table 2-7 shows the effects on enhancements for various percentages of reusable material.

TABLE 2-6 Customer Support Value of High-Quality Reusable Materials

Application size (function points) = 10,000

Installations = 500

Application users = 25,000

Reuse Percent	Staff	Effort (months)	Prod. (FP/Mon.)	Schedule (months)	Defect Potential	Removal Efficiency	Latent Defects
0.00%	25	300	33.33	12.00	12,619	80.00%	2,524
10.00%	23	270	37.04	12.00	9,452	83.00%	1,607
20.00%	20	243	41.15	12.00	6,276	87.00%	816
30.00%	18	219	45.72	12.00	4,113	90.00%	411
40.00%	16	197	50.81	12.00	2,393	93.00%	167
50.00%	15	177	56.45	12.00	1,373	95.00%	69
60.00%	13	159	62.72	12.00	630	97.00%	19
70.00%	12	143	69.69	12.00	298	98.00%	6
80.00%	11	129	77.44	12.00	137	98.50%	2
90.00%	10	116	86.04	12.00	40	99.00%	0
100.00%	9	105	95.60	12.00	3	99.50%	0

TABLE 2-7 Enhancement Value of High-Quality Reusable Materials

Application size (function points) = 10,000

Enhancements (function points) = 800

Years of usage = 10

Installations = 1,000

Application users = 50,000

Reuse Percent	Staff	Effort (months)	Prod. (FP/Mon.)	Schedule (months)	Defect Potential	Removal Efficiency	Latent Defects
0.00%	6	77	130.21	12.00	3,046	80.00%	609
10.00%	5	58	173.61	12.00	2,741	83.00%	466
20.00%	4	51	195.31	12.00	2,437	87.00%	317
30.00%	4	45	223.21	12.00	2,132	90.00%	213
40.00%	3	38	260.42	12.00	1,828	93.00%	128
50.00%	3	32	312.50	12.00	1,523	95.00%	76
60.00%	2	26	390.63	12.00	1,218	97.00%	37
70.00%	2	19	520.83	12.00	914	98.00%	18
80.00%	1	13	781.25	12.00	609	98.50%	9
90.00%	1	6	1562.50	12.00	305	99.00%	3
100.00%	1	4	2500.00	12.00	2	99.50%	0

Although total cost of ownership (TCO) is largely driven by defect removal and repair costs, there are other factors, too. Table 2-8 shows a hypothetical result for development plus ten years of usage for 0 percent reuse and 80 percent reuse. Obviously, Table 2-8 oversimplifies TCO calculations, but the intent is to show the significant economic value of certified high-quality reusable materials.

Constructing applications that are 100 percent reusable is not likely to be a common event. However, experiments indicate that almost any application could achieve reuse levels of 85 percent to 90 percent if certified reusable components were available. A study done some years ago at IBM for accounting applications found that about 85 percent of the code in the applications was common and generic and involved the logistics of putting accounting onto a computer. About 15 percent of the code actually dealt with accounting per se.

Not only code but also architecture, requirements, design, test materials, user manuals, and other items need to be part of the reusable package, which also has to be under strict configuration control and of course certified to zero-defect levels for optimum value. Software reuse has been a promising technology for many years, but has never achieved its true potential, due primarily to mediocre quality control. If service-oriented architecture (SOA) is to fulfill its promise, then it must achieve excellent quality levels and thereby allow development to make full use of certified reusable components.

In addition to approaching zero-defect quality levels, certified components should also be designed and developed to be much more secure against hacking, viruses, botnets, and other kinds of security attacks. In fact, a strong case can be made that developing reusable materials

TABLE 2-8 Total Cost of Ownership of High-Quality Reusable Materials (0% and 80% reuse volumes)

Application size (function points) =	10,000		
Annual enhancements (function points) =	800		
Monthly cost =	\$10,000		
Support cost =	\$4,000		
Useful life after deployment =	10 years		
	0% Reuse	80% Reuse	Difference
Development	\$26,540,478	\$2,788,372	-\$23,752,106
Enhancement	\$7,680,000	\$1,280,000	-\$6,400,000
Maintenance	\$16,000,000	\$3,200,000	-\$12,800,000
Customer support	\$12,000,000	\$5,165,607	-\$6,834,393
TOTAL COST	\$62,220,478	\$12,433,979	-\$49,786,499
TCO Cost per Function Point	\$3,456.69	\$690.78	-\$2,765.92

with better boundary controls and more secure programming languages, such as E, would add even more value to certified reusable objects.

As the global economy descends into severe recession, every company will be seeking methods to lower costs. Since software costs historically have been large and difficult to control, it may be that the recession will attract renewed interest in software reuse. To be successful, both quality and security certification will be a critical part of the process.

28. Best Practices for Programming or Coding

Programming or coding remains the central activity of software development, even though it is no longer the most expensive. In spite of the promise of software reuse, object-oriented (OO) development, application generators, service-oriented architecture (SOA), and other methods that attempt to replace manual coding with reusable objects, almost every software project in 2009 depends upon coding to a significant degree.

Because programming is a manual and error-prone activity, the continued reliance upon programming or coding places software among the most expensive of all human-built products.

Other expensive products whose cost structures are also driven by the manual effort of skilled craftspeople include constructing 12-meter yachts and constructing racing cars for Formula One or Indianapolis. The costs of the custom-built racing yachts are at least ten times higher than normal class-built yachts of the same displacements. Indy cars or Formula 1 cars are close to 100 times more costly than conventional sedans built on assembly lines and that include reusable components.

One unique feature of programming that is unlike any other engineering discipline is the existence of more than 700 different programming languages. Not only are there hundreds of programming languages, but also some large software applications may have as many as 12 to 15 languages used at the same time. This is partly due to the fact that many programming languages are specialized and have a narrow focus. Therefore if an application covers a wide range of functions, it may be necessary to include several languages. Examples of typical combinations include COBOL and SQL from the 1970s and 1980s; Visual Basic and C from the 1990s; and Java Beans and HTML from the current century.

New programming languages have been developed at a rate of more than one a month for the past 30 years or so. The current table of programming languages maintained by Software Productivity Research (SPR) now contains more than 700 programming languages and continues to grow at a dozen or more languages per calendar year. Refer to www.SPR.com for additional information.

Best practices for programming circa 2009 include the following topics:

- Selection of programming languages to match application needs
- Utilization of structured programming practices for procedural code
- Selection of reusable code from certified sources, before starting to code
- Planning for and including security topics in code, including secure languages such as E
- Avoidance of “spaghetti bowl” code
- Minimizing cyclomatic complexity and essential complexity
- Including clear and relevant comments in the source code
- Using automated static analysis tools for Java and dialects of C
- Creating test cases before or concurrently with the code
- Formal code inspections of all modules
- Re-inspection of code after significant changes or updates
- Renovating legacy code before starting major enhancements
- Removing error-prone modules from legacy code

The U.S. Department of Commerce does not classify programming as a profession, but rather as a craft or skilled trade. Good programming also has some aspects of an art form. As a result, individual human skill and careful training exert a very strong influence on the quality and suitability of software programming.

Experiments in the industry use *pair programming*, where two programmers work concurrently on the same code; one does the programming and the other does real-time review and inspection. Anecdotal evidence indicates that this method may achieve somewhat higher quality levels than average. However, the method seems intrinsically inefficient. Normal development by one programmer, followed by static analysis and peer reviews of code, also achieves better than average quality at somewhat lower costs than pair programming.

It is a proven fact that people do not find their own errors with high efficiency, primarily because they think many errors are correct and don’t realize that they are wrong. Therefore peer reviews, inspections, and other methods of review by other professionals have demonstrable value.

The software industry will continue with high costs and high error rates so long as software applications are custom-coded. Only the substitution of high-quality reusable objects is likely to make a fundamental change in overall software costs, schedules, quality levels, and failure rates.

It should be obvious that if software applications were assembled from reusable materials, then the costs of each reusable component could be much higher than today, with the additional costs going to developing very sophisticated security controls, optimizing performance levels, creating state-of-the-art specifications and user documents, and achieving zero-defect quality levels. Even if a reusable object were to cost 10 times more than today's custom-code for the same function, if the reused object were used 100 times, then the effective economic costs would be only 1/10th of today's cost.

29. Best Practices for Software Project Governance

Over the past few years an alarming number of executives in major corporations have been indicted and convicted for insider trading, financial misconduct, deceiving stockholders with false claims, and other crimes and misdemeanors. As a result, the U.S. Congress passed the Sarbanes-Oxley Act of 2002, which took effect in 2004.

The Sarbanes-Oxley (SOX) Act applies to major corporations with earnings above \$75 million per year. The SOX Act requires a great deal more accountability on the part of corporate executives than was normal in the past. As a result, the topic of governance has become a major issue within large corporations.

Under the concept of *governance*, senior executives can no longer be passive observers of corporate financial matters or of the software applications that contain financial data. The executives are required to exercise active oversight and guidance on all major financial and stock transactions and also on the software used to keep corporate books.

In addition, some added reports and data must be provided in the attempt to expand and reform corporate financial measurements to ensure absolute honesty, integrity, and accountability. Failure to comply with SOX criteria can lead to felony charges for corporate executives, with prison terms of up to 20 years and fines of up to \$500,000.

Since the Sarbanes-Oxley measures apply only to major public companies with revenues in excess of \$75 million per annum, private companies and small companies are not directly regulated by the law. However, due to past irregularities, executives in all companies are now being held to a high standard of trust. Therefore governance is an important topic.

The first implementation of SOX measures seemed to require teams of 25 to 50 executives and information technology specialists working for a year or more to establish the SOX control framework. Many financial applications required modification, and of course all new applications must be SOX compliant. The continuing effort of administering and adhering to SOX criteria will probably amount to the equivalent of

perhaps 20 personnel full time for the foreseeable future. Because of the legal requirements of SOX and the severe penalties for noncompliance, corporations need to get fully up to speed with SOX criteria. Legal advice is very important.

Because of the importance of both governance and Sarbanes-Oxley, a variety of consulting companies now provide assistance in governance and SOX adherence. There are also automated tools that can augment manual methods of oversight and control. Even so, executives need to understand and become more involved with financial software applications than was common before Sarbanes-Oxley became effective.

Improper governance can lead to fines or even criminal charges for software developed by large U.S. corporations. However, good governance is not restricted to very large public companies. Government agencies, smaller companies, and companies in other countries not affected by Sarbanes-Oxley would benefit from using best practices for software governance.

30. Best Practices for Software Project Measurements and Metrics

Leading companies always have software measurement programs for capturing productivity and quality historical data. The state of the art of software measurements for projects in the nominal 10,000–function point domain includes measures of:

1. Accumulated effort
2. Accumulated costs
3. Accomplishing selected milestones
4. Development productivity
5. Maintenance and enhancement productivity
6. The volume of requirements changes
7. Defects by origin
8. Defect removal efficiency
9. Earned value (primarily for defense projects)

The measures of effort are often granular and support work breakdown structures (WBS). Cost measures are complete and include development costs, contract costs, and costs associated with purchasing or leasing packages. There is one area of ambiguity even for top companies: the overhead or burden rates associated with software costs vary widely and can distort comparisons between companies, industries, and countries.

Many military applications use the *earned value* approach for measuring progress. A few civilian projects use the earned value method, but the usage is far more common in the defense community.

Development productivity circa 2008 normally uses function points in two fashions: function points per staff month and/or work hours per function point.

Measures of quality are powerful indicators of top-ranked software producers. Laggards almost never measure quality, while top software companies always do. Quality measures include data on defect volumes by origin (i.e., requirements, design, code, bad fixes) and severity level.

Really sophisticated companies also measure defect removal efficiency. This requires accumulating all defects found during development and also after release to customers for a predetermined period. For example, if a company finds 900 defects during development and the clients find 100 defects in the first three months of use, then the company achieved a 90 percent defect removal efficiency level. Top companies are usually better than 95 percent in defect removal efficiency, which is about 10 percent better than the U.S. average of 85 percent.

One of the uses of measurement data is for comparison against industry benchmarks. The nonprofit International Software Benchmarking Standards Group (ISBSG) has become a major industry resource for software benchmarks and has data on more than 4000 projects as of late 2008. A best practice circa 2009 is to use the ISBSG data collection tool from the start of requirements through development, and then to routinely submit benchmark data at the end. Of course classified or proprietary applications may not be able to do this.

Sophisticated companies know enough to avoid measures and metrics that are not effective or that violate standard economic assumptions. Two common software measures violate economic assumptions and cannot be used for economic analysis:

- Cost per defect penalizes quality and makes buggy software look better than it is.
- The lines of code metric penalizes high-level languages and makes assembly language look more productive than any other.

As will be discussed later, both lines of code and cost per defect violate standard economic assumptions and lead to erroneous conclusions about both productivity and quality. Neither metric is suitable for economic study.

Measurement and metrics are embarrassingly bad in the software industry as this book is written. Not only do a majority of companies measure little or nothing, but some of the companies that do try to measure

use invalid metrics such as lines of code or cost per defect. Measurement of software is a professional embarrassment as of 2009 and urgently needs improvement in both the quantity and quality of measures.

Software has perhaps the worst measurement practices of any “engineering” field in human history. The vast majority of software organizations have no idea of how their productivity and quality compare with other organizations. Lack of good historical data also makes estimating difficult, process improvement difficult, and is one of the factors associated with the high cancellation rate of large software projects. Poor software measurement should be viewed as professional malpractice.

31. Best Practices for Software Benchmarks and Baselines

A *software benchmark* is a comparison of a project or organization against similar projects and organizations from other companies. A *software baseline* is a collection of quality and productivity information gathered at a specific time. Baselines are used to evaluate progress during software process improvement periods.

Although benchmarks and baselines have different purposes, they collect similar kinds of information. The primary data gathered during both benchmarks and baselines includes, but is not limited to, the following:

1. Industry codes such as the North American Industry Classification (NAIC) codes
2. Development countries and locations
3. Application taxonomy of nature, scope, class, and type
4. Application complexity levels for problems, data, and code
5. Application size in terms of function points
6. Application size in terms of logical source code statements
7. Programming languages used for the application
8. Amount of reusable material utilized for the application
9. Ratio of source code statements to function points
10. Development methodologies used for the application (Agile, RUP, TSP, etc.)
11. Project management and estimating tools used for the application
12. Capability maturity level (CMMI) of the project
13. Chart of accounts for development activities
14. Activity-level productivity data expressed in function points
15. Overall net productivity expressed in function points

16. Cost data expressed in function points
17. Overall staffing of project
18. Number and kinds of specialists employed on the project
19. Overall schedule for the project
20. Activity schedules for development, testing, documentation, and so on
21. Defect potentials by origin (requirements, design, code, documents, bad fixes)
22. Defect removal activities used (inspections, static analysis, testing)
23. Number of test cases created for application
24. Defect removal efficiency levels
25. Delays and serious problems noted during development

These 25 topics are needed not only to show the results of specific projects, but also to carry out regression analysis and to show which methods or tools had the greatest impact on project results.

Consulting groups often gather the information for the 25 topics just shown. The questionnaires used would contain about 150 specific questions. About two days of effort are usually needed to collect all of the data for a major application benchmark in the 10,000–function point size range. The work is normally carried out on-site by a benchmark consulting group. During the data collection, both project managers and team members are interviewed to validate the results. The interview sessions usually last about two hours and may include half a dozen developers and specialists plus the project manager.

Collecting full benchmark and baseline data with all 25 factors only occurs within a few very sophisticated companies. More common would be to use partial benchmarks that can be administered by web surveys or remote means without requiring on-site interviews and data collection. The ten most common topics gathered for these partial benchmarks and baselines include, in descending order:

1. Application size in terms of function points
2. Amount of reusable material utilized for the application
3. Development methodologies used for the application (Agile, RUP, TSP, etc.)
4. Capability maturity level (CMMI) of the project
5. Overall net productivity expressed in function points
6. Cost data expressed in function points
7. Overall staffing of project

8. Overall schedule for the project
9. Delays and serious problems noted during development
10. Customer-reported bugs or defects

These partial benchmarks and baselines are of course useful, but lack the granularity for a full and complete statistical analysis of all factors that affect application project results. However, the data for these partial benchmarks and baselines can be collected remotely in perhaps two to three hours once the application is complete.

Full on-site benchmarks can be performed by in-house personnel, but more often are carried out by consulting companies such as the David Consulting Group, Software Productivity Research, Galorath, and a number of others.

As this book is written in 2009, the major source of remote benchmarks and baselines is the International Software Benchmarking Standards Group (ISBSG). ISBSG is a nonprofit organization with headquarters in Australia. They have accumulated data on perhaps 5000 software projects and are adding new data at a rate of perhaps 500 projects per year.

Although the data is not as complete as that gathered by a full on-site analysis, it is still sufficient to show overall productivity rates. It is also sufficient to show the impact of various development methods such as Agile, RUP, and the like. However, quality data is not very complete as this book is written.

The ISBSG data is expressed in terms of function point metrics, which is a best practice for both benchmarks and baselines. Both IFPUG function points and COSMIC function points are included, as well as several other variants such as Finnish and Netherlands function points.

The ISBSG data is heavily weighted toward information technology and web applications. Very little data is available at present for military software, embedded software, systems software, and specialized kinds of software such as scientific applications. No data is available at all for classified military applications.

Another gap in the ISBSG data is due to the intrinsic difficulty of counting function points. Because function point analysis is fairly slow and expensive, very few applications above 10,000 function points have ever been counted. As a result, the ISBSG data lacks applications such as large operating systems and enterprise resource planning packages that are in the 100,000 to 300,000–function point range.

As of 2009, some high-speed, low-cost function point methods are available, but they are so new that they have not yet been utilized for benchmark and baseline studies. However, by 2010 or 2012 (assuming the economy improves), this situation may change.

Because the data is submitted to ISBSG remotely by clients themselves, there is no formal validation of results, although obvious errors are corrected. As with all forms of self-analysis, there may be errors due to misunderstandings or to local variations in how topics are measured.

Two major advantages of the ISBSG data are the fact that it is available to the general public and the fact that the volume of data is increasing rapidly.

Benchmarks and baselines are both viewed as best practices because they have great value in heading off irrational schedule demands or attempting applications with inadequate management and development methods.

Every major project should start by reviewing available benchmark information from either ISBSG or from other sources. Every process improvement plan should start by creating a quantitative baseline against which progress can be measured. These are both best practices that should become almost universal.

32. Best Practices for Software Project Milestone and Cost Tracking

Milestone tracking refers to having formal closure on the development of key deliverables. Normally, the closure milestone is the direct result of some kind of review or inspection of the deliverable. A milestone is not an arbitrary calendar date.

Project management is responsible for establishing milestones, monitoring their completion, and reporting truthfully on whether the milestones were successfully completed or encountered problems. When serious problems are encountered, it is necessary to correct the problems before reporting that the milestone has been completed.

A typical set of project milestones for software applications in the nominal 10,000–function point range would include completion of:

1. Requirements review
2. Project plan review
3. Cost and quality estimate review
4. External design reviews
5. Database design reviews
6. Internal design reviews
7. Quality plan and test plan reviews
8. Documentation plan review
9. Deployment plan review

10. Training plan review
11. Code inspections
12. Each development test stage
13. Customer acceptance test

Failing or delayed projects usually lack serious milestone tracking. Activities might be reported as finished while work was still ongoing. Milestones might be simple dates on a calendar rather than completion and review of actual deliverables. Some kinds of reviews may be so skimpy as to be ineffective. Other topics such as training may be omitted by accident.

It is important that the meaning of *milestone* is ambiguous in the software industry. Rather than milestones being the result of a formal review of a deliverable, the term milestone often refers to arbitrary calendar dates or to the delivery of unreviewed and untested materials.

Delivering documents or code segments that are incomplete, contain errors, and cannot support downstream development work is not the way milestones are used by industry leaders.

Another aspect of milestone tracking among industry leaders is what happens when problems are reported or delays occur. The reaction is strong and immediate: corrective actions are planned, task forces assigned, and correction begins to occur. Among laggards, on the other hand, problem reports may be ignored, and corrective actions very seldom occur.

In a dozen legal cases involving projects that failed or were never able to operate successfully, project tracking was inadequate in every case. Problems were either ignored or brushed aside, rather than being addressed and solved.

An interesting form of project tracking for object-oriented projects has been developed by the Shoulders Corporation. This method uses a 3-D model of software objects and classes using Styrofoam balls of various sizes that are connected by dowels to create a kind of mobile.

The overall structure is kept in a location visible to as many team members as possible. The mobile makes the status instantly visible to all viewers. Color-coded ribbons indicate status of each component, with different colors indicating design complete, code complete, documentation complete, and testing complete (gold). There are also ribbons for possible problems or delays.

This method provides almost instantaneous visibility of overall project status. The same method has been automated using a 3-D modeling package, but the physical structures are easier to see and have proven more useful on actual projects. The Shoulders Corporation method condenses a great deal of important information into a single visual representation that nontechnical staff can readily understand.

33. Best Practices for Software Change Control Before Release

Applications in the nominal 10,000–function point range run from 1 percent to 3 percent per month in new or changed requirements during the analysis and design phases. The total accumulated volume of changing requirements can top 50 percent of the initial requirements when function point totals at the requirements phase are compared with function point totals at deployment. Therefore successful software projects in the nominal 10,000–function point range must use state-of-the-art methods and tools to ensure that changes do not get out of control. Successful projects also use change control boards to evaluate the need for specific changes. And of course all changes that have a significant impact on costs and schedules need to trigger updated development plans and new cost estimates.

The state of the art of change control for applications in the 10,000–function point range includes the following:

- Assigning “owners” of key deliverables the responsibility for approving changes
- Locked “master copies” of all deliverables that change only via formal methods
- Planning contents of multiple releases and assigning key features to each release
- Estimating the number and rate of development changes before starting
- Using function point metrics to quantify changes
- A joint client/development change control board or designated domain experts
- Use of joint application design (JAD) to minimize downstream changes
- Use of formal requirements inspections to minimize downstream changes
- Use of formal prototypes to minimize downstream changes
- Planned usage of iterative development to accommodate changes
- Planned usage of Agile development to accommodate changes
- Formal review of all change requests
- Revised cost and schedule estimates for all changes greater than 10 function points
- Prioritization of change requests in terms of business impact

- Formal assignment of change requests to specific releases
- Use of automated change control tools with cross-reference capabilities

One of the observed byproducts of the usage of formal JAD sessions is a reduction in downstream requirements changes. Rather than having unplanned requirements surface at a rate of 1 percent to 3 percent per month, studies of JAD by IBM and other companies have indicated that unplanned requirements changes often drop below 1 percent per month due to the effectiveness of the JAD technique.

Prototypes are also helpful in reducing the rates of downstream requirements changes. Normally, key screens, inputs, and outputs are prototyped so users have some hands-on experience with what the completed application will look like.

However, changes will always occur for large systems. It is not possible to freeze the requirements of any real-world application, and it is naïve to think this can occur. Therefore leading companies are ready and able to deal with changes, and do not let them become impediments to progress. Consequently, some form of iterative development is a logical necessity.

The newer Agile methods embrace changing requirements. Their mode of operation is to have a permanent user representative as part of the development team. The Agile approach is to start by building basic features as rapidly as possible, and then to gather new requirements based on actual user experiences with the features already provided in the form of running code.

This method works well for small projects with a small number of users. It has not yet been deployed on applications such as Microsoft Vista, where users number in the millions and the features number in the thousands. For such massive projects, one user or even a small team of users cannot possibly reflect the entire range of usage patterns.

Effective software change management is a complicated problem with many dimensions. Software requirements, plans, specifications, source code, test plans, test cases, user manuals, and many other documents and artifacts tend to change frequently. Changes may occur due to external factors such as government regulations, competitive factors, new business needs, new technologies, or to fix bugs.

Furthermore some changes ripple through many different deliverables. A new requirement, for example, might cause changes not only to requirements documentation but also to internal and external specifications, source code, user manuals, test libraries, development plans, and cost estimates.

When function points are measured at the end of requirements and then again at delivery, it has been found that the rate of growth of

“creeping requirements” averages between 1 percent and 3 percent per calendar month during the design and coding phases. Therefore if an application has 1000 function points at the end of the requirements phase, expect it to grow at a rate of perhaps 20 function points per month for the next eight to ten months. Maximum growth has topped 5 percent per month, while minimum growth is about 0.5 percent per month. Agile projects, of course, are at the maximum end.

For some government and military software projects, *traceability* is a requirement, which means that all changes to code or other deliverables must be traced back to an explicit requirement approved by government stakeholders or sponsors.

In addition, a number of people may be authorized to change the same objects, such as source code modules, test cases, and specifications. Obviously, their separate changes need to be coordinated, so it may be necessary to “lock” master copies of key deliverables and then to serialize the updates from various sources.

Because change control is so complex and so pervasive, many automated tools are available that can aid in keeping all changes current and in dealing with the interconnections of change across multiple deliverables.

However, change control cannot be entirely automated since it is necessary for stakeholders, developers, and other key players to agree on significant changes to application scope, to new requirements, and to items that may trigger schedule and cost changes.

At some point, changes will find their way into source code, which implies new test cases will be needed as well. Formal integration and new builds will occur to include sets of changes. These builds may occur as needed, or they may occur at fixed intervals such as daily or weekly.

Change control is a topic that often causes trouble if it is not handled well throughout the development cycle. As will be discussed later, it also needs to be handled well after deployment during the maintenance cycle. Change control is a superset of configuration control, since change control also involves decisions of corporate prestige and competitive issues that are outside the scope of configuration control.

34. Best Practices for Configuration Control

Configuration control is a subset of change control in general. Formal configuration control originated in the 1950s by the U.S. Department of Defense as a method of keeping track of the parts and evolution of complex weapons systems. In other words, hardware configuration control is older than software configuration control. As commonly practiced, configuration control is a mechanical activity that is supported by many tools and substantial automation. Configuration control deals

with keeping track of thousands of updates to documents and source code. Ascertaining whether a particular change is valuable is outside the scope of configuration control.

Software configuration control is one of the *key practice* areas of the Software Engineering Institute's capability maturity model (CMM and CMMI). It is also covered by a number of standards produced by the IEEE, ANSI (American National Standards Institute), ISO, and other organizations. For example, ISO standard 10007-2003 and IEEE standard 828-1998 both cover configuration control for software applications.

Although configuration control is largely automated, it still requires human intervention to be done well. Obviously, features need to be uniquely identified, and there needs to be extensive mapping among requirements, specifications, source code, test cases, and user documents to ensure that every specific change that affects more than one deliverable is correctly linked to other related deliverables.

In addition, the master copy of each deliverable must be locked to avoid accidental changes. Only formal methods with formal validation should be used to update master copies of deliverables.

Automated configuration control is a best practice for all applications that are intended for actual release to customers.

35. Best Practices for Software Quality Assurance (SQA)

Software quality assurance is the general name for organizations and specialists who are responsible for ensuring and certifying the quality levels of software applications before they are delivered to customers.

In large corporations such as IBM, the SQA organizations are independent of software development organizations and report to a corporate vice president of quality. The reason for this is to ensure that no pressure can be brought to bear on SQA personnel by means of threats of poor appraisals or career damage if they report problems against software.

SQA personnel constitute between 3 percent and 5 percent of software engineering personnel. If the SQA organization is below 3 percent, there usually are not enough personnel to staff all projects. It is not a best practice to have "token SQA" organizations who are so severely understaffed that they cannot review deliverables or carry out their roles.

SQA organizations are staffed by personnel who have some training in quality measurements and quality control methodologies. Many SQA personnel are certified as black belts in Six Sigma practices. SQA is not just a testing organization and indeed may not do testing at all. The roles normally played by SQA groups include

- Estimating quality levels in terms of defect potentials and defect removal efficiency
- Measuring defect removal and assigning defect severity levels
- Applying Six Sigma practices to software applications
- Applying quality function deployment (QFD) to software applications
- Moderating and participating in formal inspections
- Teaching classes in quality topics
- Monitoring adherence to relevant corporate, ANSI, and ISO quality standards
- Reviewing all deliverables to ensure adherence to quality standards and practices
- Reviewing test plans and quality plans to ensure completeness and best practices
- Measuring the results of testing
- Performing root-cause analysis on serious defects
- Reporting on potential quality problems to higher management
- Approving release of applications to customers by certifying acceptable quality levels

At IBM and some other companies, formal approval by software quality assurance is a prerequisite for actually delivering software to customers. If the SQA organization recommends against delivery due to quality issues, that recommendation can only be overturned by appeal to the division's vice president or to the president of the corporation. Normally, the quality problems are fixed.

To be definitive about quality issues, the SQA organizations are the primary units that measure software quality, including but not limited to:

Customer satisfaction Leaders perform annual or semiannual customer satisfaction surveys to find out what their clients think about their products. Leaders also have sophisticated defect reporting and customer support information available via the Web. Many leaders in the commercial software world have active user groups and forums. These groups often produce independent surveys on quality and satisfaction topics. There are also focus groups, and some large software companies even have formal *usability labs*, where new versions of products are tried out by customers under controlled conditions. (Note: customer satisfaction is sometimes measured by marketing organizations rather than by SQA groups.)

Defect quantities and origins Industry leaders keep accurate records of the bugs or defects found in all major deliverables, and they start early, during requirements or design. At least five categories of

defects are measured: requirements defects, design defects, code defects, documentation defects, and bad fixes, or secondary bugs introduced accidentally while fixing another bug. Accurate defect reporting is one of the keys to improving quality. In fact, analysis of defect data to search for root causes has led to some very innovative defect prevention and defect removal operations in many companies. Overall, careful measurement of defects and subsequent analysis of the data is one of the most cost-effective activities a company can perform.

Defect removal efficiency Industry leaders know the average and maximum efficiency of every major kind of review, inspection, and test, and they select optimum series of removal steps for projects of various kinds and sizes. The use of pretest reviews and inspections is normal among Baldrige winners and organizations with ultrahigh quality, since testing alone is not efficient enough. Leaders remove from 95 percent to more than 99 percent of all defects prior to delivery of software to customers. Laggards seldom exceed 80 percent in terms of defect removal efficiency and may drop below 50 percent.

Delivered defects by application Industry leaders begin to accumulate statistics on errors reported by users as soon as the software is delivered. Monthly reports are prepared and given to executives, which show the defect trends against all products. These reports are also summarized on an annual basis. Supplemental statistics such as defect reports by country, state, industry, client, and so on, are also included.

Defect severity levels All of the industry leaders, without exception, use some kind of a severity scale for evaluating incoming bugs or defects reported from the field. The number of plateaus varies from one to five. In general, “Severity 1” problems cause the system to fail completely, and the severity scale then descends in seriousness.

Complexity of software It has been known for many years that complex code is difficult to maintain and has higher than average defect rates. A variety of complexity analysis tools are commercially available that support standard complexity measures such as cyclomatic and essential complexity. It is interesting that the systems software community is much more likely to measure complexity than the information technology (IT) community.

Test case coverage Software testing may or may not cover every branch and pathway through applications. A variety of commercial tools are available that monitor the results of software testing and that help to identify portions of applications where testing is sparse or nonexistent. Here, too, the systems software domain is much more likely to measure test coverage than the information technology (IT) domain.

Cost of quality control and defect repairs One significant aspect of quality measurement is to keep accurate records of the costs

and resources associated with various forms of defect prevention and defect removal. For software, these measures include the costs of (1) software assessments, (2) quality baseline studies, (3) reviews, inspections, and testing, (4) warranty repairs and postrelease maintenance, (5) quality tools, (6) quality education, (7) your software quality assurance organization, (8) user satisfaction surveys, and (9) any litigation involving poor quality or customer losses attributed to poor quality. In general, the principles of Crosby's "Cost of Quality" topic apply to software, but most companies extend the basic concept and track additional factors relevant to software projects. The general topics of Cost of Quality include the costs of prevention, appraisal, internal failures, and external failures. For software more details are needed due to special topics such as toxic requirements, security vulnerabilities, and performance issues, which are not handled via normal manufacturing cost of quality.

Economic value of quality One topic that is not well covered in the quality assurance literature is that of the economic value of quality. A phrase that Phil Crosby, the former ITT vice president of quality, made famous is "quality is free." For software it is better than free; it more than pays for itself. Every reduction of 120 delivered defects can reduce maintenance staffing by about one person. Every reduction of about 240 delivered defects can reduce customer support staffing by about one person. In today's world, software engineers spend more days per year fixing bugs than doing actual development. A combination of quality-centered development methods such as Team Software Process (TSP), joint application design (JAD), quality function deployment (QFD), static analysis, inspections, and testing can reduce costs throughout the life cycle and also shorten development schedules. Unfortunately, poor measurement practices make these improvements hard to see except among very sophisticated companies.

As previously stated, a key reason for this is that the two most common metrics for quality, lines of code and cost per defect, are flawed and cannot deal with economics topics. Using defect removal costs per function point is a better choice, but these metrics need to be deployed in organizations that actually accumulate effort, cost, and quality data simultaneously. From studies performed by the author, combinations of defect prevention and defect removal methods that lower defect potentials and raise removal efficiency greater than 95 percent benefit development costs, development schedules, maintenance costs, and customer support costs simultaneously.

Overall quality measures are the most important of almost any form of software measurement. This is because poor quality always causes schedule delays and cost overruns, while good quality is associated with on-time completions of software applications and effective cost controls.

Formal SQA organizations occur most often in companies that build large and complex physical devices such as airplanes, mainframe computers, telephone switching systems, military equipment, and medical equipment. Such organizations have long recognized that quality control is important to success.

By contrast, organizations such as banks and insurance companies that build information technology software may not have SQA organizations. If they do, the organizations are usually responsible for testing and not for a full range of quality activities.

Studies of the delivered quality of software applications indicate that companies with formal SQA organizations and formal testing organizations tend to exceed 95 percent in cumulative defect removal efficiency levels.

36. Best Practices for Inspections and Static Analysis

Formal design and code inspections originated more than 35 years ago in IBM. They still are among the top-ranked methodologies in terms of defect removal efficiency. (Michael Fagan, formerly of IBM Kingston, first published the inspection method with his colleagues Lew Priven, Ron Radice, and then some years later, Roger Stewart.) Further, inspections have a synergistic relationship with other forms of defect removal such as testing and static analysis and also are quite successful as defect prevention methods.

Automated static analysis is a newer technology that originated perhaps 12 years ago. Automated static analysis examines source code for syntactical errors and also for errors in boundary conditions, calls, links, and other troublesome and tricky items. Static analysis may not find embedded requirements errors such as the notorious Y2K problem, but it is very effective in finding thousands of bugs associated with source code issues. Inspections and static analysis are synergistic defect removal methods.

Recent work on software inspections by Tom Gilb, one of the more prominent authors dealing with inspections, and his colleagues continues to support the early finding that the human mind remains the tool of choice for finding and eliminating complex problems that originate in requirements, design, and other noncode deliverables. Indeed, for finding the deeper problems in source code, formal code inspections still outrank testing in defect removal efficiency levels. However, both static analysis and automated testing are now fairly efficient in finding an increasingly wide array of problems.

If an application is written in a language where static analysis is supported (Java, C, C++, and other C dialects), then static analysis is

a best practice. Static analysis may top 87 percent in finding common coding defects. Occasionally there are false positives, however. But these can be minimized by “tuning” the static analysis tools to match the specifics of the applications. Code inspections after static analysis can find some deeper problems such as embedded requirements defects, especially in key modules and algorithms.

Because neither code inspections nor static analysis are fully successful in finding performance problems, it is also necessary to use dynamic analysis for performance issues. Either various kinds of controlled performance test suites are run, or the application is instrumented to record timing and performance data.

Most forms of testing are less than 35 percent efficient in finding errors or bugs. The measured defect removal efficiency of both formal design inspections and formal code inspections averages more than 65 percent efficient, or twice as efficient as most forms of testing. Some inspections top 85 percent in defect removal efficiency levels. Tom Gilb reports that some inspection efficiencies have been recorded that are as high as 88 percent.

A combination of formal inspections of requirements and design, static analysis, formal testing by test specialists, and a formal (and active) software quality assurance (SQA) group are the methods most often associated with projects achieving a cumulative defect removal efficiency higher than 99 percent.

Formal inspections are manual activities in which from three to six colleagues go over design specifications page by page, using a formal protocol. The normal complement is four, including a moderator, a recorder, a person whose work is being inspected, and one other. (Occasionally, new hires or specialists such as testers participate, too.) *Code* inspections are the same idea, but they go over listings or screens line by line. To term this activity an *inspection*, certain criteria must be met, including but not limited to the following:

- There must be a moderator to keep the session moving.
- There must be a recorder to keep notes.
- There must be adequate preparation time before each session.
- Records must be kept of defects discovered.
- Defect data should not be used for appraisals or punitive purposes.

The original concept of inspections was based on actual meetings with live participants. The advent of effective online communications and tools for supporting remote inspections now means that inspections can be performed electronically, which saves on travel costs for teams that are geographically dispersed.

Any software deliverable can be subject to a formal inspection, and the following deliverables have now developed enough empirical data to indicate that the inspection process is generally beneficial:

- Architecture inspections
- Requirements inspections
- Design inspections
- Database design inspections
- Code inspections
- Test plan inspections
- Test case inspections
- User documentation inspections

For every software artifact where formal inspections are used, the inspections range from just under 50 percent to more than 80 percent in defect removal efficiency and have an average efficiency level of roughly 65 percent. This is overall the best defect removal efficiency level of any known form of error elimination.

Further, thanks to the flexibility of the human mind and its ability to handle inductive logic as well as deductive logic, inspections are also the most versatile form of defect removal and can be applied to essentially any software artifact. Indeed, inspections have even been applied recursively to themselves, in order to fine-tune the inspection process and eliminate bottlenecks and obstacles.

It is sometimes asked “If inspections are so good, why doesn’t everyone use them?” The answer to this question reveals a basic weakness of the software industry. Inspections have been in the public domain for more than 35 years. Therefore no company except a few training companies tries to “sell” inspections, while there are many vendors selling testing tools. If you want to use inspections, you have to seek them out and adopt them.

Most software development organizations don’t actually do research or collect data on effective tools and technologies. They make their technology decisions to a large degree by listening to tool and methodology vendors and adopting those where the sales personnel are most persuasive. It is even easier if the sales personnel make the tool or method sound like a silver bullet that will give miraculous results immediately upon deployment, with little or no training, preparation, or additional effort. Since inspections are not sold by tool vendors and do require training and effort, they are not a glamorous technology. Hence many software organizations don’t even know about inspections and have no idea of their versatility and effectiveness.

It is a telling point that all of the top-gun software quality houses and even industries in the United States tend to utilize pretest inspections. For example, formal inspections are very common among computer manufacturers, telecommunication manufacturers, aerospace manufacturers, defense manufacturers, medical instrument manufacturers, and systems software and operating systems developers. All of these need high-quality software to market their main products, and inspections top the list of effective defect removal methods.

It is very important not to allow toxic requirements, requirements errors, and requirements omissions to flow downstream into code, because requirements problems cannot be found and removed by testing.

Design problems should also be found prior to code development, although testing can find some design problems.

The key message is that defects should be found within no more than a few hours or days from when they originate. Defects that originate in a specific phase such as requirements should never be allowed downstream into design and code.

Following are the most effective known methods for finding defects within a specific phase or within a short time interval from when the defects originate:

Defect Origins	Optimal Defect Discovery Methods
Requirements defects	Formal requirements inspections
Design defects	Formal design inspections
Coding defects	Static analysis
	Formal code inspections
	Testing
Document defects	Editing of documents
	Formal document inspections
Bad fixes	Re-inspection after defect repairs
	Rerunning static analysis tools after defect repairs
	Regression testing
Test case defects	Inspection of test cases

As can be seen, inspections are not the only form of defect removal, but they are the only form that has proven to be effective against requirements defects, and they are also very effective against other forms of defects as well.

A new nonprofit organization was created in 2009 that is intended to provide instruction and quantified data about formal inspections. The organization is being formed as this book is written.

As of 2009, inspections are supported by a number of tools that can predict defects, defect removal efficiency, costs, and other relevant factors. These tools also collect data on defects and effort, and can consolidate the data with similar data from static analysis and testing.

Formal inspections are a best practice for all mission-critical software applications.

37. Best Practices for Testing and Test Library Control

Software testing has been the main form of defect removal since software began more than 60 years ago. At least 20 different forms of testing exist, and typically between 3 and 12 forms of testing will be used on almost every software application.

Note that testing can also be used in conjunction with other forms of defect removal such as static analysis and formal inspections. In fact, such synergistic combinations are best practices, because testing by itself is not sufficient to achieve high levels of defect removal efficiency.

Unfortunately, when measured, testing is rather low in defect removal efficiency levels. Many forms of testing such as unit test are below 35 percent in removal efficiency, or find only about one bug out of three.

The cumulative efficiency of all forms of testing seldom tops 80 percent, so additional steps such as inspections and static analysis are needed to raise defect removal efficiency levels above 95 percent, which is a minimum safe level.

Because of the many forms of testing and the existence of hundreds or thousand of test cases, test libraries are often huge and cumbersome, and require automation for successful management.

Testing has many varieties, including *black box testing* (no knowledge of application structure), *white box testing* (application structure is known), and *gray box testing* (data structures are known).

Another way of dividing testing is to look at test steps performed by developers, by testing specialists or quality assurance, and by customers themselves. Testing in all of its forms can utilize 20 percent to more than 40 percent of total software development effort. Given the low efficiency of testing in terms of defect removal, alternatives that combine higher efficiency levels with lower costs are worth considering.

There are also very specialized forms of testing such as tests concerned with performance issues, security issues, and usability issues. Although not testing in the normal sense of the word, applications with high security criteria may also use professional hackers who seek to penetrate the application's defenses. Common forms of software testing include

Testing by Developers

- Subroutine testing
- Module testing
- Unit testing

Testing by Test Specialists or Software Quality Assurance

- New function testing
- Component testing
- Regression testing
- Performance testing
- Security testing
- Virus and spyware testing
- Usability testing
- Scalability testing
- Standards testing (ensuring ISO and other standards are followed)
- Nationalization testing (foreign languages versions)
- Platform testing (alternative hardware or operating system versions)
- Independent testing (military applications)
- Component testing
- System testing

Testing by Customers or Users

- External beta testing (commercial software)
- Acceptance testing (information technology; outsource applications)
- In-house customer testing (special hardware devices)

In recent years automation has facilitated test case development, test script development, test execution, and test library management. However, human intelligence is still very important in developing test plans, test cases, and test scripts.

Several issues with testing are underreported in the literature and need more study. One of these is the error density in test cases themselves. Studies of samples of test libraries at selected IBM locations sometimes found more errors in test cases than in the software being tested. Another issue is that of redundant test cases, which implies that two or more test cases are duplicates or test the same conditions. This adds costs, but not rigor. It usually occurs when multiple developers or multiple test personnel are engaged in testing the same software.

A topic that has been studied but which needs much more study is that of testing defect removal efficiency. Since most forms of testing seem to be less than 35 percent efficient, or find only about one bug out of three, there is an urgent need to examine why this occurs.

A related topic is the low coverage of testing when monitored by various test coverage analysis tools. Usually, only 75 percent or less of the

source code in applications is executed during the course of testing. Some of this may be dead code (which is another problem), some may be paths that are seldom traversed, but some may be segments that are missed by accident.

The bottom line is that testing alone is not sufficient to achieve defect removal efficiency levels of 95 percent or higher. The current best practice would be to use testing in conjunction with other methods such as requirements and design inspections, static analysis, and code inspections prior to testing itself. Both defect prevention and defect removal should be used together in a synergistic fashion.

Effective software quality control is the most important single factor that separates successful projects from delays and disasters. The reason for this is because finding and fixing bugs is the most expensive cost element for large systems and takes more time than any other activity.

Successful quality control involves defect prevention, defect removal, and defect measurement activities. The phrase *defect prevention* includes all activities that minimize the probability of creating an error or defect in the first place. Examples of defect prevention activities include the Six Sigma approach, joint application design (JAD) for gathering requirements, usage of formal design methods, use of structured coding techniques, and usage of libraries of proven reusable material.

The phrase *defect removal* includes all activities that can find errors or defects in any kind of deliverable. Examples of defect removal activities include requirements inspections, design inspections, document inspections, code inspections, and all kinds of testing. Following are the major forms of defect prevention and defect removal activities practiced as of 2009:

Defect Prevention

- Joint application design (JAD) for gathering requirements
- Quality function deployment (QFD) for quality requirements
- Formal design methods
- Structured coding methods
- Renovation of legacy code prior to updating it
- Complexity analysis of legacy code prior to updating it
- Surgical removal of error-prone modules from legacy code
- Formal defect and quality estimation
- Formal security plans
- Formal test plans
- Formal test case construction
- Formal change management methods

- Six Sigma approaches (customized for software)
- Utilization of the Software Engineering Institute's capability maturity model (CMM or CMMI)
- Utilization of the new team and personal software processes (TSP, PSP)
- Embedded users with development teams (as in the Agile method)
- Creating test cases before code (as with Extreme programming)
- Daily SCRUM sessions

Defect Removal

- Requirements inspections
- Design inspections
- Document inspections
- Formal security inspections
- Code inspections
- Test plan and test case inspection
- Defect repair inspection
- Software quality assurance reviews
- Automated software static analysis (for languages such as Java and C dialects)
- Unit testing (automated or manual)
- Component testing
- New function testing
- Regression testing
- Performance testing
- System testing
- Security vulnerability testing
- Acceptance testing

The combination of defect prevention and defect removal activities leads to some very significant differences when comparing the overall numbers of software defects in successful versus unsuccessful projects. For projects in the 10,000–function point range, the successful ones accumulate development totals of around 4.0 defects per function point and remove about 95 percent of them before delivery to customers. In other words, the number of delivered defects is about 0.2 defect per function point, or 2,000 total latent defects. Of these, about 10 percent or 200 would be fairly serious defects. The rest would be minor or cosmetic defects.

By contrast, the unsuccessful projects accumulate development totals of around 7.0 defects per function point and remove only about 80 percent of them before delivery. The number of delivered defects is about 1.4 defects per function point, or 14,000 total latent defects. Of these, about 20 percent or 2,800 would be fairly serious defects. This large number of latent defects after delivery is very troubling for users.

Unsuccessful projects typically omit design and code inspections, static analysis, and depend purely on testing. The omission of upfront inspections causes three serious problems: (1) The large number of defects still present when testing begins slows down the project to a standstill; (2) The “bad fix” injection rate for projects without inspections is alarmingly high; and (3) The overall defect removal efficiency associated only with testing is not sufficient to achieve defect removal rates higher than about 80 percent.

38. Best Practices for Software Security Analysis and Control

As this book is written in 2009, software security is becoming an increasingly critical topic. Not only are individual hackers attempting to break into computers and software applications, but also organized crime, drug cartels, terrorist organizations such as al Qaeda, and even hostile foreign governments are.

As computers and software become more pervasive in business and government operations, the value of financial data, military data, medical data, and police data is high enough so that criminal elements can afford to mount major attacks using very sophisticated tools and also very sophisticated hackers. Cybersecurity is becoming a major battleground and needs to be taken very seriously.

Modern software applications that contain sensitive data such as financial information, medical records, personnel data, or military and classified information are at daily risk from hackers, viruses, spyware, and even from deliberate theft by disgruntled employees. Security control of software applications is a serious business, associated with major costs and risks. Poor security control can lead to serious damages and even to criminal charges against the software and corporate executives who did not ensure high security levels.

Modern security control of critical software applications requires a combination of specialized skills; sophisticated software tools; proper architecture, design, and coding practices; and constant vigilance. Supplemental tools and approaches such as hardware security devices, electronic surveillance of premises, careful background checks of all personnel, and employment of hackers who deliberately seek out weaknesses and vulnerabilities are also very common and may be necessary.

However, software security starts with careful architecture, design, and coding practices. In addition, security inspections and the employment of security specialists are key criteria for successful security control. Both “blacklisting” and “whitelisting” of applications that interface with applications undergoing security analysis are needed. Also, programming languages such as E (a Java variation) that are aimed at security topics are important and also a best practice.

Security leaks or vulnerabilities come from a variety of sources, including user inputs, application interfaces, and of course leaks due to poor error-handling or poor coding practices. One of the reasons that specialists are required to reduce security vulnerabilities is because ordinary training of software engineers is not thorough in security topics.

Dozens of companies are now active in the security area. The U.S. Department of Homeland Security is planning on building a new research lab specifically for software security. Nonprofit organizations such as the Center for Internet Security (CIS) are growing rapidly in membership, and joining such a group would be a best practice for both corporations and government agencies.

In addition, security standards such as ISO 17799 also offer guidance on software security topics.

Although hacking and online theft of data is the most widespread form of security problem, physical security of computers and data centers is important, too. Almost every month, articles appear about loss of confidential credit card and medical records due to theft of notebook computers or desktop computers.

Because both physical theft and hacking attacks are becoming more and more common, encryption of valuable data is now a best practice for all forms of proprietary and confidential information.

From about 2000 forward into the indefinite future, there has been an escalating contest between hackers and security experts. Unfortunately, the hackers are becoming increasingly sophisticated and numerous.

It is theoretically possible to build some form of artificial intelligence or neural network security analysis tools that could examine software applications and find security flaws with very high efficiency. Indeed, a similar kind of AI tool applied to architecture and design could provide architects and designers with optimal security solutions.

A general set of best practices for software applications under development includes

- Improve the undergraduate and professional training of software engineers in security topics.
- For every application that will connect to the Internet or to other computers, develop a formal security plan.
- Perform security inspections of requirements and specifications.

- Develop topnotch physical security for development teams.
- Develop topnotch security for home offices and portable equipment.
- Utilize high-security programming languages such as E.
- Utilize automated static analysis of code to find potential security vulnerabilities.
- Utilize static analysis on legacy applications that are to be updated.

Automation will probably become the overall best practice in the future. However, as of 2009, security analysis by human experts remains the best practice. While security experts are common in military and classified areas, they are not yet used as often as they should be for civilian applications.

39. Best Practices for Software Performance Analysis

As any user of Windows XP or Windows Vista can observe, performance of large and complex software applications is not as sophisticated as it should be. For Windows, as an example, application load times slow down over time. A combination of increasing Internet clutter and spyware can degrade execution speed to a small fraction of optimum values.

While some utility applications can restore a measure of original performance, the fact remains that performance optimization is a technology that needs to be improved. Microsoft is not alone with sluggish performance. A frequent complaint against various Symantec tools such as the Norton AntiVirus package is that of extremely slow performance. The author has personally observed a Norton AntiVirus scan that did not complete after 24 hours, although the computer did not have the latest chips.

Since performance analysis is not always a part of software engineering or computer science curricula, many software engineers are not qualified to deal with optimizing performance. Large companies such as IBM employ performance specialists who are trained in such topics. For companies that build large applications in the 100,000–function point range, employment of specialists would be considered a best practice.

There are a number of performance tools and measurement devices such as *profilers* that collect data on the fly. It is also possible to embed performance measurement capabilities into software applications themselves, which is called *instrumentation*.

Since instrumentation and other forms of performance analysis may slow down application speed, care is needed to ensure that the data is correct. Several terms derived from physics and physicists have moved into the performance domain. For example, a *heisenbug* is named after

Heisenberg's uncertainty principle. It is a bug that disappears when an attempt is made to study it. Another physics-based term is *bohrbug* named after Nils Bohr. A bohrbug occurs when a well-defined set of conditions occur, and does not disappear. A third term from physics is that of *mandelbug* named after Benoit Mandelbrot, who developed chaos theory. This form of bug is caused by such random and chaotic factors that isolation is difficult. A fourth and very unusual form of bug is a *schrodenbug* named after Ernst Schrodinger. This form of bug does not occur until someone notices that the code should not have worked at all, and as soon as the bug is discovered, the software stops working (reportedly).

Performance issues also occur based on business cycles. For example, many financial and accounting packages tend to slow down at the end of a quarter or the end of a fiscal year when usage increases dramatically.

One topic that is not covered well in the performance literature is the fact that software performance drops to zero when a high-severity bug is encountered that stops it from running. Such problems can be measured using mean-time-to-failure. These problems tend to be common in the first month or two after a release, but decline over time as the software stabilizes. Other stoppages can occur due to denial of service attacks, which are becoming increasingly common.

This last point brings up the fact that performance best practices overlap best practices in quality control and security control. A general set of best practices includes usage of performance specialists, excellence in quality control, and excellence in security control.

As with security, it would be possible to build an artificial intelligence or neural net performance optimization tool that could find performance problems better than testing or perhaps better than human performance experts. A similar tool applied to architecture and design could provide performance optimization rules and algorithms prior to code development.

In general, AI and neural net approaches for dealing with complex problems such as security flaws and performance issues have much to recommend them. These topics overlap *autonomous computing*, or applications that tend to monitor and improve their own performance and quality.

40. Best Practices for International Software Standards

Because software is not a recognized engineering field with certification and licensing, usage of international standards has been inconsistent. Further, when international standards are used, not very much empirical

data is available that demonstrates whether they were helpful, neutral, or harmful for the applications being developed. Some of the international standards that apply to software are established by the International Organization for Standards commonly known as the *ISO*. Examples of standards that affect software applications include

- ISO/IEC 10181 Security Frameworks
- ISO 17799 Security
- Sarbanes-Oxley Act
- ISO/IEC 25030 Software Product Quality Requirements
- ISO/IEC 9126-1 Software Engineering Product Quality
- IEEE 730-1998 Software Quality Assurance Plans
- IEEE 1061-1992 Software Metrics
- ISO 9000-9003 Quality Management
- ISO 9001:2000 Quality Management System

There are also international standards for functional sizing. As of 2008, data on the effectiveness of international standards in actually generating improvements is sparse.

Military and defense applications also follow military standards rather than ISO standards. Many other standards will be dealt with later in this book.

41. Best Practices for Protecting Intellectual Property in Software

The obvious first step and also a best practice for protecting intellectual property in software is to seek legal advice from a patent or intellectual property law firm. Only an intellectual property lawyer can provide proper guidance through the pros and cons of copyrights, patents, trademarks, service marks, trade secrets, nondisclosure agreements, noncompetition agreements, and other forms of protection. The author is of course not an attorney, and nothing in this section or this book should be viewed as legal advice.

Over and above legal advice, technical subjects also need to be considered, such as encryption of sensitive information, computer firewalls and hacking protection, physical security of offices, and for classified military software, perhaps even isolation of computers and using protective screens that stop microwaves. Microwaves can be used to collect and analyze what computers are doing and also to extract confidential data.

Many software applications contain proprietary information and algorithms. Some defense and weapons software may contain classified

information as well. Patent violation lawsuits and theft of intellectual property lawsuits are increasing in number, and this trend will probably continue. Overt theft of software and data by hackers or bribery of personnel are also occurring more often than in the past.

Commercial software vendors are also concerned about piracy and the creation of unauthorized copies of software. The solutions to this problem include registration, activation, and in some cases actually monitoring the software running on client computers, presumably with client permission. However, these solutions have been only partially successful, and unlawful copying of commercial software is extremely common in many developing countries and even in the industrialized nations.

One obvious solution is to utilize encryption of all key specifications and code segments. However, this method raises logistical issues for the development team, since unencrypted information is needed for human understanding.

A possible future solution may be associated with cloud computing, where applications reside on network servers rather than on individual computers. Although such a method might protect the software itself, it is not trouble free and may be subject to hacking, interception from wireless networks, and perhaps even denial of service attacks.

Since protection of intellectual property requires expert legal advice and also specialized advice from experts in physical security and online security, only a few general suggestions are given here.

Be careful with physical security of office spaces, notebook computers, home computers that may contain proprietary information, and of course e-mail communications. Theft of computers, loss of notebook computers while traveling, and even seizure of notebook computers when visiting foreign countries might occur. Several companies prohibit employees from bringing computers to various overseas locations.

In addition to physical security of computers, it may be necessary to limit usage of thumb drives, DVDs, writable CD disks, and other removable media. Some companies and government organizations prohibit employees from carrying removable media in and out of offices.

If your company supports home offices or telecommuting, then your proprietary information is probably at some risk. While most employees are probably honest, there is no guarantee that their household members might not attempt hacking just for enjoyment. Further, you may not have any control over employee home wireless networks, some of which may not have any security features activated.

For employees of companies with proprietary intellectual property, some form of employment agreement and noncompetition agreement would normally be required. This is sometimes a troublesome area, and a few companies demand ownership of all employee inventions,

whether or not they are job related. Such a Draconian agreement often suppresses innovation.

Outsource agreements should also be considered as part of protecting intellectual property. Obviously, outsource vendors need to sign confidentiality agreements. These may be easier to enforce in the United States than in some foreign locations, which is a factor that needs to be considered also.

If the intellectual property is embedded in software, it may be prudent to include special patterns of code that might identify the code if it is pirated or stolen.

If the company downsizes or goes out of business, special legal advice should be sought to deal with the implications of handling intellectual property. For downsizing, obviously all departing employees will probably need to sign noncompete agreements. For going out of business, intellectual property will probably be an asset under bankruptcy rules, so it still needs to be protected.

While patents are a key method of protecting intellectual property, they are hotly debated in the software industry. One side sees patents as the main protective device for intellectual property; the other side sees patents as merely devices to extract enormous fees. Also, there may be some changes in patent laws that make software patents difficult to acquire in the future. The topic of software patents is very complex, and the full story is outside the scope of this book.

One curious new method of protecting algorithms and business rules in software is derivative of the “Bible code” and is based on equidistant letter spacing (ELS).

A statistical analysis of the book of Genesis found that letters that were equally spaced sometimes spelled out words and even phrases. It would be possible for software owners to use the same approach either with comments or actual instructions and to embed a few codes using the ELS method that identified the owner of the software. Equally spaced letters that spelled out words or phrases such as “stop thief” could be used as evidence of theft. Of course this might backfire if thieves inserted their own ELS codes.

42. Best Practices for Protecting Against Viruses, Spyware, and Hacking

As of 2009, the value of information is approaching the value of gold, platinum, oil, and other expensive commodities. In fact, as the global recession expands, the value of information is rising faster than the value of natural products such as metals or oil. As the value of information goes up, it is attracting more sophisticated kinds of thievery. In the past, hacking and viruses were often individual efforts, sometimes

carried out by students and even by high-school students at times just for the thrill of accomplishing the act.

However, in today's world, theft of valuable information has migrated to organized crime, terrorist groups, and even to hostile foreign governments. Not only that, but denial of service attacks and "search bots" that can take over computers are powerful and sophisticated enough to shut down corporate data centers and interfere with government operations. This situation is going to get worse as the global economy declines.

Since computers are used to store valuable information such as financial records, medical records, patents, trade secrets, classified military information, customer lists, addresses and e-mail addresses, phone numbers, and social security numbers, the total value of stored information is in the range of trillions of dollars. There is no other commodity in the modern world that is simultaneously so valuable and so easy to steal as information stored in a computer.

Not only are there increasing threats against software and financial data, but it also is technically within the realm of possibility to hack into voting and election software as well. Any computer connected to the outside world by any means is at risk. Even computers that are physically isolated may be at some risk due to their electromagnetic emissions.

Although many individual organizations such as Homeland Security, the Department of Defense, the FBI, NSA (National Security Agency), IBM, Microsoft, Google, Symantec, McAfee, Kaspersky, Computer Associates, and scores of others have fairly competent security staffs and also security tools, the entire topic needs to have a central coordinating organization that would monitor security threats and distribute data on best practices for preventing them. The fragmentation of the software security world makes it difficult to organize defenses against all known threats, and to monitor the horizon for future threats.

The FBI started a partnership organization with businesses called InfraGuard that is intended to share data on software and computer security issues. According to the InfraGuard web site, about 350 of the Fortune 500 companies are members. This organization has local branches affiliated with FBI field offices in most major cities such as Boston, Chicago, San Francisco, and the like. However, smaller companies have not been as proactive as large corporations in dealing with security matters. Membership in InfraGuard would be a good first step and a best practice as well.

The Department of Homeland Security (DHS) also has a joint government-business group for Software Assurance (SwA). This group has published a Software Security State of the Art Report (SOAR) that summarizes current best practices for prevention, defense, and recovery from security flaws. Participation in this group and following the principles discussed in the SOAR would be best practices, too.

As this book is being written, Homeland Security is planning to construct a major new security research facility that will probably serve as a central coordination location for civilian government agencies and will assist businesses as well.

A new government security report chaired by Representative James Langevin of Rhode Island is also about to be published, and it deals with all of the issues shown here as well as others, and in greater detail. It will no doubt provide additional guidance beyond what is shown here.

Unfortunately, some of the security literature tends to deal with threats that occur after development and deployment. The need to address security as a fundamental principle of architecture, design, and development is poorly covered. A book related to this one, by Ken Hamer-Hodges, *Authorization Oriented Architecture*, will deal with more fundamental subjects. Among the subjects is automating computer security to move the problem from the user to the system itself. The way to do this is through detailed boundary management. That is why objects plus capabilities matter. Also, security frames such as Google Caja, which prevents redirection to phishing sites, are best practices. The new E programming language is also a best practice, since it is designed to ensure optimum security.

The training of business analysts, systems analysts, and architects in security topics has not kept pace with the changes in malware, and this gap needs to be bridged quickly, because threats are becoming more numerous and more serious.

It is useful to compare security infections with medical infections. Some defenses against infections, such as firewalls, are like biohazard suits, except the software biohazard suits tend to leak.

Other defenses, such as antivirus and antispyware applications, are like antibiotics that stop some infections from spreading and also kill some existing infections. However, as with medical antibiotics, some infections are resistant and are not killed or stopped. Over time the resistant infections tend to evolve more rapidly than the infections that were killed, which explains why polymorphic software viruses are now the virus of choice.

What might be the best long-term strategy for software would be to change the DNA of software applications and to increase their natural immunity to infections via better architecture, better design, more secure programming languages, and better boundary controls.

The way to solve security problems is to consider the very foundations of the science and to build boundary control in physical terms based on the Principle of Least Authority, where each and every subroutine call is treated as an instance of a protected class of object. There should be no Global items, no Global Name Space, no Global path names like C:/directory/file or URL <http://123.456.789/file>. Every subroutine should

be a protected call with boundary checking, and all program references are dynamically bound from a local name at run time with access control check included at all times. Use secure languages and methods (for example, E and Caja today). Some suggested general best practices from the Hamer-Hodges draft include

- Change passwords frequently (outdated by today's technology).
- Don't click on e-mail links—type the URL in manually.
- Disable the preview pane in all inboxes.
- Read e-mail in plain text.
- Don't open e-mail attachments.
- Don't enable Java, JS, or particularly ActiveX.
- Don't display your e-mail address on your web site.
- Don't follow links without knowing what they link to.
- Don't let the computer save your passwords.
- Don't trust the "From" line in e-mail messages.
- Upgrade to latest security levels, particularly for Internet Explorer.
- Consider switching to Firefox or Chrome.
- Never run a program unless it is trusted.
- Read the User Agreement on downloads (they may sell your personal data).
- Expect e-mail to carry worms and viruses.
- Just say no to pop-ups.
- Say no if an application asks for additional or different authorities.
- Say no if it asks to read or edit anything more than a Desktop folder.
- Say no if an application asks for edit authority on other stuff.
- Say no if it asks for read authority on odd stuff, with a connection to the Web.
- During an application install, supply a new name, new icon, and a new folder path.
- Say no when anything asks for web access beyond a specific site.
- Always say no unless you want to be hit sooner or later.

Internet security is so hazardous as of 2009 that one emerging best practice is for sophisticated computer users to have two computers. One of these would be used for web surfing and Internet access. The second

computer would not be connected to the Internet and would accept only trusted inputs on physical media that are of course checked for viruses and spyware.

It is quite alarming that hackers are now organized and have journals, web sites, and classes available for teaching hacking skills. In fact, a review of the literature indicates that there is more information available about how to hack than on how to defend against hacking. As of 2009, the hacking “industry” seems to be larger and more sophisticated than the security industry, which is not surprising, given the increasing value of information and the fundamental flaws in computer security methods. There is no real census of either hackers or security experts, but as of 2009, the hacking community may be growing at a faster rate than the security community.

Standard best practices include use of firewalls, antivirus packages, antispyware packages, and careful physical security. However, as the race between hackers and security companies escalates, it is also necessary to use constant vigilance. Virus definitions should be updated daily, for example. More recent best practices include biological defenses such as using fingerprints or retina patterns in order to gain access to software and computers.

Two topics that have ambiguous results as of 2009 are those of identifying theft insurance and certification of web sites by companies such as VeriSign. As to identity theft insurance, the idea seems reasonable, but what is needed is more active support than just reimbursement for losses and expenses. What would perhaps be a best practice would be a company or nonprofit that had direct connections to all credit card companies, credit bureaus, and police departments and could offer rapid response and assistance to consumers with stolen identities.

As to certification of web sites, an online search of that subject reveals almost as many problems and mistakes as benefits. Here, too, the idea may be valid, but the implementation is not yet perfect. Whenever problem reports begin to approach benefit reports in numbers, the topic is not suitable for best practice status.

Some examples of the major threats in today’s cyberworld are discussed below in alphabetical order:

Adware Because computer usage is so common, computers have become a primary medium for advertising. A number of software companies generate income by placing ads in their software that are displayed when the software executes. In fact, for shareware and freeware, the placing of ads may be the primary source of revenue. As an example, the Eudora e-mail client application has a full-featured version that is supported by advertising revenue. If adware were nothing but a passive display of information, it would be annoying but not hazardous. However, adware can also collect information as well as display it.

When this occurs, adware tends to cross a line and become spyware. As of 2009, ordinary consumers have trouble distinguishing between adware and spyware, so installation of antispyspyware tools is a best practice, even if not totally effective. In fact, sophisticated computer users may install three or four different antispyspyware tools, because none are 100 percent effective by themselves.

Authentication, authorization, and access Computers and software tend to have a hierarchy of methods for protection against unauthorized use. Many features are not accessible to ordinary users, but require some form of *administrative access*. Administrative access is assigned when the computer or software is first installed. The administrator then grants other users various permissions and access rights. To use the computer or software, users need to be *authenticated* or identified to the application with the consent of the administrator. Not only human users but also software applications may need to be authenticated and given access rights. While authenticating human users is not trivial, it can be done without a great deal of ambiguity. For example, retina prints or fingerprints provide an unambiguous identification of a human user. However, authenticating and authorizing software seems to be a weak link in the security chain. Access control lists (ACLs) are the only available best practice, but just for static files, services, and networks. ACL cannot distinguish identities, so a virus or Trojan has the same authorization as the session owner! If some authorized software contains worms, viruses, or other forms of malware, they may use access rights to propagate. As of 2009, this problem is complex enough that there seems to be no best practice for day-to-day authorization. However, a special form of authorization called *capability-based security* is at least in theory a best practice. Unfortunately, capability-based security is complex and not widely utilized. Historically, the Plessey 250 computer implemented a hardware-based capability model in order to prevent hacking and unauthorized changes of access lists circa 1975. This approach dropped from use for many years, but has resurfaced by means of Google's Caja and the E programming language.

Back door Normally, to use software, some kind of login process and password are needed. The term *back door* refers to methods for gaining access to software while bypassing the normal entry points and avoiding the use of passwords, user names, and other protocols. Error-handling routines and buffer overruns are common backdoor entry points. Some computer worms install back doors that might be used to send spam or to perform harmful actions. One surprising aspect of back doors is that occasionally they are deliberately put into software by the programmers who developed the applications. This is why classified software and software that deals with financial data needs careful inspection, static analysis, and of course background security checks

of the software development team. Alarming, back doors can also be inserted by compilers if the compiler developer put in such a function. The backdoor situation is subtle and hard to defend against. Special artificial intelligence routines in static analysis software may become a best practice, but the problem remains complex and hard to deal with. Currently, several best practice rules include (1) assume errors are signs of an attack in process; (2) never let user-coded error recovery run at elevated privileged levels; (3) never use global (path) addressing for URL or networked files; and (4) local name space should be translated only by a trusted device.

Botnets The term *botnet* refers to a collection of “software robots” that act autonomously and attempt to seize control of hundreds or thousand of computers on a network and turn them into “zombie computers.” The bots are under control of a *bot herder* and can be used for a number of harmful purposes such as denial of service attacks or sending spam. In fact, this method has become so pervasive that bot herders actually sell their services to spammers! Botnets tend to be sophisticated and hard to defend against. While firewalls and fingerprinting can be helpful, they are not 100 percent successful. Constant vigilance and top-gun security experts are a best practice. Some security companies are now offering botnet protection using fairly sophisticated artificial intelligence techniques. It is alarming that cybercriminals and cyberdefenders are apparently in a heated technology race. Lack of boundary controls is what allow botnets to wander at will. Fundamental architectural changes, use of Caja, and secure languages such as E could stop botnets.

Browser hijackers This annoying and hazardous security problem consists of software that overrides normal browser addresses and redirects the browser to some other site. Browser hijackers were used for marketing purposes, and sometimes to redirect to porn sites or other unsavory locations. A recent form of browser hijacking is termed *rogue security sites*. A pop-up ad will display a message such as “YOUR COMPUTER IS INFECTED” and direct the user to some kind of security site that wants money. Of course, it might also be a phishing site. Modern antispymware tools are now able to block and remove browser hijackers in most cases. They are a best practice for this problem, but they must be updated frequently with new definitions. Some browsers such as Google Chrome and Firefox maintain lists of rogue web sites and caution users about them. This keeping of lists is a best practice.

Cookies These are small pieces of data that are downloaded from web sites onto user computers. Once downloaded, they then go back and forth between the user and the vendor. Cookies are not software but rather passive data, although they do contain information about the user. Benign uses of cookies are concerned with online shopping and with

setting up user preferences on web sites such as Amazon. Harmful uses of cookies include capturing user information for unknown or perhaps harmful purposes. For several years, both the CIA and NSA downloaded cookies into any computer that accessed their web sites for any reason, which might have allowed the creation of large lists of people who did nothing more than access web sites. Also, cookies can be hijacked or changed by a hacker. Unauthorized change of a cookie is called *cookie poisoning*. It could be used, for example, to change the amount of purchase at an online store. Cookies can be enabled or disabled on web browsers. Because cookies can be either beneficial or harmful, there is no general best practice for dealing with them. The author's personal practice is to disable cookies unless a specific web site requires cookies for a business purpose originated by the author.

Cyberextortion Once valuable information such as bank records, medical records, or trade secrets are stolen, what next? One alarming new form of crime is cyberextortion, or selling the valuable data back to the original owner under threat of publishing it or selling it to competitors. This new crime is primarily aimed at companies rather than individuals. The more valuable the company's data, the more tempting it is as a target. Best practices in this area involve using topnotch security personnel, constant vigilance, firewalls and the usual gamut of security software packages, and alerting authorities such as the FBI or the cybercrime units of large police forces if extortion is attempted.

Cyberstalking The emergence of social networks such as YouTube, MySpace, and Facebook has allowed millions of individuals to communicate who never (or seldom) meet each other face to face. These same networks have also created new kinds of threats for individuals such as cyberbullying and cyberstalking. Using search engines and the Internet, it is fairly easy to accumulate personal information. It is even easier to plant rumors, make false accusations, and damage the reputations of individuals by broadcasting such information on the Web or by using social networks. Because cyberstalking can be done anonymously, it is hard to trace, although some cyberstalkers have been arrested and charged. As this problem becomes more widespread, states are passing new laws against it, as is the federal government. Defenses against cyberstalking include contacting police or other authorities, plus contacting the stalker's Internet service provider if it is known. While it might be possible to slow down or prevent this crime by using anonymous avatars for all social networks, that more or less defeats the purpose of social networking.

Denial of service This form of cybercrime attempts to stop specific computers, networks, or servers from carrying out normal operations by saturating them with phony messages or data. This is a sophisticated form of attack that requires considerable skill and effort to set

up, and of course considerable skill and effort to prevent or stop. Denial of service (DoS) attacks seemed to start about 2001 with an attack against America Online (AOL) that took about a week to stop. Since then numerous forms of DoS attacks have been developed. A precursor to a denial of service attack may include sending out worms or search robots to infiltrate scores of computers and turn them into *zombies*, which will then unknowingly participate in the attack. This is a complex problem, and the best practice for dealing with it is to have topnotch security experts available and to maintain constant vigilance.

Electromagnetic pulse (EMP) A byproduct of nuclear explosions is a pulse of electromagnetic radiation that is strong enough to damage transistors and other electrical devices. Indeed, such a pulse could shut down almost all electrical devices within perhaps 15 miles. The damage may be so severe that repair of many devices—that is, computers, audio equipment, cell phones, and so on—would be impossible. The electromagnetic pulse effect has led to research in *e-bombs*, or high-altitude bombs that explode perhaps 50 miles up and shut down electrical power and damage equipment for hundreds of square miles, but do not kill people or destroy buildings. Not only nuclear explosions but other forms of detonation can trigger such pulses. While it is possible to shield electronic devices using Faraday cages or surrounding them in metallic layers, this is unfeasible for most civilians. The major military countries such as the United States and Russia have been carrying out active research in e-bombs and probably have them already available. It is also possible that other countries such as North Korea may have such devices. The presence of e-bombs is a considerable threat to the economies of every country, and no doubt the wealthier terrorist organizations would like to gain access to such devices. There are no best practices to defend against this for ordinary citizens.

Electromagnetic radiation Ordinary consumers using home computers probably don't have to worry about loss of data due to electromagnetic radiation, but this is a serious issue for military and classified data centers. While operating, computers radiate various kinds of electromagnetic energy, and some of these can be picked up remotely and deciphered in order to collect information about both applications and data. That information could be extracted from electromagnetic radiation was first discovered in the 1960s. Capturing electromagnetic radiation requires rather specialized equipment and also specialized personnel and software that would be outside the range of day-to-day hackers. Some civilian threats do exist, such as the possibility of capturing electromagnetic radiation to crack "smart cards" when they are being processed. Best practices include physical isolation of equipment behind copper or steel enclosures, and of course constant vigilance and topnotch security experts. Another best practice would be to install

electromagnetic generators in data centers that would be more powerful than computer signals and hence interfere with detection. This approach is similar to jamming to shut down pirate radio stations.

Hacking The word “hack” is older than the computer era and has meaning in many fields, such as golf. However, in this book, *hacking* refers to deliberate attempts to penetrate a computer or software application with the intent to modify how it operates. While some hacking is harmful and malicious, some may be beneficial. Indeed, many security companies and software producers employ hackers who attempt to penetrate software and hardware to find vulnerabilities that can then be fixed. While firewalls, antivirus, and antispyware programs are all good practices, what is probably the best practice is to employ ethical hackers to attempt penetration of key applications and computer systems.

Identity theft Stealing an individual’s identity in order to make purchases, set up credit card accounts, or even to withdraw funds from banks is one of the fastest-growing crimes in human history. A new use of identity theft is to apply for medical benefits. In fact, identity theft of physicians’ identities can even be used to bill Medicare and insurance companies with fraudulent claims. Unfortunately, this crime is far too easy to commit, since it requires only moderate computer skills plus commonly available information such as social security numbers, birth dates, parents’ names, and a few other topics. It is alarming that many identity thefts are carried out by relatives and “friends” of the victims. Also, identity information is being sold and traded by hackers. Almost every computer user receives daily “phishing” e-mails that attempt to trick them into providing their account numbers and other identifying information. As the global economy declines into recession, identity theft will accelerate. The author estimates that at least 15 percent of the U.S. population is at risk. Best practices to avoid identity theft include frequent credit checks, using antivirus and anti-spyware software, and also physical security of credit cards, social security cards, and other physical media.

Keystroke loggers This alarming technology represents one of the most serious threats to home computer users since the industry began. Both hardware and software keystroke logging methods exist, but computer users are more likely to encounter software keystroke logging. Interestingly, keystroke logging also has benign uses in studying user performance. In today’s world, not only keystrokes but also mouse movements and touch-screen movements need to be recorded for the technology to work. The most malicious use of keystroke logging is to intercept passwords and security codes so that bank accounts, medical records, and other proprietary data can be stolen. Not only computers are at risk, but also ATM machines. In fact, this technology could also be used on voting machines; possibly with the effect of influencing elections.

Antispyware programs are somewhat useful, as are other methods such as one-time passwords. This is such a complex problem that the current best practice is to do almost daily research on the issue and look for emerging solutions.

Malware This is a hybrid term that combines one syllable from “malicious” and one syllable from “software.” The term is a generic descriptor for a variety of troublesome security problems including viruses, spyware, Trojans, worms, and so on.

Phishing This phrase is derived from “fishing” and refers to attempts to get computer users to reveal confidential information such as account numbers by having them respond to bogus e-mails that appear to be from banks or other legitimate businesses. A classic example of phishing are e-mails that purport to be from a government executive in Nigeria who is having trouble getting funds out of the country and wants to deposit them in a U.S. account. The e-mails ask the readers to respond by sending back their account information. This early attempt at phishing was so obviously bogus that hardly anyone responded to it, but surprisingly, a few people might have. Unfortunately, modern attempts at phishing are much more sophisticated and are very difficult to detect. The best practice is never to respond to requests for personal or account information that you did not originate. However, newer forms are more sophisticated and can intercept browsers when they attempt to go to popular web sites such as eBay or PayPal. The browser can be redirected to a phony web site that looks just like the real one. Not only do phony web sites exist, but also phony telephone sites. However, as phishing becomes more sophisticated, it is harder to detect. Fortunately credit card companies, banks, and other institutions at risk have formed a nonprofit Anti-Phishing Working Group. For software companies, affiliation with this group would be a best practice. For individuals, verifying by phone and refusing to respond to e-mail requests for personal and account data are best practices. Many browsers such as Firefox and Internet Explorer have anti-phishing *blacklists* of known phishing sites and warn users if they are routed to them. Boundary control, Caja, and languages such as E are also effective against phishing.

Physical security Physical security of data centers, notebook computers, thumb drives, and wireless transmission remains a best practice. Almost every week, articles appear in papers and journals about loss or theft of confidential data when notebook computers are lost or stolen. There are dozens of effective physical security systems, and all of them should be considered. A modern form of physical security involves using fingerprints or retina patterns as passwords for computers and applications.

Piracy Piracy in several forms is a major problem in the modern world. The piracy of actual ships has been a problem off the African

coast. However, software piracy has also increased alarmingly. While China and the Asia Pacific region are well known as sources of piracy, the disputes between Iran and the USA have led Iran to allow unlimited copying of software and intellectual property, which means that the Middle East is also a hotbed of software piracy. In the United States and other countries with strong intellectual property laws, Microsoft and other large software vendors are active in bringing legal charges against pirates. The nonprofit Business Software Alliance even offers rewards for turning in pirates. However, unauthorized copies of software remain a serious problem. For smaller software vendors, the usual precautions include registration and activation of software before it can be utilized. It is interesting that the open-source and freeware communities deal with the problem in rather different ways. For example, open-source softwares commonly use static analysis methods, which can find some security flaws. Also having dozens of developers looking at the code raises the odds that security flaws might be identified.

Rootkits In the Unix operating system, the term *root user* refers to someone having authorization to modify the operating system or the kernel. For Windows, having *administrative rights* is equivalent. Rootkits are programs that infiltrate computers and seize control of the operating system. Once that control is achieved, then the rootkit can be used to launch denial of service attacks, steal information, reformat disk drives, or perform many other kinds of mischief. In 2005, the Sony Corporation deliberately issued a rootkit on music CDs in an attempt to prevent music piracy via peer-to-peer and computer copying. However, an unintended consequence of this rootkit was to open up backdoor access to computers that could be used by hackers, spyware, and viruses. Needless to say, once the Sony rootkit was revealed to the press, the outcry was sufficient for Sony to withdraw the rootkit. Rootkits tend to be subtle and not only slip past some antivirus software, but indeed may attack the antivirus software itself. There seem to be no best practices as of 2009, although some security companies such as Kaspersky and Norton have development methods for finding some rootkits and protecting themselves as well.

Smart card hijacking A very recent threat that has only just started to occur is that of remote-reading of various “smart cards” that contain personal data. These include some new credit cards and also new passports with embedded information. The government is urging citizens to keep such cards in metal containers or at least metal foil, since the data can be accessed from at least 10 feet away. Incidentally, the “EZ Pass” devices that commuters use to go through tolls without stopping are not secure either.

Spam Although the original meaning of *spam* referred to a meat product, the cyberdefinition refers to unwanted ads, e-mails, or instant

messages that contain advertising. Now that the Internet is the world's primary communication medium and reaches perhaps one-fifth of all humans on the planet, using the Internet for ads and marketing is going to continue. The volume of spam is alarming and is estimated at topping 85 percent of all e-mail traffic, which obviously slows down the Internet and slows down many servers as well. Spam is hard to combat because some of it comes from zombie computers that have been hijacked by worms or viruses and then unknowingly used for transmitting spam. Some localities have made spamming illegal, but it is easy for spammers to outsource to some other locality where it is not illegal. Related to spamming is a new subindustry called *e-mail address harvesting*. E-mail addresses can be found by search robots, and once found and created, the lists are sold as commercial products. Another form of address harvesting is from the fine print of the service agreements of social networks, which state that a user's e-mail address may not be kept private (and will probably be sold as a profit-making undertaking). A best practice against spam is to use spyware and spam blockers, but these are not 100 percent effective. Some spam networks can be *de-peered*, or cut off from other networks, but this is technically challenging and may lead to litigation.

Spear phishing The term *spear phishing* refers to a new and very sophisticated form of phishing where a great deal of personal information is included in the phishing e-mail to deceive possible victims. The main difference between phishing and spear phishing is the inclusion of personal information. For example, an e-mail that identifies itself as coming from a friend or colleague is more likely to be trusted than one coming from a random source. Thus, spear phishing is a great deal harder to defend against. Often hackers break into corporate computers and then send spear phishing e-mails to all employees, with disinformation indicating that the e-mail is from accounting, human factors, or some other legitimate organization. In fact, the real name of the manager might also be included. The only best practice for spear phishing is to avoid sending personal or financial information in response to any e-mail. If the e-mail seems legitimate, check by phone before responding. However, spear phishing is not just a computer scam, but also includes phony telephone messages and text messages as well.

Spyware Software that installs itself on a host computer and takes partial control of the operating system and web browser is termed *spyware*. The purpose of spyware is to display unwanted ads, redirect browsers to specific sites, and also to extract personal information that might be used for purposes such as identity theft. Prior to version 7 of Microsoft Internet Explorer, almost any ActiveX program could be downloaded and start executing. This was soon discovered by hackers as

a way to put ads and browser hijackers on computers. Because spyware often embedded itself in the registry, it was difficult to remove. In today's world circa 2009, a combination of firewalls and modern antispyware software can keep most spyware from penetrating computers, and can eliminate most spyware as well. However, in the heated technology race between hackers and protectors, sometimes the hackers pull ahead. Although Macintosh computers have less spyware directed their way than computers running Microsoft Windows do, no computers or operating systems in the modern world are immune to spyware.

Trojans This term is of course derived from the famous Trojan horse. In a software context, a Trojan is something that seems to be useful so that users are deceived into installing it via download or by disk. Once it's installed, some kind of malicious software then begins to take control of the computer or access personal data. One classic form of distributing Trojans involves screensavers. Some beautiful view such as a waterfall or a lagoon is offered as a free download. However, malicious software routines that can cause harm are hidden in the screensaver. Trojans are often involved in denial of service attacks, in identity theft, in keystroke logging, and in many other harmful actions. Modern antivirus software is usually effective against Trojans, so installing, running, and updating such software is a best practice.

Viruses Computer viruses originated in the 1970s and started to become troublesome in the 1980s. As with disease viruses, computer viruses attempt to penetrate a host, reproduce themselves in large numbers, and then leave the original host and enter new hosts. Merely reproducing and spreading can slow networks and cause performance slowdowns, but in addition, some viruses also have functions that deliberately damage computers, steal private information, or perform other malicious acts. For example, viruses can steal address books and then send infected e-mails to every friend and contact of the original host. *Macro* viruses transmitted by documents created using Microsoft Word or Microsoft Excel have been particularly common and particularly troublesome. Viruses spread by instant messaging are also troublesome. Viruses are normally transmitted by attaching themselves to a document, e-mail, or instant message. While antivirus software is generally effective and a best practice, virus developers tend to be active, energetic, and clever. Some newer viruses morph or change themselves spontaneously to avoid antivirus software. These mutating viruses are called *polymorphic* viruses. Although viruses primarily attack Microsoft Windows, all operating systems are at risk, including Linux, Unix, Mac OS, Symbian, and all others. Best practices for avoiding viruses are to install antivirus software and to keep the virus definitions up to date. Taking frequent checkpoints and restore points is also a best practice.

Whaling This is a form of phishing that targets very high-level executives such as company presidents, senior vice presidents, CEOs, CIOs, board members, and so forth. Whaling tends to be very sophisticated. An example might be an e-mail that purports to be from a well-known law firm and that discusses possible litigation against the target or his or her company. Other devices would include “who’s who” e-mail requests, or requests from famous business journals. The only best practice is to avoid responding without checking out the situation by phone or by some other method.

Wireless security leaks In the modern world, usage of wireless computer networks is about as popular as cell phone usage. Many homes have wireless networks as do public buildings. Indeed some towns and cities offer wireless coverage throughout. As wireless communication becomes a standard method for business-to-business and person-to-person communication, it has attracted many hackers, identify thieves, and other forms of cybercriminals. Unprotected wireless networks allow cybercriminals to access and control computers, redirect browsers, and steal private information. Other less overt activities are also harmful. For example, unprotected wireless networks can be used to access porn sites or to send malicious e-mails to third parties without the network owner being aware of it. Because many consumers and computer users are not versed in computer and wireless network issues, probably 75 percent of home computer networks are not protected. Some hackers even drive through large cities looking for unprotected networks (this is called *war driving*). In fact, there may even be special signs and symbols chalked on sidewalks and buildings to indicate unprotected networks. Many networks in coffee shops and hotels are also unprotected. Best practices for avoiding wireless security breaches include using the latest password and protection tools, using encryption, and frequently changing passwords.

Worms Small software applications that reproduce themselves and spread from computer to computer over networks are called *worms*. Worms are similar to viruses, but tend to be self-propagating rather than spreading by means of e-mails or documents. While a few worms are benign (Microsoft once tried to install operating system patches using worms), many are harmful. If worms are successful in reproducing and moving through a network, they use bandwidth and slow down performance. Worse, some worms have *payloads* or subroutines that perform harmful and malicious activities such as erasing files. Worms can also be used to create zombie computers that might take part in denial of service attacks. Best practices for avoiding worms include installing the latest security updates from operating vendors such as Microsoft, using antivirus software (with frequent definition updates), and using firewalls.

As can be seen from the variety of computer and software hazards in the modern world, protection of computers and software from harmful attacks requires constant vigilance. It also requires installation and usage of several kinds of protective software. Finally, both physical security and careless usage of computers by friends and relatives need to be considered. Security problems will become more pervasive as the global economy sinks into recession. Information is one commodity that will increase in value no matter what is happening to the rest of the economy. Moreover, both organized crime and major terrorist groups are now active players in hacking, denial of service, and other forms of cyberwarfare.

If you break down the economics of software security, the distribution of costs is far from optimal in 2009. From partial data, it looks like about 60 percent of annual corporate security costs are spent on defensive measures for data centers and installed software, about 35 percent is spent on recovering from attacks such as denial of service, and only about 5 percent is spent on preventive measures. Assuming an annual cost of \$50 million for security per Fortune 500 company, the breakdown might be \$30 million on defense, \$17.5 million for recovery, and only \$2.5 million on prevention during development of applications.

With more effective prevention in the form of better architecture, design, secure coding practices, boundary controls, and languages such as E, a future cost distribution for security might be prevention, 60 percent; defense, 35 percent; and recovery, 5 percent. With better prevention, the total security costs would be lower: perhaps \$25 million per year instead of \$50 million per year. In this case the prevention costs would be \$15 million; defensive costs would be \$8.75 million; and recovery costs would be only \$1.25 million. Table 2-9 shows the two cost profiles.

So long as software security depends largely upon human beings acting wisely by updating virus definitions and installing antispyware, it cannot be fully successful. What the software industry needs is to design and develop much better preventive methods for building applications and operating systems, and then to fully automate defensive approaches with little or no human intervention being needed.

TABLE 2-9 Estimated Software Security Costs in 2009 and 2019 (Assumes Fortune 500 Company)

	2009	2019	Difference
Prevention	\$2,500,000	\$15,000,000	\$12,500,000
Defense	\$30,000,000	\$8,750,000	-\$21,250,000
Recovery	\$17,500,000	\$1,250,000	-\$16,250,000
TOTAL	\$50,000,000	\$25,000,000	-\$25,000,000

43. Best Practices for Software Deployment and Customization

Between development of software and the start of maintenance is a gray area that is seldom covered by the software literature: deployment and installation of software applications. Considering that the deployment and installation of large software packages such as enterprise resource planning (ERP) tools can take more than 12 calendar months, cost more than \$1 million, and involve more than 25 consultants and 30 in-house personnel, deployment is a topic that needs much more research and much better coverage in the literature.

For most of us who use personal computers or Macintosh computers, installation and deployment are handled via the Web or from a CD or DVD. While some software installations are troublesome (such as Symantec or Microsoft Vista), many can be accomplished in a few minutes.

Unfortunately for large mainframe applications, they don't just load up and start working. Large applications such as ERP packages require extensive customization in order to work with existing applications.

In addition, new releases are frequently buggy, so constant updates and repairs are usually part of the installation process.

Also, large applications with hundreds or even thousands of users need training for different types of users. While vendors may provide some of the training, vendors don't know the specific practices of their clients. So it often happens that companies themselves have to put together more than a dozen custom courses. Fortunately, there are tools and software packages that can help in doing in-house training for large applications.

Because of bugs and learning issues, it is unfeasible just to stop using an old application and to start using a new commercial package. Usually, side-by-side runs occur for several months, both to check for errors in the new package and to get users up to speed as well.

To make a long (and expensive) story short, deployment of a major new software package can run from six months to more than 12 months and involve scores of consultants, educators, and in-house personnel who need to learn the new software. Examples of best practices for deployment include

- Joining user associations for the new application, if any exist
- Interviewing existing customers for deployment advice and counsel
- Finding consultants with experience in deployment
- Acquiring software to create custom courses
- Acquiring training courses for the new application
- Customizing the new application to meet local needs

- Developing interfaces between the new application and legacy applications
- Recording and reporting bugs or defects encountered during deployment
- Installing patches and new releases from the vendor
- Evaluating the success of the new application

Installation and deployment of large software packages are common, but very poorly studied and poorly reported in the software literature. Any activity that can take more than a calendar year, cost more than \$1 million, and involve more than 50 people in full-time work needs careful analysis.

The costs and hazards of deployment appear to be directly related to application size and type. For PC and Macintosh software, deployment is usually fairly straightforward and performed by the customers themselves. However, some companies such as Symantec make it difficult by requiring the prior versions of their applications be removed, but normal Windows removal of it leaves traces that can interfere with the new installation.

Big applications such as mainframe operating systems, ERP packages, and custom software are very troublesome and expensive to deploy. In addition, such applications often require extensive customization for local conditions before they can be utilized. And, of course, this complex situation also requires training users.

44. Best Practices for Training Clients or Users of Software Applications

It is an interesting phenomenon of the software industry that commercial vendors do such a mediocre job of providing training and tutorial information that a major publishing subindustry has come into being providing books on popular software packages such as Vista, Quicken, Microsoft Office, and dozens of other popular applications. Also, training companies offer interactive CD training for dozens of software packages. As this book is written, the best practice for learning to use popular software packages from major vendors is to use third-party sources rather than the materials provided by the vendors themselves.

For more specialized mainframe applications such as those released by Oracle and SAP, other companies also provide supplemental training for both users and maintenance personnel, and usually do a better job than the vendors themselves.

After 60 years of software, it might be thought that standard user-training materials would have common characteristics, but they do not.

What is needed is a sequence of learning material including but not limited to:

- Overview of features and functions
- Installation and startup
- Basic usage for common tasks
- Usage for complex tasks
- HELP information by topic
- Troubleshooting in case of problems
- Frequently asked questions (FAQ)
- Operational information
- Maintenance information

Some years ago, IBM performed a statistical analysis of user evaluations for all software manuals provided to customers with IBM software. Then the top-ranked manuals were distributed to all IBM technical writers with a suggestion that they be used as guidelines for writing new manuals.

It would be possible to do a similar study today of third-party books by performing a statistical analysis of the user reviews listed in the Amazon catalog of technical books. Then the best books of various kinds could serve as models for new books yet to be written.

Because hard-copy material is static and difficult to modify, tutorial material will probably migrate to online copy plus, perhaps, books formatted for e-book readers such as the Amazon Kindle, Sony PR-505, and the like.

It is possible to envision even more sophisticated online training by means of virtual environments, avatars, and 3-D simulations, although these are far in the future as of 2009.

The bottom line is that tutorial materials provided by software vendors are less than adequate for training clients. Fortunately, many commercial book publishers and education companies have noted this and are providing better alternatives, at least for software with high usage.

Over and above vendor and commercial books, user associations and various web sites have volunteers who often can answer questions about software applications. Future trends might include providing user information via e-books such as the Amazon Kindle or Sony PR-505.

45. Best Practices for Customer Support of Software Applications

Customer support of software applications is almost universally unsatisfactory. A few companies such as Apple, Lenovo, and IBM have reasonably

good reviews for customer support, but hundreds of others garner criticism for long wait times and bad information.

Customer support is also labor-intensive and very costly. This is the main reason why it is not very good. On average it takes about one customer support person for every 10,000 function points in a software application. It also takes one customer support person for about every 150 customers. However, as usage goes up, companies cannot afford larger and larger customer support teams, so the ratio of support to customers eventually tops 1000 to 1, which of course means long wait times. Thus, large packages in the 100,000–function point range with 100,000 customers need either enormous support staff, or smaller staffs that trigger very difficult access by customers.

Because of the high costs and labor intensity of customer support, it is one of the most common activities outsourced to countries with low labor costs such as India.

Surprisingly, small companies with only a few hundred customers often have better customer support than large companies, due to the fact that their support teams are not overwhelmed.

A short-range strategy for improving customer support is to improve quality so that software is delivered with fewer bugs. However, not many companies are sophisticated enough to even know how to do this. A combination of inspections, static analysis, and testing can raise defect removal efficiency levels up to perhaps 97 percent from today's averages of less than 85 percent. Releasing software with fewer bugs or defects would yield a significant reduction in the volume of incoming requests for customer support.

The author estimates that reducing delivered bugs by about 220 would reduce customer support staffing by one person. This is based on the assumption that customer support personnel answer about 30 calls per day, and that each bug will be found by about 30 customers. In other words, one bug can occupy one day for a customer support staff member, and there are 220 working days per year.

A more comprehensive long-range strategy would involve many different approaches, including some that are novel and innovative:

- Develop artificial-intelligence virtual support personnel who will serve as the first tier of telephone support. Since live humans are expensive and often poorly trained, virtual personnel could do a much better job. Of course, these avatars would need to be fully stocked with the latest information on bug reports, work-arounds, and major issues.
- Allow easy e-mail contacts between customers and support organizations. For small companies or small applications, these could be screened by live support personnel. For larger applications or those

with millions of customers, some form of artificial-intelligence tool would scan the e-mails and either offer solutions or route them to real personnel for analysis.

- Standardize HELP information and user's guides so that all software applications provide similar data to users. This would speed up learning and allow users to change software packages with minimal disruptions. Doing this would perhaps trigger the development of new standards by the International Standards Organization (ISO), by the IEEE, and by other standards bodies.
- For reusable functions and features, such as those used in service-oriented architecture, provide reusable HELP screens and tutorial information as well as reusable source code. As software switches from custom development to assembly from standard components, the tutorial materials for those standard components must be part of the package of reusable artifacts shared among many applications.

46. Best Practices for Software Warranties and Recalls

Almost every commercial product comes with a warranty that offers repairs or replacement for a period of time if the product should be defective: appliances, automobiles, cameras, computers, optics, and so on. Software is a major exception. Most “software warranties” explicitly disclaim fitness for use, quality, or causing harm to consumers. Most software products explicitly deny warranty protection either “express or implied.”

Some software vendors may offer periodic updates and bug repairs, but if the software should fail to operate or should produce incorrect results, the usual guarantee is merely to provide another copy, which may have the same flaws. Usually, the software cannot be returned and the vendor will not refund the purchase price, much less fix any damage that the software might have caused such as corrupting files or leaving unremoveable traces.

What passes for software warranties are part of *end user license agreements* (EULA), which users are required to acknowledge or sign before installing software applications. These EULA agreements are extremely one-sided and designed primarily to protect the vendors.

The reason for this is the poor quality control of software applications, which has been a major weakness of the industry for more than 50 years.

As this book is being written, the federal government is attempting to draft a Uniform Computer Information Transaction Act (UCITA) as part of the Uniform Commercial Code. UCITA has proven to be very controversial, and some claim it is even weaker in terms of consumer

protection than current EULA practices, if that is possible. Because state governments can make local changes, the UCITA may not even be very uniform.

If software developers introduced the best practices of achieving greater than 95 percent defect removal efficiency levels coupled with building software from certified reusable components, then it would also be possible to create the best practice fair warranties that benefit both parties. Clauses within such warranties might include

- Vendors would make a full refund of purchase price to any dissatisfied customer within a fixed time period such as 30 days.
- Software vendors would guarantee that the software would operate in conformance to the information provided in user guides.
- The vendors would offer free updates and bug repairs for at least a 12-month period after purchase.
- Vendors would guarantee that software delivered on physical media such as CD or DVD disks would be free of viruses and malware.

Over and above specific warranty provisions, other helpful topics would include

- Methods of reporting bugs or defects to the vendor would be included in all user guides and also displayed in HELP screens.
- Customer support would be promptly available by phone with less than three minutes of wait time.
- Responses to e-mail requests for help would occur within 24 business hours of receipt (weekends might be excluded in some cases).

As of 2009, most EULA agreements and most software warranties are professionally embarrassing.

47. Best Practices for Software Change Management After Release

In theory, software change management after release of a software application should be almost identical to change management before the release; that is, specifications would be updated as needed, configuration control would continue, and customer-reported bugs would be added to the overall bug database.

In practice, postrelease change management is often less rigorous than change management prior to the initial release. While configuration control of code might continue, specifications are seldom kept current. Also, small bug repairs and minor enhancements may occur that lack

permanent documentation. As a result, after perhaps five years of usage, the application no longer has a full and complete set of specifications.

Also, code changes may have occurred which triggered islands of “dead code” that is no longer reached. Code comments may be out of date. Complexity as measured using cyclomatic or essential complexity will probably have gone up, so changes tend to become progressively more difficult. This situation is common enough so that for updates, many companies depend primarily on the tenure of long-term maintenance employees, whose knowledge of the structure of aging legacy code is vital for successful updates.

However, legacy software systems do have some powerful tools that can help in bringing out new versions and even in developing replacements. Because the source code does exist in most cases, it is possible to apply automation to the source code and extract hidden business rules and algorithms that can then be carried forward to replacement applications or to renovated legacy applications. Examples of such tools include but are not limited to:

- Complexity analysis tools that can illustrate all paths and branches through code
- Static analysis tools that can find bugs in legacy code, in selected languages
- Static analysis tools that can identify error-prone modules for surgical removal
- Static analysis tools that can identify dead code for removal or isolation
- Data mining tools that can extract algorithms and business rules from code
- Code conversion tools that can convert legacy languages into Java or modern languages
- Function point enumeration tools that can calculate the sizes of legacy applications
- Renovation workbenches that can assist in handling changes to existing software
- Automated testing tools that can create new test cases after examining code segments
- Test coverage tools that can show gaps and omissions from current test case libraries

In addition to automated tools, formal inspection of source code, test libraries, and other artifacts of legacy applications can be helpful, too, assuming the artifacts have been kept current.

As the global economy continues to sink into a serious recession, keeping legacy applications running for several more years may prove to have significant economic value. However, normal maintenance and enhancement of poorly structured legacy applications with marginal quality is not cost-effective. What is needed is a thorough analysis of the structure and features of legacy applications. Since manual methods are likely to be ineffective and costly, automated tools such as static analysis and data mining should prove to be valuable allies during the next few years of the recession cycle.

48. Best Practices for Software Maintenance and Enhancement

Software maintenance is more difficult and complex to analyze than software development because the word “maintenance” includes so many different kinds of activities. Also, estimating maintenance and enhancement work requires evaluation not only of the changes themselves, but also detailed and complete analysis of the structure and code of the legacy application that is being modified.

As of 2009, some 23 different forms of work are subsumed under the single word “maintenance.”

Major Kinds of Work Performed Under the Generic Term “Maintenance”

1. Major enhancements (new features of greater than 50 function points)
2. Minor enhancements (new features of less than 5 function points)
3. Maintenance (repairing defects for good will)
4. Warranty repairs (repairing defects under formal contract)
5. Customer support (responding to client phone calls or problem reports)
6. Error-prone module removal (eliminating very troublesome code segments)
7. Mandatory changes (required or statutory changes)
8. Complexity or structural analysis (charting control flow plus complexity metrics)
9. Code restructuring (reducing cyclomatic and essential complexity)
10. Optimization (increasing performance or throughput)
11. Migration (moving software from one platform to another)
12. Conversion (changing the interface or file structure)

13. Reverse engineering (extracting latent design information from code)
14. Reengineering/renovation (transforming legacy application to modern forms)
15. Dead code removal (removing segments no longer utilized)
16. Dormant application elimination (archiving unused software)
17. Nationalization (modifying software for international use)
18. Mass updates such as Euro or Year 2000 repairs
19. Refactoring, or reprogramming, applications to improve clarity
20. Retirement (withdrawing an application from active service)
21. Field service (sending maintenance members to client locations)
22. Reporting bugs or defects to software vendors
23. Installing updates received from software vendors

Although the 23 maintenance topics are different in many respects, they all have one common feature that makes a group discussion possible: they all involve modifying an existing application rather than starting from scratch with a new application.

Each of the 23 forms of modifying existing applications has a different reason for being carried out. However, it often happens that several of them take place concurrently. For example, enhancements and defect repairs are very common in the same release of an evolving application. There are also common sequences or patterns to these modification activities. For example, reverse engineering often precedes reengineering, and the two occur so often together as to almost constitute a linked set. For releases of large applications and major systems, the author has observed from six to ten forms of maintenance all leading up to the same release!

In recent years the Information Technology Infrastructure Library (ITIL) has begun to focus on many key issues that are associated with maintenance, such as change management, reliability, availability, and other topics that are significant for applications in daily use by many customers.

Because aging software applications increase in complexity over time, it is necessary to perform some form of renovation or *refactoring* from time to time. As of 2009, the overall set of best practices for aging legacy applications includes the following:

- Use maintenance specialists rather than developers.
- Consider maintenance outsourcing to specialized maintenance companies.
- Use maintenance renovation workbenches.

- Use formal change management procedures.
- Use formal change management tools.
- Use formal regression test libraries.
- Perform automated complexity analysis studies of legacy applications.
- Search out and eliminate all error-prone modules in legacy applications.
- Identify all dead code in legacy applications.
- Renovate or refactor applications prior to major enhancements.
- Use formal design and code inspections on major updates.
- Track all customer-reported defects.
- Track response time from submission to repair of defects.
- Track response time from submission to completion of change requests.
- Track all maintenance activities and costs.
- Track warranty costs for commercial software.
- Track availability of software to customers.

Because the effort and costs associated with maintenance and enhancement of aging software are now the dominant expense of the entire software industry, it is important to use state-of-the-art methods and tools for dealing with legacy applications.

Improved quality before delivery can cut maintenance costs. Since maintenance programmers typically fix about 10 bugs per calendar month, every reduction in delivered defects of about 120 could reduce maintenance staffing by one person. Therefore combinations of defect prevention, inspections, static analysis, and better testing can reduce maintenance costs. This is an important consideration in a world facing a serious recession as we are in 2009.

Some of the newer approaches circa 2009 include maintenance or renovation workbenches, such as the tools offered by Relativity Technologies. This workbench also has a new feature that performs function point analysis with high speed and good precision. Renovation prior to major enhancements should be a routine activity.

Since many legacy applications contain error-prone modules that are high in complexity and receive a disproportionate share of defect reports, it is necessary to take corrective actions before proceeding with significant changes. As a rule of thumb, less than 5 percent of the modules in large systems will receive more than 50 percent of defect reports. It is usually impossible to fix such modules, so once they are identified, surgical removal followed by replacement is the normal therapy.

As of 2009, maintenance outsourcing has become one of the most popular forms of software outsourcing. In general, maintenance outsource agreements have been more successful than development outsource agreements and seem to have fewer instances of failure and litigation. This is due in part to the sophistication of the maintenance outsource companies and in part to the fact that existing software is not prone to many of the forms of catastrophic risk that are troublesome for large development projects.

Both maintenance and development share a need for using good project management practices, effective estimating methods, and very careful measurement of productivity and quality. While development outsourcing ends up in litigation in about 5 percent of contracts, maintenance outsourcing seems to have fewer issues and to be less contentious. As the economy moves into recession, maintenance outsourcing may offer attractive economic advantages.

49. Best Practices for Updates and Releases of Software Applications

Once software applications are installed and being used, three things will happen: (1) bugs will be found that must be fixed; (2) new features will be added in response to business needs and changes in laws and regulations; and (3) software vendors will want to make money either by bringing out new versions of software packages or by adding new features for a fee. This part of software engineering is not well covered by the literature. Many bad practices have sprung up that are harmful to customers and users. Some of these bad practices include

- Long wait times for customer support by telephone.
- Telephone support that can't be used by customers who have hearing problems.
- No customer support by e-mail, or very limited support (such as Microsoft).
- Incompetent customer support when finally reached.
- Charging fees for customer support, even for reporting bugs.
- Inadequate methods of reporting bugs to vendors (such as Microsoft).
- Poor response times to bugs that are reported.
- Inadequate repairs of bugs that are reported.
- Stopping support of older versions of software prematurely.
- Forcing customers to buy new versions.
- Changing file formats of new versions for arbitrary reasons.

- Refusing to allow customers to continue using old versions.
- Warranties that cover only replacement of media such as disks.
- One-sided agreements that favor only the vendor.
- Quirky new releases that can't be installed over old releases.
- Quirky new releases that drop useful features of former releases.
- Quirky new releases that don't work well with competitive software.

These practices are so common that it is not easy to even find companies that do customer support and new releases well, although there are a few. Therefore the following best practices are more theoretical than real as of 2009:

- Ideally, known bugs and problems for applications should be displayed on a software vendor's web site.
- Bug reports and requests for assistance should be easily handled by e-mail. Once reported, responses should be returned within 48 hours.
- Reaching customer support by telephone should not take more than 5 minutes.
- When customer support is reached by phone, at least 60 percent of problems should be resolved by the first tier of support personnel.
- Reaching customer support for those with hearing impairments should be possible.
- Fee-based customer support should exclude bug reports and problems caused by vendors.
- Bug repairs should be self-installing when delivered to clients.
- New versions and new features should not require manual uninstalls of prior versions.
- When file formats are changed, conversion to and from older formats should be provided free of charge by vendors.
- Support of applications with thousands of users should not be arbitrarily withdrawn.
- Users should not be forced to buy new versions annually unless they wish to gain access to the new features.

In general, mainframe vendors of expensive software packages (greater than \$100,000) are better at customer support than are the low-end, high-volume vendors of personal computer and Macintosh packages. However, poor customer support, inept customer support, sluggish bug repairs, and forced migration to new products or releases of questionable value remain endemic problems of the software industry.

50. Best Practices for Terminating or Withdrawing Legacy Applications

Large software applications tend to have surprisingly long life expectancies. As of 2009, some large systems such as the U.S. air traffic control system have been in continuous usage for more than 30 years. Many large internal applications in major companies have been in use more than 20 years.

Commercial applications tend to have shorter life expectancies than information systems or systems software, since vendors bring out new releases and stop supporting old releases after a period of years. Microsoft, Intuit, and Symantec, for example, are notorious for withdrawing support for past versions of software even if they still have millions of users and are more stable than the newer versions.

Intuit, for example, deliberately stops support for old versions of Quicken after a few years. Microsoft is about to stop support for Windows XP even though Vista is still somewhat unstable and unpopular. Even worse, Symantec, Intuit, and Microsoft tend to change file formats so that records produced on new versions can't be used on old versions. Repeated customer outrage finally got the attention of Microsoft, so that they usually provide some kind of conversion method. Intuit and Symantec are not yet at that point.

Nonetheless at some point aging legacy applications will need replacement. Sometimes the hardware on which they operate will need replacement, too.

For small PC and Macintosh applications, replacement is a minor inconvenience and a noticeable but not unbearable expense. However, for massive mainframe software or heavy-duty systems software in the 10,000–function point range, replacement can be troublesome and expensive.

If the software is custom-built and has unique features, replacement will probably require development of a new application with all of the original features, plus whatever new features appear to be useful. The patient-record system of the Veterans Administration is an example of an aging legacy system that has no viable commercial replacements. An additional difficulty with retiring or replacing legacy systems is that often the programming languages are “dead” and no longer have working compilers or interpreters, to say nothing of having very few programmers available.

Best practices for retiring aging systems (assuming they still are in use) include the following:

- Mine the application to extract business rules and algorithms needed for a new version.
- Survey all users to determine the importance of the application to business operations.

- Do extensive searches for similar applications via the Web or with consultants.
- Attempt to stabilize the legacy application so that it stays useful as the new one is being built.
- Consider whether service-oriented architecture (SOA) may be suitable.
- Look for certified sources of reusable material.
- Consider the possibility of automated language conversion.
- Utilize static analysis tools if the language(s) are suitable.

Make no mistake, unless an application has zero users, replacement and withdrawal are likely to cause trouble.

Although outside the scope of this book, it is significant that the life expectancies of all forms of storage are finite. Neither magnetic disks nor solid-state devices are likely to remain in fully operational mode for more than about 25 years.

Summary and Conclusions

The most obvious conclusions are six:

First, software is not a “one size fits all” occupation. Multiple practices and methods are needed.

Second, poor measurement practices and a lack of solid quantified data have made evaluating practices difficult. Fortunately, this situation is improving now that benchmark data is readily available.

Third, given the failure rates and number of cost and schedule overruns, normal development of software is not economically sustainable. Switching from custom development to construction using certified reusable components is needed to improve software economics.

Fourth, effective quality control is a necessary precursor that must be accomplished before software reuse can be effective. Combinations of defect prevention method, inspections, static analysis, testing, and quality assurance are needed.

Fifth, as security threats against software increase in numbers and severity, fundamental changes are needed in software architecture, design, coding practices, and defensive methods.

Sixth, large software applications last for 25 years or more. Methods and practices must support not only development, but also deployment and many years of maintenance and enhancements.

Readings and References

Chapter 2 is an overview of many different topics. Rather than provide a conventional reference list, it seems more useful to show some of the key

books and articles available that deal with the major topics discussed in the chapter.

Project Management, Planning, Estimating, Risk, and Value Analysis

- Boehm, Barry Dr. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.
- Booch Grady. *Object Solutions: Managing the Object-Oriented Project*. Reading, MA: Addison Wesley, 1995.
- Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1974, rev. 1995.
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Charette, Bob. *Application Strategies for Risk Management*. New York: McGraw-Hill, 1990.
- Chrissies, Mary Beth; Konrad, Mike; Shrum, Sandy; CMMI®: Guidelines for Product Integration and Process Improvement; Second Edition; Addison Wesley, Reading, MA; 2006; 704 pages.
- Cohn, Mike. *Agile Estimating and Planning*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005.
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- Ewusi-Mensah, Kwaku. *Software Development Failures* Cambridge, MA: MIT Press, 2003.
- Galarath, Dan. *Software Sizing, Estimating, and Risk Management: When Performance Is Measured Performance Improves*. Philadelphia: Auerbach Publishing, 2006.
- Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Harris, Michael, David Herron, and Stasia Iwanicki. *The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results*. Boca Raton, FL: CRC Press (Auerbach), 2008.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989.
- Johnson, James, et al. *The Chaos Report*. West Yarmouth, MA: The Standish Group, 2000.
- Jones, Capers. *Assessment and Control of Software Risks*.: Prentice Hall, 1994.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007.
- Jones, Capers. "Estimating and Measuring Object-Oriented Software." *American Programmer*, 1994.
- Jones, Capers. *Patterns of Software System Failure and Success*. Boston: International Thomson Computer Press, December 1995.
- Jones, Capers. *Program Quality and Programmer Productivity*. IBM Technical Report TR 02.764. San Jose, CA: January 1977.
- Jones, Capers. *Programming Productivity*. New York: McGraw-Hill, 1986.
- Jones, Capers. "Why Flawed Software Projects are not Cancelled in Time." *Cutter IT Journal*, Vol. 10, No. 12 (December 2003): 12–17.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000.
- Jones, Capers. "Software Project Management Practices: Failure Versus Success." *Crosstalk*, Vol. 19, No. 6 (June 2006): 4–8.
- Laird, Linda M. and Carol M. Brennan. *Software Measurement and Estimation: A Practical Approach*. Hoboken, NJ: John Wiley & Sons, 2006.
- McConnell, Steve. *Software Estimating: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Park, Robert E., et al. *Software Cost and Schedule Estimating - A Process Improvement Initiative*. Technical Report CMU/SEI 94-SR-03. Pittsburgh, PA: Software Engineering Institute, May 1994.
- Park, Robert E., et al. *Checklists and Criteria for Evaluating the Costs and Schedule Estimating Capabilities of Software Organizations* Technical Report CMU/SEI

- 95-SR-005. Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon Univ., January 1995.
- Roetzheim, William H. and Reyna A. Beasley. *Best Practices in Software Cost and Schedule Estimation*. Saddle River, NJ: Prentice Hall PTR, 1998.
- Strassmann, Paul. *Governance of Information Management: The Concept of an Information Constitution*, Second Edition. (eBook) Stamford, CT: Information Economics Press, 2004.
- Strassmann, Paul. *Information Productivity*. Stamford, CT: Information Economics Press, 1999.
- Strassmann, Paul. *Information Payoff*. Stamford, CT: Information Economics Press, 1985.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.
- Stukes, Sherry, Jason Deshoretz, Henry Apgar, and Ilona Macias. *Air Force Cost Analysis Agency Software Estimating Model Analysis*. TR-9545/008-2 Contract F04701-95-D-0003, Task 008. Management Consulting & Research, Inc., Thousand Oaks, CA 91362. September 30, 1996.
- Symons, Charles R. *Software Sizing and Estimating—Mk II FPA (Function Point Analysis)*. Chichester, UK: John Wiley & Sons, 1991.
- Wellman, Frank. *Software Costing: An Objective Approach to Estimating and Controlling the Cost of Computer Software*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Whitehead, Richard. *Leading a Development Team*. Boston: Addison Wesley, 2001.
- Yourdon, Ed. *Death March - The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.
- Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

Measurements and Metrics

- Abran, Alain and Reiner R. Dumke. *Innovations in Software Measurement*. Aachen, Germany: Shaker-Verlag, 2005.
- Abran, Alain, Manfred Bundschuh, Reiner Dumke, Christof Ebert, and Horst Zuse. ["article title"?] *Software Measurement News*, Vol. 13, No. 2 (Oct. 2008). (periodical).
- Bundschuh, Manfred and Carol Dekkers. *The IT Measurement Compendium*. Berlin: Springer-Verlag, 2008.
- Chidamber, S. R. and C. F. Kemerer. "A Metrics Suite for Object-Oriented Design," *IEEE Trans. On Software Engineering*, Vol. SE20, No. 6 (June 1994): 476–493.
- Dumke, Reiner, Rene Braungarten, Günter Büren, Alain Abran, Juan J. Cuadrado-Gallego, (editors). *Software Process and Product Measurement*. Berlin: Springer-Verlag, 2008.
- Ebert, Christof and Reiner Dumke. *Software Measurement: Establish, Extract, Evaluate, Execute*. Berlin: Springer-Verlag, 2007.
- Garmus, David & David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Garmus, David and David Herron. *Function Point Analysis – Measurement Practices for Successful Software Projects*. Boston: Addison Wesley Longman, 2001.
- International Function Point Users Group. *IFPUG Counting Practices Manual*, Release 4. Westerville, OH: April 1995.
- International Function Point Users Group (IFPUG). *IT Measurement – Practical Advice from the Experts*. Boston: Addison Wesley Longman, 2002.
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York: McGraw-Hill, 2008.
- Jones, Capers. "Sizing Up Software." *Scientific American Magazine*, Vol. 279, No. 6 (December 1998): 104–111.
- Jones Capers. *A Short History of the Lines of Code Metric*, Version 4.0. (monograph) Narragansett, RI: Capers Jones & Associates LLC, May 2008.
- Kemerer, C. F. "Reliability of Function Point Measurement – A Field Experiment." *Communications of the ACM*, Vol. 36, 1993: 85–97.
- Parthasarathy, M. A. *Practical Software Estimation – Function Point Metrics for Insourced and Outsourced Projects*. Upper Saddle River, NJ: Infosys Press, Addison Wesley, 2007.

- Putnam, Lawrence H. *Measures for Excellence -- Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Yourdon Press – Prentice Hall, 1992.
- Putnam, Lawrence H. and Ware Myers. *Industrial Strength Software – Effective Management Using Measurement*. Los Alamitos, CA: IEEE Press, 1997.
- Stein, Timothy R. *The Computer System Risk Management Book and Validation Life Cycle*. Chico, CA: Paton Press, 2006.
- Stutzke, Richard D. *Estimating Software-Intensive Systems*. Upper Saddle River, NJ: Addison Wesley, 2005.

Architecture, Requirements, and Design

- Ambler, S. *Process Patterns – Building Large-Scale Systems Using Object Technology*. Cambridge University Press, SIGS Books, 1998.
- Artow, J. and I. Neustadt. *UML and the Unified Process*. Boston: Addison Wesley, 2000.
- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Boston: Addison Wesley, 1997.
- Berger, Arnold S. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CMP Books, 2001.
- Booch, Grady, Ivar Jacobsen, and James Rumbaugh. *The Unified Modeling Language User Guide*, Second Edition. Boston: Addison Wesley, 2005.
- Cohn, Mike. *User Stories Applied: For Agile Software Development*. Boston: Addison Wesley, 2004.
- Fernandini, Patricia L. *A Requirements Pattern Succeeding in the Internet Economy*. Boston: Addison Wesley, 2002.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Design*. Boston: Addison Wesley, 1995.
- Inmon William H., John Zachman, and Jonathan G. Geiger. *Data Stores, Data Warehousing, and the Zachman Framework*. New York: McGraw-Hill, 1997.
- Marks, Eric and Michael Bell. *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology*. New York: John Wiley & Sons, 2006.
- Martin, James & Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice Hall, 1985.
- Orr, Ken. *Structured Requirements Definition*. Topeka, KS: Ken Orr and Associates, Inc, 1981.
- Robertson, Suzanne and James Robertson. *Mastering the Requirements Process*, Second Edition. Boston: Addison Wesley, 2006.
- Warnier, Jean-Dominique. *Logical Construction of Systems*. London: Van Nostrand Reinhold.
- Wiegiers, Karl E. *Software Requirements*, Second Edition. Bellevue, WA: Microsoft Press, 2003.

Software Quality Control

- Beck, Kent. *Test-Driven Development*. Boston: Addison Wesley, 2002.
- Chelf, Ben and Raoul Jetley. “Diagnosing Medical Device Software Defects Using Static Analysis.” Coverity Technical Report. San Francisco: 2008.
- Chess, Brian and Jacob West. *Secure Programming with Static Analysis*. Boston: Addison Wesley, 2007.
- Cohen, Lou. *Quality Function Deployment – How to Make QFD Work for You*. Upper Saddle River, NJ: Prentice Hall, 1995.
- Crosby, Philip B. *Quality is Free*. New York: New American Library, Mentor Books, 1979.
- Everett, Gerald D. and Raymond McLeod. *Software Testing*. Hoboken, NJ: John Wiley & Sons, 2007.
- Gack, Gary. *Applying Six Sigma to Software Implementation Projects*. <http://software.isixsigma.com/library/content/c040915b.asp>.
- Gilb, Tom and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.

- Hallowell, David L. *Six Sigma Software Metrics, Part 1*. <http://software.isixsigma.com/library/content/03910a.asp>.
- International Organization for Standards. "ISO 9000 / ISO 14000." <http://www.iso.org/iso/en/iso9000-14000/index.html>.
- Jones, Capers. *Software Quality – Analysis and Guidelines for Success*. Boston: International Thomson Computer Press, 1997.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston: Addison Wesley Longman, 2003.
- Land, Susan K., Douglas B. Smith, John Z. Walz. *Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards*. Wiley-Blackwell, 2008.
- Mosley, Daniel J. *The Handbook of MIS Application Software Testing*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1993.
- Myers, Glenford. *The Art of Software Testing*. New York: John Wiley & Sons, 1979.
- Nandyal Raghav. *Making Sense of Software Quality Assurance*. New Delhi: Tata McGraw-Hill Publishing, 2007.
- Radice, Ronald A. *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing, 2002.
- Wieggers, Karl E. *Peer Reviews in Software – A Practical Guide*. Boston: Addison Wesley Longman, 2002.

Software Security, Hacking, and Malware Prevention

- Acochido, Byron and John Swartz. *Zero Day Threat: The Shocking Truth of How Banks and Credit Bureaus Help Cyber Crooks Steal Your Money and Identity*. Union Square Press, 2008.
- Allen, Julia, Sean Barnum, Robert Ellison, Gary McGraw, and Nancy Mead. *Software Security: A Guide for Project Managers*. (An SEI book sponsored by the Department of Homeland Security) Boston: Addison Wesley Professional, 2008.
- Anley, Chris, John Heasman, Felix Lindner, and Gerardo Richarte. *The Shellcoders Handbook: Discovering and Exploiting Security Holes*. New York: Wiley, 2007.
- Chess, Brian. *Secure Programming with Static Analysis*. Boston: Addison Wesley Professional, 2007.
- Dowd, Mark, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Boston: Addison Wesley Professional, 2006.
- Ericson, John. *Hacking: The Art of Exploitation*, Second Edition. No Starch Press, 2008.
- Gallager, Tom, Lawrence Landauer, and Brian Jeffries. *Hunting Security Bugs*. Redmond, WA: Microsoft Press, 2006.
- Hamer-Hodges, Ken. *Authorization Oriented Architecture – Open Application Networking and Security in the 21st Century*. Philadelphia: Auerbach Publications, to be published in December 2009.
- Hogland, Greg and Gary McGraw. *Exploiting Software: How to Break Code*. Boston: Addison Wesley Professional, 2004.
- Hogland, Greg and Jamie Butler. *Rootkits: Exploiting the Windows Kernel*. Boston: Addison Wesley Professional, 2005.
- Howard, Michael and Steve Lippner. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006.
- Howard, Michael and David LeBlanc. *Writing Secure Code*. Redmond, WA: Microsoft Press, 2003.
- Jones, Andy and Debi Ashenden. *Risk Management for Computer Security: Protecting Your Network and Information Assets*. Butterworth-Heinemann, 2005.
- Landoll, Douglas J. *The Security Risk Assessment Handbook: A Complete Guide for Performing Security Risk Assessments*. Boca Raton, FL: CRC Press (Auerbach), 2005.
- McGraw, Gary. *Software Security – Building Security In*. Boston: Addison Wesley Professional, 2006.

- Rice, David. *Geekonomics: The Real Cost of Insecure Software*. Boston: Addison Wesley Professional, 2007.
- Scambray, Joel. *Hacking Exposed Windows: Microsoft Windows Security Secrets and Solutions*, Third Edition. New York: McGraw-Hill, 2007.
- . *Hacking Exposed Web Applications*, Second Edition. New York: McGraw-Hill, 2006.
- Sherwood, John, Andrew Clark, and David Lynas. *Enterprise Security Architecture: A Business-Driven Approach*. CMP, 2005.
- Shostack, Adam and Andrews Stewart. *The New School of Information Security*. Boston: Addison Wesley Professional, 2008.
- Skudis, Edward and Tom Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses*. Englewood Cliffs, NJ: Prentice Hall PTR, 2006.
- Skudis, Edward and Lenny Zeltzer. *Malware: Fighting Malicious Code*. Englewood Cliffs, NJ: Prentice Hall PTR, 2003.
- Stuttard, Dafydd and Marcus Pinto. *The Web Application Hackers Handbook: Discovering and Exploiting Security Flaws*. New York: Wiley, 2007.
- Szor, Peter. *The Art of Computer Virus Research and Defense*. Boston: Addison Wesley Professional, 2005.
- Thompson, Herbert and Scott Chase. *The Software Vulnerability Guide*. Boston: Charles River Media, 2005.
- Viega, John and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston: Addison Wesley Professional, 2001.
- Whittaker, James A. and Herbert H. Thompson. *How to Break Software Security*. Boston: Addison Wesley Professional, 2003.
- Wysopal, Chris, Lucas Nelson, Dino Dai Zovi, and Elfriede Dustin. *The Art of Software Security Testing: Identifying Software Security Flaws*. Boston: Addison Wesley Professional, 2006.

Software Engineering and Programming

- Barr, Michael and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, 2006.
- Beck, K. *Extreme Programming Explained: Embrace Change*. Boston: Addison Wesley, 1999.
- Bott, Frank, A. Coleman, J. Eaton, and D. Roland. *Professional Issues in Software Engineering*. Taylor & Francis, 2000.
- Glass, Robert L. *Facts and Fallacies of Software Engineering (Agile Software Development)*. Boston: Addison Wesley, 2002.
- Hans, Professor van Vliet. *Software Engineering Principles and Practices*, Third Edition. London, New York: John Wiley & Sons, 2008.
- Hunt, Andrew and David Thomas. *The Pragmatic Programmer*. Boston: Addison Wesley, 1999.
- Jeffries, R., et al. *Extreme Programming Installed*. Boston: Addison Wesley, 2001.
- Marciniak, John J. (Editor). *Encyclopedia of Software Engineering* (two volumes). New York: John Wiley & Sons, 1994.
- McConnell, Steve. *Code Complete*. Redmond, WA: Microsoft Press, 1993.
- Morrison, J. Paul. *Flow-Based Programming: A New Approach to Application Development*. New York: Van Nostrand Reinhold, 1994.
- Pressman, Roger. *Software Engineering – A Practitioner's Approach*, Sixth Edition. New York: McGraw-Hill, 2005.
- Sommerville, Ian. *Software Engineering*, Seventh Edition. Boston: Addison Wesley, 2004.
- Stephens M. and D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Berkeley, CA: Apress L.P., 2003.

Software Development Methods

- Boehm, Barry. "A Spiral Model of Software Development and Enhancement." Proceedings of the Int. Workshop on Software Process and Software Environments. *ACM Software Engineering Notes* (Aug. 1986): 22–42.

- Cockburn, Alistair. *Agile Software Development*. Boston: Addison Wesley, 2001.
- Cohen, D., M. Lindvall, and P. Costa. "An Introduction to agile methods." *Advances in Computers*, New York: Elsevier Science, 2004.
- Highsmith, Jim. *Agile Software Development Ecosystems*. Boston: Addison Wesley, 2002.
- Humphrey, Watts. *TSP – Leading a Development Team*. Boston: Addison Wesley, 2006.
- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Krutchen, Phillippe. *The Rational Unified Process – An Introduction*. Boston: Addison Wesley, 2003.
- Larman, Craig and Victor Basili. "Iterative and Incremental Development – A Brief History." *IEEE Computer Society* (June 2003): 47–55.
- Love, Tom. *Object Lessons*. New York: SIGS Books, 1993.
- Martin, Robert. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall, 2002.
- Mills, H., M. Dyer, and R. Linger. "Cleanroom Software Engineering." *IEEE Software*, 4, 5 (Sept. 1987): 19–25.
- Paulk Mark, et al. *The Capability Maturity Model Guidelines for Improving the Software Process*. Reading, MA: Addison Wesley, 1995.
- Rapid Application Development. http://en.wikipedia.org/wiki/Rapid_application_development.
- Stapleton, J. *DSDM – Dynamic System Development Method in Practice*. Boston: Addison Wesley, 1997.

Software Deployment, Customer Support, and Maintenance

- Arnold, Robert S. *Software Reengineering*. Los Alamitos, CA: IEEE Computer Society Press, 1993.
- Arthur, Lowell Jay. *Software Evolution – The Software Maintenance Challenge*. New York: John Wiley & Sons, 1988.
- Gallagher, R. S. *Effective Customer Support*. Boston: International Thomson Computer Press, 1997.
- Parikh, Girish. *Handbook of Software Maintenance*. New York: John Wiley & Sons, 1986.
- Pigoski, Thomas M. *Practical Software Maintenance – Best Practices for Managing Your Software Investment*. Los Alamitos, CA: IEEE Computer Society Press, 1997.
- Sharon, David. *Managing Systems in Transition – A Pragmatic View of Reengineering Methods*. Boston: International Thomson Computer Press, 1996.
- Takang, Armstrong and Penny Grubb. *Software Maintenance Concepts and Practice*. Boston: International Thomson Computer Press, 1997.
- Ulrich, William M. *Legacy Systems: Transformation Strategies*. Upper Saddle River, NJ: Prentice Hall, 2002.

Social Issues in Software Engineering

- Brooks, Fred. *The Mythical Manmonth*, Second Edition. Boston: Addison Wesley, 1995.
- DeMarco, Tom. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1999.
- Glass, Robert L. *Software Creativity*, Second Edition. Atlanta, GA: developer.*books, 2006.
- Humphrey, Watts. *Winning with Software: An Executive Strategy*. Boston: Addison Wesley, 2002.
- Johnson, James, et al. *The Chaos Report*. West Yarmouth, MA: The Standish Group, 2007.
- Jones, Capers. "How Software Personnel Learn New Skills," Sixth Edition (monograph). Narragansett, RI: Capers Jones & Associates LLC, July 2008.

- Jones, Capers. "Conflict and Litigation Between Software Clients and Developers" (monograph). Narragansett, RI: Software Productivity Research, Inc., 2008.
- Jones, Capers. "Preventing Software Failure: Problems Noted in Breach of Contract Litigation." Narragansett, RI: Capers Jones & Associates LLC, 2008.
- Krasner, Herb. "Accumulating the Body of Evidence for the Payoff of Software Process Improvement – 1997" Austin, TX: Krasner Consulting.
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. University of Chicago Press, 1996.
- Starr, Paul. *The Social Transformation of American Medicine*.: Basic Books Perseus Group, 1982.
- Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
- Weinberg, Gerald M. *Becoming a Technical Leader*. New York: Dorset House, 1986.
- Yourdon, Ed. *Death March – The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.
- Zoellick, Bill. *CyberRegs – A Business Guide to Web Property, Privacy, and Patents*. Boston: Addison Wesley, 2002.

Web Sites

There are hundreds of software industry and professional associations. Most have a narrow focus. Most are more or less isolated and have no contact with similar associations. Exceptions to this rule include the various software process improvement network (SPIN) groups and the various software metrics associations.

This partial listing of software organizations and web sites is to facilitate communication and sharing of data across both organization and national boundaries. Software is a global industry. Problems occur from the first day of requirements to the last day of usage, and every day in between. Therefore mutual cooperation across industry and technical boundaries would benefit software and help it toward becoming a true profession rather than a craft of marginal competence.

What might be useful for the software industry would be reciprocal memberships among the major professional associations along the lines of the American Medical Association. There is a need for an umbrella organization that deals with all aspects of software as a profession, as does the AMA for medical practice.

American Electronics Association (AEA) www.aeanet.org (may merge with ITAA)

American Society for Quality www.ASQ.org

Anti-Phishing Working Group www.antiphishing.org

Association for Software Testing www.associationforsoftwaretesting.org

Association of Computing Machinery www.ACM.org

Association of Competitive Technologies (ACT) www.actonline.org

Association of Information Technology Professionals www.aitp.org

Brazilian Function Point Users Group www.BFPUG.org

Business Application Software Developers Association www.basda.org

Business Software Alliance (BSA) www.bsa.org

Center for Internet Security www.cisecurity.org

Center for Hybrid and Embedded Software Systems (CHESS) <http://chess.eecs.berkeley.edu>

China Software Industry Association www.CSIA.org
Chinese Software Professional Association www.CSPA.com
Computing Technology Industry Association (CTIA) www.comptia.org
Embedded Software Association (ESA) www.esofta.com
European Design and Automation Association (EDAA) www.edaa.com
Finnish Software Measurement Association www.fisma.fi
IEEE Computer Society www.computer.org
Independent Computer Consultants Association (ICCA) www.icca.org
Information Technology Association of America (ITAA) www.itaa.org (may merge with AEA)
Information Technology Metrics and Productivity Institute (ITMPI)
www.ITMPI.org
InfraGuard www.InfraGuard.net
Institute for International Research (IIR) eee.irusa.com
Institute of Electrical and Electronics Engineers (IEEE) www.IEEE.org
International Association of Software Architects www.IASAHOME.org
International Function Point Users Group (IFPUG) www.IFPUG.org
International Institute of Business Analysis www.IIBAorg
International Software Benchmarking Standards Group (ISBSG) www.ISBSG.org
Japan Function Point Users Group www.jfpug.org
Linux Professional Institute www.lpi.org
National Association of Software and Service Companies (India)
www.NASCOM.in
Netherlands Software Metrics Association www.NESMA.org
Process Fusion www.process-fusion.com
Programmers' Guild www.programmersguild.org
Project Management Institute www.PMI.org
Russian Software Development Organization (RUSSOFT) www.russoft.org
Society of Information Management (SIM) www.simnet.org
Software and Information Industry Association www.siiia.net
Software Engineering Body of Knowledge www.swebok.org
Software Engineering Institute (SEI) www.SEI.org
Software Productivity Research (SPR) www.SPR.com
Software Publishers Association (SPA) www.spa.org
United Kingdom Software Metrics Association www.UKSMA.org
U.S. Internet Industry Association (USIIA) www.usiia.org
Women in Technology International www.witi.com

This page intentionally left blank

A Preview of Software Development and Maintenance in 2049

Introduction

From the 1960s through 2009, software development has been essentially a craft where complicated applications are designed as unique artifacts and then constructed from source code on a line-by-line basis. This method of custom development using custom code written line by line can never be efficient, economical, or achieve consistent levels of quality and security.

Composing and painting a portrait in oil paint and developing a software application are very similar in their essential nature. Each of these artifacts is unique, and each is produced using individual “brushstrokes” that need to be perfectly placed and formed in order for the overall results to be aesthetic and effective. Neither portraits nor software applications are engineering disciplines.

Hopefully, by 2049, a true engineering discipline will emerge that will allow software to evolve from a form of artistic expression to a solid engineering discipline. This section presents a hypothetical analysis of the way software applications might be designed and constructed circa 2049.

If software should become a true engineering discipline, then much more than code development needs to be included. Architecture, requirements, design, code development, maintenance, customer support, training, documentation, metrics and measurements, project management, security, quality, change control, benchmarks, and many other topics need to be considered.

The starting point in both 2009 and 2049 will of course be the requirements for the new application. In 2009 users are interviewed to develop the requirements for new applications, but in 2049 a different method may be available.

Let us assume that the application to be developed circa 2049 is a new form of software planning and cost-estimating tool. The tool will provide software cost estimates, schedule estimates, quality estimates, and staffing estimates as do a number of existing tools. However, the tool will also introduce a number of new features, such as:

1. Early sizing prior to knowledge of full requirements
2. Estimates of requirements changes during development
3. Estimates of defect quantities in creeping requirements
4. Integrated risk analysis
5. Integrated value analysis
6. Integrated security analysis
7. Prediction of effects of any CMMI level on productivity and quality
8. Prediction of effects of various quantities of reusable materials
9. Prediction of effects of intelligent agents of software development
10. Prediction of effects of intelligent agents on software maintenance
11. Prediction of effects of intelligent agents on software documentation
12. Prediction of effects of intelligent agents on software customer support
13. Prediction of effects of intelligent agents on software failures
14. Automated conversion between function points, LOC, story points, and so on
15. Estimates of learning curves on the part of users of the application
16. Estimates of mistakes made while users learn the application
17. Estimates of customer support and maintenance for 10+ years after deployment
18. Estimates of application growth for 10+ years after initial deployment
19. Integrated capture of historical data during development and maintenance
20. Automated creation of benchmarks for productivity and quality
21. Expert advice on software quality control
22. Expert advice on software security control

23. Expert advice on software governance
24. Expert advice on intellectual property
25. Expert advice on relevant standards and regulations

The 19th and 20th new features of the estimating tool would involve establishing an overall license with the International Software Benchmarking Standards Group (ISBSG) so that customers would be able to use the tool to gather and analyze benchmarks of similar applications while estimating new applications. Each client would have to pay for this service, but it should be integrated into the tool itself. Thus, not only would estimates be produced by the tool, but also benchmarks for similar applications would be gathered and used to support the estimate by providing historical data about similar applications.

The new estimating tool is intended to be used to collect historical data and create benchmarks semiautomatically. These benchmarks would utilize the ISBSG question set, with some additional questions included for special topics such as security, defect removal efficiency, and customer support not included in the ISBSG questions.

Because the tool will be used to both predict and store confidential and perhaps classified information, security is a stringent requirement, and a number of security features will be implemented, including encryption of all stored information.

We can also assume that the company building the new estimating tool has already produced at least one prior tool in the same business area; in other words, existing products are available for analysis within the company.

Requirements Analysis Circa 2049

The first step in gathering requirements circa 2049 will be to dispatch an intelligent agent or avatar to extract all relevant information about software estimating and planning tools from the Web. All technical articles and marketing information will be gathered and analyzed for similar tools such as Application Insight, Artemis Views, Checkpoint, COCOMO and its clones, KnowledgePlan, Microsoft Project, Price-S, SEER, SLIM, SPQR/20, SoftCost, and all other such tools.

The intelligent agent will also produce a consolidated list of all of the functions currently available in all similar tools; that is, sizing methods, currency conversion, inflation-rate adjustments, quality predictions, total cost of ownership, and so on.

Hopefully, by 2049, software reuse will have reached a level of maturity so that comprehensive catalogs of reusable artifacts will be available; certification for quality and security will be commonplace; and architecture

and design will have reached the point where standard structural descriptions for applications, attachment points, and other relevant issues will be easily accessible.

The intelligent agent will also gather information from public records about numbers of copies of such tools sold, revenues from the tools, user associations for the tools, litigation against tool vendors, and other relevant business topics.

If the tool is used to estimate financial software applications, the intelligent agent will also scan the Web for all government regulations that may be applicable such as Sarbanes-Oxley and other relevant rules. Due to the financial crisis and recession, scores of new regulations are about to surface, and only an intelligent agent and expert system can keep up.

For other forms of software, the intelligent agent might also scan the Web for regulations, standards, and other topics that affect governance and also government mandates—for example, software applications that deal with medical devices, process medical records, or that need legal privacy protection.

Once the universe of existing tools and feature sets has been analyzed, the next step is to consider the new features that will add value over and above what is already available in existing project planning and estimating tools. Here, requirements in 2049 will resemble those of 2009, in that inputs from a number of stakeholders will be collected and analyzed.

Since the application to be developed is an expert-system, much of the information about new features must come from experts in software planning and estimating. Although the views of customers via surveys or focus groups will be helpful, and the views of the marketing organization of the company will be helpful, only experts are qualified to specify the details of the more unique features.

That being said, as requirements for new features are being planned, a parallel effort will take place to develop patent applications for some or all of the unique features. Here too an intelligent agent will be dispatched to gather and analyze all existing patents that cover features that might be similar to those planned for the new estimating tool.

Assuming that most of the new features truly are unique and not present in any current estimating tool, somewhere between half a dozen to perhaps 20 new patent applications will probably be prepared as the requirements are assembled. This is an important step in building applications that contain new intellectual content: violating a patent can cause huge expenses and stop development cold. In particular, the patents of companies such as Intellectual Ventures, whose main business is patent licensing, need to be considered.

In addition to or perhaps in place of patents, there may also be trade secrets, invention disclosures, copyrights, and other forms of protection for confidential and proprietary information and algorithms.

For the tool discussed in this example, patent protection will be needed for the early sizing feature, for the feature that predicts requirements changes during development, and also for the feature that predicts customer learning-curve costs. Other topics might also require patent protection, but the three just cited are novel and unique and not found in competitive tools. For example, no current estimating tools have any algorithms that deal with the impacts of intelligent agents.

The requirements analysis phase will also examine the possible platforms for the new estimating tool; that is, what operating systems will host the tool, what hardware platforms, and so on. No doubt a tool of this nature would be a good candidate for personal computers, but perhaps a subset of the features might also be developed for hand-held devices. In any case, a tool of this sort will probably run on multiple platforms and therefore needs to be planned for Windows, Apple, Linux, Unix, and so on.

Not only will the tool operate on multiple platforms, but also it is obviously a tool that would be valuable in many countries. Here too an intelligent agent would be dispatched to look for similar tools that are available in countries such as China, Japan, Russia, South Korea, Brazil, Mexico, and so on. This information will be part of market planning and also will be used to ascertain how many versions must be built with information translated into other natural languages.

Using information gathered via intelligent agents on current market size, another aspect of requirements analysis will be to predict the market potentials of the new tool and its new features in terms of customers, revenue, competitive advantages, and so forth. As with any other company, the value of the new features will have to generate revenues perhaps ten times greater than development and maintenance costs to commit funds for the new product.

The outputs from the requirements phase would include the requirements for the new tool, summary data on all patents that are relative to the application area, and a summary of the current market for estimating and project planning tools in every country where the tool is likely to generate significant revenues. Summaries of relevant government regulations would also be included. It is interesting that about 85 percent of these outputs could be produced by intelligent agents and expert systems with little human effort other than setting up search criteria.

Superficially, applications designed for service-oriented architecture (SOA) also envision collections of standard reusable components. The object-oriented (OO) paradigm has incorporated reusable objects for more than 30 years. However, neither SOA nor the OO paradigm includes formal mining of legacy applications for algorithms and business rules. Neither uses intelligent agents for searching the Web. Neither SOA nor OO envisions developing all-new features as reusable objects, although

the OO paradigm comes close. Also, neither the quality control nor the security practices of the SOA and OO methods are as rigorous as needed for truly safe applications. For example, certification of the reused code is spotty in both domains.

Design Circa 2049

Because many similar applications already exist, and because the company itself has built similar applications, design does not start with a clean piece of paper or a clean screen. Instead design starts by a careful analysis of the architecture and design of all similar applications.

One very important difference between design circa 2009 and design circa 2049 will be the use of many standard reusable features from in-house sources, commercial sources, or possibly from libraries of certified reusable functions.

For example, since the application is a cost-estimating tool, no doubt currency conversion, inflation rate adjustments, internal and accounting rates of return, and many other features are available in reusable form from either commercial vendors or in-house tools already developed.

Some of the printed output may use report generation tools such as Crystal Reports or something similar. Some application data may be stored in normal commercial databases such as Access, Bento, or similar packages.

Since the company building the application already has similar applications, no doubt many features such as quality estimation, schedule estimation, and basic cost estimation will be available. The caveat is that reuse needs to be certified to almost zero-defect levels to be economically successful.

Ideally, at least 85 percent of the features and design elements will be available in reusable form, and only 15 percent will be truly new and require custom design. For the new features, it is important to ensure high levels of quality and security, so design inspections would be performed on all new features that are to be added.

However, custom development for a single application is never cost-effective. Therefore, a major difference in design circa 2049 from design circa 2009 is that almost every new feature will be designed as a reusable artifact, rather than being designed as a unique artifact for a single application.

Along with formal reuse as a design goal for all important features, security, quality, and portability among platforms (Windows, Apple, Unix, Linux, etc.) are fundamental aspects of design. Custom design for a single application needs to be eliminated as a general practice, and replaced by design for reuse that supports many applications and many platforms.

For example, the new feature that permits early sizing without knowledge of full requirements is obviously a feature that might be licensed to other companies or used in many other applications. Therefore it needs to be designed for multiple uses and multiple platforms. It would also need patent protection.

It may be that the design environment circa 2049 will be quite different from 2009. For example, since most applications are based on prior applications, descriptions of previous features will be extracted from the legacy applications. The extraction of design and algorithms from legacy code can be done automatically via data mining of the source code, assuming that past specifications have not been fully updated or may even be missing.

Therefore in the future, software designers can concentrate more on what is new and novel rather than dealing with common generic topics from legacy applications. The design of the carryover features from legacy applications will be generated by means of an expert system, augmented by web searches for similar applications by using an intelligent agent.

An expert-system design tool will be needed in order to mine information from similar legacy applications. This tool will include the features of static analysis, complexity analysis, security analysis, architecture and design structural analysis, and also the capability of extracting algorithms and business rules from legacy code.

Outputs from the tool will include structural design graphs, control flow information, information on dead code, and also textual and mathematical descriptions of business rules and algorithms embedded in the legacy code.

Even sample use cases and “user stories” could be constructed automatically by an intelligent agent based on examining information available on the Web and from published literature. Data dictionaries of all applications could also be constructed using expert systems with little human involvement.

Because software is dynamic, it can be expected that animation and simulation will also be part of design circa 2049. Perhaps a 3-D dynamic model of the application might be created to deal with issues such as performance, security vulnerabilities, and quality that are not easily understood using static representations on paper.

The completed design would show both old and new features, and would even include comparisons between the new application and competitive applications, with most of this work being done automatically through the aid of intelligent agents and the design engine. Manual design and construction of new algorithms by human experts would be primarily for the new features such as early sizing, requirements growth, and customer learning curves.

For software engineering to become a true engineering discipline, it will be necessary to have effective methods for analyzing and identifying optimal designs of software applications. Designing every application as a unique custom product is not really engineering. An expert system that can analyze the structure, features, performance, and usability of existing applications is a fundamental part of moving software from a craft to an engineering discipline.

Indeed, catalogs of hundreds of optimal designs augmented by catalogs of certified reusable components should be standard features of software architecture and design circa 2049. To do this, a taxonomy of application types and a taxonomy of features are needed. Also, standard architectural structures are needed and may perhaps follow the method of the Zachman architectural approach.

Software Development Circa 2049

Assuming that perhaps 85 percent of software application features will be in the form of standard reusable components, software development circa 2049 will be quite different from today's line-by-line coding for unique applications.

The first stage of software development circa 2049 is to accumulate all existing reusable components and put them together into a working prototype, with placeholders for the new features that will be added later. This prototype can be used to evaluate basic issues such as usability, performance, security, quality, and the like.

As new features are created and tested, they can be appended to the initial working prototype. This approach is somewhat similar to Agile development, except that most instances of Agile do not start by data mining of legacy applications.

Some of the logistical portions of Agile development such as daily progress meetings or Scrum sessions may also be of use.

However, because development is aimed at constructing reusable objects rather than unique single-use objects, other techniques that emphasize and measure quality will also be utilized. The Team Software Process (TSP) and Personal Software Process (PSP) approaches, for example, have demonstrated very high levels of quality control.

Due to very stringent security and quality requirements for the new application, these reusable components must be certified to near zero-defect levels. If such certification is not available, then the candidate reusable components must be put through a very thorough examination that will include automated static analysis, dynamic analysis, testing, and perhaps inspections. In addition, the histories of all reusable components will be collected and analyzed to evaluate any quality and security flaws that might have been previously reported.

Because the new features for the application are not intended for a single use, but are planned to become reusable components themselves, it is obvious that they need to be developed very carefully. Of the available development methods for new development, the Team Software Process (TSP) and the Personal Software Process (PSP) seem to have the rigor needed for creating reusable artifacts. Some of the logistical methods of Agile or other approaches may be utilized, but rigor and high quality levels are the primary goals for successful reuse.

Because of the need for quality, automated static and dynamic analysis, careful testing, and live inspections will also be needed. In particular, special kinds of inspections such as those concentrating on security flaws and vulnerabilities will be needed.

Because of security issues, languages such as E that support security might be used for development. However, some of the older reusable components will no doubt be in other languages such as C, Java, C++, and so on, so language conversions may be required. However, by 2049, hopefully, secure versions of all reusable components may be available.

Software cost-estimating applications of the type discussed in this example are usually about 2,500 function points in size circa 2009. Such applications typically require about two and a half calendar years to build and achieve productivity rates between 10 and 15 function points per staff month.

Defect potentials for such applications average about 4.5 per function point, while defect removal efficiency is only about 87 percent. As a result, about 1,400 defects are still present when the software first goes to users. Of these, about 20 percent, or 280, would be serious enough to cause user problems.

By switching from custom design and custom code to construction based on certified reusable components, it can be anticipated that productivity rates will be in the range of 45 to 50 function points per staff month. Schedules would be reduced by about one year, for a development cycle of 1.5 calendar years instead of 2.5 calendar years.

Defect potentials would be only about 1.25 per function point, while defect removal efficiency would be about 98 percent. As a result, only about 60 latent defects would remain at delivery. Of these, only about 10 percent would be serious, so users might encounter as few as six significant defects after release.

These improvements in quality will of course benefit customer support and maintenance as well as initial development.

Since the tool used as an example is designed to capture historical data and create a superset of ISBSG benchmarks, obviously the development of the tool itself will include productivity, schedule, staffing, and quality benchmarks. In fact, it is envisioned that every major software application

would include such benchmark data, and that it would routinely be added to the ISBSG data collection. However, some applications' benchmark data may not be made publicly available due to competitive situations, classified military security, or for some other overriding factor.

It is interesting to speculate on what would be needed to develop 100 percent of a new application entirely from reusable materials. First, an expert system would have to analyze the code and structure of a significant number of existing legacy applications: perhaps 100 or more. The idea of this analysis is to examine software structures and architecture from examination of code, and then to use pattern-matching to assemble optimal design patterns.

Another criterion for 100 percent development would be to have access to all major sources of reusable code, and, for that matter, access to reusable test cases, reusable user documentation, reusable HELP text, and other deliverables. Not all of these would come from a single source, so a dynamic and constantly updated catalog would be needed with links to the major sources of reusable materials.

Needless to say, interfaces among reusable components need to be rigorously defined and standardized for large-scale reuse to be feasible when components are available from multiple companies and are created using multiple methods and languages.

Because quality and security are critical issues, selected code segments would either have to be certified to high standards of excellence, or run through a very careful quality vetting process that included static analysis, dynamic analysis, security analysis, and usability analysis.

Assuming all of these criteria were in place, the results would be impressive. Productivity rates might top 100 function points per month for an application of 2,500 function points, while development schedules would probably be in the range of three to six calendar months.

Defect potentials would drop below one per function point, while defect removal efficiency might hit 99 percent. At these levels, an application of 2500 function points would contain about 25 defects still present at delivery, of which perhaps 10 percent would be serious. Therefore, only about three serious defects would be present at delivery.

It is unlikely that automatic development of sophisticated applications will occur even by 2049, but at least the technologies that would be needed can be envisioned. It is even possible to envision a kind of robotic assembly line for software where intelligent agents and expert systems perform more than 90 percent of the tasks now performed by humans.

User Documentation Circa 2049

In 2009 both customer support and user documentation are weak links for software applications, and usually range between "unacceptable"

and “marginal.” A few companies such as Apple, IBM, and Lenovo occasionally reach levels of “good,” but not very often.

Since applications constructed from reusable components will have HELP text and user information as part of the package, the first step is to assemble all of the document sections for the reusable materials that are planned for the new application. However, documentation for specific functions lacks any kind of overall information for the entire application with dozens or hundreds of features, so quite a lot of new information must be created.

For user documentation and HELP text, the next step would be to dispatch an intelligent agent or avatar to check the user reviews of all customer manuals, third-party user guides, and HELP text as discussed on the Web. Obviously, both praise and complaints about these topics are plentiful in forums and discussion groups, but an intelligent agent will be needed to gather and assemble a full picture. The reviews of third-party books at web sites such as Amazon will also be analyzed.

Once the intelligent agent has finished collecting information, the sample of books and text with the highest and most favorable reviews from customers will be analyzed, using both automated tools such as the FOG and Fleisch indexes, and also reviews by human writers and authors.

The goal of this exercise is to find the structure and patterns of books and user information that provides the best information based on evaluations of similar applications by live customers. Once excellent documents have been identified, it might be a good idea to subcontract the work of producing user information to the authors whose books have received the best reviews for similar applications.

If these authors are not available, then at least their books can be provided to the authors who are available and who will create the user guides. The purpose is to establish a solid and successful pattern to follow for all publications. Note that violation of copyrights is not intended. It is the overall structure and sequence of information that is important.

Some years ago IBM did this kind of analysis for their own users' guides. Customer evaluation reports were analyzed, and all IBM technical writers received a box of books and guides that users had given the highest evaluation scores.

Other kinds of tutorial material include instructional DVDs, webinars, and perhaps live instruction for really large and complex applications such as ERP packages, operating systems, telephone switching systems, weapon systems, and the like. Unless such material is on the Web, it would be hard to analyze using intelligent agents. Therefore, human insight will probably still play a major part in developing training materials.

Since the application is intended to be marketed in a number of countries, documentation and training materials will have to be translated into several national languages, using automated translation as the starting point. Hopefully, in 2049, automated translation will result in smoother and more idiomatic text than translations circa 2009. However, a final edit by a human author may be needed.

Because tools such as this have global markets, it can be expected that documentation routinely will be converted into Japanese, Russian, German, French, Korean, Chinese, Spanish, Portuguese, and Arabic versions. In some cases, other languages such as Polish, Danish, Norwegian, Swedish, or Lithuanian may also occur.

Customer Support in 2049

As to customer support, it currently is even worse than user information. The main problems with customer support include, but are not limited to:

1. Long wait times when attempting to reach customer support by phone
2. Limited phone support for deaf or hard-of-hearing customers
3. Poorly trained first-line support personnel who can't resolve many questions
4. Limited hours for customer support; that is, working hours for one time zone
5. Slow responses to e-mail queries for support
6. Charges for customer support even to report bugs in the vendor's software
7. Lack of analysis of frequently reported bugs or defects
8. Lack of analysis for "frequently asked questions" and responses

Some of these issues are due to software being routinely released with so many serious bugs or defects that about 75 percent of customer service calls for the first year of application usage are about bugs and problems. When software is developed from certified reusable materials, and when new development aims at near zero-defect quality levels, the numbers of bug-related calls circa 2049 should be reduced by at least 65 percent compared with 2009 norms. This should help in terms of response times for phone and e-mail customer queries.

The next issue is inadequate support for the deaf and hard-of-hearing customers. This issue needs more substantial work on the part of software vendors. Automatic translation of voice to text should be

available using technologies that resemble Dragon Naturally Speaking or other voice translators, but hopefully will have improved in speed and accuracy by 2049.

While TTY devices and telephone companies may offer assistance for the deaf and hard of hearing, these approaches are inconvenient for dealing with software trouble reports and customer service. Long wait times before vendor support phones answer and the need to deal with technical terms makes such support awkward at best.

Ideally, cell phones and landlines might have a special key combination that indicates usage by a deaf or hard-of-hearing person. When this occurs, automatic translation of voice into screen text might be provided by the vendors, or perhaps even made available by cell phone manufacturers.

The main point is that there are millions of deaf and hard-of-hearing computer users, and the poor quality of today's software combined with marginal user guides and HELP text makes access to software customer support very difficult for deaf customers.

Other forms of physical disability such as blindness or loss of limbs may also require special assistive tools.

Because some bugs and issues occur for hundreds or thousands of customers, all bug reports need an effective taxonomy of symptoms so they can be entered into a repository and analyzed by an expert system for common causes and symptoms. These high-frequency problems need to be conveyed to everyone in the customer-support organization. As the bugs or problems are fixed or temporary solutions are developed, these need to be provided to all support personnel in real time.

Some vendors charge for customer support calls. The main reason for such charges is to cut down on the numbers of calls and thereby reduce the need for customer support staff. Charging customers to report bugs or for help in fixing bugs is a cynical and misguided policy. Companies that do this usually have very unhappy customers who would gladly migrate to other vendors. Better quality control is a more effective solution than charging for customer support.

All incoming problem reports that seem to be indicative of real bugs in the software should trigger an immediate set of actions on the part of the vendors:

1. The symptoms of the bug need to be analyzed using a standard taxonomy.
2. Analysis of the bug via static or dynamic analysis should be performed at once.
3. The location of the bug in the application should be narrowed down.

4. The bug should be immediately routed to the responsible change team.
5. Customers reporting the same bug should be alerted about its status.
6. Repairs should be made available to customers as soon as possible.
7. If the bug is in reusable code from an external source, notification should be made.
8. Severity levels and other topics should be included in monthly defect reports.

Some large software companies such as IBM have fairly sophisticated defect reporting tools that analyze bugs, catalog symptoms, route bugs to the appropriate change team, and update defect and quality statistics.

Incidentally, since the example discussed here includes quality and defect estimation capabilities, the tool should of course be used recursively to estimate its own defect levels. That brings up the corollary point that development methods such as TSP and PSP, static analysis, and inspections that improve quality should also be used.

It is technically feasible to construct a customer-support expert system that includes voice recognition; voice to text translation; and an artificial intelligence engine that could speak to customers, listen to their problems, match the problems against other reports, provide status to the customer, and for unique or special cases, transfer the customer to a live human expert for additional consultation and support.

Indeed if expert analysis of reported defects and previous customer calls were included in the mix, the AI engine could probably outperform human customer support personnel.

Since this kind of an expert system does not depend upon human specialists to answer the initial phone calls, it could lower the wait time from less than 10 minutes, which is a typical value circa 2009, to perhaps three rings of the phone, or less than 3 seconds.

A combination of high-quality reusable materials and support of expert systems to analyze software defects could make significant improvements in customer support.

Deployment and Customer Training in 2049

Applications such as the estimating tool used in this example are normally deployed in one (or more) of four different ways:

- They are released on CD or DVD.
- They are downloaded from the Web and installed by customers.

- They can be run from the Web without installation (software as a service).
- They are installed by vendors or by vendor agents.

In 2009, the distribution among these four methods is shifting. The relative proportions are CD installation about 60 percent, downloads about 25 percent, vendor installs 10 percent, and web access about 5 percent.

If current trends continue, by 2049 the distribution might be web access 40 percent, downloads 25 percent, CD installation 20 percent, and vendor installation 15 percent. (Vendor installation usually is for very large or specialized applications such as ERP packages, telephone switching systems, robotic manufacturing, process control, medical equipment, weapons systems, and the like. These require extensive customization during the installation process.)

Although some applications are simple enough for customers to use with only minimal training, a significant number of applications are complicated and difficult to learn. Therefore, tutorial information and training courses are necessary adjuncts for most large software packages. This training may be provided by the vendors, but a significant third-party market exists of books and training materials created by other companies such as book publishers and specialized education groups.

Because of the high costs of live instruction, it can be anticipated that most training circa 2049 will be done using prerecorded webinars, DVDs, or other methods that allow training material to be used many times and scheduled at the convenience of the customers.

However, it is also possible to envision expert systems and avatars that operate in virtual environments. Such avatars might appear to be live instructors and even answer questions from students and interact with them, but in reality they would be AI constructs.

Because of the high cost of producing and distributing paper books and manuals, by 2049 it can be expected that close to 100 percent of instructional materials will be available either online, or in portable forms such as e-book readers, and even cell phones and hand-held devices. Paper versions could be produced on demand, but by 2049 the need for paper versions should be much lower than in 2009.

Maintenance and Enhancement in 2049

Since the average life expectancies of software applications runs from 10 to more than 30 years, a development process by itself is not adequate for a true engineering discipline. It is also necessary to include maintenance

(defect repairs) and enhancements (new features) for the entire life of applications once they are initially developed and deployed.

In the software cost-estimating field discussed in this section, COCOMO first came out in 1981, while Price-S is even older, and many estimating tools were first marketed in the mid-1980s. As can be seen, this business sector is already approaching 30 years of age. In fact, the maximum life expectancy for large applications is currently unknown, because many of them are still in service. A few applications, such as air traffic control, may eventually top 50 years of continuous service.

Incidentally, the growth rate of software applications after their initial deployment is about 8 percent per calendar year, so after 20 to 30 years of usage, applications have ballooned to more than twice their original size. Unfortunately, this growth is usually accompanied by serious increases in cyclomatic and essential complexity; as a result maintenance becomes progressively more expensive and “bad fixes” or secondary defect injections made during changes increase over time.

To slow down the entropy or decay of aging legacy applications, they need to be renovated after perhaps five to seven years of service. Renovation would eliminate error-prone modules, refactor the applications or simplify the complexity of the code, eliminate security flaws, and possibly even convert the code to more modern languages such as E. Automated renovation tools are available from several vendors and seem to work well. One of these tools includes the ability to calculate the function point totals of applications as renovation takes place, which is useful for benchmarks and studies of productivity and quality.

For the example estimating tool used here, new features will be added at least once a year and possibly more often. These releases will also include bug repairs, as they occur.

Because new programming languages come out at rates of about one per month, and because there are already more than 700 programming languages in existence, it is obvious that any estimating tool that supports estimates for coding must keep current on new languages as they occur. Therefore, an intelligent agent will be kept busy scanning the Web for descriptions of new languages, and for published reports on their effects on quality and productivity.

Other new features will be gathered as an intelligent agent scans the release histories of competitive estimating tools. For any commercial application, it is important to be cognizant of the feature sets of direct competitors and to match their offerings.

Of course, to achieve a position near the top of the market for software estimating, mere passive replication of competitive features is not an effective strategy. It is necessary to plan novel and advanced features that are not currently offered by competitive estimating tools.

For the estimating example used in this discussion, a suite of new and interesting features is being planned for several years out. These include but are not limited to:

1. Side-by-side comparison of development methods (Agile, RUP, TSP, etc.)
2. Inclusion of “design to cost” and “staff to cost” estimates
3. Inclusion of earned-value estimates and tracking
4. Estimates of impact of Six Sigma, quality function deployment, and so on
5. Estimates of impact of ISO9000 and other standards
6. Estimates of impact of certification of personnel for testing, QA, and so on
7. Estimates of impact of specialists versus generalists
8. Estimates of impact of large teams versus small teams
9. Estimates of impact of distributed and international development
10. Estimates of impact of multinational, multiplatform applications
11. Estimates of impact of released defects on customer support
12. Estimates of deployment costs for large ERP and SOA projects
13. Estimates of recovery costs for denial of service and other security attacks
14. Estimates of odds of litigation occurring for outsource projects
15. Estimates of costs of litigation should it occur (breach of contract)
16. Estimates of patent licensing costs
17. Estimates of cost of patent litigation should it occur
18. Estimates of consequential damages for major business software defects
19. Estimates of odds of litigation due to serious bugs in application
20. Integration of project history with cost accounting packages

It should be obvious that maintenance of software applications that are constructed almost completely from reusable components derived from a number of sources is going to be more complicated than maintenance in 2009. For the example application in this section, features and code may have been acquired from more than a dozen vendors and possibly from half a dozen in-house applications as well.

Whenever a bug is reported against the application, that same bug may also be relevant to scores of other applications that utilize the same

reusable component. Therefore, it is necessary to have accurate information on the sources of every feature in the application. When bugs occur, the original source of the feature needs to be notified. If the bug is from an existing in-house application, the owners and maintenance teams of that application need to be notified.

Because the example application operates on multiple platforms (Windows, Apple, Linux, Unix, etc.), there is also a good chance that a defect reported on one platform may also be present in the versions that operate on the other platforms. Therefore, a key kind of analysis would involve running static and dynamic analysis tools for every version whenever a significant bug is reported. Obviously, the change teams for all versions need to be alerted if a bug appears to have widespread impact.

Of course, this requires very sophisticated analysis of bugs to identify which specific feature is the cause. In 2009, this kind of analysis is done by maintenance programming personnel, but in 2049, extended forms of static and dynamic analysis tools should be able to pin down bugs faster and more reliably than today.

Maintenance or defect repairs circa 2049 should have access to a powerful workbench that integrates bug reporting and routing, automated static and dynamic analysis, links to test libraries and test cases, test coverage analyzers, and complexity analysis tools. There may also be automatic test case generators, and perhaps more specialized tools such as code restructuring tools and language translators.

Because function point metrics are standard practices for benchmarks, no doubt the maintenance workbench will also generate automated function point counts for legacy applications and also for enhancements that are large enough to change the function point totals.

Historically, software applications tend to grow at about 8 percent per calendar year, using the size of the initial release in function points as the starting point. There is no reason to think that growth in 2049 will be slower than in 2009, but there's some reason to think it might be even faster.

For one thing, the utilization of intelligent agents will identify possible features very rapidly. Development using standard reusable components is quick enough so that the lag between identifying a useful feature and adding it to an application will probably be less than 6 months circa 2049, as opposed to about 18 months circa 2009.

It is not uncommon circa 2009 for the original requirements and design materials to fall out of use as applications age over the years. In 2049, a combination of intelligent agents and expert systems will keep the design current for as long as the application is utilized. The same kinds of expert systems that are used to mine business rules and algorithms could be kept in continuous use to ensure that the

software and its supporting materials are always at the same levels of completeness.

This brings up the point that benchmarks for productivity and quality may eventually include more than 30 years of history and perhaps even more than 50 years. Therefore, submission of data to benchmark repositories such as ISBSG will be a continuous activity rather than a one-time event.

Software Outsourcing in 2049

Dozens of outsourcing companies are in the United States, India, China, Russia, and scores of other countries. Not only do outsource companies have to be evaluated, but larger economic issues such as inflation rates, government stability, and intellectual property protection need to be considered too. In today's world of financial fraud, due diligence in selecting an outsourcer will also need to consider the financial integrity of the outsource company (as demonstrated by the financial irregularities of Satyam Consulting in India).

In 2009, potential clients of outsource companies are bombarded by exaggerated claims of excellence and good results, often without any real history to back it up. From working as an expert witness in a dozen lawsuits involving breach of contract by outsourcers, the author finds it astonishing to compare the marketing claims made by the vendors to the actual way the projects in court were really developed. The marketing claims enumerated best practices throughout, but in reality most of the real practices were astonishingly bad: inadequate estimating, deceitful progress reports, inadequate quality control, poor change management, and a host of other failures tended to be rampant.

By 2049, a combination of intelligent agents and expert systems should add some rigor and solid business insight into the topic of finding suitable outsource partners. Outsourcing is a business decision with two parts: (1) whether outsourcing is the right strategy for a specific company to follow, and (2) if outsourcing is the right strategy, how the company can select a really competent and capable outsource vendor.

The first step in determining if outsourcing is a suitable strategy is to evaluate your current software effectiveness and strategic direction.

As software operations become larger, more expensive, and more widespread, the executives of many large corporations are asking a fundamental question: *Should software be part of our core business?*

This is not a simple question to answer, and the exploration of some of the possibilities is the purpose of this chapter. You would probably

want to make software a key component of your core business operations under these conditions:

1. You sell products that depend upon your own proprietary software.
2. Your software is currently giving your company significant competitive advantage.
3. Your company's software development and maintenance effectiveness are far better than your competitors'.

You might do well to consider outsourcing of software if its relationship to your core business is along the following lines:

1. Software is primarily used for corporate operations; not as a product.
2. Your software is not particularly advantageous compared against your competitors.
3. Your development and maintenance effectiveness are marginal.

Over the past few years, the Information Technology Infrastructure Library (ITIL) and service-oriented-architecture (SOA) have emerged. These methods emphasize the business value of software and lead to thinking about software as providing a useful service for users and executives, rather than as an expensive corporate luxury.

Some of the initial considerations for dealing with the topic of whether software should be an integral part of corporate operations or perhaps outsourced include the following 20 points:

1. Are you gaining significant competitive advantage from your current software?
2. Does your current software contain trade secrets or valuable proprietary data?
3. Are your company's products dependent upon your proprietary software?
4. How much does your current software benefit these business functions:
 - A. Corporate management
 - B. Finance
 - C. Manufacturing and distribution
 - D. Sales and marketing
 - E. Customer support
 - F. Human resources

5. How much software does your company currently own?
6. How much new software will your company need in the next five years?
7. How much of your software is in the form of aging legacy systems?
8. How many of your aging legacy systems are ITIL-compliant?
9. How many of your aging legacy systems are SOA-ready?
10. Is your software development productivity rate better than your competitors?
11. Is your software maintenance more efficient than your competitors?
12. Is your time to market for software-related products better than your competitors?
13. Is your software quality level better than your competitors?
14. Are you able to use substantial volumes of reusable artifacts?
15. How many software employees are currently on board?
16. How many software employees will be hired over the next five years?
17. How many users of software are there in your company?
18. How many users of software will there be in five years?
19. Are you considering enterprise software packages such as SAP or Oracle?
20. Are you finding it hard to hire new staff due to the personnel shortage?

The patterns of answers can vary widely from company to company, but will fall within this spectrum of possibilities:

- A. If your company is a software “top gun” and a notable leader within your industry, then you probably would not consider outsourcing at all.
- B. At the opposite extreme, if your company trails all major competitors in software topics, then outsourcing should be on the critical path for immediate action.

In two other situations, the pros and cons of outsourcing are more ambiguous:

- C. Your software operations seem to be average within your industry, neither better nor worse than your competitors in most respects. In this case, outsourcing can perhaps offer you some cost reductions or at least a stable software budget in the future, if you select the right outsourcing partner.

- D. Another ambiguous outsourcing situation is this: you don't have the vaguest idea whether your software operations are better or worse than your competitors due to a chronic lack of data about software in your industry or in your company.

In this situation, ignorance is dangerous. If you don't know in a quantitative way whether your software operations are good, bad, or indifferent, then you can be very sure that your company is not a top gun and is probably no better than mediocre in overall software performance. It may be much worse, of course. This harsh statement is because all of the really good top-gun software groups have quality and productivity measurement programs in place, so they know how good they are.

Your company might also compare a sample of recent in-house software projects against industry benchmarks from a public source such as the International Software Benchmarking Standards Group (ISBSG).

Once a company decides that outsourcing is a suitable business strategy, the second part of the problem is to find a really competent outsource partner. All outsource companies claim to be competent, and many really are competent, but not all of them. Because outsourcing is a long-term arrangement, companies need to perform serious due-diligence studies when selecting outsource partners.

You may choose to evaluate potential outsource partners with your own staff, or you can choose one or more of the external management consultants who specialize in this area. In either case, the first step is to dispatch an intelligent agent to bring back information on all of the outsourcing companies whose business lines are similar to your business needs: Computer Aid Incorporated (CAI), Electronic Data Systems, IBM, Lockheed, Tata, Satyam (if it still exists), and many others.

Some of the information brought back by the intelligent agent would include financial data if the company is public, information on past or current lawsuits filed by customers, regulatory investigations against the company by the SEC or state governments, and also benchmarks that show productivity and quality results.

A fundamental decision in outsourcing in 2009 is to decide whether a domestic or an international outsource partner is preferred. The international outsource companies from countries such as India, China, or Russia can sometimes offer attractive short-term cost reductions. However, communication with international outsource partners is more complex than with domestic partners, and other issues should be evaluated as well.

Recent economic trends have raised the inflation rates in India, China, and Russia. The decline of the value of the dollar against foreign currencies such as the yen and pound have led to the situation that the United States now is being considered as a major outsource location. For example, IBM is about to open up a large new outsource center in

Dubuque, Iowa, which is a good choice because of the favorable business climate and low labor costs.

Already costs in the United States are lower than in Japan, Germany, France, and other major trading partners. If these trends continue (and if the United States enters a recessionary period), the United States might end up with cost structures that are very competitive in global outsourcing markets.

However, by 2049, a completely different set of players may be involved in global outsourcing. For example, as this is written, Vietnam is developing software methods fairly rapidly, and software expertise is expanding in Mexico, Brazil, Argentina, Venezuela, and many other countries south of the United States.

In fact, assuming some sort of lasting peace can be arranged for the Middle East, by 2049, Iraq, Iran, Syria, and Lebanon may be significant players in global technology markets. The same might occur for Sri Lanka, Bangladesh, and possibly a dozen other countries.

By 2049, you should be able to dispatch an intelligent agent to bring back information on every country's inflation rates, intellectual property protection laws, numbers of outsource companies, software engineering populations, software engineering schools and graduates, local tax structures, outsource company turnover rates; and other information for helping to select an optimum location for long-range contracts.

If you are considering an international outsource partner, some of the factors to include in your evaluation are (1) the expertise of the candidate partners for the kinds of software your company utilizes; (2) the availability of satellite or reliable broadband communication between your sites and the outsource location; (3) the local copyright, patent, and intellectual property protection within the country where the outsource vendor is located; (4) the probability of political upheavals or factors that might interfere with transnational information flow; and (5) the basic stability and economic soundness of the outsource vendor, and what might occur should the vendor encounter a severe financial downturn.

The domestic outsource companies can usually offer some level of cost reduction or cost stabilization, and also fairly convenient communication arrangements. Also, one sensitive aspect of outsourcing is the future employment of your current software personnel. The domestic outsource companies may offer an arrangement where some or all of your personnel become their employees.

One notable aspect of outsourcing is that outsource vendors who specialize within particular industries such as banking, insurance, telecommunications, or some other sector may have substantial quantities of reusable material available. Since reuse is the technology that gives the best overall efficiency for software, the reuse factor is one of the key reasons why some outsource vendors may be able to offer cost savings.

There are ten software artifacts where reuse is valuable, and some of the outsource vendors may have reusable material from many of these ten categories: reusable architecture, plans, estimates, requirements, design, source code, data, human interfaces, user documentation, and test materials.

Some of the general topics to consider when evaluating potential outsource partners that are either domestic or international include the following:

- The expertise of the outsource vendor within your industry, and for the kinds of software your company utilizes. (If the outsource vendor serves your direct competitors, be sure that adequate confidentiality can be assured.)
- The satisfaction levels of current clients who use the outsource vendor's services. You may wish to contact several clients and find out their firsthand experiences. It is particularly useful to speak with clients who have had outsource contracts in place for more than two or three years, and hence who can talk about long-term satisfaction. An intelligent agent might be able to locate such companies, or you can ask the vendors for lists of clients (with the caveat that only happy clients will be provided by the vendors).
- Whether any active or recent litigation exists between the outsource company and either current or past clients. Although active litigation may not be a "showstopper" in dealing with an outsource vendor, it is certainly a factor you will want to find out more about if the situation exists.
- How the vendor's own software performance compares against industry norms in terms of productivity, quality, reuse, and other quantitative factors using standard benchmarks such as those provided by the ISBSG. For this kind of analysis, the usage of the function point metric is now the most widely used in the world, and far superior to any alternative metrics. You should require that outsource vendors have comprehensive productivity and quality measurements and use function points as their main metric. If the outsource vendor has no data on their own quality or productivity, be cautious. You might also require some kind of proof of capability, such as requiring that the outsource vendor be at or higher than level 3 on the capability maturity model integration (CMMI) of the Software Engineering Institute (SEI).
- The kinds of project management tools that the vendor utilizes. Project management is a weak link of the software industry, and the leaders tend to utilize a suite of software project management tools, including cost estimation tools, quality estimation tools, software planning tools, software tracking tools, "project office" tools, risk management

tools, and several others. If your candidate outsource vendor has no quantitative estimating or measurement capabilities, it is unlikely that their performance will be much better than your own.

These five topics are only the tip of the iceberg. Some of the topics included in contractor evaluation assessments include (1) the project management tools and methods used by the vendor, (2) the software engineering tools and methods used by the vendor, (3) the kinds of quality assurance approaches used by the vendor, (4) the availability or lack of availability of reusable materials, (5) the configuration control and maintenance approaches used by the vendor, (6) the turnover or attrition rate of the vendors management and technical staff, and (7) the basic measurements and metrics used by the vendor for cost control, schedule control, quality control, and so on.

The International Software Benchmarking Standards Group (ISBSG) has collected data on more than 5,000 software projects. New data is being collected at a rate of perhaps 500 projects per year. This data is commercially available and provides useful background information for ascertaining whether your company's costs and productivity rates are better or worse than average.

Before signing a long-term outsource agreement, customers should request and receive quantitative data on these topics from potential outsource vendors:

1. Sizes of prior applications built in both function points and lines of code
2. Defect removal efficiency levels (average, maximum, minimum)
3. Any certification such as CMMI levels
4. Staff turnover rates on an annual basis
5. Any past or current litigation against the outsourcer
6. Any past or present government investigations against the out-sourcer
7. References to other clients
8. Quality control methods utilized by the outsourcer
9. Security control methods utilized by the outsourcer
10. Progress tracking methods utilized by the outsourcer
11. Cost-tracking methods utilized by the outsourcer
12. Certified reusable materials utilized by the outsourcer

Automated software cost-estimating tools are available (such as the example tool used in this chapter) that allow side-by-side estimates for

the same project, with one version showing the cost and schedule profile using your current in-house development approaches, and the second version giving the results based on how the outsource contractor would build the same product using their proprietary or unique approaches and reusable materials.

From working as an expert witness in a dozen lawsuits between outsource vendors and their dissatisfied clients, the author has found several key topics that should be clearly defined in outsource contracts:

1. Include anticipated learning curves for bringing the outsource vendor up to speed for all of the applications that are included in the agreement. Assume about one-third of an hour per function point for each outsource team member to get up to speed. In terms of the schedule for getting up to speed, assume about two weeks for 1,000 function points, or six weeks for 10,000 function points.
2. Clear language is needed to define how changing requirements will be handled and funded. All changes larger than 50 function points will need updated cost and schedule estimates, and also updated quality estimates. Requirements *churn*, which are changes that do not affect function point totals, also need to be included in agreements.
3. The quality control methods used by the outsource vendor should be provably effective. A requirement to achieve higher than 95 percent defect removal efficiency would be a useful clause in outsource agreements. Defect tracking and quality measurements should be required. For applications in C, Java, or other supported languages static analysis should also be required.
4. Tracking and reporting progress during software development projects has been a weak link in outsource agreements. Every project should be tracked monthly, and the reports to the client should address all issues that may affect the schedule, costs, or quality of the projects under development. If litigation does occur, these reports will be part of the discovery process, and the vendors will be deposed about any inaccuracies or concealment of problems.
5. Rules for terminating the agreement by both parties should be included, and these rules need to be understood by both parties before the agreement is signed.
6. If penalties for late delivery and cost overruns are included in the agreement, they should be balanced by rewards and bonuses for finishing early. However, quality and schedule clauses need to be linked together.

Many outsource contracts are vague and difficult to administer. Outsource agreements should clearly state the anticipated quality

results, methods for handling requirements changes, and methods of monitoring progress.

Some of the software your company owns may have such a significant competitive value that you may not want to outsource it, or even to let any other company know of its existence. One of the basic preparatory steps before initiating an outsource arrangement is to survey your major current systems and to arrange security or protection for valuable software assets with high competitive value.

This survey of current systems will have multiple benefits for your company, and you might want to undertake such a survey even if you are not considering outsource arrangements at all. The survey of current and planned software assets should deal with the following important topics.

This is an area where intelligent agents and automated business-rule extraction tools should be able to offer great assistance by 2049. In fact, most of the business rules, algorithms, and proprietary data should have been mined from legacy applications and put into expandable and accessible forms by means of AI tools and intelligent agents.

- Identification of systems and programs that have high competitive value, or that utilize proprietary or trade-secret algorithms. These systems may well be excluded from more general outsource arrangements. If they are to be included in an outsource contract, then special safeguards for confidential factors should be negotiated. Note also that preservation of proprietary or competitive software and data is very delicate when international outsource contracts are utilized. Be sure that local patent, copyright, and intellectual property laws are sufficient to safeguard your sensitive materials. You may need attorneys in several countries.
- Analysis of the databases and files utilized by your software applications, and the development of a strategy for preservation of confidential data under the outsource arrangement. If your databases contain valuable and proprietary information on topics such as trade secrets, competitors, specific customers, employee appraisals, pending or active litigation, or the like, you need to ensure that this data is carefully protected under any outsource arrangement.
- Quantification of the number of users of your key system, and their current levels of satisfaction and dissatisfaction with key applications. In particular, you will want to identify any urgent enhancements that may need to be passed on to an outsource vendor.
- Quantification of the size of the portion of your current portfolio that is to be included in the outsource contract. Normally, this quantification will be based on the function point metric and will include the size

in function points of all current systems and applications for which the outsource vendor will assume maintenance responsibility.

- Analysis of the plans and estimates for future or partly completed software projects that are to be included in the outsource arrangement and hence developed by the outsource vendor. You will want to understand your own productivity and quality rates, and then compare your anticipated results against those the outsource vendor will commit to. Here, too, usage of the function point metric is now the most common and the best choice for outsourcing contracts.

Because outsource contracts may last for many years and cost millions of dollars, it is well to proceed with care and thoroughness before completing an outsource contract.

As of 2009, there is no overall census of how long typical outsource agreements last, how many are mutually satisfactory, how many are terminated, and how many end up in court. However, the author's work in litigation and with many customers indicates that 75 percent of outsource agreements are mutually satisfactory; about 15 percent are troubled; and perhaps 10 percent may end up in court.

By utilizing careful due-diligence augmented by intelligent agents and expert systems, it is hoped that by 2049 more than 90 percent of outsource agreements are mutually satisfactory, and less than 1 percent might end up in litigation.

As the global recession lengthens and deepens, outsourcing may be affected in unpredictable ways. On the downside, some outsource companies and their clients may either (or both) go bankrupt. On the upside, cost-effective outsourcing is a way to save money for companies that are experiencing revenue and profitability drops.

A major new topic that should be added to outsource agreements from 2009 forward is that of what happens to the contract and to the software under development in cases where one or both partners go bankrupt.

Software Package Evaluation and Acquisition in 2049

In 2009, buying or leasing a software package is a troublesome area. Vendor claims tend to be exaggerated and unreliable; software warranties and guarantees are close to being nonexistent, and many are actually harmful to clients; quality control even on the part of major vendors such as Microsoft is poor to marginal; and customer support is both difficult to access and not very good when it is accessed. There may also be serious security vulnerabilities that invite hacking and theft of proprietary data, or that facilitate denial of service attacks, as discussed in Chapter 2 of this book.

In spite of these problems, more than 50 percent of the software run on a daily basis in large corporations comes from external vendors or from open-source providers. Almost all systems software such as operating systems and telephone switching systems comes from vendors, as does embedded software. Other large commercial packages include databases, repositories, and enterprise-resource planning (ERP) applications.

Will this situation be much better in 2049 than it is in 2009? Hopefully, a migration to construction from certified components (as discussed earlier) will improve commercial software quality, security, and reliability by 2049. It is hoped that improvements in customer support will occur due to methods also discussed earlier in this chapter.

Prior to acquiring a software package in 2049, the starting point would be to dispatch an intelligent agent that would scan the Web and bring back information on these topics:

1. Information on all packages that provide the same or similar services as needed
2. Reviews of all packages by journals and review organizations
3. Lists of all user associations for packages that have such associations
4. Information on the finances of public software vendors
5. Information on current and past litigation filed against software vendors
6. Information on government investigations against software vendors
7. Information on quality results by static analysis tools and other methods
8. Information on security flaws or vulnerabilities in the package

In 2009, software vendors usually refuse to provide any quantitative data at all. Information on the size of applications, on productivity, on customer-reported bugs, and even on the results of running static analysis tools is not released to customers, with the exception of some open-source packages. They also refuse to provide anything that passes for a warranty or guarantee, other than something trivial or possibly harmful (such as selling client information). Almost all software warranties include specific disclaimers of any responsibilities for harm or damages caused by bugs or security flaws.

A hidden but fundamental reason for poor software warranties is that software controls so many key aspects of business, medicine, government, and military operations that software failures can cause more problems and expense than failures of almost any other kind of product. Software bugs can cause death with medical equipment failures, airplane and rocket malfunctions, air-traffic failure, weapons system

failure, manufacturing shutdowns, errors in critical business data, and scores of other really serious problems. If software companies should ever become liable for consequential damages or business losses due to software bugs, successful litigation could wipe out even major software vendors.

Individuals and small companies that buy software packages at the retail level have no power to change the very unprofessional marketing approaches of software vendors. However, large companies, military agencies, federal and state governments, and other large enterprises do have enough clout to insist on changes in software package development, warranties, guarantees, security control, quality control, and other pertinent issues.

While intelligent agents and expert systems can help in minimizing risks of buying packages with major quality and security flaws, it may take government intervention to improve warranties and guarantees. However, a good warranty would be such a powerful marketing tool that if a major vendor such as IBM were to start to offer meaningful warranties, all competitors would be forced to follow suit or lose most of their business.

At the very least, software vendors should offer a full refund to dissatisfied customers for at least 90 days after purchase. While the vendors might lose a small amount of money, they would probably make quite a bit of additional revenue if this warranty were featured in their ads and packaging.

For large clients that are acquiring major software packages from vendors such as Microsoft, IBM, SAP, Oracle, and so forth, the following information should be a precursor to actually leasing or purchasing a commercial software product in 2049:

1. Size of the application in function points and lines of code
2. Quality control steps used during development
3. Security control steps used during development
4. Numbers of bugs and defects found prior to release of the product
5. Numbers of bugs and defects reported by customers of the product
6. Litigation against the vendor by dissatisfied customers
7. Anticipated customer support for major defect repairs
8. Anticipated defect repair turnaround after defects are reported
9. Guarantee of no charges to customers for reporting defects
10. Guarantee of no charges to customers for support by phone or e-mail
11. Guarantee of refund for product returns within 90 days of installation

Much of this information would rightly be regarded by the vendors as being proprietary and confidential. However, since the information would be going to major customers, no doubt it could be provided under nondisclosure agreements.

The deepening and lengthening global recession is going to add new problems to the software industry, including to commercial vendors. A new clause that needs to be included in major software contracts from 2009 forward is what happens to the software, the warranty, and to the maintenance agreements should either the vendor or the client go bankrupt.

Technology Selection and Technology Transfer in 2049

Two major weaknesses of the software industry since its inception have been that of technology selection and technology transfer. The software industry seldom selects development methods based on solid empirical data of success. Instead, the software industry has operated more or less like a collection of cults, with various methods being developed by charismatic leaders. Once developed, these methods then acquire converts and disciples who defend the methods, often with little or no historical data to demonstrate either success or failure.

Of course, some of these methods turn out to be fairly effective, or at least effective for certain sizes and types of software. Examples of effective methods include (in alphabetical order) Agile development, code inspections, design inspections, iterative development, object-oriented development (OO), Rational Unified Process (RUP), and Team Software Process (TSP). Other methods that do not seem to accomplish much include CASE, I-CASE, ISO quality standards, and of course the traditional waterfall method. For a number of newer methods, there is not yet enough data to be certain of effectiveness. These include extreme programming, service-oriented architecture (SOA), and perhaps 20 more. That few projects actually measure either productivity or quality is one of the reasons why it is difficult to judge effectiveness.

If software is to make real progress as an engineering discipline, rather than an art form, then measurement and empirical results need to be more common than they have been. What would be useful for the software industry is a nonprofit evaluation laboratory that resembles the Consumers Union or the Underwriters Laboratory, or even the Food and Drug Administration.

This organization would evaluate methods under controlled conditions and then report on how well they operate for various kinds of software, various sizes of applications, and various technical areas such as requirements, design, development, defect removal, and the like.

It would be very interesting and useful to have side-by-side comparisons of the results of using Agile development, clean-room development, intelligent-agent development, iterative development, object-oriented development, rapid application development, the Rational Unified Process (RUP), the Team Software Process (TSP), various ISO standards, and other approaches compared against standard benchmark examples.

In the absence of a formal evaluation laboratory, a second tier for improving software selection would be for every software project to collect reliable benchmark data on productivity and quality, and to submit it to a nonprofit clearinghouse such as the International Software Benchmarking Standards Group (ISBSG).

Historical data and benchmarks take several years to accumulate enough information for statistical studies and multiple regression analysis. However, benchmarks are extremely useful for measuring progress over time, whereas evaluations at a consumer lab only deal with a fixed point in time.

Even if development methods are proven to be visibly successful, that fact by itself does not guarantee adoption or utilization. Normally, social factors are involved, and most people are reluctant to abandon current methods unless their colleagues have done so.

This is not just a software problem, but has been an issue with innovation and new practices in every field of human endeavor: medical practice, military science, physics, geology, and scores of others.

Several important books deal with the issues of technology selection and technology transfer. Although these books are not about software, they have much to offer to the software community. One book is Thomas Kuhn's book *The Structure of Scientific Revolutions*. Another book is Paul Starr's *The Social Transformation of American Medicine* (winner of the Pulitzer Prize in 1982). A third and very important book is Leon Festinger's *The Theory of Cognitive Dissonance*, which deals with the psychology of opinion formation.

Another social problem with technology transfer is the misguided attempts of some executives and managers to force methodologies on unwilling participants. Forced adoption of methodologies usually fails and causes resentment as well.

A more effective approach to methodology deployment is to start using the method as a controlled experiment, with the understanding that after a suitable trial period (six weeks to six months), the method will be evaluated and either rejected or accepted.

When this experimental approach is used with methods such as formal inspections, it almost always results in adoption of the technique.

Another troubling issue with technology selection is the fact that many development methods are narrow in focus. Some work best for

small applications, but are ineffective for large systems. Others were designed with large systems in mind and are too cumbersome for small projects and small companies (such as the higher levels of the CMMI). It is a mistake to assume that because a methodology gives good results for a small sample, that it will give good results for every known size and type of software application.

One of the valuable aspects of dispatching intelligent agents is that they may have the ability to capture and display information about the pros and cons of popular development methods such as Agile, TSP, and other related topics such as CMMI, TickIT, ISO standards, and so on.

It would be good if software practices were based on actual data and empirical results in 2049, but this is by no means certain. Moving to actual data will take at least 15 years, because hundreds of companies will need to establish measurement programs and train practitioners in effective measurement methods. Automated tools will need to be acquired, too, and of course their costs need to be justified.

Another sociological issue that affects the software industry is that a number of widely used measures either violate the assumptions of standard economics, or are so ambiguous that they can't be used for benchmarks and comparative studies. Both "lines of code" and "cost per defect" violate economic principles and should probably be viewed as professional malpractice for economic analysis. Other metrics such as "story points" and "use-case points" may have limited usefulness for specific projects, but cannot be used for wide-scale economic analysis. Neither can such measures be used for side-by-side comparisons with projects that don't utilize user stories or use-cases.

For meaningful benchmarks and economic studies to be carried out, either the data must be collected initially using standard metrics such as IFPUG function points, or there should be automated conversion tools so the metrics such as "lines of code" or "story points" or "Cosmic function points" could be converted into standard metrics. It is obvious that large-scale economic studies of either portfolios or the entire software industry need to have all data expressed in terms of standard metrics.

The common practice circa 2009 of using quirky and nonstandard metrics is a sign that the software industry is not really an engineering discipline. The best that can be said about software in 2009 is that it is a craft or art form that sometimes yields valuable results, but often fails.

A study of technology transfer in IBM some years ago found that only about one-third of applications were using what at the time were viewed as being best practices. This led IBM to expend considerable resources on improving technology transfer within the company.

Similar studies at Hewlett-Packard and ITT also revealed rather sluggish technology transfer and extremely subjective technology acquisition. These are chronic problems that need a great deal more study on the part of sociologists, industrial psychologists, and of course the software engineering community itself.

Enterprise Architecture and Portfolio Analysis in 2049

Once intelligent agents, expert design tools, and expert maintenance workbenches become widely deployed, these will open up new forms of work that deal with higher levels of software ownership at the enterprise and portfolio levels.

Today in 2009, corporations and government agencies own thousands of software applications developed over many years and using scores of different architectural approaches, design methods, development methods, and programming languages. In addition, many applications in the portfolios may be commercial packages such as ERP packages, office suites, financial applications, and the like. These applications are maintained at random intervals. Most contain significant quantities of latent bugs. Some even contain “error-prone modules,” which are highly complex and very buggy code segments where bad-fix injection rates of new bugs introduced via changes may top 50 percent.

It would make good business sense to dispatch the same intelligent agents and use the same expert systems to perform a full and careful analysis of entire portfolios. The goal of this exercise is to identify quality and security flaws in all current applications, map out how current applications interact, and to place every application and its feature set on the map of standard taxonomies and standard features that are being used to support development from reusable components.

An additional feature that needs expert analysis and intelligent agents is identifying the portions of software that might need updates due to changes in various government regulations and laws, such as changes in tax laws, changes in governance policies, changes in privacy requirements, and scores of others. Hardly a day goes by without some change in either state or federal laws and regulations, so only a combination of intelligent agents and expert systems could keep track of what might be needed in a portfolio of thousands of applications.

In other words, it would be possible to perform large-scale data mining of entire portfolios and extract all algorithms and business rules utilized by entire corporations or government agencies. Corporate data dictionaries would also be constructed via data mining. Since large portfolios may include more than 10,000 applications and 10 million function

points in their entirety, this work cannot easily be done by human beings and requires automation to be performed at all.

No doubt there would be many thousands of business rules and many thousands of algorithms. Once extracted, these obviously need to be classified and assembled into meaningful patterns based on various taxonomies such as the Zachman architectural approach and also other taxonomies such as those that define application types, feature types, and a number of others.

Not only would this form of data mining consolidate business rules and assist in rationalizing portfolio maintenance and government, but it would also introduce much better rigor in terms of economic analysis, governance, quality control, and security controls.

A huge data dictionary and catalog could be created that showed the impacts of all known government regulations on every application in the corporate portfolio. This kind of work exceeds the unaided capabilities of human beings, and only expert systems and AI tools and intelligent agents are likely to be able to do it at all.

Few companies actually know the sizes of their portfolios in terms of either function points or lines of code. Few companies actually know their maintenance cost breakdowns in terms of defect repairs, enhancements, and other kinds of work. Few companies know current quality levels and security flaws in existing software. Few companies know how many users utilize each application, or the value of the applications to the organization.

By 2049, it is possible to envision a suite of intelligent agents and expert systems constantly at work identifying flaws and sections of legacy applications that need attention due to quality and security flaws. The agents would be geographically dispersed among perhaps 50 different corporate development and maintenance locations. However, the results of these tools would be consolidated at the enterprise level.

As this data is gathered and analyzed, it would have to be stored in an active repository so that it could be updated essentially every day as new applications were added, current applications were updated, and old applications were retired. Some of the kinds of data stored in this repository would include application size in function points and LOC, defect and change histories, security status and known vulnerabilities, numbers of users, features based on standard taxonomies, and relationships to other applications owned by the enterprise or by suppliers or customers to which it connects.

It is also possible to envision much better planning at the level of enterprise architecture and portfolio management when corporate business needs and corporate software portfolios are reliably mapped and all known business rules and business algorithms have been consolidated from existing portfolios via automated tools.

Software portfolios and the data they contain are simultaneously the most valuable assets that most corporations own, and also the most troublesome, error-prone, and expensive to develop, replace, and maintain.

It is obvious that software needs to migrate from a craft that builds applications line by line to an engineering discipline that can construct high-quality and high-security applications from standard components. A combination of intelligent agents, expert systems, architectural methods, and several kinds of taxonomies are needed to accomplish this. In addition, automated methods of security analysis and quality analysis using both static and dynamic analysis should be in constant use to keep applications secure and reliable.

Some of the business purposes for this kind of automated portfolio analysis would include corporate governance, mergers and acquisitions, assessing the taxable value of software assets, maintenance planning, litigation for intellectual property and breach of contract, and of course security and quality improvement. As the economy moves through another recessionary year, every company needs to find ways of lowering portfolio maintenance costs. Only when portfolios can be completely scanned and analyzed by expert applications rather than by human beings can really significant economies be realized.

Portfolio analysis is especially important in the case of mergers and acquisitions between large corporations. Attempting to merge the portfolios and software organizations of two large companies is a daunting task that often damages both partners. Careful analysis of both portfolios, both data dictionaries, and both sets of business rules and algorithms needs to be carried out, but is very difficult for unaided human beings. Obviously, intelligent agents and expert systems would be very helpful both for due diligence and later when the merger actually occurs.

At the level of enterprise architecture and portfolio analysis, graphical representations would be valuable for showing software usage and status throughout the enterprise. A capability similar to that used today for Google Earth might start with a high-level view of the entire corporation and portfolio, and then narrow the view down to the level of individual applications, individual business units, and possibly even individual functions and users.

The main difference between Google Earth and an overall representation of a corporate portfolio is that the portfolio would be shown using animation and real-time information. The idea is to have continuous animated representation of the flow of business information from unit to unit, from the company to and from suppliers, and also to and from customers.

One additional point is significant. Software portfolios are taxable assets in the view of the Internal Revenue Service. There is frequent

tax litigation after mergers and acquisitions that deals with the original development costs of legacy applications. It would be prudent from the point of view of minimizing tax consequences for every company to know the size of each application in the portfolio, the original development cost, and the continuous costs of maintenance and enhancements over time.

A Preview of Software Learning in 2049

Because technology transfer is a weak link in 2009, it is interesting to consider how software topics might be learned by software professionals in 2049.

Considering technologies that are available in 2009 and projecting them forward, education and learning are likely to be very different in the future. This short discussion provides a hypothetical scenario of learning circa 2049.

Assume that you are interested in learning about current software benchmarks for productivity and quality circa 2049.

By 2049, almost 100 percent of all published material will be available online in various formats. Conversion from one format to another will be common and automatic. Automatic translation from one language to another such as Russian to English will no doubt be available, too.

Copyrights and payments for published material will hopefully be resolved by 2049. Ideally, text mining of this huge mass of material will have established useful cross-references and indexing across millions of documents.

First, your computer in 2049 will probably be somewhat different from today's normal computers. It will perhaps have several screens and also independent processors. One will be highly secure and deal primarily with web access, while the other, also secure, will not be directly connected to the Web. The second unit is available for writing, spreadsheets, graphics, and other activities. Hardware security will be a feature of both processors.

Computer keyboards may still exist, but no doubt voice commands and touch-screens will be universally available. Since the technology of creating 3-D images exists today, you may also have the capability of looking at information in 3-D form, with or without using special glasses. Virtual reality will no doubt be available as a teaching aid.

Because reading in a fixed position is soon tiring, one of the screens or a supplemental screen will be detachable and can be held like a book. The most probable format is for a screen similar to today's Amazon Kindle or Sony PR-505. These devices are about the size and shape of a paperback book. No doubt by 2049, high-resolution graphics and full colors will also be available for e-books, and probably animation as well.

Voice commands and touch screens will probably be standard, too. Batteries will be more effective in 2049 as well, and using a hand-held device for eight to ten hours on battery power should be the norm rather than an exception as it is in 2009.

Other technical changes might modify the physical appearance of computers. For example, flat and flexible screens exist in 2009, as do eyeglasses that can show images on the lenses. Regardless of the physical shape of computers, access to the Web and to online information will remain a major function; security will remain a major problem.

By 2049, basically all information will be online, and you will have a personal avatar librarian available to you that is programmed with all of your major interests. On a daily basis you will have real-time summaries of changes in the topics that concern you.

You start your search for benchmark information by entering your personal learning area. The area might appear to be a 3-D image of your favorite campus with trees, buildings, and avatars of other students and colleagues.

You might begin by using a voice or keyed query such as “Show me current software productivity and quality benchmarks.”

Your avatar might respond by asking for additional information to narrow the search, such as: “Do you want development, maintenance, customer support, quality, or security benchmarks?” You might narrow the focus to “development productivity benchmarks.”

A further narrowing of the search might be the question, “Do you want web applications, embedded software, military software, commercial applications, or some specific form of software?”

You might narrow the issue to “embedded software.” Your avatar might then state, “The International Software Benchmarking Standards Group has 5,000 embedded applications from the United States, 7,500 from China, 6,000 from Japan, 3,500 from Russia, and 12,000 from other countries. There are also 5,000 embedded benchmarks from other organizations. Do you want overall benchmarks, or do you wish to compare one country with another?”

You might respond by saying “I’m interested in comparing the United States, Japan, China, India, and Russia. For consistency, use only the ISBSG benchmark data.”

The avatar might also ask, “Are you interested in specific languages such as E, Java, Objective C, or in all languages?” In this case, you might respond with “all languages.”

The avatar might also ask, “Are you interested in specific methods such as Agile and Team Software Process, or in capability maturity levels?” You might respond by saying, “I’m interested in comparing Agile against Team Software Process.”

Your avatar might then say, “For embedded applications about 1,000 in each country used Agile methods and about 2,000 used TSP methods. Almost all embedded applications were at CMMI level 3 or higher.”

At this point, you might say something like, “Create graphs that compare embedded productivity levels by country for embedded applications between 1,000 and 25,000 function points in size. Show a comparison of Agile and TSP methods. Also show the highest productivity levels for embedded applications of 1,000, 5,000, and 10,000 function points.”

Within a few seconds, your initial set of graphs will be displayed. You might then decide to refine your search by asking for annual trends for the past ten years, or by including other factors such as looking at military versus civilian embedded applications.

You might also ask your avatar librarian for the schedules of upcoming webinars and seminars on benchmarks. You might also ask for summary highlights of webinars and seminars on benchmarks held within the past six months.

At this point, you might also ask your avatar to send copies of the graphs to selected colleagues who are working in the same area of research. No doubt by 2049, all professionals will be linked into a number of social networks that deal with topics of shared interest.

These networks occur already in 2009 using commercial services such as Plaxo, LinkedIn, various forums, wiki groups, and other means. But today’s networks are somewhat awkward for sharing large volumes of information.

Although this scenario is hypothetical and may not occur, the major differences between learning in 2049 and learning in 2009 are likely to include these key topics:

1. Much better security of computers than what is available in 2009.
2. The existence of AI avatars or intelligent agents that can assist in dealing with vast quantities of information based on profiles of personal interests.
3. Much better indexing and cross-referencing capabilities among documents than what is available in 2009.
4. Workable methods for dealing with copyrights and payments across millions of documents.
5. The accumulation of private “libraries” of online information that meet your personal criteria. To be useful, intelligent agents will create cross-references and indexes across your entire collection. The bulk of the information will be available online, and much of it can be accessed from hand-held devices equivalent to the Kindle as well as from your computers, smart phones, and other wireless devices.

6. Schedules of all webinars, seminars, and other forms of communications that are in topics that match your personal interest profiles. These can either be viewed as they occur, or stored for later viewing. Your personal avatar librarian can also extract relevant information in summary form.
7. Existence of specialized social networks that allow colleagues to communicate and share research and data in topics such as software productivity, security, quality, and other key issues.
8. Existence of virtual communities associated with social networks so that you and your colleagues can participate in online discussions and meetings in virtual environments.
9. Utilization of standard taxonomies of knowledge to facilitate organizing millions of documents that cover thousands of topics.
10. The development of fairly sophisticated filters to separate low-value information from high-value information. For example, articles on productivity that lack quantitative data would probably be of lower value than articles containing quantitative data.

In 2009, vast quantities of data and information are available on the Web and Internet. But the data is chaotic, unstructured, and very inconsistent in terms of intellectual content. Hopefully by 2049, a combination of standard taxonomies, metadata, and the use avatars and intelligent agents will make it possible to gather useful information on any known topic by filtering out low-value data and condensing high-value data into meaningful collections.

Also by 2049, hundreds of colleagues in various fields will be linked together into social networks that enable them to share data on a daily basis, and to rapidly examine the state of the art in any field of knowledge.

With so much information available, copyright and payment methods must be robust and reliable. Also, security of both personal data collections and libraries of online documents must be very robust compared with 2009 norms. Much of the information may be encrypted. Hardware security methods will probably augment software security methods. But the key topic for extracting useful information from billions of source documents will be the creation of intelligent agents that can act on your behalf.

Due Diligence in 2049

Although the recession has slowed down venture capital investments and brought software IPOs almost to a standstill, it has not slowed down mergers and acquisitions. In fact, several merger companies such as Corum had record years in 2008, which is counter to the recessionary trend.

Whenever due diligence is required, and it is always required for mergers and acquisitions and private investments, it is obvious that the combination of intelligent agents and expert systems would be dispatched to evaluate the portfolios and applications of both parties.

If the companies are medium to large in size, then they will each own more than 1,000 applications that total to more than 1 million function points. Really big companies can own ten times as much software as this. Due diligence of an entire portfolio is far too difficult for unaided humans; only intelligent agents and expert systems can handle such large volumes of software.

After a merger is complete, both the software portfolios and software development organizations of both parties will need to be consolidated, or at least some applications will need to operate jointly.

Therefore, every application should be examined by intelligent agents and expert systems for security flaws, latent defects, interfaces, presence of data dictionaries, reusable materials, and many other topics.

For venture investments in startup companies with perhaps only one or two software applications, expert analysis of the software's quality, security vulnerabilities, and other topics would assist in judging whether the investment is likely to be profitable, or may end up with negative returns.

As previously mentioned, software is a taxable asset. Therefore, every software application needs to keep permanent records of size, original development costs, maintenance and enhancement costs, marketing costs, and other financial data. Quality and reliability data should be kept too, for aid in defense against possible lawsuits from clients or users.

Some of the topics that need to be evaluated during due diligence activities include, but are not limited to, the following:

1. Protection of intellectual property in software assets (patents, trade secrets)
2. On-going litigation (if any) for breach of contract, taxes, and so on
3. Benchmarks of productivity and quality for past applications
4. Quality control methods used for software development
5. Data on defects and reliability of legacy software
6. Data on customer satisfaction of legacy software
7. Security control methods used in software applications (encryption, E, etc.)
8. Security control methods used at the enterprise level (firewalls, antivirus, etc.)
9. Existence of business rules, algorithms, and so on, for legacy applications

10. Enterprise architectural schema
11. Open-source applications used by the companies
12. Similar applications owned by both companies
13. How easily applications can be modified
14. Architectural compatibilities or differences
15. Compensation differences between organizations

Unless a company is a conglomerate and frequently acquires other companies, the logistics of due diligence can be daunting. Professional advice is needed from attorneys and also from specialists in mergers and acquisitions. Additional advice may be needed from security and quality consultants, and also advice from architecture specialists may be needed.

By 2049, a combination of intelligent agents and AI tools should also be available to assist in due diligence for mergers, venture capital investments, and other key business purposes.

Certification and Licensing in 2049

Certification and licensing of software personnel are controversial topics. Certification and licensing were also controversial in the medical field and the legal field as well. If certification exists, then the opposite case of decertification for malpractice would also exist, which is even more contentious and controversial.

The history of medical certification is spelled out in Paul Starr's book *The Social Transformation of American Medicine*, which won a Pulitzer Prize in 1982. Since medicine in 2009 is the most prestigious learned profession, it is interesting to read Starr's book and consider how medical practice in the 1850s resembled software circa 2009.

Curricula for training physicians were two-year programs, and there were no residencies or internships. Many medical schools were run for profit and did not require college degrees or even high school diplomas for entry. Over half of U.S. physicians never went to college.

During training in medical schools, most physicians never entered a hospital or dealt with actual patients. In addition to medical schools that taught "standard" medical topics, a host of arcane medical schools taught nonstandard medicine such as homeopathy. There were no legal distinctions among any of these schools.

Hospitals themselves were not certified or regulated either, nor were they connected to medical schools. Many hospitals required that all patients be treated only by the hospital's staff physicians. When patients entered a hospital, they could not be treated or even visited by their regular physicians.

These small excerpts from Paul Starr's book illustrate why the American Medical Association was formed, and why it wished to improve physician training and also introduce formal specialties, licensing, and certification. As it happened, it required about 50 years for the AMA to achieve these goals.

If certification and licensing should occur for software, the approach used for medical certification is probably the best model. As with early medical certification, some form of "grandfathering" would be needed for existing practitioners who entered various fields before certification began.

What is an interesting question to consider is: What are the actual topics that are so important to software engineering that certification and licensing might be of value? In the medical field, general practitioners and internists deal with the majority of patients, but when certain conditions are found, patients are referred to specialists: oncology for cancer, cardiology, obstetrics, and so forth. There are currently 24 board-certified medical specialties and about 60 total specialties.

For software engineering, the topics that seem important enough to require specialized training and perhaps examinations and board certification are the following:

1. General software engineering
2. Software maintenance engineering
3. Software security engineering
4. Software quality engineering
5. Large-system engineering (greater than 10,000 function points)
6. Embedded software engineering
7. Business software engineering
8. Medical software engineering
9. Weapons-system software engineering
10. Artificial-intelligence software engineering

There would also be some specialized topics where the work might or might not be performed by software engineers:

1. Software metrics and measurement
2. Software contracts and litigation
3. Software patents and intellectual property
4. Software customer training
5. Software documentation and HELP information

6. Software customer support
7. Software testing and static analysis
8. Software configuration control
9. Software reusability
10. Software pathology and forensic analysis
11. Software due diligence
12. Data and business rule mining
13. Deployment of intelligent agents

As time goes by, other topics would probably be added to these lists. The current set considers topics where formal training is needed, and where either certification or licensing might possibly be valuable.

As of 2009, more than a dozen software topics have various forms of voluntary certification available. Some of these include software project management, function point counting (for several flavors of function points), Six Sigma, testing (several different certificates by different groups), Zachman architectural method, and quality assurance.

As of 2009, there seems to be no legal distinction between certified and uncertified practitioners in the same fields. There is not a great deal of empirical data on the value of certification in terms of improved performance. An exception is that some controlled studies have demonstrated that certified function-point counters have higher accuracy levels than uncertified function-point counters.

By 2049, no doubt other forms of certification will exist for software, but whether software will achieve the same level of training, licensing, and certification as medicine is uncertain.

In 2009, about one-third of large software projects are terminated due to excessive cost and schedule overruns. A majority of those that are finished run late and exceed their budgets. When delivered, almost all software applications contain excessive quantities of defects and numerous very serious security flaws.

It is obvious from the current situation that software is not a true engineering discipline in 2009. If software engineering were a true discipline, there would not be so many failures, disasters, quality problems, security flaws, and cost overruns.

If software engineering should become a licensed and certified occupation, then the issue of professional malpractice will become an important one. Only when the training and performance of software personnel reaches the point where project failures drop below 1 percent and defect removal efficiency approaches 99 percent would “software engineering” performance be good enough to lower the odds of wiping out the industry due to malpractice charges. In fact, even 2049 may be an optimistic date.

Software Litigation in 2049

Litigation seems to be outside of the realm of the rest of the economy, and lawsuits for various complaints will apparently increase no matter what the recession is doing. The author often works as an expert witness in software breach of contract litigation, but many other kinds of litigation including, but not limited to, the following are

1. Patent or copyright violations
2. Tax litigation on the value of software assets
3. Theft of intellectual property
4. Plagiarism or copying code and document segments
5. Violations of noncompetition agreements
6. Violations of nondisclosure agreements
7. Fraud and misrepresentation by software vendors
8. Fraud and misrepresentation by software outsourcers
9. Damages, death, or injuries caused by faulty software
10. Recovery of stolen assets due to computer fraud
11. Warranty violations for excessive time to repair defects
12. Litigation against executives for improper governance of software
13. Litigation against companies whose lax security led to data theft
14. Antitrust suits against major companies such as Microsoft
15. Fraud charges and suits against executives for financial irregularities

The legal and litigation arena has much to offer the software community when it comes to searching and consolidating information. The legal reference firm of Lexis-Nexis is already able to search more than 5 million documents from more than 30,000 sources in 2009. Not only that, but legal information is already cross-indexed and much easier to use for tracing relevant topics than software literature is.

From working as an expert witness in a number of lawsuits, the author finds it very interesting to see how trial attorneys go about their preparation. On the whole, a good litigator will know much more about the issues of a case than almost any software engineer or software manager knows about the issues of a new software application. In part this is due to the excellent automation already available for searching legal materials, and in part it is due to the organizations and support teams in law firms, where paralegals support practicing attorneys in gathering key data.

Even the structure of a lawsuit might be a useful model for structuring software development. The first document in a lawsuit is a complaint filed by the plaintiff. Since most software applications are started because of dissatisfaction with older legacy applications or dissatisfaction with particular business practices, using the format of a legal complaint might be a good model for initial requirements.

During the discovery period of a lawsuit, the defendants and the plaintiffs are asked to provide written answers to written questions prepared by the attorneys, often with the assistance of expert witnesses. A discovery phase would be a good model for gathering more detailed requirements and initial design information for software projects.

At some point between the initial complaint and the completion of the discovery phase, expert witnesses are usually hired to deal with specific topics and to assist the lawyers in writing the deposition questions. The experts also write their own expert-opinion reports that draw upon their knowledge of industry topics. For software litigation, experts in quality control and software costs are often used. During software projects, it would also be useful to bring outside experts for critical topics such as security and quality where in-house personnel may not be fully qualified.

After the discovery phase is complete, the next phase of a lawsuit involves depositions, where the defendants, plaintiffs, witnesses, and experts are interviewed and examined by attorneys for both sides of the case. There is no exact equivalent to depositions in most software development projects, although some aspects of quality function deployment (QFD) and joint application design (JAD) do have slight similarities in that they involve personnel with many points of view trying to zero in on critical issues in face-to-face meetings.

Depositions are where the real issues of a lawsuit tend to surface. Good litigators use depositions to find out all of the possible weaknesses of the opposing side's case and personnel. It might be very useful to have a form of deposition for large software projects, where stakeholders and software architects and designers were interviewed by consultants who played the parts of both plaintiff and defendant attorneys.

The value of this approach for software is that someone would play the role of a devil's advocate and look for weaknesses in architecture, development plans, cost estimates, security plans, quality plans, and other topics that often cause major software projects to fail later on. Usually, software projects are one-sided and tend to be driven by enthusiasts who don't have any interest in negative facts. The adversarial roles of plaintiff and defendant attorneys and expert witnesses might stop a lot of risky software projects before they got out of control or spend so much money that cancellation would be a major financial loss.

For software, it would be useful if we could achieve the same level of sophistication in searching out facts and insights about similar projects that lawyers have for searching out facts about similar cases.

Once intelligent agents and expert systems begin to play a role in software development and software maintenance, they will of course also play a role in software litigation. A few examples of how intelligent agents and expert systems can support software litigation are shown next:

- The search engines used by Lexis-Nexis and other litigation support groups are already somewhat in advance of equivalent search capabilities for software information.
- Software cost-estimating tools are already in use for tax cases, where they are used to model the original development costs of applications that failed to collect historical data.
- Static analysis of code segments in litigation where allegations of poor quality or damages are part of the plaintiff claims should add a great deal of rigor to either the side of the plaintiff or the side of the defendant.
- A new kind of litigation may soon appear. This is litigation against companies whose data has been stolen, thus exposing thousands of customers or patients to identity theft or other losses. Since the actual criminals may be difficult to apprehend or live in other countries (or even be other national governments), it may be that litigation will blame the company whose data was stolen for inadequate security precautions. This is a form of consequential damages, which are seldom allowed by U.S. courts. But if such litigation should start, it would probably increase rapidly. Static analysis and other expert systems could analyze the applications from which the data was stolen and identify security flaws.
- Automatic sizing methods for legacy applications that create function point totals can be used for several kinds of litigation (tax cases, breach of contract) to provide comparative information about the application involved in the case and similar applications. Size correlates with both quality and productivity, so ascertaining size is useful for several kinds of litigation.
- A new form of software cost-estimating tool (used as an example in this chapter) can predict the odds of litigation occurring for outsource and contract software development. The same tool predicts delivered defects and problems encountered by users when attempting to install and use buggy software.
- The same software cost-estimating tool used in this chapter, already operational in prototype form in 2009, can predict the costs of litigation for both the plaintiff and defendant. It often happens that neither party

entering litigation has any idea of the effort involved, the costs involved, the interruption of normal business activities, and the possible freezing of software projects. The prototype estimates legal effort, expert-witness effort, employee and executive effort, and the probable duration of the trial unless it settles out of court.

- Static analysis tools can be used to find identical code segments in different applications, in cases involving illegal copying of code. (Occasionally, software companies deliberately insert harmless errors or unusual code combinations that can serve as telltale triggers in case of theft. These can be identified using intelligent agents or as special factors for static analysis tools.)
- A combination of static analysis tools and other forms of intelligent agents can be used to search out prior knowledge and similar designs in patent violation cases.
- Software benchmarks and software quality benchmarks can be used to buttress expert opinions in cases of breach of contract or cases involving claims of unacceptable quality levels.
- For litigation, depositions are held face-to-face, and the statements are taken down by a court stenographer. However, for software meetings and fact-gathering in 2049, more convenient methods might be used. Many meetings could take place in virtual environments where the participants interacted through avatars, which could either be symbolic or actually based on images of the real participants. Court stenographers would of course not be necessary for ordinary discussions of requirements and design for software, but it might be of interest to record at least key discussions using technologies such as Dragon Naturally Speaking. The raw text of the discussions could then be analyzed by an expert system to derive business rules, key algorithms, security and quality issues, and other relevant facts.
- A powerful analytical engine that could examine source code, perform static analysis, perform cyclomatic and essential complexity analysis, seek out segments of code that might be copied illegally, quantify size in terms of function points, examine test coverage, find error-prone modules, look for security flaws, look for performance bottlenecks, and perform other kinds of serious analysis would be a very useful support tool for litigation, and also for maintenance of legacy applications. The pieces of such a tool exist in 2009, but are not all owned by one company, nor are they yet fully integrated into a single tool.

Software litigation is unfortunate when it occurs, and also expensive and disruptive of normal business. Hopefully, improvements in quality control and the utilization of certified reusable material will reduce breach of contract and warranty cases. However, tax cases, patent violations, theft

of intellectual property, and violations of employment agreements can occur no matter how the software is built and maintained.

In conclusion, the software industry should take a close look at the legal profession in terms of how information is gathered, analyzed, and used.

Summary and Conclusions

A major change in development between 2009 and 2049 will be that the starting point circa 2049 assumes the existence of similar applications that can be examined and mined for business rules and algorithms. Another major change is the switch from custom design and line-by-line coding to construction from reusable designs and reusable code segments.

For these changes to occur, a new kind of design and development support tools will be needed that can analyze existing applications and extract valuable information via data mining and pattern matching. Intelligent agents that can scan the Web for useful data and patent information are also needed. Not only patents, but government rules, laws, international standards, and other topics also need intelligent agents.

A final change is that every application circa 2049 should routinely gather and collect data for productivity, quality, and other benchmarks. Some tools are available for these purposes in 2009 as are the ISBSG questionnaires, but they are not yet as widely utilized as they should be.

The goal of the software industry should be to replace custom design and labor-intensive line-by-line coding with automated construction from zero-defect materials.

As the global economy continues another year of recession, all companies need to find ways of reducing software development and maintenance costs. Line-by-line software development is near the limit of its effective productivity, and it seldom achieved effective quality or security. New methods are needed that replace custom design and custom line-by-line coding with more automated approaches.

Maintenance and portfolio costs also need to be reduced, and here too intelligent agents and expert systems that can extract latent business rules and find quality and security flaws are on the critical path for improving software portfolio economics and security.

Readings and References

- Festinger, Leon. *A Theory of Cognitive Dissonance*. Palo Alto, CA: Stanford University Press, 1957.
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press, 1970.
- Pressman, Roger. *Software Engineering – A Practitioners' Approach*, Sixth Edition. New York: McGraw-Hill, 2005.
- Starr, Paul. *The Social Transformation of American Medicine*. Basic Books, 1982.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.

This page intentionally left blank

How Software Personnel Learn New Skills

Introduction

The combination of the financial meltdown of 2008 and 2009 followed by the global recession will make profound changes in training of professional personnel. Low-cost methods such as e-learning will expand rapidly. High-cost methods such as live conferences, classroom training, and several forms of paper publications will decline rapidly. Fortunately, the technologies associated with e-learning are at the point that their effectiveness is increasing.

The rate of change of software technology is extremely rapid. New programming languages appear almost monthly. New programming tools appear almost daily. New software development methodologies appear several times a year.

The fast rate of software technology change means that software professionals are faced with a continuing need to learn new skills. What channels are available to software professionals for learning new skills? How good are the available ways of learning, and what new ways are likely to occur?

Even as software learning methods improve, there are a number of critical topics where software education lags far behind what is truly needed to achieve professional status for software. The most glaring omissions include

1. Software security practices for building low-risk applications
2. Software quality control practices for minimizing delivered defects
3. Software measures and metrics for effective economic analysis

4. Software estimating and planning methods for on-time delivery
5. Software architecture for optimizing use of reusable components
6. Software methods for effective renovation of legacy applications
7. Software technology evaluation and technology transfer
8. Software intellectual property protection

These gaps and omissions need to be quickly filled if software is to evolve from an art form into a true engineering discipline.

Due to the continuing recession, attendance at conferences is going down, some in-house training is being cancelled, and some software instructors are being laid off. These are likely to be permanent changes. From 2009 through the indefinite future, e-learning and webinars are likely to be the major education channel for professional education.

Even newer methods such as virtual environments, avatars, and text mining are likely to grow as the recession continues. Over and above their low costs, these newer methods may offer actual advantages as approaches to learning.

The Evolution of Software Learning Channels

The world of software is evolving new technologies as rapidly as any industry in human history. This means that software professionals are faced with a need to acquire new knowledge and new skills at a very rapid clip.

As of 2009, the United States currently employs about 2.6 million personnel in the technical areas of programming or software development and maintenance, about 280,000 software managers, and perhaps another 1.1 million ancillary personnel in related specialties such as software sales, customer support, software technical writing, and many others.

The U.S. total is probably over 3.8 million professionals if all software-related occupations are considered. The European total is slightly larger than that of the United States, and the world total is approaching 18 million. Exact counts are not available for India, China, and Russia, but these three countries combined probably are equivalent to the U.S. total, and software work is growing very rapidly in all three. Globally, all software personnel need constant refreshment of their knowledge to stay current.

The financial crisis and recession of 2008 and 2009 (and beyond) are likely to disrupt the normal channels of software education. To conserve funds, many companies will cut back on training expenses. A significant number of software personnel may lose their jobs due to downsizing or to actual bankruptcy of their companies. As a result, attendance at conferences will probably diminish sharply, as will attendance at commercial education classes. The effect of a long recession on university and graduate school education is uncertain. It may be that lack of normal

job opportunities may actually increase university and graduate school enrollments, assuming that loans and funding sources do not dry up.

Webinars and online educational channels are likely to expand due in large part to their low costs for both sponsors and students. Indeed, webinars are exploding so rapidly that there is a need for a central catalog of all webinars, organized by topic. On any business day, no fewer than 50 software webinars are being offered in the United States, and no doubt this number will soon rise into the hundreds.

It is conceivable that the recession may cause significant new research into hypermodern training methods such as virtual reality “classrooms,” text mining to convert paper documents into web-accessible documents, and shifting information from paper form into e-book and web-accessible form.

When this report was first produced in 1995, there were only ten major channels for software personnel to acquire new information (see Table 4-1). These channels varied in effectiveness and costs.

Today in 2009, there are 17 channels available for software personnel to acquire new information. New forms of electronic books, blogs, Twitter, web browsing, and webinars, and simulation web sites such as Second Life have been added to the suite of learning methods. In addition, Google and Microsoft are in the process of converting millions of paper documents into online web-accessible documents.

The current report uses a new way of evaluating learning methods. Each method is ranked in terms of four factors using a scale from 1 (best) to 10 (worst):

1. Cost
2. Learning efficiency
3. Learning effectiveness
4. Currency of information

TABLE 4-1 Software Education Channels Available in 1995

1.	In-house education
2.	Commercial education
3.	Vendor education
4.	University education
5.	Self-study from workbooks
6.	Self-study from CD-ROMs or DVDs
7.	Live conferences
8.	Online education via the Internet and World Wide Web
9.	Books
10.	Journals

The number “1” is the top rank or score for each category. All of the available methods are then listed in descending order for each topic.

The first factor, *cost*, does not need much explanation. It simply refers to the expenses a student is likely to have in order to use the learning channel. Costs range from almost free for activities such as web browsing, to extremely expensive for attending a major university or going to graduate school.

The second factor, *learning efficiency*, refers to the amount of calendar time required to impart a given amount of new knowledge to students. A score of “1” indicates the most efficient form of learning. Online education and web browsing are the quickest methods of learning.

The third factor, *learning effectiveness*, refers to the volume of information that the learning channel can transmit to a student. A score of “1” indicates the most effective form of learning. Live instructors in universities and in-house training transmit more information than any other channels.

The fourth factor, *currency*, refers to the average age of the information that is being transmitted, with a score of “1” ranking as highest or having the most recent data. For currency, the online sources of education tend to have the most recent data, followed by conferences and commercial education. Universities lag in currency.

Table 4-2 lists the 17 channels evaluated in the 2009 versions of this report.

Between 1995 and 2009, two computer-aided forms of learning, web browsing and webinars (e-learning), have not only been added to the list, but also have now achieved top ranking in the categories of cost, currency, and efficiency. They are still in the middle of the list in terms of effectiveness, however.

As the recession deepens and lengthens, and as technologies change, we can expect to see many future changes in learning methods and especially changes that achieve lower costs. Hopefully, higher effectiveness might be achieved at the same time.

What Topics Do Software Engineers Need to Learn Circa 2009?

Table 4-3 is a sample of current terms, acronyms, and abbreviations that have come into prominence within the software community over the past few years. The terms give flavor to the kinds of technologies that software personnel need to know about in 2009, technologies which in some cases did not exist even as late as 2000.

Even with 75 entries, this list covers no more than perhaps 30 percent of the topics that are part of the overall knowledge of the software engineering domain as of 2009. Indeed there are more than 700 known programming languages and dialects, more than 50 software design

TABLE 4-2 Ranking of Software Learning Channels as of January 2009

Average Score	Form of Education	Cost Ranking	Efficiency Ranking	Effectiveness Ranking	Currency Ranking
3.00	Web browsing	1	1	9	1
3.25	Webinars/e-learning	3	2	6	2
3.50	Electronic books	4	3	3	4
5.25	In-house training	9	4	1	7
6.00	Self-study CD/DVD	4	3	7	10
7.25	Vendor training	13	6	5	5
7.25	Commercial training	14	5	4	6
7.50	Wiki sites	2	9	16	3
8.25	Live conferences	12	8	8	5
9.00	Simulation web sites	8	7	13	8
10.25	Self-study from books	5	13	12	11
10.25	Journals	7	11	14	9
10.75	On-the-job training	11	10	10	12
11.75	Mentoring	10	12	11	14
12.00	Books	6	14	15	13
12.25	Undergraduate training	15	15	3	16
12.25	Graduate training	16	16	2	15

approaches, and at least a dozen named software development methods, to say nothing of an almost infinite variety of combinations of things.

The topics in Table 4-3 are primarily concerned with software engineering topics. Many other topics affect software outside of software engineering, such as asset management, licensing, protecting intellectual property, governance, the Information Technology Infrastructure Library (ITIL), and hundreds of other topics.

Each topic in this list is relatively new. Each is relatively complicated. How can software personnel stay current with the latest technologies? Even more important, how can software personnel actually learn how to use these new concepts well enough to be effective in their jobs?

Since not every new topic is appropriate for every project, and since some topics are mutually exclusive, it is also important for software engineers and managers to know enough about each to select the appropriate methods for specific projects.

An interesting hypothesis is that one reason why traditional topics tend to fall out of use is that the volume of new topics in the software world is so large. For example, most personnel are limited in the amount of time that can be spent on training. If there is a choice between a new topic such as “Agile development” and a traditional topic such as “design inspections,” the new is likely to be chosen over the old.

TABLE 4-3 Software Knowledge Areas Circa 2009

1. Agile development	35. ITIL (Information Technology Infrastructure Library)
2. ASP (application service provider—software available via the World Wide Web)	36. JAD (joint application design)
3. Automated static analysis	37. Java
4. Automated testing	38. JavaScript
5. B2B (acronym for “business to business” or business via the World Wide Web)	39. OO (object-oriented)
6. BPR (business process reengineering)	40. OLAP (online analytical processing)
7. Caja	41. OLE (object linking and embedding)
8. Client-server computing	42. Orthogonal defect tracking
9. Cloud computing	43. Process improvement
10. Computer security	44. PSP (Personal Software Process)
11. CMM (capability maturity model)	45. QFD (quality function deployment)
12. CMMI (capability maturity model integration)	46. RAD (rapid application development)
13. COSMIC (a function point variant from Canada and Europe)	47. REST (Representational State Transfer)
14. Configuration control	48. RPC (Remote Procedure Call)
15. CRM (Customer Relationship Management)	49. Ruby
16. Data mining	50. Ruby with Rails
17. Data quality	51. RUP (Rational Unified Process)
18. Data warehouses	52. Renovation of legacy applications
19. Dot-com (a company doing business via the World Wide Web)	53. Reusability
20. E-business (electronic business)	54. Scrum
21. E-learning	55. Security vulnerabilities
22. E programming language	56. Software as a Service (SaaS)
23. EA (Enterprise Architecture)	57. SOA (service-oriented architecture)
24. ERP (enterprise resource planning)	58. SOAP (Simple Object Access Protocol)
25. Extreme programming (XP)	59. Six Sigma for software
26. Formal design and code inspections	60. Static analysis
27. Function point metrics	61. Story points
28. GUI (graphical user interface)	62. Supply-chain integration
29. Hacking defenses	63. TCO (total cost of ownership)
30. HTML (Hypertext Markup Language)	64. TickIT
31. I-CASE (integrated computer-aided software engineering)	65. TQM (total quality management)
32. IE (information engineering)	66. TSP (Team Software Process)
33. ISO (International Standards Organization)	67. Trusted computing
34. ISP (Internet service provider)	68. UML (unified modeling language)
	69. Use cases
	70. Use-case points
	71. Virtualization
	72. Web-based applications
	73. Web object points
	74. Wiki sites
	75. XML

In the 1500s, the English financier Sir Thomas Gresham noted that when two forms of currency were in circulation that had different values compared with a fixed standard such as gold, people would hoard the more valuable currency and use the less valuable currency. Gresham's law describing this phenomenon was quite succinct: "bad drives out good."

For software education and training, students with finite available time whose choice is between learning a brand-new technology or learning an older technology, the odds favor signing up for a class in the new technology. For software, a variation of Gresham's law for technology education is "new drives out old."

The fact that courses in new topics tend to be more popular than courses in older topics has nothing to do with the effectiveness or value of the two topics. It is merely a phenomenon that new topics seem to be favored over older topics.

Some older topics such as formal inspections of requirements, design, and code are still among the most effective forms of defect removal ever developed and are also valuable in terms of defect prevention. Yet courses in formal inspections seldom draw as many attendees as courses in newer subjects such as test-driven development, Agile development, or automated testing.

In this book, the way of evaluating the overall effectiveness of various education channels is to combine data on the numbers of students using the channel, the students' satisfaction with the materials, and the students' abilities to actually acquire new skills.

The rankings in this section are derived from interviews and client benchmark studies that the author and his company have performed. Overall, about 600 companies have been visited, including roughly 150 large enough to be included in the Fortune 500 set. About 35 government sites have also been visited, at both state and national levels. More than a dozen of the enterprises visited employed at least 10,000 software personnel. Examples of major software employers include IBM, Microsoft, and Electronic Data Systems (EDS). More than 100 of the companies employed more than 1000 software personnel.

Software Engineering Specialists Circa 2009

The author was commissioned to carry out a study of software specialization within large enterprises. Some of the enterprises that participated included AT&T, the U.S. Air Force, Texas Instruments, and IBM. These enterprises were visited by the author and his colleagues, and they participated in detailed surveys. A number of other corporations were contacted by telephone, or as part of assessment and benchmark consulting studies. These demographic studies are still continuing, and more

recent data was published in the author's book *Software Assessments, Benchmarks, and Best Practices* (Addison-Wesley Professional, 2000).

The original study revealed several interesting facts about the software workforce, but also revealed considerable ambiguity in software demographics. For example:

- Large corporations employ a huge variety of specialists.
- Small corporations employ few specialists and utilize generalists.
- Systems software and information technology groups use different specialists.
- Generic titles such as “member of the technical staff” confuse demographic data.
- There is no consistency from company to company in job titles used.
- There is no consistency among companies in work done by the same job titles.
- The e-business domain is creating scores of new software job-title variants.
- Human resource departments seldom have accurate software employee data.
- Software has many degrees besides computer science or software engineering.
- Some software personnel prefer not to be counted as software professionals.

The last point was a surprise. Many of the personnel who write embedded software are electrical or mechanical engineers by training. Some of these engineers refuse to call themselves programmers or software engineers, even though their primary work is actually that of creating software.

In one company visited, several hundred engineers were developing embedded software for automotive purposes, but were not considered to be software workers by the human resources group, since their academic degrees were in other forms of engineering. When interviewed, many of these engineers preferred to be identified by their academic engineering degrees, rather than by their actual jobs. Seemingly, software engineering is viewed as of somewhat lower status than electrical engineering, telecommunications engineering, automotive engineering, and many others.

In none of the companies and government organizations surveyed could the human resources groups accurately enumerate the numbers of software personnel employed or their job titles. In several of the companies and government agencies visited, the human resources organizations had

no job descriptions of the specialist occupations now part of the overall software community, or even of some mainstream jobs such as “software engineer” or “programmer.”

Assume you were interested in learning the roles and knowledge required to perform a certain job such as “business analyst,” and you visit ten companies who would seem likely to employ such specialists. What you might find as of 2009 is the following:

- Two of the companies employ business analysts, who use similar job descriptions maintained by the company human resource organizations.
- Three of the companies employ business analysts, but use unique local jobs descriptions maintained locally and unknown to the human resource organizations.
- Three of the companies use the title of “business analyst,” but have no written job descriptions either locally or with the human resource organizations.
- Two of the companies have people who seem to work as business analysts, but have different job titles such as “member of the technical staff” or “advisory analyst.” These two have generic job descriptions used by many other specialties.

Given the randomness of today’s situation, it is very difficult to ascertain even basic facts such as how many software employees work for large organizations. Ascertaining more granular data such as the exact kind and number of specialists employed is impossible as of 2009. Ascertaining the specific kinds of knowledge and training these specialists need is not quite impossible, but has thousands of local variations and no consistent overall patterns.

The responsibilities for defining positions and recruiting software technical personnel is often delegated to the software executives and managers, with the role of the human resource organization being secondary and involving nothing more than processing offers and entering new employees into various payroll and administrative systems.

The interviews carried out during the study revealed that statistical analysis of software occupations using only employment data provided by human resources organizations would understate the numbers of software professionals by perhaps 30 percent in typical large high-technology companies. This is because specialized positions such as embedded software development, quality assurance, technical writers, and testing specialists may not be included in counts of programmers or software engineers.

Another source of confusion is the use of generic occupation titles such as “member of the technical staff,” which can subsume more than

20 occupational specialties that include software, electrical engineers, telecommunications engineers, avionics engineers, and many other technical occupations such as business analysts and quality assurance personnel. In some companies, the title “member of the technical staff” includes hardware engineers, software engineers, and support personnel such as technical writers.

The gaps and missing data from human resource organizations explain some of the ambiguity in software population studies published by journals and by government organizations such as the Bureau of Labor Statistics. While many studies no doubt are accurate in terms of reporting the information supplied by human resources organizations, they are not able to deal with errors in the incoming raw data itself.

Varieties of Software Specialization Circa 2009

The total number of kinds of software specialists exceeds 100 and appears to be growing. The overall classes of specialization observed fall into five discrete domains:

- Specialists in specific tools, methods, or languages such as Java or object-orientation
- Specialists in particular business, industry, or technology domains such as banking
- Specialists in support tasks such as testing, quality assurance, or documentation
- Specialists in managerial tasks such as planning, estimating, and measurement
- Specialists in portions of software life cycles such as development or maintenance

Table 4-4 lists the major kinds of specialists observed in the course of software demographic studies and assessment studies among large corporations and government agencies.

Although Table 4-4 includes some 115 job titles or forms of specialization (and is not even 100 percent complete), no company yet studied has employed more than about 50 software specialties that could be correctly identified. Organizations using “member of the technical staff” as a generic job title may have other specialties that were not identified during the author’s studies.

An additional recent title found in more than a dozen companies that specialize in software is that of “evangelist.” This is an intriguing title because it demonstrates that software technology selection tends to be a faith-based activity rather than a scientific or knowledge-based activity.

TABLE 4-4 Software Specialization in Large Software Organizations

-
1. Accounting/financial specialists
 2. Agile development specialists
 3. Architects (software/systems)
 4. Assessment specialists
 5. Audit specialists
 6. Baldrige Award specialists
 7. Baseline specialists
 8. Benchmarking specialists
 9. Business analysts
 10. Business Process Reengineering (BPR) specialists
 11. Capability Maturity Model Integration (CMMI) specialists
 12. Complexity specialists
 13. Component development specialists
 14. Configuration control specialists
 15. Cost estimating specialists
 16. Consulting specialists (various topics)
 17. Curriculum planning specialists
 18. Customer liaison specialists
 19. Customer support specialists
 20. Database administration specialists
 21. Data center support specialists
 22. Data quality specialists
 23. Data warehouse specialists
 24. Decision support specialists
 25. Development specialists
 26. Distributed systems specialists
 27. Domain specialists
 28. Education specialists (various topics)
 29. Embedded systems specialists
 30. Encryption specialists
 31. Enterprise Resource Planning (ERP) specialists
 32. Frame specialists
 33. Expert-system specialists
 34. Function point specialists (COSMIC certified)
 35. Function point specialists (IFPUG certified)
 36. Function point specialists (Finnish certified)
 37. Function point specialists (Netherlands certified)
 38. Generalists (who perform a variety of software-related tasks)
 39. Globalization and nationalization specialists
 40. Graphics artist specialists
 41. Graphics production specialists
 42. Graphical user interface (GUI) specialists
-

(Continued)

TABLE 4-4 Software Specialization in Large Software Organizations (*continued*)

43.	Hacking specialists (for defensive purposes)
44.	Human factors specialists
45.	Information engineering (IE) specialists
46.	Instructors (Management topics)
47.	Instructors (Software topics)
48.	Instructors (Quality topics)
49.	Instructors (Security topics)
50.	Integration specialists
51.	Internet specialists
52.	ISO certification and standards specialists
53.	Joint application design (JAD) specialists
54.	Knowledge specialists
55.	Library specialists (for project libraries)
56.	Litigation support specialists
57.	Maintenance specialists
58.	Marketing specialists
59.	Member of the technical staff (multiple specialties)
60.	Measurement specialists
61.	Metric specialists
62.	Microcode specialists
63.	Multimedia specialists
64.	Nationalization specialists
65.	Network maintenance specialists
66.	Network specialists (LAN)
67.	Network specialists (WAN)
68.	Network specialists (Wireless)
69.	Neural net specialists
70.	Object-oriented specialists
71.	Outsource evaluation specialists
72.	Package evaluation specialists
73.	Performance specialists
74.	Personal Software Process (PSP) specialists
75.	Project cost analysis specialists
76.	Project managers
77.	Project planning specialists
78.	Process improvement specialists
79.	Productivity specialists
80.	Quality assurance specialists
81.	Quality function deployment (QFD) specialists
82.	Quality measurement specialists
83.	Rapid application development (RAD) specialists
84.	Rational Unified Process (RUP) specialists

TABLE 4-4 Software Specialization in Large Software Organizations (*continued*)

85.	Research fellow specialists
86.	Reliability specialists
87.	Repository specialists
88.	Reengineering specialists
89.	Renovation specialists
90.	Reverse engineering specialists
91.	Reusability specialists
92.	Risk management specialists
93.	Sales specialists
94.	Sales support specialists
95.	Scope managers
96.	Scrum masters
97.	Security specialists
98.	Service-oriented architecture (SOA) specialists
99.	Six Sigma specialists
100.	Standards specialists (ANSI, IEEE, etc.)
101.	Statistical specialists
102.	Systems analysis specialists
103.	Systems support specialists
104.	Technology evaluation specialists
105.	Team Software Process (TSP) specialists
106.	Technical translation specialists
107.	Technical writing specialists
108.	Testing specialists
109.	Test library control specialists
110.	Total quality management (TQM) specialists
111.	Value analysis specialists
112.	Virtual reality specialists
113.	Web development specialists
114.	Web page design specialists
115.	Webmasters

The context of the evangelist title usually occurs with fairly new technical concepts such as “Java evangelist” or “Linux evangelist.” The title is common enough so that a Global Network of Technology Evangelists (GNoTE) was formed a few years ago by a Microsoft employee.

While the evangelist job title seems to be benign and is often associated with interesting new concepts, it does highlight some differences between software engineering, medicine, and older engineering fields in terms of how ideas are examined and tested prior to widespread adoption. For software, charismatic leadership seems to be more widely associated with

technology selection and technology transfer than empirical data based on experiment or actual usage.

The book *Selling the Dream* by Guy Kawasaki (formerly of Apple) in 1991 may be the first published use of “evangelist” in a software context. However, in an interview, Kawasaki said that Apple was already using “evangelist” when he first arrived, so he did not originate the title. Even so, Apple is a likely choice as the origin of this interesting job title.

A few other software job titles have occurred whose origins are worth noting. The Six Sigma community has adopted some of the terminology from martial arts and uses “yellow belt,” “green belt,” and “black belt” to indicate various degrees of proficiency.

If licensing and board certification should come to pass in the future of software engineering, it is interesting to speculate how titles such as “evangelist” and “black belt” might be handled by state licensing examinations.

There are interesting correlations between specialization and company size, and also by specialization and industry.

In general, large corporations with more than 1000 total software workers have the greatest numbers of specialists. Small companies with fewer than about 25 software personnel may have no specialists at all in terms of job titles, even though there may be some distribution of work among the software staff.

The high-technology industries such as telecommunications, defense, and aerospace employ more specialists than do service industries and companies in the retail, wholesale, finance, and insurance sectors.

The software domain is expanding rapidly in terms of technologies. The time has arrived when one person cannot know more than a fraction of the information needed to do an effective job in the software industry. When this situation occurs, specialization is the normal response.

If the 75 topics shown in Table 4-3 were combined with the 115 specialists shown in Table 4-4, the result would be a matrix with some 8,625 cells. Since each cell would include its own unique form of knowledge and information requirements, it can be seen that keeping up to speed in modern software engineering topics is a very difficult problem.

If a full spectrum of software topics with about 1000 entries were combined with a full range of software and business specialists with about 250 occupations, the resulting matrix would contain some 250,000 cells. It is apparent that unaided human intelligence could not encompass all of the knowledge in all of the cells. Some form of simplification and a formal taxonomy of knowledge branches would be needed to keep this mass of information in order. No doubt intelligent agents and expert systems would also be needed to extract and analyze knowledge for specific activities and occupations.

The emergence of specialization has occurred in most scholarly disciplines such as medicine and law. In fact, specialization has been a common phenomenon for science in general. The specialties of chemistry, physics, biology, geology, and so on are all less than 200 years old. Prior to the development of these specialized domains, those with scholarly and scientific aspirations were identified by generic titles such as “natural philosopher.”

If licensing and certification of software specialties should become a reality, the presence of more than 115 different specialists will present considerable difficulty. This is almost three times the number of medical specialties that exist in 2009 and more than four times the number of legal specialists.

There would seem to be a necessity to prune and consolidate some forms of specialization in order to bring the total number of specialists into line with law and medicine; that is, fewer than 50 formal domains of specialization and perhaps 25 primary specialties that have significant numbers of practitioners.

It is interesting that software academic curricula appear to be aimed more at generalists than at specialization. For example, most of the quality assurance personnel interviewed learned the elements of testing and quality control via on-the-job training or in-house courses rather than at the university level.

Currently, the explosive growth of the World Wide Web and the emergence of e-business as a major corporate focus are expanding the numbers of job titles. For example, the titles of “webmaster” and “web page developer” appear to be less than ten years old.

New forms of specialization in the software domain occur very rapidly: more than five new kinds of specialists tend to occur almost every calendar year. Chapter 3 of this book deals with the topic of software certification and specialization. Chapter 3 offers suggestions about the minimum number of specialists that might be needed. This is an attempt to begin to identify the correlation between software occupations, and the knowledge and learning channels that will be required to perform at professional levels.

Approximate Ratios of Specialists to General Software Personnel

One of the major topics in the domain of specialization is how many specialists of various kinds are needed to support the overall work of the software generalist community? Table 4-5 uses the term “generalist” to refer to employees who perform software development programming and whose job titles are “programmer,” “programmer/analyst,” or “software engineer.”

TABLE 4-5 Approximate Ratios of Specialists to General Software Populations

Specialist Occupations	Specialists to Generalists	Generalist %
Maintenance and enhancement specialists	1 to 4	25.0%
Testing specialists	1 to 8	12.5%
Systems analysts	1 to 12	12.0%
Technical writing specialists	1 to 15	6.6%
Business analyst	1 to 20	5.0%
Quality assurance specialists	1 to 25	4.0%
Database administration specialists	1 to 25	4.0%
Configuration control specialists	1 to 30	3.3%
Systems software support specialists	1 to 30	3.3%
Function point counting specialists	1 to 50	2.0%
Integration specialists	1 to 50	2.0%
Measurement specialists	1 to 50	2.0%
Network specialists (local, wide area)	1 to 50	2.0%
Performance specialists	1 to 75	1.3%
Architecture specialists	1 to 75	1.3%
Cost estimating specialists	1 to 100	1.0%
Reusability specialists	1 to 100	1.0%
Scope managers	1 to 125	0.8%
Package acquisition specialists	1 to 150	0.6%
Security specialists	1 to 175	0.6%
Process improvement specialists	1 to 200	0.5%
Education and training specialists	1 to 250	0.4%
Standards specialists	1 to 300	0.3%

The generalist category refers to workers whose main role is computer programming, although they often do other work as well, such as requirements analysis, design, testing, and perhaps creating user documentation.

The topic of specialization is only just starting to be explored in depth, so the following ratios have a high margin of error. Indeed, for some kinds of specialization there are not yet any normative ratios available. Not all of the specialists shown here occur within the same company. They are ranked here in descending order. The data in Table 4-5 comes from large corporations that employ more than 1000 software personnel in total.

Although the purpose of the author’s study of software specialization was to explore software occupations in major companies, a secondary result of the study was to recommend some significant improvements in the nature of the data kept by human resource organizations.

In general, the demographic data maintained by human resource organizations is inadequate to predict long-range trends or even current

employment in software occupations. The data appears to be so incomplete for software occupations as to invalidate many demographic studies published by government agencies and large information companies such as Gartner Group. If we do not even know how many software personnel are employed today, it is hard to discuss future needs in a realistic fashion.

Not only is there a huge gap in software demographic data, but there also is an even larger gap in what all these specialized occupations really need to know to perform their jobs with high levels of professional competence. This gap affects university curricula and graduate schools, and also affects every single learning channel.

The author hypothesizes that software has such a large number of informal specialties because it is not yet a true engineering occupation. Software is slowly evolving from a craft or art form, and therefore has not yet achieved a stable point in either the kinds of knowledge that are needed or the kinds of specialists that are required to utilize that knowledge in a professional manner.

Evaluating Software Learning Channels Used by Software Engineers

For more than 50 years, major software employers have tried to keep their employees up to speed by offering training and education on a continuing basis. The best-in-class software employers typically have a pattern that resembles the following:

1. From four to ten weeks of intensive training for new technical employees
2. Annual in-house training that runs from five to ten days per year
3. From one to three external commercial seminars per year
4. Monthly or at least quarterly “webinars” on newer topics
5. A growing library of DVD training courses for self-study
6. A significant library of technical books and journals
7. Tuition-refund programs for managers and technical staff
8. Access to web-based technical sites
9. Subscriptions to software and management journals such as *CrossTalk*
10. Subscriptions to executive journals such as *CIO*
11. Subscriptions to information providers such as Gartner Group
12. Increasing use of webinars, e-books, and web-based information

Unfortunately, the recession of 2009 is likely to cause significant reductions in many of these methods, as companies scramble to stay solvent. Not only are some of the educational approaches likely to be cut back, but also some of the instructors may face layoffs. Hiring itself may be reduced to such low levels that training needs are also reduced.

Because multiple channels of education are available, it is interesting to consider the topics for which each kind of channel is likely to be selected, and their strengths and weaknesses. The following educational channels are ranked in terms of their overall scores. In the following ranking, “1” is best. There are 17 channels shown in the full discussion.

Number 1: Web Browsing

Costs = 1; Efficiency = 1; Effectiveness = 9; Currency = 1; Overall Score = 3.00

Prognosis: Expanding rapidly

Web browsing using search engines such as Google or Ask is now the most rapid and cost-effective method for finding out about almost any technical topic in the world. By using a few choice search phrases such as “software quality” or “software cost estimating,” within a few moments, millions of pages of information will be at hand.

The downside of web browsing is that the information is likely to be scattered, chaotic, and a mixture of good and bad information. Even so, web browsing is now a powerful research tool not only for software engineers, but also for all knowledge workers.

A number of web portals provide unusually large numbers of links to other relevant sources of information. One of these is the portal of the Information Technology Metrics and Productivity Institute (ITMPI), which provides an extensive fan-out to topics in areas such as software engineering, quality assurance, project management, and maintenance of legacy applications (www.ITMPI.org).

Among academic links, one of the most complete is that of Dave W. Farthing of the University of Glamorgan in the United Kingdom (www.comp.glam.ac.uk). This interesting portal has links to dozens of project management sites and to the publishers of scores of project management books.

A third portal to useful software topics is the web site of the Software Engineering Institute (SEI), which is somewhat slanted toward the government and defense sectors. However, it still covers a host of interesting topics and provides many useful links (www.SEI.org).

However, consolidation of information by topic, cross indexing, and knowledge extraction remain somewhat primitive in the software domain.

Number 2: Webinars, Podcasts, and E-Learning

Costs = 3; Efficiency = 2; Effectiveness = 6; Currency = 2; Overall Score = 3.25

Prognosis: Expanding rapidly in usage; improving in effectiveness

Using computers for training has long been possible, but new technologies are making rapid improvements. Within a few years, students may be able to take extensive training in virtual environments augmented by many new learning methods.

Webinars are a new kind of seminar that is exploding in popularity. With webinars, the speakers and the attendees are located in their own offices or homes, and all use their own local computers. A webinar hosting company connects all of the participants and also provides back-up support in case any participant gets disconnected. The hosting company also provides evaluations of the event.

Podcasts are similar to webinars, except that they may or may not be broadcast live. If they are recorded, then the information is available on demand at any time. Podcasts can also include quizzes and tests, with automatic scoring and rerouting to different parts of the material based on student answers.

In webinars, the primary speaker and also an MC communicate with the attendees by phone, but PowerPoint slides or other computer-generated information appears on the attendees' screens, under the control of the primary speaker or the MC.

Because no travel is involved, the efficiency and cost scores of webinars are quite good. The currency score is also quite good. As of 2008, the effectiveness was only mid-range but increasing rapidly. Webinars at the moment are primarily used for single-purpose presentations of up to about 90 minutes in duration.

Webinars and podcasts have expanded in numbers so rapidly that on any given day, perhaps 50 such webinars are being offered concurrently by various organizations. It is almost impossible to keep track of the numbers of webinars. What would be of value to the industry is a nonprofit catalog that would allow companies and universities to post the schedules and abstracts of all webinars for at least two months in advance.

In the future, it can be expected that because of the cost-effectiveness of the webinar and podcast approach, entire courses of perhaps 10 to 20 hours in duration will migrate to the webinar method. At some point, virtual environments using avatars will join the mix, and then e-learning will have a good chance of becoming more effective than any other form of training in human history.

As of 2009, the technology of webinars is still immature. Disconnects, low volume, and intermittent telephone problems are not uncommon. The VOIP form of telephone calls, for example, often does not work at all. No doubt these issues will be resolved over the next year or two.

It is theoretically possible to envision entire courses taught on a global basis without the students and instructors ever meeting face to face, by using webinars and computer communications exclusively.

It is only a short step from today's webinars to using a virtual classroom where students and instructors can see each other, either as abstract avatars or by using actual images of real people.

Some universities such as MIT have already integrated video and audio connections into lecture halls, so that remote students can participate in live classes. It is only a small step to extend this technology to hand-held devices or to record the lectures and discussions for later use.

In addition to merely having webinars and then saying goodbye, astute companies will note that the connections to several hundred possible customers might be a good marketing opportunity as well. Attendees can register with the instructor or the company sponsoring the event to receive additional information.

Indeed, it would also be possible to invite webinar attendees to participate in other forms of information transfer such as wiki sites. There are also blogs and Twitter sites, which are essentially channels for personal views and opinions.

The long-range direction in online learning is positive. More and more information will be available, and more and more personnel will be able to communicate and share ideas without travel or face-to-face contact in real life.

Eventually, some combination of avatars, virtual reality, and improved methods of using intelligent agents to extract and consolidate knowledge should lead to the ability to offer entire educational curricula from high-school through graduate school via online and web-accessible methods. No doubt wireless technologies will also be involved, and information will also become available on hand-held devices such as smart phones.

Number 3: Electronic Books (e-books)

Costs = 4; Efficiency = 3; Effectiveness = 3; Currency = 4; Overall Score = 3.50

Prognosis: Expanding in usage; improving in effectiveness; declining in costs

Electronic books, or e-books, have been on the periphery of education and entertainment for more than 20 years. In the past, electronic books were characterized by dim screens, awkward interfaces, slow downloads, limited content, and general inconvenience compared with paper books, or even compared with normal computers whose screens are easier to see.

There have long been organizations such as Project Gutenberg that make books available in HTML, PDA, Word, or other web-accessible forms. This trend is now exploding as Google, Microsoft, and other major players are attempting to use automated text-conversion tools to convert almost

100 percent of hard-copy books into web-accessible formats. Needless to say, these attempts have raised serious issues on copyrights, intellectual property, and compensation for authors and copyright holders.

Also, given the huge volumes of online text now available, there are also technical issues that need to be improved such as cross-references, abstracts, indexes, inclusive catalogs, and so on. However, the trend toward web-accessible text is expanding rapidly and will continue to do so.

Interestingly, the legal profession seems to be somewhat ahead of the software profession in terms of concentrating knowledge, cross-referencing it, and making information accessible to professionals who need it. The Lexis-Nexis company, for example, has access to more than 5 million documents from perhaps 30,000 sources. There is no equivalent organization that has such a well-organized collection of software information.

More recently, starting in 2007 and 2008, Amazon and Sony began to change the equation for electronic books with new models that solved most of the problems of older e-books and simultaneously introduced powerful new features.

The new Amazon Kindle and the Sony PR-505 have at least an outside chance at transforming not only software learning, but also learning in all other scientific disciplines. The best features of these new devices include excellent screen clarity, very fast downloads (including wireless connections), long battery life, and much-improved interfaces. In addition, they have some features that are even superior to regular paper books. These include the ability to keep permanent annotations, to skip from book to book on the same topic, and to get frequent updates as new materials become available from publishers.

Using e-books as software learning tools, it would be easy to select and download the top ten books on software project management, the top ten books on software quality, the top ten books on software maintenance, the top ten books on software cost estimating, and the top ten books on software development, and have all of them present and available simultaneously.

For colleges and universities, it would easily be possible to download every book for every course each semester and have them all available concurrently.

The current models of the Sony and Kindle devices are already quite good, but as with all new products, improvements can be expected to appear rapidly over the next few years. Within perhaps five years, it can be anticipated that e-books will be making a substantial dent in the market for conventional paper books. Some features to anticipate include the ability to deal with graphics and animation downloads; e-book subscriptions from major technical journals; and perhaps inclusions of links to relevant web sites.

Number 4: In-House Education

Costs = 9; Efficiency = 4; Effectiveness = 1; Currency = 7; Overall Score = 5.25

Prognosis: Effective, but declining due to recession

Due to the economic crisis and recession of 2008 and 2009, in-house training is likely to diminish over the next few years due to layoffs and cost-cutting on the part of major corporations, to say nothing of some major corporations filing for bankruptcy and going out of business.

In-house education was number one in overall effectiveness from 1985, when our surveys started, through the end of 2007, before the financial crisis and the recession. In-house education still ranks as number 1 in effectiveness and is number 4 in efficiency. That means that a great deal of information can be transmitted to students with relative ease and speed.

However, this channel is only available for employees of fairly large corporations such as IBM, Microsoft, Google, and the like. The author estimates that roughly half of the U.S. software personnel currently work in organizations large enough to have in-house training, that is, just over 1.6 million U.S. software professionals have access to this channel.

More recent studies in 2001, 2002, 2003, and 2004 indicated a decline in this channel. The economic downturn of 2008 and 2009 caused some in-house training to be cancelled, due in part to the instructors being laid off or taking early retirement.

In addition, the reduction in entry-level hiring has reduced the need for education of recent college graduates who are joining large companies in junior positions. In-house education is likely to continue to be diminished through 2010 as the recession lengthens and deepens. If other channels such as virtual education and webinars continue to expand, the high-water mark for in-house education may have passed.

Some large software employers such as Accenture, AT&T, EDS, IBM, Microsoft, and many others have in-house curricula for software professionals and managers that are more diverse and current than most universities in the United States. A former chairman of ITT observed that the Fortune 500 companies in the United States have more software instructors than all universities put together. Employees within large companies have more student days of in-house training than all other channels combined.

The in-house courses within major corporations are usually very concentrated and very intensive. An average course would run from two to three business days, starting at about 8:30 in the morning and finishing at about 4:30 in the afternoon. However, to optimize student availability, some courses continue on into the evening.

From observations of the curricula and attendance at some of the courses, the in-house education facilities of large corporations are among the most effective ways of learning current technologies.

Another advantage of in-house training is the ease of getting approval to take the courses. It requires far less paperwork to gain approval for a corporation's in-house training than it does to deal with a tuition-refund program for university courses.

As this is written in 2009, it is uncertain if in-house education will regain the importance that it had during the 1980s and 1990s. The economic future is too uncertain to make long-range predictions.

One interesting finding about in-house education is the impact on software development productivity. Companies that provide ten days of training per year for software engineers have higher annual productivity rates than companies that provide zero days of training, even though ten working days are set aside for courses. The value of the training appears to pay off in better performance.

Although the information was not explored in depth and there may not be a provable relationship, it was interesting that companies with ten days of training per year had lower rates of voluntary attrition than companies with zero days of training. Apparently training has a beneficial impact on job satisfaction.

Number 5: Self-Study Using CD-ROMs or DVDs

Costs = 4; Efficiency = 3; Effectiveness = 7; Currency = 10; Overall Score = 6.00

Prognosis: Improving slowly in usage; improving faster in effectiveness

The technology of self-study courses is on the verge of being transformed by new CD-ROM and DVD approaches. It may also be transformed by e-books. The older form of self-study courses consisted of tutorial materials, exercises, and quizzes often assembled into loose-leaf notebooks. The CD-ROM or DVD varieties include all of the prior approaches, but can also feature hypertext links and a huge variety of supplemental information.

The newest form of DVD training actually allows new content to be added to the course while it is in session. This method is so new that little empirical data is available. When it becomes widespread, the "currency" score should climb rapidly.

The prospect of fully interactive learning via DVDs is an exciting one, since graphics, animation, voices, and other topics can now be included. However, the costs and skill required to put together an effective DVD course are significantly greater than those needed for traditional workbooks. The costs for the students are not particularly high, but the production costs are high.

Until about 1995, the number of CD-ROM drives in the hands of potential students was below critical mass levels, and many of these were older single-speed drives with sluggish access times and jerky animation.

However, by 2009 the author estimates that more than 99 percent of software personnel have DVD or CD-ROM drives on their office computers. (Ultralight net-book devices weighing less than 3 pounds often lack drives for DVDs and CDs. Also, some organizations that do highly classified work prohibit all forms of removable media on the premises.)

As of early 2009, the author estimates that more than 250,000 software personnel have taken self-study courses via CD-ROMs or DVDs. Probably fewer than 125 such courses currently are available due to the difficulty and costs of production.

As the 21st century advances, it is possible that self-study courses using CD-ROM and DVD approaches will expand in numbers and improve in effectiveness. Giving this self-study channel a boost are lightweight portable CD-ROM or DVD viewers in notebook computers that can be carried and used on airplanes. The same is true of electronic books.

Even newer technologies will allow the equivalent to DVDs to be downloaded not only to computers, but also directly to television sets, iPods, smart phones, and other hand-held devices.

The impact of the new Blu-Ray format can potentially improve the interactive capabilities of DVD education, but so far this idea remains largely unexploited by educational companies due to the fairly high costs of Blu-Ray production.

Number 6: Commercial Education

Costs = 14; Efficiency = 5; Effectiveness = 4; Currency = 6; Overall Score = 7.25

Prognosis: Declining due to recession

The economic crisis of 2008 and 2009 and the continuing recession are having a depressing impact on commercial education. Numbers of students are in decline, and air-travel is becoming more and more expensive. No doubt lower-cost methods such as e-learning will start to supplant normal commercial education.

For many years, commercial education ranked number two in overall effectiveness. There is a significant subindustry of commercial education providers for the software domain. In 2009, it is now in fifth place, but still quite effective.

Companies within this subindustry include Computer Aid (CAI), Construx, Coverity, Cutter, Data-Tech, Digital Consulting Inc. (DCI), Delphi, FasTrak, the Quality Assurance Institute (QAI), Learning Tree,

Technology Transfer Institute (TTI), and many others who teach on a national or even international level. These companies provide education in both technical and managerial topics.

Operating at a higher level are specialized information companies such as Gartner Group and Forrester Research. These companies both provide standard reports and do customized research for clients. The main client bases of such high-end companies are executives and top management. In keeping with these target markets, the fees and costs of such research are high enough so that they appeal primarily to major corporations and some large government software executives, such as the CIOs of states and major cities.

Nonprofit organizations also offer fee-based training. For example, the nonprofit International Function Point Users Group (IFPUG) offers training and workshops in function point topics. The Project Management Institute (PMI) also offers commercial education, as does the Software Engineering Institute (SEI). The International Software Benchmarking Standards Group (ISBSG) also offers training in estimation and benchmark topics, and publishes books and monographs on both topics.

Hundreds of local companies and thousands of individual consultants teach courses on a contract basis within companies and sometimes as public courses as well. Many courses are offered by nonprofit organizations such as the ACM (Association for Computing Machinery), DPMA (Data Processing Management Association), IEEE, IFPUG, SEI, and scores of others.

The author estimates that about 500,000 U.S. software personnel take at least one commercial software seminar in the course of a normal business year. However, from 2009 onward, there will be a sharp decline due to the global recession.

Since the major commercial educators run their training as a business, they have to be pretty good to survive. A primary selling point of the larger commercial education companies is to use famous people as instructors on key topics. For example, such well-known industry figures as Bill Curtis, Chris Date, Tom DeMarco, Tim Lister, Howard Rubin, Ed Yourdon, Watts Humphrey, Dr. Gerry Weinberg, Dr. James Martin, and Dr. Carma McClure all offer seminars through commercial education companies.

A typical commercial seminar will run for two days, cost about \$895 to enroll, and attract 50 students. A minor but popular aspect of commercial education is the selection of very good physical facilities. Many commercial software courses are taught at resort hotels in areas such as Aspen, Orlando, Phoenix, or San Francisco.

However, the more recent recession of 2009 (and beyond) needs to be considered. For several years, business activities involving travel to

other cities were reduced somewhat and have never fully recovered. As the recession grows more widespread, even further reductions can be anticipated.

The main strengths of commercial education remain and are twofold:

- Very current topics are the most salable.
- Top-notch instructors are the most salable.

This means that commercial seminars are likely to cover material that is not available from a university or even from an in-house curriculum. It also means that students get a chance to interact with some of the leading thinkers of the software domain. For example, in 2009, the commercial education channel is much more likely than most to cover current hot topics such as Agile development, Six Sigma for software, or web-based topics.

The commercial education market has been most widely utilized by companies in the top quartile of software productivity and with quality levels as noted during SPR assessment and benchmark studies. In other words, companies that want top performance from their managers and technical workers realize that they need to bring in top-gun educators.

Number 7: Vendor Education

Costs = 13; Efficiency = 6; Effectiveness = 5; Currency = 5; Overall Score = 7.25

Prognosis: Declining due to recession

Vendor education was formerly ranked number three in overall effectiveness and is now number six. This is not because of slipping quality on the part of vendors, but because of the rapid emergence of webinars and new DVD technologies.

Vendor-supplied education has been a staple of the software world for almost 50 years. Because vendor education in tools such as spreadsheets and word processors are taken by non-software personnel, the total number of students is enormous. However, within the software domain, the author estimates that about 500,000 software personnel will take at least one vendor course in a normal business year.

Vendor education used to be free in the days when hardware and software were still bundled together. Some vendor education is still free today, when used as part of marketing programs. Normally, vendor education is sold to clients at the same time that they buy the tools or packages for which training is needed.

Almost every large commercial software application is complicated. Even basic applications such as word processors and spreadsheets

now have so many features that they are not easily mastered. Thus large software companies such as Artemis, IBM, Oracle, and Computer Associates offer fee-based education as part of their service offerings.

The size, feature set, and complexity of software products mean that every major vendor now has some kind of education offering available. For really popular tools in the class of Microsoft Word, WordPerfect, Excel, Artemis Views, KnowledgePlan, and so on, there may be local consultants and even high-school and college courses that supplement vendor-supplied education.

For very large applications such as ERP packages from Oracle and SAP, it is hardly possible to learn to use the software without extensive education either from the vendors, or from specialized education companies that support these packages.

Vendor education is a mainstay for complicated topics such as learning how to deploy and utilize enterprise resource planning packages by vendors such as BAAN, Oracle, PeopleSoft, SAP, and others.

Vendor-provided education runs the gamut from very good to very poor, but on the whole does a creditable job of getting clients up and running on the applications in question.

Vendor education is usually a lot cheaper than commercial education, and the effective costs are sometimes less than \$100 per student day. Vendor education is often offered on a company's own premises, so it is generally very convenient. On the other hand, you don't expect big name instructors to constitute the faculty of vendor training either.

However, newer methods such as e-learning are even cheaper and have the advantage that courses can be taken 24 hours a day at the convenience of the students. Therefore, vendor education will decline in future years, and e-learning methods will become the major vehicle.

Further, as the recession deepens and lengthens, instructors are usually among the first to lose their jobs due to cost-cutting measures. Therefore, vendor education with live instructors is entering a period of decline.

As software packages continue to evolve new features and more complexity, vendor-supplied education will remain a stable feature of the software world well into the next century, but moving from live instructors toward perhaps e-learning or even using virtual environments with avatars.

Vendor education has also been negatively affected by the economic downturn circa 2008–2009. Some vendors lost money due to reduced sales, so course offerings were naturally reduced as well. Other vendors were acquired, and some went out of business. In spite of the reduction in courses and instructors, vendor-supplied education remains an important channel of instruction, even though the numbers of vendors, students, and courses are in decline.

Number 8: Live Conferences

Costs = 12; Efficiency = 8; Effectiveness = 8; Currency = 5; Overall Score = 8.25

Prognosis: Declining due to recession

Unfortunately, the financial crisis and recession of 2008 and 2009 (and beyond) are causing a severe decline in live conferences, and this trend may continue indefinitely. The author estimates that attendance at live conferences in 2009 will probably be about 30 percent less than the same events in 2006, due to layoffs, cost cutting, and other financial issues.

Software-related conferences rank number eight in effectiveness in teaching new skills. However, they would rank number one in highlighting interesting new technologies. Unfortunately, the combined impact of the September 11 disaster and the economic downturn since 2000 caused many companies to cut back corporate travel, which affected conference attendance.

The author estimates that attendance at software conferences in 2003 was probably 15 percent below what might have been the norm in 1999. However, 2006 and 2007 saw increases in conference attendance. The recession of 2008 is still too new to have impacted many conferences as this is written, but no doubt conference attendance will decline before the end of 2008 and perhaps in 2009 based on the poor results of the U.S. and world economies.

Even so, there are major software conferences every month of the year, and some months may have multiple events. Conferences are sponsored both by nonprofit organizations, such as the U.S. Air Force STSC (Software Technology Support Center), IEEE, IFPUG, GUIDE (Global Uniform Interoperable Data Exchange), SHARE (Software-Hardware Asset Reuse Enterprise), or ASQC (American Society for Quality Control), and also by commercial conference companies such as Computer Aid (CAI), Cutter, Digital Consulting Inc. (DCI), Software Productivity Group (SPG), or Technology Transfer Institute (TTI). There are also high-end conferences for executives sponsored both by research companies such as Gartner Group, and also by academia such as the Harvard Business School or the Sloan School.

In addition, there are vendor-hosted conferences for users by companies such as Apple, Microsoft, Computer Associates, Oracle, CADRE, COGNOS, SAS, SAP, and the like. These are often annual events that sometimes draw several thousand participants.

The author estimates that about 250,000 U.S. software professionals attended at least one conference in 2007, and some professionals may have attended multiple conferences. The year 2008 will probably see a drop due to the declining status of the U.S. economy.

Software conferences are where the cutting edge of software technologies are explained and sometimes revealed for the very first time. Many conferences also feature training seminars before or after the main event, and hence overlap commercial education channels and vendor education channels.

However, most of us go to conferences to find out what is new and exciting in our chosen domains. The mix of speakers and topics at conferences can range from brilliant to boring. Conferences are arranged with many concurrent sessions so that it is easy to leave a boring session and find a better one.

Most conferences combine keynote speeches to the whole audience with parallel sessions that cover more specialized topics. A typical U.S. software conference will run for three days, attract from 200 to more than 3000 attendees, and feature from 20 to more than 75 speakers.

In addition to primary speakers and seminar leaders, many conferences also have “vendor showcases,” where companies in related businesses can display their goods and perhaps make sales or at least contacts. The fees that vendors pay for participating in such conferences defray the administrative costs and sometimes even allow conferences to be run as profit-making opportunities.

On the whole, conferences do a good job in their primary role of exposing leading-edge technologies to the audience. You seldom come away from a conference with an in-depth working knowledge of a new technology. But you often come away with a solid understanding of what technologies merit closer analysis.

Within recent years, several conferences have become so large that the proceedings are now starting to be released on CD-ROM rather than in traditional notebooks or bound hard copies.

In the future, it is possible that live conferences will merge with webinars and with DVD production. It is technically possible to have simultaneous live events and webinars so that the speakers are seen by a live audience and a remote audience at the same time. It would also be possible to record the proceedings for later offline use or for downloading to computers and hand-held devices.

If the economy continues to decline and live conferences lose attendees, as happened during the dot-com crash and earlier recessions, it may be that webinars will become effective substitutes for live instruction.

Number 9: Wiki Sites

Costs = 2; Efficiency = 9; Effectiveness = 16; Currency = 3; Overall Score = 7.50

Prognosis: Increasing rapidly in usage; increasing in effectiveness

The word “wiki” means “fast” in Hawaiian. The term has started to be applied to web sites that allow multiple people to participate toward common goals.

Wiki sites are making their first appearance in this list of educational channels. This is a new and emerging technology that allows many participants to collaborate on a common theme or common undertaking.

The most famous example of a wiki site is Wikipedia, which has become the largest encyclopedia in the world. Each entry or article in Wikipedia is written by a volunteer. Readers or other volunteers can modify the entry and input their own thoughts and ideas.

At first glance, wikis would seem to lead to chaotic and perhaps offensive final products. But many wikis, including Wikipedia, are temperate in tone and rich in content. This is partly because readers can immediately delete offensive comments.

Some wiki sites try to screen and monitor inputs and changes before posting them; others simply allow the material to be self-corrected by other readers. Both methods seem to be effective.

What wiki sites do best is allow people with common interests to quickly produce documents or web sites that contain their accumulated and shared wisdom. Thus wiki sites would be very appropriate for technical issues such as Agile development, software quality, software security, testing, maintenance, and other topics where there are many practitioners with a need to share new data and experiences.

Number 10: Simulation Web Sites

Costs = 8; Efficiency = 7; Effectiveness = 13; Currency = 8; Overall Score = 9.00

Prognosis: Increasing rapidly in usage; increasing rapidly in effectiveness

Simulators for teaching mechanical skills such as how to fly an airplane or how to assemble a machine gun have been used by commercial and military organizations for many years.

But in recent years, new forms of broader simulation sites have emerged, such as Second Life, which is a kind of virtual world where avatars (symbolic surrogates for computer users) wander through a virtual landscape and interact with other avatars and with resources such as text documents and graphics.

This new form of virtual reality has other purposes besides training and education, but it is starting to move in the direction of education. For example, it would be possible to teach formal design and code inspections using avatars in a virtual room. Not only could inspections be taught via simulation, but indeed they could be performed as well.

Many other new technologies could also be taught in the same fashion. Currently, the costs of producing tutorial materials are fairly high and

the process is complex, but both of these issues should be eliminated fairly soon.

Virtual reality in simulated worlds is likely to become a fast-moving educational method within the next five years. It is theoretically possible to create a virtual “university” in which avatars for both students and professors will interact exclusively in online environments.

Although simulation and virtual worlds are not in the mainstream of education as of 2009, the odds are good that they will rise to the top of the list within ten years.

Number 11: Software Journals

Costs = 7; Efficiency = 11; Effectiveness = 14; Currency = 9; Overall Score = 10.25

Prognosis: Declining in numbers due to recession; stable in effectiveness; migrating rapidly from paper to online publication formats; some migration to e-book formats

Now that journals are available for the Amazon Kindle and other e-book platforms, it can be anticipated that e-journals will begin to replace paper journals due to economic reasons. Electronic publishing is much quicker and cheaper than paper publishing, and far more friendly to the environment. Further, electronic journals are distributed within minutes, so there is no delay due to surface mailing.

Conventional software journals tend to rank 14th in transferring skills. The main strength of software journals is in popularizing concepts and presenting the surfaces of technologies rather than the depths of technologies.

There are scores of software-related journals. Some are commercial journals published for profit and depending upon advertising revenues. Many others are produced by nonprofit professional associations. For example, *IEEE Computer* is produced by a nonprofit association as are the other IEEE journals.

A literate and very broad-based software journal is published by the U.S. Air Force Software Technology Support Center (STSC). This journal, *CrossTalk*, has become one of the few software journals to strive for depth in articles, rather than shallow coverage consisting primarily of short quotes from industry figures.

Another interesting journal is the *Cutter Journal*, originally founded by software guru Ed Yourdon under the name of *American Programmer*. As with the *CrossTalk* journal, the *Cutter Journal* publishes full-length technical articles.

There are so many journals that a number of them are quite specialized and occupy fairly narrow niches. An example of one of these specialized niche journals is *Metric Views*, the journal of the International Function Point Users Group.

Many software journals are available to software professionals for free, providing the potential subscribers bother to fill out a questionnaire (and have some responsibility for acquiring tools or software). On the other hand, some journals have annual subscriptions that can exceed \$1000.

The software professional ends up with subscriptions to quite a few journals, even though few may actually be read. SPR estimates that software technical personnel subscribe to or have access to about four journals per month on average.

Comparatively few software journals contain articles of lasting technical value. When journals do discuss technologies, it is seldom an in-depth treatment. This is understandable, given that neither journalists nor professional contributors can spare more than a minimum amount of time to assemble information before writing articles.

The best articles in terms of technology transfer are those on specific topics, often in special issues. For example, several journals have had special issues on topics such as quality assurance, measurement and metrics, software maintenance, object-oriented approaches, change management, and the like.

The least effective articles are the typical broad-brush surveys produced by journalists that consist largely of short quotes from 20 to 50 different people. This approach can seldom do more than bring a topic into public view.

An interesting new trend is to publish online journals as well as paper journals. Among the best and most interesting of the new online journals is the *Information Technology Metrics and Productivity Institute (ITMPI) Journal*. The web site for this journal is www.ITMPI.org. This journal is published by Computer Aid and includes interviews with famous software personages such as Watts Humphrey and Ed Yourdon, technical articles, and citations to hundreds of relevant web sites.

As online communication becomes the pervasive medium for information exchange, both software journals and other print-based information media are moving to create parallel online versions.

Number 12: Self-Study Using Books, E-Books, and Training Material

Costs = 5; Efficiency = 13; Effectiveness = 12; Currency = 11; Overall Score = 10.25

Prognosis: Decreasing in numbers due to recession; stable in effectiveness; rapidly migrating away from paper documents toward DVD or online information

The self-study market using books ranks number 11 in overall effectiveness. The market for traditional self-study courses is not fast growing, but has been relatively solid and stable for 50 years. The author

estimates that a total of about 50,000 software professionals will take some kind of self-study course over a normal business year.

The usual format of self-study material is a loose-leaf notebook. This form of self-study material can be effective for those who are self-motivated, but tends to be bland and frequently boring. Some self-study courses also include video- or audiocassettes.

The most common topics for self-study are those that have relatively large market potentials. This means that basic subjects such as “Introduction to COBOL Programming” are most likely to be found in self-study form.

Really new technologies are seldom found as self-study courses, because of the time required to produce the materials and the uncertainty of the market. There are always exceptions to rules, and fairly new topics such as ISO 9000-9004 standards have already arrived in self-study form due to the obviously large market.

In theory, electronic books on hand-held devices would seem to be a useful medium for studies of any kind. However, among the author’s clients, no formal course materials were produced for such devices as of 2009. We did not encounter enough students using these devices to form an opinion. Overall, usage of hand-held reading devices is still at a very early stage of deployment, although they show great potential for the future.

From 2006 through 2009, there has been an increase in books and video materials available on hand-held devices such as PDAs and Apple iPods. Some educational material is available, but the volume is too small to form a solid opinion of effectiveness.

The new Amazon and Sony hand-held reading devices of 2008 are more sophisticated than their predecessors. If these devices succeed, then it is a sign that electronic reading devices are about to enter the mainstream. The fact that one device can hold many books will certainly prove advantageous to travelers and even to sales personnel.

Number 13: On-the-Job Training

Costs = 11; Efficiency = 10; Effectiveness = 10; Currency = 12; Overall Score = 10.75

Prognosis: Declining in numbers due to the recession

On-the-job training has been most often utilized either for special techniques developed and used within companies, or for basic skills that are seldom taught at universities, such as formal design and code inspections.

With on-the-job training, new hires are instructed in selected methods by experienced users of the method. The most effective kinds of on-the-job training are topics where performing a task is the best way to learn the tasks. For example, formal inspections, quality function

deployment (QFD), learning to use proprietary specification methods, and learning to use programming languages that are unique to specific companies are good choices for in-house training.

The downside of on-the-job training is that the experts who are doing the training need to take time away from their own work. This is why the costs are fairly high. Also, it sometimes happens that what the older employees teach the new employees may be somewhat out of date.

As the recession lengthens and deepens, on-the-job training will probably decline due to layoffs and downsizing. The remaining personnel may no longer have time to engage in extensive on-the-job training.

Number 14: Mentoring

Costs = 10; Efficiency = 12; Effectiveness = 11; Currency = 15; Overall Score = 11.75

Prognosis: Declining in numbers due to recession

The concept of “mentoring” involves a one-to-one relationship between a new employee and an older and more experienced employee. Mentoring is most often used for proprietary methods, such as teaching a new employee about local corporate standards, or teaching a new employee about custom tools that are only available in one location.

Mentoring is often effective at a social level, but it is somewhat expensive if it requires that the mentor take too much time away from his or her normal work.

Informal mentoring is likely to continue as a useful method for teaching new personnel local methods, tools, development methods, maintenance methods, and other topics that may have been customized.

Unfortunately, there are no statistics on the number of mentors or people being mentored, so this channel of learning is ambiguous as to usage and overall effectiveness.

Number 15: Professional Books, Monographs, and Technical Reports

Costs = 6; Efficiency = 14; Effectiveness = 16; Currency = 13; Overall Score = 12.00

Prognosis: Decline in paper publication; rapid migration to e-book, online, and web publication

Software books tend to rank 15th in overall effectiveness in transferring software skills in the United States. The low ranking is not because of the books themselves, but because the U.S. software industry does not appear to be especially literate, which is something of a surprise given the nature of the work.

Many of the books are excellent. For example, Dr. Barry Boehm’s classic *Software Engineering Economics* (Prentice Hall), Watts Humphrey’s book on *Team Software Processes (TSP)* (Addison Wesley), and Dr. Roger

Pressman's classic *Software Engineering—A Practitioner's Approach* (McGraw-Hill) have all seen numerous editions and sold close to 1 million copies each.

Software books are a staple for reference purposes even if not for pedagogical purposes. A typical software engineer will have a library of between 20 and 100 volumes on topics of importance such as the programming languages, operating systems, applications, and hardware used in daily work.

As of 2009, there are more than 3 million software personnel in the United States, and the numbers are growing fairly rapidly. Yet software book publishers regard sales of more than 10,000 copies as a fairly good volume. Sales of 25,000 copies are remarkably good for software titles, and only a few software titles have approached 1 million copies.

The high-volume software books often aim at the end-user market and not the professional or education markets. For example, books on Visual Basic or primers on Windows can exceed 1 million copies in sales because there are about 10 million users of these products who are not professional software personnel.

It is possible to learn a variety of software-related skills from books, but this approach is not as widely utilized as seminars or some kind of formal training. One possible exception is that of learning personal topics, such as Watts Humphrey's Personal Software Process (PSP) method. As of 2009, Watts' books are the primary channel for this topic.

Another situation where books are actually used for self-learning of new skills is the area of new programming languages. New programming languages have been coming out at a rate of more than one per month for the past 30 years, and more than 700 programming languages have been listed in the Software Productivity Research (SPR) "table of programming languages and levels." Not many languages reach the mainstream, but for those that do, such as Ruby or N or E, experienced programmers can teach themselves the new language from books faster than from most others methods.

There are many excellent software books on the market by publishers such as Addison Wesley Longman, Auerbach, Dorset House, IEEE Computer Society Press, McGraw-Hill, Prentice Hall, Que, Microsoft Press, and the like.

Also included under the heading of books would be monographs published by various companies and associations. For example, software-related monographs of 50 to more than 100 pages in size are published by Accenture, IBM, IFPUG, ISBSG, various IEEE associations, McKinsey, Gartner Group, Meta Group, the Software Engineering Institute (SEI), and Software Productivity Research (SPR).

These privately published monographs are distributed primarily to the clients of the publishing organization. The costs range from zero up

to more than \$25,000 based on whether the monograph is intended as a marketing tool, or contains proprietary or special data of interest to the clients.

There are many software bookstores and large software sections within general bookstores such as Borders. And of course software books are featured in all of the major web-based bookstores such as Amazon and Barnes & Noble. The total volume of good books on software topics probably exceeds 2,500 titles for technical books and 250 titles for software management books.

Yet in spite of the plentiful availability of titles, many software managers and quite a few technical software personnel don't read more than one or two technical books a year, based on responses to assessment interviews.

The author estimates that software professionals purchase about four books per year on average (more than seem to be read). In any case, it would be hard to keep current just from books since software technology changes are so volatile.

There is a curious omission in the book domain for software and project management topics. Among the more mature professions such as medicine and law, a significant number of books are available in audio form such as cassettes or CDs so they can be listened to at home or in automobiles. We have not yet encountered any audio titles in the software engineering or project management fields, although some may be available.

An increasing number of technical books are becoming available online and can be viewed on personal computer screens or downloaded. This method of gaining access to books is expanding, but has not yet reached a critical mass. The author estimates that fewer than perhaps 100 software-related titles are available in online form, and that perhaps fewer than 125,000 software professionals have utilized this channel of gaining access to books. However, both the number of titles and accesses should increase rapidly over the next ten years.

New electronic devices such as the Amazon and Sony hand-held book readers, the iPhone, and various PDA devices can be used to store books, although this is not yet a major market for book publishers.

Of course, personal ownership of books may not be necessary if a company or government agency maintains a good technical library. One interesting observation from the author's assessments over the years is that companies with large technical libraries have higher annual productivity rates than companies of the same size without such libraries.

Having a library is probably not a causative factor for high productivity, though. The most likely reason for the correlation is that companies with good technical libraries also tend to have better than average salary and compensation levels, so they select top performers for software occupations.

Vast numbers of monographs and technical reports usually range between 20 and 75 pages. Usually, such documents are devoted to a single topic, such as service-oriented architecture (SOA) or perhaps marketing issues.

Some companies such as Gartner Group in the United States and Research and Markets in Ireland do a surprisingly large business from marketing both paper and online versions of monographs and technical reports.

The primary markets for these shorter documents are either corporate libraries or executives interested in business trends and high enough in company hierarchies to be authorized to spend money on subscriptions or individual studies, many of which are far more expensive than ordinary books.

As tools for informing executives about emerging business trends, the technical report business is fairly effective. As tools for learning the specifics of technical topics such as testing or inspections, these shorter reports are not as effective as books.

One technical problem with books published on paper is that the software industry changes faster than the books can be revised and new editions brought out.

A possible new business model for book publishers as they migrate to e-books would be to sell subscriptions to updates at the same time as the book is originally downloaded. For example, a downloaded software textbook in e-book format might sell for \$25, and subscriptions to updates might sell for \$10 per year. Not only would e-books be cheaper than paper books, but offering them as subscriptions would lead to recurring revenue streams.

Number 16: Undergraduate University Education

Costs = 15; Efficiency = 15; Effectiveness = 3; Currency = 16; Overall Score = 12.25

Prognosis: May decline in numbers due to recession; stable in effectiveness; curricula lag technology changes by more than five years

From the author's studies, undergraduate university education was only number 15 in overall effectiveness for software professionals. Universities are often not very current in the topics that they teach. In general, university curricula lag the actual state of the art of software by between five years and ten years. This lag is because of the way universities develop their teaching materials and their curricula.

There are a number of critical topics where software education at the university level lags far behind what is truly needed to achieve professional status for software. The most glaring omissions include

1. Software security practices for safeguarding high-risk applications
2. Software quality control practices for minimizing delivered defects

3. Software measures and metrics for effective economic analysis
4. Software estimating and planning for on-time delivery and costs
5. Software architecture for optimizing use of reusable components
6. Software methods for effective renovation of legacy applications
7. Software technology evaluation and technology transfer

These gaps and omissions need to be quickly filled if software is to evolve from an art form into a true engineering discipline.

There are some exceptions to this rule that universities always lag. Many universities have established fairly close ties with the local business community and attempt to offer courses that match the needs of the area's software employers. For example, Stevens Institute of Technology in New Jersey has established close ties with AT&T and is offering a master's program that includes topics suggested by AT&T. Bentley College in the Boston area, Washington University in St. Louis, Georgia State University in Atlanta, St. Thomas University in the St. Paul area, and many other schools adjacent to large software producers have adopted similar policies of curricula based on the needs of major software employers.

An important strength of undergraduate education is that what gets taught tends to be used throughout the rest of the professional lives of the students. University education ranks number three in effectiveness, or the volume of information transmitted.

The author estimates that perhaps 95,000 U.S. software professionals will take university or college courses during a normal business year.

Having performed a consulting study on continuing software education, the author noted a few practical issues. The way companies fund tuition-refund programs is often remarkably cumbersome. Sometimes several layers of management approval are required. The courses themselves must be job-related within fairly narrow boundaries. Some companies reserve the option of having the students withdraw "because of the needs of the company."

Also, the tuition-refund policies are based on achieving passing grades. This is not an unreasonable policy, but it does raise the mathematical probability that the student will end up with significant out-of-pocket expenses.

On the whole, university training appears to be more expensive and less effective than in-house training, commercial education, or vendor education for practicing software professionals.

A former chairman of the ITT Corporation once noted in a speech that it took an average of about three years of in-house training and on-the-job experience before a newly graduated software engineer could be entrusted with serious project responsibilities. This was about two

years longer than the training period needed by electrical or mechanical engineers. The conclusion was that software engineering and computer science curricula lagged traditional engineering curricula in teaching subjects of practical value.

Indeed, a quick review of several university software engineering and computer science curricula found some serious gaps in academic training. Among the topics that seemingly went untaught were software cost estimating, design and code inspections, statistical quality control, maintenance of legacy applications, metrics and measurements, Six Sigma methods, risk and value analysis, function points, and joint application design (JAD) for requirements analysis. While basic technical topics are fairly good at the university level, topics associated with project management are far from state-of-the-art levels.

Universities are also being impacted by the recession, and the future of university training in terms of numbers of software engineering and computer science students and graduates is uncertain. Whether or not the recession will improve or degrade curricula is uncertain as this is written in 2009.

Number 17: Graduate University Education

Costs = 16; Efficiency = 16; Effectiveness = 2; Currency = 15; Overall Score = 12.25

Prognosis: May decline in numbers due to recession; stable in effectiveness; curricula lag technology changes by more than five years

Graduate education in software engineering or computer science unfortunately tends to bring up the rear and is ranked number 16. Graduate school does rank number two in effectiveness, and it does transmit a great deal of information to graduate students.

The downside is that a lot of the information that is transmitted may be obsolete, since university curricula often lag the business and technical worlds by five to ten years.

Graduate education could be improved by greater concentration on special topics such as the economics of software. Software costs are so heavily dominated by defect removal expenses, catastrophic failures, and huge cost and schedule overruns that there is a need to teach all MBA students as well as specialist software engineer and computer science graduate students the state of the art for defect prevention, defect removal, and cost of quality economics.

Also, software security problems are not only rampant in 2009, but also are becoming more numerous and more serious at a rapid pace. It is obvious that universities have lagged severely in this area, and also in the area of software quality control. For that matter, other key topics such as construction from reusable materials also lag at both the undergraduate and graduate levels.

There appears to be a significant need to improve the speed at which universities are able to incorporate new material and new technologies into their curricula. For fast-moving industries such as computing and software, the technologies are changing much faster than university curricula.

For that matter, the recession has demonstrated that economic and financial curricula are not only severely out of date, but severely in error as well.

Software Areas Where Additional Education Is Needed

From working as an expert witness in a number of software lawsuits for breach of contract or for litigation involving claims of poor quality, the author finds that some important topics need additional education or reeducation. Table 4-6 shows 25 major technology areas where

TABLE 4-6 Gaps in Software Training Circa 2009

1.	Security vulnerability prevention
2.	Security recovery after attacks
3.	Quality control (defect prevention)
4.	Quality control (defect removal)
5.	Quality estimating
6.	Quality measurement
7.	Measurement and metrics of software
8.	Change management
9.	Tracking of software projects
11.	Cost, quality, and schedule estimating
10.	Intellectual property protection
12.	Reuse of software artifacts
13.	Risk analysis and abatement
14.	Value analysis of software applications
15.	Technology analysis and technology transfer
16.	Renovation of legacy applications
17.	Requirements collection and analysis
18.	Software project management
19.	Formal inspections
22.	Test case design
20.	Performance analysis
21.	Customer support of software applications
23.	Contract and outsource agreements
24.	User training
25.	User documentation

performance in the software world seems to be deficient, in approximate order of importance to the software profession.

Failures and problems associated with these topics appear to be endemic in the software world, and especially so for large software applications. From the frequency with which large software projects fail, and the even larger frequency that have cost and schedule overruns, it can be concluded that training in software topics urgently needs major improvement.

New Directions in Software Learning

The global recession is causing thousands of organizations to reduce costs in order to stay in business. Almost all labor-intensive activities such as training are going to be scrutinized very carefully.

At the same time the technologies of virtual reality, e-books, and distributing information over hand-held devices are increasing in sophistication and numbers of users.

Within a period of perhaps ten years, the combination of recessionary pressures and technology changes will probably make major differences in software learning methods. Online web-based information, e-books, and hand-held devices will no doubt replace substantial volumes of paper-based materials.

In addition, virtual reality may introduce artificial classrooms and simulated universities where students and teachers interact through avatars rather than face to face in real buildings.

The increasing sophistication of intelligent agents and expert systems will probably improve the ability to scan vast quantities of online information. The fact that companies such as Google and Microsoft are rapidly converting paper books and documents into online text will also change the access to information.

However, software has a long way to go before it achieves the ease of use and sophistication of the legal and medical professions in terms of the organization and access to vital information. For example, there is no software equivalent to the Lexis-Nexis legal reference company as of 2009.

Over the next few years, changes in learning methods may undergo changes as profound as those introduced by the printing press and television. However, the quality of software information is still poor compared with the quality of information in more mature fields such as medicine and law. The severe shortage of quantitative data on productivity, schedules, quality, and costs makes software appear to be more of a craft than a true profession.

However, the technologies of mining information, consolidating knowledge, and making knowledge accessible are rapidly improving. The overall prognosis for learning and information transfer is positive,

but no doubt the formats for presenting the information will be very different in the future than in the past.

Summary and Conclusions

In any technical field, it is hard to keep up with the latest developments. Software technologies are changing very rapidly as the 21st century advances, and this makes it difficult to stay current.

Seventeen different channels are available to the software community for acquiring information on new software technologies. The most effective historical channel is in-house training within a major corporation, but that channel is only available to the corporation's employees. Webinars and web-based research are rapidly advancing in sophistication and are already very inexpensive.

The use of virtual reality or simulation web sites is another exciting prospect. It is technically possible for avatars of virtual students to participate in simulated virtual classrooms.

Of the other channels, two appear to have strong potential for the future: self-study utilizing CD-ROM or DVD technology, and online study using the World Wide Web or an information utility. These channels are beginning to expand rapidly in terms of information content and numbers of users. Channels using wireless connectivity and hand-held devices may also be added to the mix of learning methods, as already demonstrated by the new generation of e-book readers from Amazon and Sony.

As the Internet and online services grow in usage, entirely new methods of education may be created as a byproduct of large-scale international communication channels. In the future, some form of education may become available via satellite radio, although none is currently broadcast on that channel.

Unfortunately, given the large numbers of project failures in the software domain, all 16 channels put together are probably not yet enough to raise the level of software engineering and management competence to fully professional status.

Curricula of Software Management and Technical Topics

Shown next is the author's proposed full curricula of managerial and technical topics related to the software industry. The courses range from top-level executive seminars through detailed technical courses. The set of curricula is aimed at corporations with large software staffs. The instructors are assumed to be top experts in the field.

Curricula such as this are not static. As new topics and technologies emerge, the curricula should be updated at least on an annual basis, and perhaps even more often.

Software Curricula for Executives, Management, and Technical Personnel

Capers Jones (Copyright © 2007–2009 by Capers Jones. All rights reserved)

Executive Courses	Days	Sequence
Global Finance and Economics	1.00	1
Software Development Economics	1.00	2
Software Maintenance Economics	1.00	3
Software Security Issues in 2008	1.00	4
Software Architecture Trends	1.00	5
Economics of Outsourcing	1.00	6
Pros and Cons of Offshore Outsourcing	1.00	7
Protecting Intellectual Property	0.50	8
Sarbanes-Oxley Compliance	1.00	9
Software Litigation Avoidance	0.50	10
Case Studies in Best Practices	0.50	11
Software Risk Avoidance	0.50	12
Case Studies in Software Failures	0.50	13
Overview of the Capability Maturity Model	0.25	14
Economics of Six Sigma	0.25	15
Overview of Viruses and Spyware	0.50	16
TOTAL	11.50	
Project Management Courses	Days	Sequence
Software Project Planning	2.00	1
Measurement and Metrics of Software	2.00	2
Software Cost Estimating: Manual	1.00	3
Software Cost Estimating: Automated	2.00	4
Software Quality and Defect Estimating	1.00	5
Software Security Planning	1.00	6
Software Milestone Tracking	1.00	7
Software Cost Tracking	1.00	8
Software Defect Tracking	1.00	9
Software Change Control	1.00	10
Function Points for Managers	0.50	11
Inspections for Project Managers	0.50	12
Testing for Project Managers	2.00	13
Six Sigma for Managers	2.00	14
Principles of TSP/PSP for Managers	1.00	15
Principles of Balanced Scorecards	1.00	16
Principles of Software Reuse	1.00	17
Software Risk Management	1.00	18
The Capability Maturity Model (CMM)	2.00	19
Six Sigma: Green Belt	3.00	20
Six Sigma: Black Belt	3.00	21

(Continued)

Project Management Courses	Days	Sequence
Earned Value Measurement	1.00	22
Appraisals and Employee Relations	1.00	23
Project Management Body of Knowledge	2.00	24
TOTAL	34.00	
Software Development Courses	Days	Sequence
Software Architecture Principles	1.00	1
Structured Development	2.00	2
Error-Prone Module Avoidance	1.00	3
Software Requirements Analysis	1.00	4
Software Change Control	1.00	5
Security Issues in 2008	1.00	6
Hacking and Virus Protection	2.00	7
Joint Application Design (JAD)	1.00	8
Static Analysis of Code	1.00	9
Formal Design Inspections	2.00	10
Formal Code Inspections	2.00	11
Test Case Design and Construction	2.00	12
Test Coverage Analysis	1.00	13
Reducing Bad Fix Injections	1.00	14
Defect Reporting and Tracking	1.00	15
Iterative and Spiral Development	1.00	16
Agile Development Methods	2.00	17
Using Scrum	1.00	18
Object-Oriented Design	2.00	19
Object-Oriented Development	2.00	20
Web Application Design and Development	2.00	21
Extreme Programming (XP) Methods	2.00	22
Development Using TSP/PSP	2.00	23
Principles of Database Development	2.00	24
Function Points for Developers	1.00	25
Design of Reusable Code	2.00	26
Development of Reusable Code	2.00	27
TOTAL	41.00	
Software Maintenance Courses	Days	Sequence
Principles of Legacy Renovation	1.00	1
Error-Prone Module Removal	2.00	2
Complexity Analysis and Reduction	1.00	3
Identifying and Removing Security Flaws	2.00	4
Reducing Bad-Fix Injections	1.00	5
Defect Reporting and Analysis	0.50	6
Change Control	1.00	7
Configuration Control	1.00	8

Software Maintenance Courses	Days	Sequence
Software Maintenance Workflows	1.00	9
Mass Updates to Multiple Applications	1.00	10
Maintenance of COTS Packages	1.00	11
Maintenance of ERP Applications	1.00	12
Static Analysis of Code	1.00	13
Data Mining for Business Rules	1.00	14
Regression Testing	2.00	15
Test Library Control	2.00	16
Test Case Conflicts and Errors	2.00	17
Dead Code Isolation	1.00	18
Function Points for Maintenance	0.50	19
Reverse Engineering	1.00	20
Reengineering	1.00	21
Refactoring	0.50	22
Maintenance of Reusable Code	1.00	23
Object-Oriented Maintenance	1.00	24
Maintenance of Agile and Extreme Code	1.00	25
TOTAL	27.50	
Software Quality Assurance Courses	Days	Sequence
Error-Prone Module Analysis	2.00	1
Software Defect Estimating	1.00	2
Software Defect Removal Efficiency	1.00	3
Software Defect Tracking	1.00	4
Software Design Inspections	2.00	5
Software Code Inspections	2.00	6
Software Test Inspections	2.00	7
Static Analysis of Code	2.00	8
Software Security and Quality in 2008	2.00	9
Defect Removal Using TSP/PSP	2.00	10
Software Static Analysis	2.00	11
Software Test Case Design	2.00	12
Software Test Library Management	1.00	13
Reducing Bad-Fix Injections	1.00	14
Test Case Conflicts and Errors	1.00	15
Function Points for Quality Measurement	1.00	16
ISO 9000-9004 Quality Standards	1.00	17
Overview of the CMM	1.00	18
Quality Assurance of Software Reuse	1.00	19
Quality Assurance of COTS and ERP	1.00	20
Six Sigma: Green Belt	3.00	21
Six Sigma: Black Belt	3.00	22
TOTAL	35.00	

(Continued)

Software Testing Courses	Days	Sequence
Test Case Design	2.00	1
Test Library Control	2.00	2
Security Testing Overview	2.00	3
Test Schedule Estimating	1.00	4
Software Defect Estimating	1.00	5
Defect Removal Efficiency Measurement	1.00	6
Static Analysis and Testing	1.00	7
Test Coverage Analysis	1.00	8
Reducing Bad-Fix Injections	1.00	9
Identifying Error-Prone Modules	2.00	10
Database Test Design	1.00	11
Removal of Incorrect Test Cases	1.00	12
Fundamentals of Unit Testing	1.00	13
Fundamentals of Regression Testing	1.00	14
Fundamentals of Component Testing	1.00	15
Fundamentals of Stress Testing	1.00	16
Fundamentals of Virus Testing	2.00	17
Fundamentals of Lab Testing	1.00	18
Fundamentals of System Testing	2.00	19
Fundamentals of External Beta Testing	1.00	20
Test Case Conflicts and Errors	1.00	21
Testing Web Applications	1.00	22
Testing COTS Application Packages	1.00	23
Testing ERP Applications	1.00	24
Function Points for Test Measures	1.00	25
Testing Reusable Functions	1.00	26
TOTAL	32.00	
Software Project Office Courses	Days	Sequence
Software Project Planning	3.00	1
Software Cost Estimating	3.00	2
Software Defect Estimating	2.00	3
Function Point Analysis	3.00	4
Software Architecture Issues	1.00	5
Software Security Issues	1.00	6
Software Change Management	2.00	7
Software Configuration Control	2.00	8
Overview of Software Inspections	1.00	9
Overview of Software Testing	1.00	10
Software Measurement and Metrics	2.00	11
Outsource Contract Development	1.00	12
COTS Acquisition	1.00	13
ERP Acquisition and Deployment	2.00	14

Software Project Office Courses	Days	Sequence
Sarbanes-Oxley Compliance	2.00	15
Multicompany Outsourced Projects	2.00	16
Software Risk Management	2.00	17
Case Studies of Software Failures	1.00	18
Case Studies of Best Practices	1.00	19
Software Milestone Tracking	2.00	20
Software Cost Tracking	2.00	21
Software Defect Tracking	2.00	22
Supply Chain Estimating	2.00	23
Balanced Scorecard Measurements	1.00	24
Earned-Value Measurements	1.00	25
Six Sigma Measurements	1.00	26
TSP/PSP Measurements	1.00	27
Software Value Analysis	1.00	28
ISO 9000-9004 Quality Standards	1.00	29
Backfiring LOC to Function Points	1.00	30
Metrics Conversion	1.00	31
TOTAL	49.00	
TOTAL CURRICULA	Days	Classes
Executive Education	11.50	16
Project Management Education	34.00	24
Software Development	41.00	27
Software Maintenance	27.50	25
Software Quality Assurance	35.00	22
Software Testing	32.00	26
Software Project Office	49.00	31
TOTAL	230.00	171.00

Readings and References

- Boehm, Barry, Dr. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.
- Curtis, Bill, Dr. *Human Factors in Software Development*. Washington, DC: IEEE Computer Society, 1985.
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York: McGraw-Hill, 2008.
- Jones, Capers. *Estimating Software Costs*, Second Edition. New York: McGraw-Hill, 2007.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000.
- Humphrey, Watts S. *PSP—A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Pearson Education, 2005.

- Humphrey, Watts S. *TSP—Leading a Development Team*. Upper Saddle River, NJ: Pearson Education, 2006.
- Kawasaki, Guy. *Selling the Dream*. Collins Business, 1992.
- Pressman, Roger. *Software Engineering—A Practitioner's Approach*, Sixth Edition. New York: McGraw-Hill, 2005.
- Weinberg, Dr. Gerald. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
- Yourdon, Ed. *Decline and Fall of the American Programmer*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1992.
- Yourdon, Ed. *Rise and Resurrection of the American Programmer*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1996.

Software Team Organization and Specialization

Introduction

More than almost any other technical or engineering field, software development depends upon the human mind, upon human effort, and upon human organizations. From the day a project starts until it is retired perhaps 30 years later, human involvement is critical to every step in development, enhancement, maintenance, and customer support.

Software requirements are derived from human discussions of application features. Software architecture depends upon the knowledge of human specialists. Software design is based on human understanding augmented by tools that handle some of the mechanical aspects, but none of the intellectual aspects.

Software code is written line-by-line by craftspeople as custom artifacts and involves the highest quantity of human effort of any modern manufactured product. (Creating sculpture and building special products such as 12-meter racing yachts or custom furniture require similar amounts of manual effort by skilled artisans, but these are not mainstream products that are widely utilized by thousands of companies.)

Although automated static analysis tools and some forms of automated testing exist, the human mind is also a primary tool for finding bugs and security flaws. Both manual inspections and manual creation of test plans and test cases are used for over 95 percent of software applications, and for almost 100 percent of software applications larger than 1,000 function points in size. Unfortunately, both quality and security remain weak links for software.

As the economy sinks into global recession, the high costs and marginal quality and security of custom software development are going

to attract increasingly critical executive attention. It may well be that the global recession will provide a strong incentive to begin to migrate from custom development to construction from standard reusable components. The global recession may also provide motivation for designing more secure software with higher quality, and for moving toward higher levels of automation in quality control and security control.

In spite of the fact that software has the highest labor content of any manufactured product, the topic of software team organization structure is not well covered in the software literature.

There are anecdotal reports on the value of such topics as pair-programming, small self-organizing teams, Agile teams, colocated teams, matrix versus hierarchical organizations, project offices, and several others. But these reports lack quantification of results. It is hard to find empirical data that shows side-by-side results of different kinds of organizations for the same kinds of applications.

One of the larger collections of team-related information that is available to the general public is the set of reports and data published by the International Software Benchmarking Standards Group (ISBSG). For example, this organization has productivity and average application size for teams ranging between 1 and 20 personnel. They also have data on larger teams, with the exception that really large teams in excess of 500 people are seldom reported to any benchmark organization.

Quantifying Organizational Results

This chapter will deal with organizational issues in a somewhat unusual fashion. As various organization structures and sizes are discussed, information will be provided that attempts to show in quantified form a number of important topics:

1. Typical staffing complements in terms of managers, software engineers, and specialists.
2. The largest software projects that a specific organization size and type can handle.
3. The average size of software projects a specific organization size and type handles.
4. The average productivity rates observed with specific organization sizes and types.
5. The average development schedules observed with specific organization sizes and types.
6. The average quality rates observed with specific organization sizes and types.

7. Demographics, or the approximate usage of various organization structures.
8. Demographics in the sense of the kinds of specialists often deployed under various organizational structures.

Of course, there will be some overlap among various sizes and kinds of organization structures. The goal of the chapter is to narrow down the ranges of uncertainty and to show what forms of organization are best suited to software projects of various sizes and types.

Organizations in this chapter are discussed in terms of typical departmental sizes, starting with one-person projects and working upward to large, multinational, multidisciplinary teams that may have 1,000 personnel or more.

Observations of various kinds of organization structures are derived from on-site visits to a number of organizations over a multiyear period. Examples of some of the organizations visited by the author include Aetna Insurance, Apple, AT&T, Boeing, Computer Aid Incorporated (CAI), Electronic Data Systems (EDS), Exxon, Fidelity, Ford Motors, General Electric, Hartford Insurance, IBM, Microsoft, NASA, NSA, Sony, Texas Instruments, the U.S. Navy, and more than 100 other organizations.

Organization structures are important aspects of successful software projects, and a great deal more empirical study is needed on organizational topics.

The Separate Worlds of Information Technology and Systems Software

Many medium and large companies such as banks and insurance companies only have information technology (IT) organizations. While there are organizational problems and issues within such companies, there are larger problems and issues within companies such as Apple, Cisco, Google, IBM, Intel, Lockheed, Microsoft, Motorola, Oracle, Raytheon, SAP and the like, which develop systems and embedded software as well as IT software.

Within most companies that build both IT and systems software, the two organizations are completely different. Normally, the IT organization reports to a chief information officer (CIO). The systems software groups usually report to a chief technology officer (CTO).

The CIO and the CTO are usually at the same level, so neither has authority over the other. Very seldom do these two disparate software organizations share much in the way of training, tools, methodologies, or even programming languages. Often they are located in different buildings, or even in different countries.

Because the systems software organization tends to operate as a profit center, while the IT organizations tends to operate as a cost center, there is often friction and even some dislike between the two groups.

The systems software group brings in revenues, but the IT organization usually does not. The friction is made worse by the fact that compensation levels are often higher in the systems software domain than in the IT domain.

While there are significant differences between IT and systems software, there are also similarities. As the global recession intensifies and companies look for ways to save money, sharing information between IT and systems groups would seem to be advantageous.

Both sides need training in security, in quality assurance, in testing, and in software reusability. The two sides tend to be on different business cycles, so it is possible that the systems software side might be growing while the IT side is downsizing, or vice versa. Coordinating position openings between the two sides would be valuable in a recession.

Also valuable would be shared resources for certain skills that both sides use. For example, there is a chronic shortage of good technical writers, and there is no reason why technical communications could not serve the IT organization and the systems organization concurrently.

Other groups such as testing, database administration, and quality assurance might also serve both the systems and IT organizations.

So long as the recession is lowering sales volumes and triggering layoffs, organizations that employ both systems software and IT groups would find it advantageous to consider cooperation.

Both sides usually have less than optimal quality, although systems software is usually superior to IT applications in that respect. It is possible that methods such as PSP, TSP, formal inspections, static analysis, automated testing, and other sophisticated quality control methods could be used by both the IT side and the systems side, which would simplify training and also allow easier transfers of personnel from one side to the other.

Colocation vs. Distributed Development

The software engineering literature supports a hypothesis that development teams that are colocated in the same complex are more productive than distributed teams of the same size located in different cities or countries.

Indeed a study carried out by the author that dealt with large software applications such as operating systems and telecommunication systems noted that for each city added to the development of the same applications, productivity declined by about 5 percent compared with teams of identical sizes located in a single site.

The same study quantified the costs of travel from city to city. For one large telecommunications application that was developed jointly between six cities in Europe and one city in the United States, the actual costs of airfare and travel were higher than the costs of programming or coding. The overall team size for this application was about 250, and no fewer than 30 of these software engineers or specialists would be traveling from country to country every week, and did so for more than three years.

Unfortunately, the fact that colocation is beneficial for software is an indication that “software engineering” is a craft or art form rather than an engineering field. For most engineered products such as aircraft, automobiles, and cruise ships, many components and subcomponents are built by scores of subcontractors who are widely dispersed geographically. While these manufactured parts have to be in one location for final assembly, they do not have to be constructed in the same building to be cost-effective.

Software engineering lacks sufficient precision in both design and development to permit construction from parts that can be developed remotely and then delivered for final construction. Of course, software does involve both outsourcers and remote development teams, but the current results indicate lower productivity than for colocated teams.

The author’s study of remote development was done in the 1980s, before the Web and the Internet made communication easy across geographic boundaries.

Today in 2009, conference calls, webinars, wiki groups, Skype, and other high-bandwidth communication methods are readily available. In the future, even more sophisticated communication methods will become available.

It is possible to envision three separate development teams located eight hours apart, so that work on large applications could be transmitted from one time zone to another at the end of every shift. This would permit 24-hour development by switching the work to three different countries located eight hours apart. Given the sluggish multiyear development schedules of large software applications, this form of distributed development might cut schedules down by perhaps 60 percent compared with a single colocated team.

For this to happen, it is obvious that software would need to be an engineering discipline rather than a craft or art form, so that the separate teams could work in concert rather than damaging each other’s results. In particular, the architecture, design, and coding practices would have to be understood and shared by the teams at all three locations.

What might occur in the future would be a virtual development environment that was available 24 hours a day. In this environment, avatars of the development teams could communicate “face to face” by using either their own images or generic images. Live conversations via Skype or the

equivalent could also be used as well as e-mail and various specialized tools for activities such as remote design and code inspections.

In addition, suites of design tools and project planning tools would also be available in the virtual environment so that both technical and business discussions could take place without the need for expensive travel. In fact, a virtual war room with every team's status, bug reports, issues, schedules, and other project materials could be created that might even be more effective than today's colocated organizations.

The idea is to allow three separate teams located thousands of miles apart to operate with the same efficiency as colocated teams. It is also desirable for quality to be even better than today. Of course, with 24-hour development, schedules would be much shorter than they are today.

As of 2009, virtual environments are not yet at the level of sophistication needed to be effective for large system development. But as the recession lengthens, methods that lower costs (especially travel costs) need to be reevaluated at frequent intervals.

An even more sophisticated and effective form of software engineering involving distributed development would be that of just-in-time software engineering practices similar to those used on the construction of automobiles, aircraft, and large cruise ships.

In this case, there would need to be standard architectures that supported construction from reusable components. The components might either be already in stock, or developed by specialized vendors whose geographic locations might be anywhere on the planet.

The fundamental idea is that rather than custom design and custom coding, standard architectures and standard designs would allow construction from standard reusable components.

Of course, this idea involves many software engineering technical topics that don't fully exist in 2009, such as parts lists, standard interfaces, certification protocols for quality and security, and architectural methods that support reusable construction.

As of 2009, the development of custom-built software applications ranges between \$1,000 per function point and \$3,000 per function point. Software maintenance and enhancements range between about \$100 and \$500 per function point per year, forever. These high costs make software among the most expensive business "machines" ever created.

As the recession lengthens, it is obvious that the high costs of custom software development need to be analyzed and more cost-effective methods developed. A combination of certified reusable components that could be assembled by teams that are geographically dispersed could, in theory, lead to significant cost reductions and schedule reductions also.

A business goal for software engineers would be to bring software development costs down below \$100 per function point, and annual maintenance and enhancement costs below \$50 per function point.

A corollary business goal might be to reduce development schedules for 10,000–function point applications from today’s averages of greater than 36 calendar months down to 12 calendar months or less.

Defect potentials should be reduced from today’s averages of greater than 5.00 per function point down to less than 2.50 per function point. At the same time, average levels of defect removal efficiency should rise from today’s average of less than 85 percent up to greater than 95 percent, and ideally greater than 97 percent.

Colocation cannot achieve such major reductions in costs, schedules, and quality, but a combination of remote development, virtual development environments, and standard reusable components might well turn software engineering into a true engineering field, and also lower both development and maintenance costs by significant amounts.

The Challenge of Organizing Software Specialists

In a book that includes “software engineering” in the title, you might suppose that the majority of the audience at which the book is aimed are software engineers working on development of new applications. While such software engineers are a major part of the audience, they actually comprise less than one-third of the personnel who work on software in large corporations.

In today’s world of 2009, many companies have more personnel working on enhancing and modifying legacy applications than on new development. Some companies have about as many test personnel as they do conventional software engineering personnel—sometimes even more.

Some of the other software occupations are just as important as software engineers for leading software projects to a successful outcome. These other key staff members work side-by-side with software engineers, and major applications cannot be completed without their work. A few examples of other important and specialized skills employed on software projects include architects, business analysts, database administrators, test specialists, technical writers, quality assurance specialists, and security specialists.

As discussed in Chapter 4 and elsewhere, the topic of software specialization is difficult to study because of inconsistencies in job titles, inconsistencies in job descriptions, and the use of abstract titles such as “member of the technical staff” that might encompass as many as 20 different jobs and occupations.

In this chapter, we deal with an important issue. In the presence of so many diverse skills and occupations, all of which are necessary for software projects, what is the best way to handle organization structures?

Should these specialists be embedded in hierarchical structures? Should they be part of matrix software organization structures and report in to their own chain of command while reporting via “dotted lines” to project managers? Should they be part of small self-organizing teams?

This topic of organizing specialists is surprisingly ambiguous as of 2009 and has very little solid data based on empirical studies. A few solid facts are known, however:

1. Quality assurance personnel need to be protected from coercion in order to maintain a truly objective view of quality and to report honestly on problems. Therefore, the QA organization needs to be separate from the development organization all the way up to the level of a senior vice president of quality.
2. Because the work of maintenance and bug repairs is rather different from the work of new development, large corporations that have extensive portfolios of legacy software applications should consider using separate maintenance departments for bug repairs.
3. Some specialists such as technical writers would have little opportunity for promotion or job enrichment if embedded in departments staffed primarily by software engineers. Therefore, a separate technical publications organization would provide better career opportunities.

The fundamental question for specialists is whether they should be organized in skill-based units with others who share the same skills and job titles, or embedded in functional departments where they will actually exercise those skills.

The advantage of skill-based units is that they offer specialists wider career opportunities and better educational opportunities. Also, in case of injury or incapacity, the skill-based organizations can usually assign someone else to take over.

The advantage of the functional organization where specialists are embedded in larger units with many other kinds of skills is that the specialists are immediately available for the work of the unit.

In general, if there are a great many of a certain kind of specialist (technical writers, testers, quality assurance, etc.), the skill-based organizations seem advantageous. But for rare skills, there may not be enough people in the same occupation for a skill-based group to even be created (i.e., security, architecture, etc.).

In this chapter, we will consider various alternative methods for dealing with the organization of key specialists associated with software. There are more than 120 software-related specialties in all, and for some of these, there may only be one or two employed even in fairly large companies.

This chapter concentrates on key specialties whose work is critical to the success of large applications in large companies. Assume the software organization in a fairly large company employs a total of 1,000 personnel. In this total of 1,000 people, how many different kinds of specialists and how many specific individuals are likely to be employed? For that matter, what are the specialists that are most important to success? Table 5-1 identifies a number of these important specialists and the approximate distribution out of a total of 1,000 software personnel.

TABLE 5-1 Distribution of Software Specialists for 1,000 Total Software Staff

	Number	Percent
1. Maintenance specialists	315	31.50%
2. Development software engineers	275	27.50%
3. Testing specialists	125	12.50%
4. First-line managers	120	12.00%
5. Quality assurance specialists	25	2.50%
6. Technical writing specialists	23	2.30%
7. Customer support specialists	20	2.00%
8. Configuration control specialists	15	1.50%
9. Second-line managers	9	0.90%
10. Business analysts	8	0.80%
11. Scope managers	7	0.70%
12. Administrative support	7	0.70%
13. Project librarians	5	0.50%
14. Project planning specialists	5	0.50%
15. Architects	4	0.40%
16. User interface specialists	4	0.40%
17. Cost estimating specialists	3	0.30%
18. Measurement/metric specialists	3	0.30%
19. Database administration specialists	3	0.30%
20. Nationalization specialists	3	0.30%
21. Graphical artists	3	0.30%
22. Performance specialists	3	0.30%
23. Security specialists	3	0.30%
24. Integration specialists	3	0.30%
25. Encryption specialists	2	0.20%
26. Reusability specialists	2	0.20%
27. Test library control specialists	2	0.20%
28. Risk specialists	1	0.10%
29. Standards specialists	1	0.10%
30. Value analysis specialists	1	0.10%
TOTAL SOFTWARE EMPLOYMENT	1000	100.00%

As can be seen from Table 5-1, software engineers do not operate all by themselves. A variety of other skills are needed in order to develop and maintain software applications in the modern world. Indeed, as of 2009, the number and kinds of software specialists are increasing, although the recession may reduce the absolute number of software personnel if it lengthens and stays severe.

Software Organization Structures from Small to Large

The observed sizes of software organization structures range from a low of one individual up to a high that consists of multidisciplinary teams of 30 personnel or more.

For historical reasons, the “average” size of software teams tends to be about eight personnel reporting to a manager or team leader. However, both smaller and larger teams are quite common.

This section of Chapter 5 examines the sizes and attributes of software organization structures from small to large, starting with one-person projects.

One-Person Software Projects

The most common corporate purpose for one-person projects is that of carrying out maintenance and small enhancements to legacy software applications. For new development, building web sites is a typical one-person activity in a corporate context.

However, a fairly large number of one-person software companies actually develop small commercial software packages such as iPhone applications, shareware, freeware, computer games, and other small applications. In fact, quite a lot of innovative new software and product ideas originate from one-person companies.

Demographics Because small software maintenance projects are common, on any given day, probably close to 250,000 one-person projects are under way in the United States, with the majority being maintenance and enhancements.

In terms of one-person companies that produce small applications, the author estimates that as of 2009, there are probably more than 10,000 in the United States. This has been a surprisingly fruitful source of innovation, and is also a significant presence in the open-source, freeware, and shareware domains.

Project size The average size of new applications done by one-person projects is about 50 function points, and the maximum size is below 1,000

function points. For maintenance or defect repair work, the average size is less than 1 function point and seldom tops 5 function points. For enhancement to legacy applications, the average size is about 5 to 10 function points for each new feature added, and seldom tops 15 function points.

Productivity rates Productivity rates for one-person efforts are usually quite good, and top 30 function points per staff month. One caveat is that if the one-person development team also has to write user manuals and provide customer support, then productivity gets cut approximately in half.

Another caveat is that many one-person companies are home based. Therefore unexpected events such as a bout of flu, a new baby, or some other normal family event such as weddings and funerals can have a significant impact on the work at hand.

A third caveat is that one-person software projects are very sensitive to the skill and work practices of specific individuals. Controlled experiments indicate about a 10-to-1 difference between the best and worst results for tasks such as coding and bug removal. That being said, quite a few of the people who migrate into one-person positions tend to be at the high end of the competence and performance scale.

Schedules Development schedules for one-person maintenance and enhancement projects usually range between a day and a week. For new development by one person, schedules usually range between about two months and six months.

Quality The quality levels for one-person applications are not too bad. Defect potentials run to about 2.5 bugs per function point, and defect removal efficiency is about 90 percent. Therefore a small iPhone application of 25 function points might have a total of about 60 bugs, of which 6 will still be present at release.

Specialization You might think that one-person projects would be the domain of generalists, since it is obvious that special skills such as testing and documentation all have to be found in the same individual. However, one of the more surprising results of examining one-person projects is that many of them are carried out by people who are not software engineers or programmers at all.

For embedded and systems software, many one-person software projects are carried out by electrical engineers, telecommunication engineers, automotive engineers, or some other type of engineer. Even for business software, some one-person projects may be carried out by accountants, attorneys, business analysts, and other domain experts who are also able to program. This is one of the reasons why such a significant

number of inventions and new ideas flow from small companies and one-person projects.

Cautions and counter indications The major caution about one-person projects for either development or maintenance is lack of backup in case of illness or incapacity. If something should happen to that one person, work will stop completely.

A second caution is if the person developing software is a domain expert (i.e., accountant, business analyst, statistician, etc.) who is building an application for personal use in a corporation, there may be legal questions involving the ownership of the application should the employee leave the company.

A third caution is that there may be liability issues in case the software developed by a knowledge worker contains errors or does some kind of damage to the company or its clients.

Conclusions One-person projects are the norm and are quite effective for small enhancement updates and for maintenance changes to legacy applications.

Although one-person development projects must necessarily be rather small, a surprising number of innovations and good ideas have originated from brilliant individual practitioners.

Pair Programming for Software Development and Maintenance

The idea of pair-programming is for two software developers to share one computer and take turns doing the coding, while the other member of the team serves as an observer. The roles switch back and forth between the two at frequent intervals, such as perhaps every 30 minutes to an hour. The team member doing the coding is called the *driver* and the other member is the *navigator* or *observer*.

As of 2009, the results of pair programming are ambiguous. Several studies indicate fewer defects from pair programming, while others assert that development schedules are improved as well.

However, all of the experiments were fairly small in scale and fairly narrow in focus. For example, no known study of pair-programming defects compared the results against an individual programmer who used static analysis and automatic testing. Neither have studies compared top-gun individuals against average to mediocre pairs, or vice versa.

There are also no known studies that compare the quality results of pair programming against proven quality approaches such as formal design and code inspections, which have almost 50 years of empirical data available, and which also utilize the services of other people for finding software defects.

While many of the pair-programming experiments indicate shorter development schedules, none indicate reduced development effort or costs from having two people perform work that is normally performed by one person.

For pair programming to lower development costs, schedules would have to be reduced by more than 50 percent. However, experiments and data collected to date indicate schedule reductions of only about 15 percent to 30 percent, which would have the effect of raising development costs by more than 50 percent compared with a single individual doing the same work.

Pair-programming enthusiasts assert that better quality will compensate for higher development effort and costs, but that claim is not supported by studies that included static analysis, automatic testing, formal inspections, and other sophisticated defect removal methods. The fact that two developers who use manual defect removal methods might have lower defects than one developer using manual defect removal methods is interesting but unconvincing.

Pair programming might be an interesting and useful method for developing reusable components, which need to have very high quality and reliability, but where development effort and schedules are comparatively unimportant. However, Watts Humphrey's Team Software Process (TSP) is also an excellent choice for reusable components and has far more historical data available than pair programming does.

Subjectively, the pair-programming concept seems to be enjoyable to many who have experienced it. The social situation of having another colleague involved with complicated algorithms and code structures is perceived as being advantageous.

As the recession of 2009 continues to expand and layoffs become more numerous, it is very likely that pair programming will no longer be utilized, due to the fact that companies will be reducing software staffs down to minimal levels and can no longer afford the extra overhead.

Most of the literature on pair programming deals with colocation in a single office. However, remote pair-programming, where the partners are in different cities or countries, is occasionally cited.

Pair programming is an interesting form of collaboration, and collaboration is always needed for applications larger than about 100 function points in size.

In the context of test-driven development, one interesting variation of pair programming would be for one of the pair to write test cases and the other to write code, and then to switch roles.

Another area where pair programming has been used successfully is that of maintenance and bug repairs. One maintenance outsource company has organized their maintenance teams along the lines of an urban police station. The reason for this is that bugs come in at random

intervals, and there is always a need to have staff available when a new bug is reported, especially a new high-severity bug.

In the police model of maintenance, a dispatcher and several pairs of maintenance programmers work as partners, just as police detectives work as partners.

During defect analysis, having two team members working side by side speeds up finding the origins of reported bugs. Having two people work on the defect repairs as partners also speeds up the repair intervals and reduces bad-fix injections. (Historically, about 7 percent of attempts to repair a bug accidentally introduce a new bug in the fix itself. These are called *bad fixes*.)

In fact, pair programming for bug repairs and maintenance activities looks as if it may be the most effective use of pairs yet noted.

Demographics Because pair programming is an experimental approach, the method is not widely deployed. As the recession lengthens, there may be even less pair-programming. The author estimates that as of 2009, perhaps 500 to 1,000 pairs are currently active in the United States.

Project size The average size of new applications done by pair-programming teams is about 75 function points, and the maximum size is fewer than 1,000 function points. For maintenance or defect repair work, the average size is less than 1 function point. For enhancement to legacy applications, the average size is about 5 to 10 function points for each new feature added.

Productivity rates Productivity rates for pair-programming efforts are usually in the range of 16 to 20 function points per staff month or 30 percent less than the same project done by one person.

Pair-programming software projects are very sensitive to the skill and work practices of specific individuals. As previously mentioned, controlled experiments indicate about a 10-to-1 range difference between the best and worst results for tasks such as coding and bug removal by individual participants in such studies.

Some psychological studies of software personnel indicate a tendency toward introversion, which may make the pair-programming concept uncomfortable to some software engineers. The literature on pair programming does indicate social satisfaction.

Schedules Development schedules for pair-programming maintenance and enhancement projects usually range between a day and a week. For new development by pairs, schedules usually range between about two months and six months. Schedules tend to be about 10 percent to 30 percent shorter than one-person efforts for the same number of function points.

Quality The quality levels for pair-programming applications are not bad. Defect potentials run to about 2.5 bugs per function point, and defect removal efficiency is about 93 percent. Therefore, a small iPhone application of 25 function points might have a total of about 60 bugs, of which 4 will still be present at release. This is perhaps 15 percent better than individual developers using manual defect removal and testing. However, there is no current data that compares pair programming with individual programming efforts where automated static analysis and automated testing are part of the equation.

Specialization There are few studies to date on the role of specialization in a pair-programming context. However, there are reports of interesting distributions of effort. For example, one of the pair might write test cases while the other is coding, or one might write user stories while the other codes.

To date there are no studies of pair programming that concern teams with notably different backgrounds working on the same application; that is, a software engineer teamed with an electrical engineer or an automotive engineer; a software engineer teamed with a medical doctor; and so forth. The pairing of unlike disciplines would seem to be a topic that might be worth experimenting with.

Cautions and counter indications The topic of pair programming needs additional experimentation before it can become a mainstream approach, if indeed it ever does. The experiments need to include more sophisticated quality control, and also to compare top-gun individual programmers. The higher costs of pair programming are not likely to gain adherents during a strong recession.

Conclusions There is scarcely enough empirical data about pair programming to draw solid conclusions. Experiments and anecdotal results are generally favorable, but the experiments to date cover only a few variables and ignore important topics such as the role of static analysis, automatic testing, inspections, and other quality factors. As the global recession lengthens and deepens, pair programming may drop from view due to layoffs and downsizing of software organizations.

Self-Organizing Agile Teams

For several years, as the Agile movement gained adherents, the concept of small self-organizing teams also gained adherents. The concept of self-organized teams is that rather than have team members reporting to a manager or formal team leader, the members of the team would migrate to roles that they felt most comfortably matched their skills.

In a self-organizing team, every member will be a direct contributor to the final set of deliverables. In an ordinary department with a manager, the manager is usually not a direct contributor to the code to deliverables that reach end users. Therefore, self-organizing teams should be slightly more efficient than ordinary departments of the same size, because they would have one additional worker.

In U.S. businesses, ordinary departments average about eight employees per manager. The number of employees reporting to a manager is called the *span of control*. (The actual observed span of control within large companies such as IBM has ranged from a low of 2 to a high of 30 employees per manager.)

For self-organizing teams, the nominal range of size is about “7 plus or minus 2.” However, to truly match any given size of software project, team sizes need to range from a low of two up to a maximum of about 12.

A significant historical problem with software has been that of decomposing applications to fit existing organization structures, rather than decomposing the applications into logical pieces based on the fundamental architecture.

The practical effect has been to divide large applications into multiple segments that can be developed by an eight-person department whether or not that matches the architecture of the application.

In an Agile context, a user representative may be a member of the team and provides inputs as to the features that are needed, and also provides experiential reports based on running the pieces of the application as they are finished. The user representative has a special role and normally does not do any code development, although some test cases may be created by the embedded user representative. Obviously, the user will provide inputs in terms of user stories, use cases, and informal descriptions of the features that are needed.

In theory, self-organizing teams are cross-functional, and everyone contributes to every deliverable on an as-needed basis. However, it is not particularly effective for people to depart from their main areas of competence. Technical writers may not make good programmers. Very few people are good technical writers. Therefore, the best results tend to be achieved when team members follow their strengths.

However, in areas where everyone (or no one) is equally skilled, all can participate. Creating effective test cases may be an example where skills are somewhat sparse throughout. Dealing with security of code is an area where so few people are skilled that if it is a serious concern, outside expertise will probably have to be imported to support the team.

Another aspect of self-organizing teams is the usage of daily status meetings, which are called *Scrum sessions*, using a term derived from the game of rugby. Typically, Scrum sessions are short and deal with three key issues: (1) what has been accomplished since the last Scrum

session, (2) what is planned between today and the next Scrum session, and (3) what problems or obstacles have been encountered.

(Scrum is not the only method of meeting and sharing information. Phone calls, e-mails, and informal face-to-face meetings occur every day. There may also be somewhat larger meetings among multiple teams, on an as-needed basis.)

One of the controversial roles with self-organizing teams is that of *Scrum master*. Nominally, the Scrum master is a form of coordinator for the entire project and is charged with setting expectations for work that spans multiple team members; that is, the Scrum master is a sort of coach. This role means that the personality and leadership qualities of the Scrum master exert a strong influence on the overall team.

Demographics Because Agile has been on a rapid growth path for several years, the number of small Agile teams is still increasing. As of 2009, the author estimates that in the United States alone there are probably 35,000 small self-organizing teams that collectively employ about 250,000 software engineers and other occupations.

Project size The average size of new applications done by self-organizing teams with seven members is about 1,500 function points, and the maximum size is perhaps 3,000 function points. (Beyond 3,000 function points, teams of teams would be utilized.) Self-organizing teams are seldom used for maintenance or defect repair work, since a bug's average size is less than 1 function point and needs only one person. For enhancements to legacy applications, self-organizing teams might be used for major enhancements in the 150– to 500–function point range. For smaller enhancements of 5 to 10 function points, individuals would probably be used for coding, with perhaps some assistance from testers, technical writers, and integration specialists.

Although there are methods for scaling up small teams to encompass teams of teams, scaling has been a problem for self-organizing teams. In fact, the entire Agile philosophy seems better suited to applications below about 2,500 function points. Very few examples of large systems greater than 10,000 function points have even been attempted using Agile or self-organizing teams.

Productivity rates Productivity rates for self-organizing teams on projects of 1,500 function points are usually in the range of 15 function points per staff month. They sometimes top 20 function points per staff month for applications where the team has significant expertise and may drop below 10 function points per staff month for unusual or complex projects.

Productivity rates for individual sprints are higher, but that fact is somewhat irrelevant because the sprints do not include final integration of all components, system test of the entire application, and the final user documentation.

Self-organizing team projects tend to minimize the performance ranges of individuals and may help to bring novices up to speed fairly quickly. However, if the range of performance on a given team exceeds about 2-to-1, those at the high end of the performance range will become dissatisfied with the work of those at the low end of the range.

Schedules Development schedules for new development by self-organizing teams for typical 1,500–function point projects usually range between about 9 months and 18 months and would average perhaps 12 calendar months for the entire application.

However, the Agile approach is to divide the entire application into a set of segments that can be developed independently. These are called *sprints* and would typically be of a size that can be completed in perhaps one to three months. For an application of 1,500 function points, there might be five sprints of about 300 function points each. The schedule for each sprint might be around 2.5 calendar months.

Quality The quality levels for self-organizing teams are not bad, but usually don't achieve the levels of methods such as Team Software Process (TSP) where quality is a central issue. Typical defect potentials run to about 4.5 bugs per function point, and defect removal efficiency is about 92 percent.

Therefore, an application of 1,500 function points developed by a self-organizing Agile team might have a total of about 6,750 bugs, of which 540 would still be present at release. Of these, about 80 might be serious bugs.

However, if tools such as automated static analysis and automated testing are used, then defect removal efficiency can approach 97 percent. In this situation, only about 200 bugs might be present at release. Of these, perhaps 25 might be serious.

Specialization There are few studies to date on the role of specialization in self-organizing teams. Indeed, some enthusiasts of self-organizing teams encourage generalists. They tend to view specialization as being similar to working on an assembly line. However, generalists often have gaps in their training and experience. The kinds of specialists who might be useful would be security specialists, test specialists, quality assurance specialists, database specialists, user-interface specialists, network specialists, performance specialists, and technical writers.

Cautions and counter indications The main caution about self-organizing teams is that the lack of a standard and well-understood structure opens up the team to the chance of power struggles and disruptive social conflicts.

A second caution is that scaling Agile up from small applications to large systems with multiple teams in multiple locations has proven to be complicated and difficult.

A third caution is that the poor measurement practices associated with Agile and with many self-organizing teams give the method the aura of a cult rather than of an engineering discipline. The failure either to measure productivity or quality, or to report benchmarks using standard metrics is a serious deficiency.

Conclusions The literature and evidence for self-organizing Agile teams is somewhat mixed and ambiguous. For the first five years of the Agile expansion, self-organizing teams were garnering a majority of favorable if subjective articles.

Since about the beginning of 2007, on the other hand, an increasing number of articles and reports have appeared that raise questions about self-organizing teams and that even suggest that they be abolished due to confusion as to roles, disruptive power struggles within the teams, and outright failures of the projects.

This is a typical pattern within the software industry. New development methods are initially championed by charismatic individuals and start out by gaining a significant number of positive articles and positive books, usually without any empirical data or quantification of results.

After several years, problems begin to be noted, and increasing numbers of applications that use the method may fail or be unsuccessful. In part this may be due to poor training, but the primary reason is that almost no software development method is fully analyzed or used under controlled conditions prior to deployment. Poor measurement practices and a lack of benchmarks are also chronic problems that slow down evaluation of software methods.

Unfortunately, self-organizing teams originated in the context of Agile development. Agile has been rather poor in measuring either productivity or quality, and creates almost no effective benchmarks. When Agile projects are measured, they tend to use special metrics such as story points or use-case points, which are not standardized and lack empirical collections of data and benchmarks.

Team Software Process (TSP) Teams

The concept of Team Software Process (TSP) was developed by Watts Humphrey based on his experiences at IBM and as the originator of the capability maturity model (CMM) for the Software Engineering Institute (SEI).

The TSP concept deals with the roles and responsibilities needed to achieve successful software development. But TSP is built on individual skills and responsibilities, so it needs to be considered in context with the Personal Software Process (PSP). Usually, software engineers and specialists learn PSP first, and then move to TSP afterwards.

Because of the background of Watts Humphrey with IBM and with the capability maturity model, the TSP approach is congruent with the modern capability maturity model integrated (CMMI) and appears to satisfy many of the criteria for CMMI level 5, which is the top or highest level of the CMMI structure.

Because TSP teams are self-organizing teams, they have a surface resemblance to Agile teams, which are also self-organizing. However, the Agile teams tend to adopt varying free-form structures based on the skills and preferences of whoever is assigned to the team.

The TSP teams, on the other hand, are built on a solid underpinning of specific roles and responsibilities that remain constant from project to project. Therefore, with TSP teams, members are selected based on specific skill criteria that have been shown to be necessary for successful software projects. Employees who lack needed skills would probably not become members of TSP teams, unless training were available.

Also, prior training in PSP is mandatory for TSP teams. Other kinds of training such as estimating, inspections, and testing may also be used as precursors.

Another interesting difference between Agile teams and TSP teams is the starting point of the two approaches. The Agile methods were originated by practitioners whose main concerns were comparatively small IT applications of 1,500 or fewer function points. The TSP approach was originated by practitioners whose main concerns were large systems software applications of 10,000 or more function points.

The difference in starting points leads to some differences in skill sets and specialization. Because small applications use few specialists, Agile teams are often populated by generalists who can handle design, coding, testing, and even documentation on an as-needed basis.

Because TSP teams are often involved with large applications, they tend to utilize specialists for topics such as configuration control, integration, testing, and the like.

While both Agile and TSP share a concern for quality, they tend to go after quality in very different fashions. Some of the Agile methods are based on test-driven development, or creating test cases prior to creating the code. This approach is fairly effective. However, Agile tends to avoid formal inspections and is somewhat lax on recording defects and measuring quality.

With TSP, formal inspections of key deliverables are an integral part, as is formal testing. Another major difference is that TSP is very rigorous in

measuring every single defect encountered from the first day of requirements through delivery, while defect measures during Agile projects are somewhat sparse and usually don't occur before testing.

Both Agile and TSP may utilize automated defect tracking tools, and both may utilize approaches such as static analysis, automated testing, and automated test library controls.

Some other differences between Agile and TSP do not necessarily affect the outcomes of software projects, but they do affect what is known about those outcomes. Agile tends to be lax on measuring productivity and quality, while TSP is very rigorous in measuring task hours, earned value, defect counts, and many other quantified facts.

Therefore, when projects are finished, Agile projects have only vague and unconvincing data that demonstrates either productivity or quality results. TSP, on the other hand, has a significant amount of reliable quantified data available.

TSP can be utilized with both hierarchical and matrix organization structures, although hierarchical structures are perhaps more common. Watts Humphrey reports that TSP is used for many different kinds of software, including defense applications, civilian government applications, IT applications, commercial software in companies such as Oracle and Adobe, and even by some of the computer game companies, where TSP has proven to be useful in eliminating annoying bugs.

Demographics TSP is most widely used by large organizations that employ between perhaps 1,000 and 50,000 total software personnel. Because of the synergy between TSP and the CMMI, it is also widely used by military and defense software organizations. These large organizations tend to have scores of specialized skills and hundreds of projects going on at the same time.

The author estimates that there are about 500 companies in the United States now using TSP. While usage may be experimental in some of these companies, usage is growing fairly rapidly due to the success of the approach. The number of software personnel using TSP in 2009 is perhaps 125,000 in the United States.

Project size The average size of new applications done by TSP teams with eight employees and a manager is about 2,000 function points. However, TSP organizations can be scaled up to any arbitrary size, so even large systems in excess of 100,000 function points can be handled by TSP teams working in concert. For large applications with multiple TSP teams, some specialist teams such as testing, configuration control, and integration also support the general development teams.

Another caveat with multiple teams attempting to cooperate is that when more than about a dozen teams are involved simultaneously,

some kind of a project office may be needed for overall planning and coordination.

Productivity rates Productivity rates for TSP departments on projects of 2,000 function points are usually in the range of 14 to 18 function points per staff month. They sometimes top 22 function points per staff month for applications where the team has significant expertise, and may drop below 10 function points per staff month for unusual or complex projects. Productivity tends to be inversely proportional to application size and declines as applications grow larger.

Schedules Development schedules for new development by TSP groups with eight team members working on a 2,000-function point project usually range between about 12 months and 20 months and would average perhaps 14 calendar months for the entire application.

Quality The quality levels for TSP organizations are exceptionally good. Average defect potentials with TSP run to about 4.0 bugs per function point, and defect removal efficiency is about 97 percent. Delivered defects would average about 0.12 per function point.

Therefore, an application of 2,000 function points developed by a single TSP department might have a total of about 8,000 bugs, of which 240 would still be present at release. Of these, about 25 might be serious bugs.

However, if in addition to pretest inspections, tools such as automated static analysis and automated testing are used, then defect removal efficiency can approach 99 percent. In this situation, only about 80 bugs might be present at release. Of these, perhaps 8 might be serious bugs, which is a rate of only 0.004 per function point.

Generally, as application sizes increase, defect potentials also increase, while defect removal efficiency levels decline. Interestingly, with TSP, this rule may not apply. Some of the larger TSP applications achieve more or less the same quality as small applications.

Another surprising finding with TSP is that productivity does not seem to degrade significantly as application size goes up. Normally, productivity declines with application size, but Watts Humphrey reports no significant reductions across a wide range of application sizes. This assertion requires additional study, because that would make TSP unique among software development methods.

Specialization TSP envisions a wide variety of specialists. Most TSP teams will have numerous specialists for topics such as architecture, testing, security, database design, and many others.

Interestingly, the TSP approach does not recommend software quality assurance (SQA) as being part of a standard TSP team. This is because

of the view that the TSP team itself is so rigorous in quality control that SQA is not needed.

In companies where SQA groups are responsible for collecting quality data, TSP teams will provide such data as needed, but it will be collected by the team's own personnel rather than by an SQA person or staff assigned to the project.

Cautions and counter indications The main caution about TSP organizations and projects is that while they measure many important topics, they do not use standard metrics such as function points. The TSP use of task hours is more or less unique, and it is difficult to compare task hours against standard resource metrics.

Another caution is that few if any TSP projects have ever submitted benchmark data to any of the formal software benchmark groups such as the International Software Benchmarking Standards Group (ISBSG). As a result, it is almost impossible to compare TSP against other methods without doing complicated data conversion.

It is technically feasible to calculate function point totals using several of the new high-speed function point methods. In fact, quantifying function points for both new applications and legacy software now takes only a few minutes. Therefore, reporting on quality and productivity using function points would not be particularly difficult.

Converting task-hour data into normal workweek and work-month information would be somewhat more troublesome, but no doubt the data could be converted using algorithms or some sort of rule-based expert system.

It would probably be advantageous for both Agile and TSP projects to adopt high-speed function point methods and to submit benchmark results to one or more of the benchmark organizations such as ISBSG.

Conclusions The TSP approach tends to achieve a high level of successful applications and few if any failures. As a result, it deserves to be studied in depth.

From observations made during litigation for projects that failed or never operated successfully, TSP has not yet had failures that ended up in court. This may change as the number of TSP applications grows larger.

TSP emphasizes the competence of the managers and technical staff, and it emphasizes effective quality control and change management control. Effective estimating and careful progress tracking also are standard attributes of TSP projects. The fact that TSP personnel are carefully trained before starting to use the method, and that experienced mentors are usually available, explains why TSP is seldom misused.

With Agile, for example, there may be a dozen or more variations of how development activities are performed, but they still use the name

“Agile” as an umbrella term. TSP activities are more carefully defined and used, so when the author visited TSP teams in multiple companies, the same activities carried out the same way were noted.

Because of the emphasis on quality, TSP would be a good choice as the construction method for standard reusable components. It also seems to be a good choice for hazardous applications where poor quality might cause serious problems; that is, in medical systems, weapons systems, financial applications, and the like.

Conventional Departments with Hierarchical Organization Structures

The concept of hierarchical organizations is the oldest method for assigning social roles and responsibilities on the planet. The etymology of the word “hierarchy” is from the Greek, and the meaning is “rule by priests.” But the concept itself is older than Greece and was also found in Egypt, Sumer, and most other ancient civilizations.

Many religions are organized in hierarchical fashion, as are military organizations. Some businesses are hierarchical if they are privately owned. Public companies with shareholders are usually semi-hierarchical, in that the operating units report upward level-by-level to the president or chief executive officer (CEO). The CEO, however, reports to a board of directors elected by the shareholders, so the very top level of a public company is not exactly a true hierarchy.

In a hierarchical organization, units of various sizes each have a formal leader or manager who is appointed to the position by higher authorities. While the appointing authority is often the leader of the next highest level of organization in the structure, the actual power to appoint is usually delegated from the top of the hierarchy. Once appointed, each leader reports to the next highest leader in the same chain of command.

While appointed leaders or managers at various levels have authority to issue orders and to direct their own units, they are also required to adhere to directives that descend from higher authorities. Progress reports flow back up to higher authorities.

In business hierarchies, lower level managers are usually appointed by the manager of the next highest level. But for executive positions such as vice presidents the appointments may be made by a committee of top executives. The purpose of this, at least in theory, is to ensure the competence of the top executives of the hierarchy. However, the recent turmoil in the financial sector and the expanding global recession indicates that top management tends to be a weak link in far too many companies.

It should be noted that the actual hierarchical structure of an organization and its power structure may not be identical. For example,

in Japan during the Middle Ages, the emperor was at the top of the formal government hierarchy, but actual ruling power was vested in a military organization headed by a commander called the shogun. Only the emperor could appoint the shogun, but the specific appointment was dictated by the military leadership, and the emperor had almost no military or political power.

A longstanding issue with hierarchical organizations is that if the leader at the top of the pyramid is weak or incompetent, the entire structure may be at some risk of failing. For hierarchical governments, weak leadership may lead to revolutions or loss of territory to strong neighbors.

For hierarchical business organizations, weak leadership at the top tends to lead to loss of market share and perhaps to failure or bankruptcy. Indeed analysis of the recent business failures from Enron through Lehmann does indicate that the top of these hierarchies did not have the competence and insight necessary to deal with serious problems, or even to understand what the problems were.

It is an interesting business phenomenon that the life expectancy of a hierarchical corporation is approximately equal to the life expectancies of human beings. Very few companies live to be 100 years old. As the global recession lengthens and deepens, a great many companies are likely to expire, although some will expand and grow stronger.

A hierarchical organization has two broad classes of employees. One of these classes consists of the workers or specialists who actually do the work of the enterprise. The second class consists of the managers and executives to whom the workers report. Of course, managers also report to higher-level managers.

The distinction between technical work and managerial work is so deeply embedded in hierarchical organizations that it has created two very distinct career paths: management and technical work.

When starting out their careers, young employees almost always begin as technical workers. For software, this means starting out as software engineers, programmers, systems analysts, technical writers, and the like. After a few years of employment, workers need to make a career choice and either get promoted into management or stay with technical work.

The choice is usually determined by personality and personal interests. Many people like technical work and never want to get into management. Other people enjoy planning and coordination of group activities and opt for a management career.

There is an imbalance in the numbers of managers and technical workers. In most companies, the managerial community totals to about 15 percent of overall employment, while the technical workers total to about 85 percent. Since managers are not usually part of the production

process of the company, it is important not to have an excessive number of managers and executives. Too many managers and executives tend to degrade operational performance. This has been noted in both business and military organizations.

It is interesting that up to a certain point, the compensation levels of technical workers and managers are approximately the same. For example, in most corporations, the top technical workers can have compensation that equals third-line managers. However, at the very top of corporations, there is a huge imbalance.

The CEOs of a number of corporations and some executive vice presidents have compensation packages that are worth millions of dollars. In fact, some executive compensation packages are more than 250 times the compensation of the average worker within the company. As the global recession deepens, these enormous executive compensation packages are being challenged by both shareholders and government regulators.

Another topic that is beginning to be questioned is the span of control, or the number of technical workers who report to one manager. For historical reasons that are somewhat ambiguous, the average department in the United States has about eight technical workers reporting to one manager. The ranges observed run from two employees per manager to about 30 employees per manager.

Assuming an average of eight technical workers per manager, then about 12.5 percent of total employment would be in the form of first-line managers. When higher-level managers are included, the overall total is about 15 percent.

From analyzing appraisal scores and examining complaints against managers in large corporations, it appears that somewhat less than 15 percent of the human population is qualified to be effective in management. In fact, only about 10 percent (or less) seem to be qualified to be effective in management.

That being said, it might be of interest to study raising the average span of control from 8 workers per manager up to perhaps 12 workers per manager. Weeding out unqualified managers and restoring them to technical work might improve overall efficiency and reduce the social discomfort caused by poor management.

Practicing managers state that increasing the span of control would lower their ability to control projects and understand the actual work of their subordinates. However, time and motion studies carried out by the author in large corporations such as IBM found that software managers tended to spend more time in meetings with other managers than in discussions or meetings with their own employees. In fact, a possible law of business is "managerial meetings are inversely proportional to the span of control." The more managers on a given project, the more time they spend with other managers rather than with their own employees.

Another and more controversial aspect of this study had to do with project failure rates, delays, and other mishaps. For large projects with multiple managers, the failure rates seem to correlate more closely to the number of managers involved with the projects than with the number of software engineers and technical workers.

While the technical workers often managed to do their jobs and get along with their colleagues in other departments, managerial efforts tend to be diluted by power struggles and debates with other managers.

This study needs additional research and validation. However, it led to the conclusion that increasing the span of control and reducing managerial numbers tends to raise the odds of a successful software project outcome. This would especially be true if the displaced managers happened to be those of marginal competence for managerial work.

In many hierarchical departments with generalists, the same people do both development and maintenance. It should be noted that if the same software engineers are responsible for both development and maintenance concurrently, it will be very difficult to estimate their development work with accuracy. This is because maintenance work involved with fixing high-severity defects tends to preempt software development tasks and therefore disrupts development schedules.

Another topic of significance is that when exit interviews are reviewed for technical workers, two troubling facts are noted: (1) technical workers with the highest appraisal scores tend to leave in the largest numbers; and (2) the most common reason cited for leaving a company is "I don't like working for bad management."

Another interesting phenomenon about management in hierarchical organizations is termed "the Peter Principle" and needs to be mentioned briefly. The Peter Principle was created by Dr. Lawrence J. Peter and Raymond Hull in the 1968 book of the same name. In essence, the Peter Principle holds that in hierarchical organizations, workers and managers are promoted based on their competence and continue to receive promotions until they reach a level where they are no longer competent. As a result, a significant percentage of older employees and managers occupy jobs for which they are not competent.

The Peter Principle may be amusing (it was first published in a humorous book), but given the very large number of cancelled software projects and the even larger number of schedule delays and cost overruns, it cannot be ignored or discounted in a software context.

Assuming that the atomic unit of a hierarchical software organization consists of eight workers who report to one manager, what are their titles, roles, and responsibilities?

Normally, the hierarchical mode of organization is found in companies that utilize more generalists than specialists. Because software specialization tends to increase with company size, the implication is

that hierarchical organizations are most widely deployed for small to midsize companies with small technical staffs. Most often, hierarchical organizations are found in companies that employ between about 5 and 50 software personnel.

The primary job title in a hierarchical structure would be programmer or software engineer, and such personnel would handle both development and maintenance work.

However, the hierarchical organization is also found in larger companies and in companies that do have specialists. In this case, an eight-person department might have a staffing complement of five software engineers, two testers, and a technical writer all reporting to the same manager.

Large corporations have multiple business units such as marketing, sales, finance, human resources, manufacturing, and perhaps research. Using hierarchical principles, each of these might have its own software organization dedicated to building the software used by a specific business unit; that is, financial applications, manufacturing support applications, and so forth.

But what happens when some kind of a corporate or enterprise application is needed that cuts across all business units? Cross-functional applications turned out to be difficult in traditional hierarchical or “stovepipe” organizations.

Two alternative approaches were developed to deal with cross-functional applications. Matrix management was one, and it will be discussed in the next section of this chapter. The second was enterprise resource planning (ERP) packages, which were created by large software vendors such as SAP and Oracle to handle cross-functional business applications.

As discussed in the next topic, the matrix-management organization style is often utilized for software groups with extensive specialization and a need for cross-functional applications that support multiple business units.

Demographics In the software world, hierarchical organizations are found most often in small companies that employ between perhaps 5 and 50 total software personnel. These companies tend to adopt a generalist philosophy and have few specialists other than some technical skills such as network administration and technical writing. In a generalist context, hierarchical organizations of about five to eight software engineers reporting to a manager handle development, testing, and maintenance activities concurrently.

The author estimates that there are about 10,000 such small companies in the United States. The number of software personnel working under hierarchical organization structures is perhaps 250,000 in the United States as of 2009.

Hierarchical structures are also found in some large companies, so perhaps another 500,000 people work in hierarchical structures inside large companies and government agencies.

Project size The average size of new applications done by hierarchical teams with eight employees and a manager is about 2,000 function points. However, one of the characteristics of hierarchical organizations is that they can cooperate on large projects, so even large systems in excess of 100,000 function points can be handled by multiple departments working in concert.

The caveat with multiple departments attempting to cooperate is that when more than about a dozen are involved simultaneously, some kind of project office may be utilized for overall planning and coordination. Some of the departments involved may handle integration, testing, configuration control, quality assurance, technical writing, and other specialized topics.

Productivity rates Productivity rates for hierarchical departments on projects of 2,000 function points are usually in the range of 12 function points per staff month. They sometimes top 20 function points per staff month for applications where the team has significant expertise, and may drop below 10 function points per staff month for unusual or complex projects. Productivity tends to be inversely proportional to application size and declines as applications grow larger.

Schedules Development schedules for new development by a single hierarchical group with eight team members working on a 2,000–function point project usually range between about 14 months and 24 months and would average perhaps 18 calendar months for the entire application.

Quality The quality levels for hierarchical departments are fairly average. Defect potentials run to about 5.0 bugs per function point, and defect removal efficiency is about 85 percent. Delivered defects would average about 0.75 per function point.

Therefore, an application of 2,000 function points developed by a single hierarchical department would have a total of about 10,000 bugs, of which 1,500 would still be present at release. Of these, about 225 might be serious bugs.

However, if pretest inspections are used, and if tools such as automated static analysis and automated testing are used, then defect removal efficiency can approach 97 percent. In this situation, only about 300 bugs might be present at release. Of these, perhaps 40 might be serious.

Specialization There are few studies to date on the role of specialization in hierarchical software organization structures. Because of common gaps in the training and experience of generalists, some kinds of specialization are needed for large applications. The kinds of specialists that might be useful would be security specialists, test specialists, quality assurance specialists, database specialists, user-interface specialists, network specialists, performance specialists, and technical writers.

Cautions and counter indications The main caution about hierarchical organization structures is that software work tends to be artificially divided to match the abilities of eight-person departments, rather than segmented based on the architecture and design of the applications themselves. As a result, some large functions in large systems are arbitrarily divided between two or more departments when they should be handled by a single group.

While communication within a given department is easy and spontaneous, communication between departments tends to slow down due to managers guarding their own territories. Thus, for large projects with multiple hierarchical departments, there are high probabilities of power struggles and disruptive social conflicts, primarily among the management community.

Conclusions The literature on hierarchical organizations is interesting but incomplete. Much of the literature is produced by enthusiasts for alternate forms of organization structures such as matrix management, Agile teams, pair programming, clean-room development, and the like.

Hierarchical organizations have been in continuous use for software applications since the industry began. While that fact might seem to indicate success, it is also true that the software industry has been characterized by having higher rates of project failures, cost overruns, and schedule overruns than any other industry. The actual impact of hierarchical organizations on software success or software failure is still somewhat ambiguous as of 2009.

Other factors such as methods, employee skills, and management skills tend to be intertwined with organization structures, and this makes it hard to identify the effect of the organization itself.

Conventional Departments with Matrix Organization Structures

The history of matrix management is younger than the history of software development itself. The early literature on matrix management seemed to start around the late 1960s, when it was used within NASA for dealing with cross-functional projects associated with complex space programs.

The idea of matrix management soon moved from NASA into the civilian sector and was eventually picked up by software organizations for dealing with specialization and cross-functional applications.

In a conventional hierarchical organization, software personnel of various kinds report to managers within a given business unit. The technical employees may be generalists, or the departments may include various specialists too, such as software engineers, testers, and technical writers. If a particular business unit has ten software departments, each of these departments might have a number of software engineers, testers, technical writers, and so forth.

By contrast, in a matrix organization, various occupation groups and specialists report to a skill or career manager. Thus all technical writers might report to a technical publications group; all software engineers might be in a software engineering group; all testers might be in a test services group; and so forth.

By consolidating various kinds of knowledge workers within skill-based organizations, greater job enrichment and more career opportunities tend to occur than when specialists are isolated and fragmented among multiple hierarchical departments.

Under a matrix organization, when specialists are needed for various projects, they are assigned to projects and report temporarily to the project managers for the duration of the projects. This of course introduces the tricky concept of employees working for two managers at the same time.

One of the managers (usually the skill manager) has appraisal and salary authority over specialist employees, while the other (usually the project manager) uses their services for completing the project. The project managers may provide inputs to the skill managers about job performance.

The manager with appraisal and salary authority over employees is said to have *solid line* reporting authority. The manager who merely borrows the specialists for specific tasks or a specific project is said to have *dotted line* authority. These two terms reflect the way organization charts are drawn.

It is an interesting phenomenon that matrix management is new enough so that early versions of SAP, Oracle, and some other enterprise resource planning (ERP) applications did not support dotted-line or matrix organization structures. As of 2009, all ERP packages now support matrix organization diagrams.

The literature on matrix management circa 2009 is very strongly polarized between enthusiasts and opponents. About half of the books and articles regard matrix management as a major business achievement. The other half of the books and articles regard matrix management as confusing, disruptive, and a significant business liability.

A Google search of the phrase “failures of matrix management” returned 315,000 citations, while a search of the phrase “successes of matrix management” returned 327,000 citations. As can be seen, this is a strong polarization of opinion that is almost evenly divided.

Over the years, three forms of matrix organization have surfaced called *weak matrix*, *strong matrix*, and *balanced matrix*.

The original form of matrix organization has now been classified as a *weak matrix*. In this form of organization, the employees report primarily to a skill manager and are borrowed by project managers on an as-needed basis. The project managers have no appraisal authority or salary authority over the employees and therefore depend upon voluntary cooperation to get work accomplished. If there are conflicts between the project managers and the skill managers in terms of resource allocations, the project managers lack the authority to acquire the skills their projects may need.

Because weak matrix organizations proved to be troublesome, the *strong matrix* variation soon appeared. In a strong matrix, the specialists may still report to a skill manager, but once assigned to a project, the needs of the project take precedence. In fact, the specialists may even be formally assigned to the project manager for the duration of the project and receive appraisals and salary reviews.

In a *balanced matrix*, responsibility and authority are nominally equally shared between the skill manager and the project manager. While this sounds like a good idea, it has proven to be difficult to accomplish. As a result, the strong matrix form seems to be dominant circa 2009.

Demographics In the software world, matrix organizations are found most often in large companies that employ between perhaps 1,000 and 50,000 total software personnel. These large companies tend to have scores of specialized skills and hundreds of projects going on at the same time.

The author estimates that there are about 250 such large companies in the United States with primarily matrix organization. The number of software personnel working under matrix organization structures is perhaps 1 million in the United States as of 2009.

Project size The average size of new applications done by matrix teams with eight employees and a manager is about 2,000 function points. However, matrix organizations can be scaled up to any arbitrary size, so even large systems in excess of 100,000 function points can be handled by multiple matrix departments working in concert.

The caveat with multiple departments attempting to cooperate is that when more than about a dozen are involved simultaneously, some kind of a project office may be needed for overall planning and coordination.

With really large applications in excess of 25,000 function points, some of the departments may be fully staffed by specialists who handle topics such as integration, testing, configuration control, quality assurance, technical writing, and other specialized topics.

Productivity rates Productivity rates for matrix departments on projects of 2,000 function points are usually in the range of 10 function points per staff month. They sometimes top 16 function points per staff month for applications where the team has significant expertise, and may drop below 6 function points per staff month for unusual or complex projects. Productivity tends to be inversely proportional to application size and declines as applications grow larger.

Schedules Development schedules for new development by a single matrix group with eight team members working on a 2,000-function point project usually ranges between about 16 months and 28 months and would average perhaps 18 calendar months for the entire application.

Quality The quality levels for matrix organizations often are average. Defect potentials run to about 5.0 bugs per function point, and defect removal efficiency is about 85 percent. Delivered defects would average about 0.75 per function point. Matrix and hierarchical organizations are identical in quality, unless special methods such as formal inspections, static analysis, automated testing, and other state-of-the-art approaches have been introduced.

Therefore, an application of 2,000 function points developed by a single matrix department might have a total of about 10,000 bugs, of which 1,500 would still be present at release. Of these, about 225 might be serious bugs.

However, if pretest inspections are used, and if tools such as automated static analysis and automated testing are used, then defect removal efficiency can approach 97 percent. In this situation, only about 300 bugs might be present at release. Of these, perhaps 40 might be serious.

As application sizes increase, defect potentials also increase, while defect removal efficiency levels decline.

Specialization The main purpose of the matrix organization structure is to support specialization. That being said, there are few studies to date on the kinds of specialization in matrix software organization structures. As of 2009, topics such as the numbers of architects needed, the number of testers needed, and the number of quality assurance personnel needed for applications of various sizes remains ambiguous.

Typical kinds of specialization are usually needed for large applications. The kinds of specialists that might be useful would be security specialists, test specialists, quality assurance specialists, database

specialists, user-interface specialists, network specialists, performance specialists, and technical writers.

Cautions and counter indications The main caution about matrix organization structures is that of political disputes between the skill managers and the project managers.

Another caution, although hard to evaluate, is that roughly half of the studies and literature about matrix organization assert that the matrix approach is harmful rather than beneficial. The other half, however, says the opposite and claims significant value from matrix organizations. But any approach with 50 percent negative findings needs to be considered carefully and not adopted blindly.

A common caution for both matrix and hierarchical organizations is that software work tends to be artificially divided to match the abilities of eight-person departments, rather than segmented based on the architecture and design of the applications. As a result, some large functions in large systems are arbitrarily divided between two or more departments when they should be handled by a single group.

While technical communication within a given department is easy and spontaneous, communication between departments tends to slow down due to managers guarding their own territories. Thus, for large projects with multiple hierarchical or matrix departments, there are high probabilities of power struggles and disruptive social conflicts, primarily among the management community.

Conclusions The literature on matrix organizations is so strongly polarized that it is hard to find a consensus. With half of the literature praising matrix organizations and the other half blaming them for failures and disasters, it is not easy to find solid empirical data that is convincing.

From observations made during litigation for projects that failed or never operated successfully, there seems to be little difference between hierarchical and matrix organizations. Both matrix and hierarchical organizations end up in court about the same number of times.

What does make a difference is the competence of the managers and technical staff, and the emphasis on effective quality control and change management control. Effective estimating and careful progress tracking also make a difference, but none of these factors are directly related to either the hierarchical or matrix organization styles.

Specialist Organizations in Large Companies

Because development software engineers are not the only or even the largest occupation group in big companies and government agencies, it is worthwhile to consider what kinds of organizations best serve the needs of the most common occupation groups.

In approximate numerical order by numbers of employees, the major specialist occupations would be

1. Maintenance software engineers
2. Test personnel
3. Business analysts and systems analysts
4. Customer support personnel
5. Quality assurance personnel
6. Technical writing personnel
7. Administrative personnel
8. Configuration control personnel
9. Project office staff
 - Estimating specialists
 - Planning specialists
 - Measurement and metrics specialists
 - Scope managers
 - Process improvement specialists
 - Standards specialists

Many other kinds of personnel perform technical work such as network administration, operating data centers, repair of workstations and personal computers, and other activities that center around operations rather than software. These occupations are important, but are outside the scope of this book.

Following are discussion of organization structures for selected specialist groups.

Software Maintenance Organizations

For small companies with fewer than perhaps 50 software personnel, maintenance and development are usually carried out by the same people, and there are no separate maintenance groups. For that matter, some forms of customer support may also be tasked to the software engineering community in small companies.

However, as companies grow larger, maintenance specialization tends to occur. For companies with more than about 500 software personnel, maintenance groups are the norm rather than the exception.

(Note: The International Software Benchmarking Standards Group (ISBSG) has maintenance benchmark data available for more than

400 projects and is adding new data monthly. Refer to www.ISBSG.org for additional information.)

The issue of separating maintenance from development has both detractors and adherents.

The detractors of separate maintenance groups state that separating maintenance from development may require extra staff to become familiar with the same applications, which might artificially increase overall staffing. They also assert that if enhancements and defect repairs are taking place at the same time for the same applications and are done by two different people, the two tasks might interfere with each other.

The adherents of separate maintenance groups assert that because bugs occur randomly and in fairly large numbers, they interfere with development schedules. If the same person is responsible for adding a new feature to an application and for fixing bugs, and suddenly a high-severity bug is reported, fixing the bug will take precedence over doing development. As a result, development schedules will slip and probably slip so badly that the ROI of the application may turn negative.

Although both sets of arguments have some validity, the author's observations support the view that separate maintenance organizations are the most useful for larger companies that have significant volumes of software to maintain.

Separate maintenance teams have higher productivity rates in finding and fixing problems than do developers. Also, having separate maintenance change teams makes development more predictable and raises development productivity.

Some maintenance groups also handle small enhancements as well as defect repairs. There is no exact definition of a "small enhancement," but a working definition is an update that can be done by one person in less than one week. That would limit the size of small enhancements to about 5 or fewer function points.

Although defect repairs and enhancements are the two most common forms of maintenance, there are actually 23 different kinds of maintenance work performed by large organizations, as shown in Table 5-2.

Although the 23 maintenance topics are different in many respects, they all have one common feature that makes a group discussion possible: they all involve modifying an existing application rather than starting from scratch with a new application.

Each of the 23 forms of modifying existing applications has a different reason for being carried out. However, it often happens that several of them take place concurrently. For example, enhancements and defect repairs are very common in the same release of an evolving application.

The maintenance literature has a number of classifications for maintenance tasks such as "adaptive," "corrective," or "perfective." These seem

TABLE 5-2 Twenty-Three Kinds of Maintenance Work

1.	Major enhancements (new features of greater than 20 function points)
2.	Minor enhancements (new features of less than 5 function points)
3.	Maintenance (repairing defects for good will)
4.	Warranty repairs (repairing defects under formal contract)
5.	Customer support (responding to client phone calls or problem reports)
6.	Error-prone module removal (eliminating very troublesome code segments)
7.	Mandatory changes (required or statutory changes)
8.	Complexity or structural analysis (charting control flow plus complexity metrics)
9.	Code restructuring (reducing cyclomatic and essential complexity)
10.	Optimization (increasing performance or throughput)
11.	Migration (moving software from one platform to another)
12.	Conversion (changing the interface or file structure)
13.	Reverse engineering (extracting latent design information from code)
14.	Reengineering (transforming legacy applications to modern forms)
15.	Dead code removal (removing segments no longer utilized)
16.	Dormant application elimination (archiving unused software)
17.	Nationalization (modifying software for international use)
18.	Mass updates such as Euro or Year 2000 repairs
19.	Refactoring, or reprogramming applications to improve clarity
20.	Retirement (withdrawing an application from active service)
21.	Field service (sending maintenance members to client locations)
22.	Reporting bugs or defects to software vendors
23.	Installing updates received from software vendors

to be classifications that derive from academia. While there is nothing wrong with them, they manage to miss the essential point. Maintenance overall has only two really important economic distinctions:

1. Changes that are charged to and paid for by customers (enhancements)
2. Changes that are absorbed by the company that built the software (bug repairs)

Whether a company uses standard academic distinctions of maintenance activities or the more detailed set of 23 shown here, it is important to separate costs into the two buckets of customer-funded or self-funded expenses.

Some companies such as Symantec charge customers for service calls, even for reporting bugs. The author regards such charges as being unprofessional and a cynical attempt to make money out of incompetent quality control.

There are also common sequences or patterns to these modification activities. For example, reverse engineering often precedes reengineering, and the two occur so often together as to almost constitute a linked set. For releases of large applications and major systems, the author has observed from six to ten forms of maintenance all leading up to the same release.

In recent years, the Information Technology Infrastructure Library (ITIL) has had a significant impact on maintenance, customer support, and service management in general. The ITIL is a rather large collection of more than 30 books and manuals that deal with service management, incident reporting, change teams, reliability criteria, service agreements, and a host of other topics. As this book is being written in 2009, the third release of the ITIL is under way.

It is an interesting phenomenon of the software world that while ITIL has become a major driving force in service agreements within companies for IT service, it is almost never used by commercial vendors such as Microsoft and Symantec for agreements with their customers. In fact, it is quite instructive to read the small print in the end-user license agreements (EULAs) that are always required prior to using the software.

When these agreements are read, it is disturbing to see clauses that assert that the vendors have no liabilities whatsoever, and that the software is not guaranteed to operate or to have any kind of quality levels.

The reason for these one-sided EULA agreements is that software quality control is so bad that even major vendors would go bankrupt if sued for the damages that their products can cause.

For many IT organizations and also for commercial software groups, a number of functions are joined together under a larger umbrella: customer support, maintenance (defect repairs), small enhancements (less than 5 function points), and sometimes integration and configuration control.

In addition, several forms of maintenance work deal with software not developed by the company itself:

1. Maintenance of commercial applications such as those acquired from SAP, Oracle, Microsoft, and the like. The maintenance tasks here involve reporting bugs, installing new releases, and possibly making custom changes for local conditions.
2. Maintenance of open-source and freeware applications such as Firefox, Linux, Google, and the like. Here, too, the maintenance tasks involve reporting bugs and installing new releases, plus customization as needed.

3. Maintenance of software added to corporate portfolios via mergers or acquisitions with other companies. This is a very tricky situation that is fraught with problems and hazards. The tasks here can be quite complex and may involve renovation, major updates, and possibly migration from one database to another.

In addition to normal maintenance, which combines defect repairs and enhancements, legacy applications may undergo thorough and extensive modernization, called *renovation*.

Software renovation can include surgical removal of error-prone modules, automatic or manual restructuring to reduce complexity, revision or replacement of comments, removal of dead code segments, and possibly even automatic conversion of the legacy application from old or obsolete programming languages into newer programming languages.

Renovation may also include data mining to extract business rules and algorithms embedded in the code but missing from specifications and written descriptions of the code. Static analysis and automatic testing tools may also be included in renovation. Also, it is now possible to generate function point totals for legacy applications automatically, and this may also occur as part of renovation activities.

The observed effect of software renovation is to stretch out the useful life of legacy applications by an additional ten years. Renovation reduces the number of latent defects in legacy code, and therefore reduces future maintenance costs by about 50 percent per calendar year for the applications renovated. Customer support costs are also reduced.

As the recession deepens and lengthens, software renovation will become more and more valuable as a cost-effective alternative to retiring legacy applications and redeveloping them. The savings accrued from renovation could reduce maintenance costs so significantly that redevelopment could occur using the savings that accrue from renovation.

If a company does plan to renovate legacy applications, it is appropriate to fix some of the chronic problems that no doubt are present in the original legacy code. The most obvious of these would be to remove security vulnerabilities, which tend to be numerous in legacy applications. The second would be to improve quality by using inspections, static analysis, automated testing, and other modern techniques such as TSP during renovations.

A combination of the Team Software Process (TSP), the Caja security architecture from Google, and perhaps the E programming language, which is more secure than most languages, might be considered for renovating applications that deal with financial or valuable proprietary data.

For predicting the staffing and effort associated with software maintenance, some useful rules of thumb have been developed based on observations of maintenance groups in companies such as IBM, EDS, Software Productivity Research, and a number of others.

Maintenance assignment scope = the amount of software that one maintenance programmer can successfully maintain in a single calendar year. The U.S. average as of 2009 is about 1,000 function points. The range is between a low of about 350 function points and a high of about 5,500 function points. Factors that affect maintenance assignment scope include the experience of the maintenance team, the complexity of the code, the number of latent bugs in the code, the presence or absence of “error-prone modules” in the code, and the available tool suites such as static analysis tools, data mining tools, and maintenance workbenches. This is an important metric for predicting the overall number of maintenance programmers needed.

(For large applications, knowledge of the internal structure is vital for effective maintenance and modification. Therefore, major systems usually have their own change teams. The number of maintenance programmers in such a change team can be calculated by dividing the size of the application in function points by the appropriate maintenance assignment scope, as shown in the previous paragraph.)

Defect repair rates = the average number of bugs or defects that a maintenance programmer can fix in a calendar month of 22 working days. The U.S. average is about 10 bugs repaired per calendar month. The range is from fewer than 5 to about 17 bugs per staff month. Factors that affect this rate include the experience of the maintenance programmer, the complexity of the code, and “bad-fix injections,” or new bugs accidentally injected into the code created to repair a previous bug. The U.S. average for bad-fix injections is about 7 percent.

Renovation productivity = the average number of function points per staff month for renovating software applications using a full suite of renovation support tools. The U.S. average is about 65 function points per staff month. The range is from a low of about 25 function points per staff month for highly complex applications in obscure languages to more than 125 function points per staff month for applications of moderate complexity in fairly modern languages. Other factors that affect this rate include the overall size of the applications, the presence or absence of “error-prone modules” in the application, and the experience of the renovation team.

(Manual renovation without automated support is much more difficult, and hence productivity rates are much lower—in the vicinity of 14 function points per staff month. This is somewhat higher than new development, but still close to being marginal in terms of return on investment.)

Software does not age gracefully. Once software is put into production, it continues to change in three important ways:

1. Latent defects still present at release must be found and fixed after deployment.
2. Applications continue to grow and add new features at a rate of between 5 percent and 10 percent per calendar year, due either to changes in business needs, or to new laws and regulations, or both.
3. The combination of defect repairs and enhancements tends to gradually degrade the structure and increase the complexity of the application. The term for this increase in complexity over time is called *entropy*. The average rate at which software entropy increases is about 1 percent to 3 percent per calendar year.

A special problem with software maintenance is caused by the fact that some applications use multiple programming languages. As many as 15 different languages have been found within a single large application.

Multiple languages are troublesome for maintenance because they add to the learning chores of the maintenance teams. Also some (or all) of these language may be “dead” in the sense that there are no longer working compilers or interpreters. This situation chokes productivity and raises the odds of bad-fix injections.

Because software defect removal and quality control are imperfect, there will always be bugs or defects to repair in delivered software applications. The current U.S. average for defect removal efficiency is only about 85 percent of the bugs or defects introduced during development. This has been the average for more than 20 years.

The actual values are about 5 bugs per function point created during development. If 85 percent of these are found before release, about 0.75 bug per function point will be released to customers.

For a typical application of 1,000 function points or 100,000 source code statements, that implies about 750 defects present at delivery. About one fourth, or 185 defects, will be serious enough to stop the application from running or will create erroneous outputs.

Since defect potentials tend to rise with the overall size of the application, and since defect removal efficiency levels tend to decline with the overall size of the application, the overall volume of latent defects delivered with the application rises with size. This explains why super-large applications in the range of 100,000 function points, such as Microsoft Windows and many enterprise resource planning (ERP) applications, may require years to reach a point of relative stability. These large systems are delivered with thousands of latent bugs or defects.

Of course, average values are far worse than best practices. A combination of formal inspections, static analysis, and automated testing can bring cumulative defect removal efficiency levels up to 99 percent. Methods such as the Team Software Process (TSP) can lower defect potentials down below 3.0 per function point.

Unless very sophisticated development practices are followed, the first year of the release of a new software application will include a heavy concentration of defect repair work and only minor enhancements.

However, after a few years, the application will probably stabilize as most of the original defects are found and eliminated. Also after a few years, new features will increase in number.

As a result of these trends, maintenance activities will gradually change from the initial heavy concentration on defect repairs to a longer-range concentration on new features and enhancements.

Not only is software deployed with a significant volume of latent defects, but the phenomenon of bad-fix injection has been observed for more than 50 years. Roughly 7 percent of all defect repairs will contain a new defect that was not there before. For very complex and poorly structured applications, these bad-fix injections have topped 20 percent.

Even more alarming, once a bad fix occurs, it is very difficult to correct the situation. Although the U.S. average for initial bad-fix injection rates is about 7 percent, the secondary injection rate against previous bad fixes is about 15 percent for the initial repair and 30 percent for the second. A string of up to five consecutive bad fixes has been observed, with each attempted repair adding new problems and failing to correct the initial problem. Finally, the sixth repair attempt was successful.

In the 1970s, the IBM Corporation did a distribution analysis of customer-reported defects against their main commercial software applications. The IBM personnel involved in the study, including the author, were surprised to find that defects were not randomly distributed through all of the modules of large applications.

In the case of IBM's main operating system, about 5 percent of the modules contained just over 50 percent of all reported defects. The most extreme example was a large database application, where 31 modules out of 425 contained more than 60 percent of all customer-reported bugs. These troublesome areas were known as *error-prone modules*.

Similar studies by other corporations such as AT&T and ITT found that error-prone modules were endemic in the software domain. More than 90 percent of applications larger than 5,000 function points were found to contain error-prone modules in the 1980s and early 1990s. Summaries of the error-prone module data from a number of companies were published in the author's book *Software Quality: Analysis and Guidelines for Success*.

Fortunately, it is possible to surgically remove error-prone modules once they are identified. It is also possible to prevent them from occurring. A combination of defect measurements, formal design inspections, formal code inspections, and formal testing and test-coverage analysis have proven to be effective in preventing error-prone modules from coming into existence.

Today, in 2009, error-prone modules are almost nonexistent in organizations that are higher than level 3 on the capability maturity model (CMM) of the Software Engineering Institute. Other development methods such as the Team Software Process (TSP) and Rational Unified Process (RUP) are also effective in preventing error-prone modules. Several forms of Agile development such as extreme programming (XP) also seem to be effective in preventing error-prone modules from occurring.

Removal of error-prone modules is a normal aspect of renovating legacy applications, so those software applications that have undergone renovation will have no error-prone modules left when the work is complete.

However, error-prone modules remain common and troublesome for CMMI level 1 organizations. They are also alarmingly common in legacy applications that have not been renovated and that are maintained without careful measurement of defects.

Once deployed, most software applications continue to grow at annual rates of between 5 percent and 10 percent of their original functionality. Some applications, such as Microsoft Windows, have increased in size by several hundred percent over a ten-year period.

The combination of continuous growth of new features coupled with continuous defect repairs tends to drive up the complexity levels of aging software applications. Structural complexity can be measured via metrics such as cyclomatic and essential complexity using a number of commercial tools. If complexity is measured on an annual basis and there is no deliberate attempt to keep complexity low, the rate of increase is between 1 percent and 3 percent per calendar year.

However, and this is important, the rate at which entropy or complexity increases is directly proportional to the initial complexity of the application. For example, if an application is released with an average cyclomatic complexity level of less than 10, it will tend to stay well structured for at least five years of normal maintenance and enhancement changes.

But if an application is released with an average cyclomatic complexity level of more than 20, its structure will degrade rapidly, and its complexity levels might increase by more than 2 percent per year. The rate of entropy and complexity will even accelerate after a few years.

As it happens, both bad-fix injections and error-prone modules tend to correlate strongly (although not perfectly) with high levels of complexity.

A majority of error-prone modules have cyclomatic complexity levels of 10 or higher. Bad-fix injection levels for modifying high-complexity applications are often higher than 20 percent.

Here, too, renovation can reverse software entropy and bring cyclomatic complexity levels down below 10, which is the maximum safe level of code complexity.

There are several difficulties in exploring software maintenance costs with accuracy. One of these difficulties is the fact that maintenance tasks are often assigned to development personnel who interleave both development and maintenance as the need arises. This practice makes it difficult to distinguish maintenance costs from development costs, because the programmers are often rather careless in recording how time is spent.

Another and very significant problem is that a great deal of software maintenance consists of making very small changes to software applications. Quite a few bug repairs may involve fixing only a single line of code. Adding minor new features, such as perhaps a new line-item on a screen, may require fewer than 50 source code statements.

These small changes are below the effective lower limit for counting function point metrics. The function point metric includes weighting factors for complexity, and even if the complexity adjustments are set to the lowest possible point on the scale, it is still difficult to count function points below a level of perhaps 15 function points.

An experimental method called *micro function points* has been developed for small maintenance changes and bug repairs. This method is similar to standard function points, but drops down to three decimal places of precision and so can deal with fractions of a single function point.

Of course, the work of making a small change measured with micro function points may be only an hour or less. But in large companies, where as many as 20,000 such changes are made in a year, the cumulative costs are not trivial. Micro function points are intended to eliminate the problem that small maintenance updates have not been subject to formal economic analysis.

Quite a few maintenance tasks involve changes that are either a fraction of a function point, or may at most be fewer than 5 function points or about 250 Java source code statements. Although normal counting of function points is not feasible for small updates, and micro function points are still experimental, it is possible to use the *backfiring* method of converting counts of logical source code statements into equivalent function points. For example, suppose an update requires adding 100 Java statements to an existing application. Since it usually takes about 50 Java statements to encode 1 function point, it can be stated that this small maintenance project is about 2 function points in size.

Because of the combination of 23 separate kinds of maintenance work mixed with both large and small updates, maintenance effort is harder to estimate and harder to measure than in conventional software development. As a result, there are many fewer maintenance benchmarks than development benchmarks. In fact, there is much less reliable information about maintenance than about almost any other aspect of software.

Maintenance activities are frequently outsourced to either domestic or offshore outsource companies. For a variety of business reasons, maintenance outsource contracts seem to be more stable and less likely to end up in court than software development contracts.

The success of maintenance outsource contracts is because of two major factors:

1. Small maintenance changes do not have the huge cost and schedule slippage rates associated with major development projects.
2. Small maintenance changes to existing software almost never fail completely. A significant number of development projects do fail and are never completed at all.

There may be other reasons as well, but the fact remains that maintenance outsource contracts seem more stable and less likely to end up in court than development outsource contracts.

Maintenance is the dominant work of the software industry in 2009 and will probably stay the dominant activity for the indefinite future. For software, as with many other industries, once the industry passes 50 years of age, more workers are involved with repairing existing products than there are workers involved with building new products.

Demographics In the software world, separate maintenance organizations are found most often in large companies that employ between perhaps 500 and 50,000 total software personnel.

The author estimates that there are about 2,500 such large companies in the United States with separate maintenance organizations. The number of software personnel working on maintenance in maintenance organizations is perhaps 800,000 in the United States as of 2009. (The number of software personnel who perform both development and maintenance is perhaps 400,000.)

Project size The average size of software defects is less than 1 function point, which is why micro-function points are needed. Enhancements or new features typically range from a low of perhaps 5 function points to a high of perhaps 500 function points. However, there are so many enhancements, that software applications typically grow at a rate of around 8 percent per calendar year for as long as they are being used.

Productivity rates Productivity rates for defect repairs are only about 10 function points per staff month, due to the difficulty of finding the exact problem, plus the need for regression testing and constructing new releases. Another way of expressing defect repair productivity is to use defects or bugs fixed per month, and a typical value would be about 10 bugs per staff month.

The productivity rates for enhancements average about 15 function points per staff month, but vary widely due to the nature and size of the enhancement, the experience of the team, the complexity of the code, and the rate at which requirements change during the enhancement. The range for enhancements can be as low as about 5 function points per staff month, or as high as 35 function points per staff month.

Schedules Development schedules for defect repairs range from a few hours to a few days, with one major exception. Defects that are *abeyant*, or cannot be replicated by the change teams, may take weeks to repair because the internal version of the application used by the change team may not have the defect. It is necessary to get a great deal more information from users in order to isolate abeyant defects.

Fixing a bug is not the same as issuing a new release. Within some companies such as IBM, maintenance schedules in the sense of defect repairs vary with the severity level of the bugs reported; that is, severity 1 bugs (most serious), about 1 week; severity 2 bugs, about two weeks; severity 3 bugs, next release; severity 4 bugs, next release or whenever it is convenient.

Development schedules for enhancements usually run from about 1 month up to 9 months. However, many companies have fixed release intervals that aggregate a number of enhancements and defect repairs and release them at the same time. Microsoft “service packs” are one example, as are the intermittent releases of Firefox. Normally, fixed release intervals are either every six months or once a year, although some may be quarterly.

Quality The main quality concerns for maintenance or defect repairs are threefold: (1) higher defect potentials for maintenance and enhancements than for new development, (2) the presence or absence of error-prone modules in the application, and (3) the bad-fix injection rates for defect repairs, which average about 7 percent.

Maintenance and enhancement defect potentials are higher than for new development and run to about 6.0 bugs per function point. Defect removal efficiency is usually lower than for new development and is only about 83 percent. As a result, delivered defects would average about 1.08 per function point.

An additional quality concern that grows slowly worse over a period of years is that application complexity (as measured by cyclomatic complexity) slowly increases because changes tend to degrade the original structure. As a result, each year, defect potentials may be slightly higher than the year before, while bad-fix injections may increase. Unless the application is renovated, these problems tend to become so bad that eventually the application can no longer be safely modified.

In addition to renovation, other approaches such as formal inspections for major enhancements and significant defect repairs, static analysis, and automatic testing can raise defect removal efficiency levels above 95 percent. However, bad-fix injections and error-prone modules are still troublesome.

Specialization The main purpose of the maintenance organization structures is to support maintenance specialization. While not everyone enjoys maintenance, it happens that quite a few programmers and software engineers do enjoy it.

Other specialist work in a maintenance organization includes integration and configuration control. Maintenance software engineers normally do most of the testing on small updates and small enhancements, although formal test organizations may do some specialized testing such as system testing prior to a major release.

Curiously, software quality assurance (SQA) is seldom involved with defect repairs and minor enhancements carried out by maintenance groups. However, SQA specialists usually do work on major enhancements.

Technical writers don't have a major role in software maintenance, but may occasionally be involved if enhancements trigger changes in user manuals or HELP text.

That being said, few studies to date deal with either personality or technical differences between successful maintenance programmers and successful development programmers.

Cautions and counter indications The main caution about maintenance specialization and maintenance organizations is that they tend to lock personnel into narrow careers, sometimes limited to repairing a single application for a period of years. There is little chance of career growth or knowledge expansion if a software engineer spends years fixing bugs in a single software application. Occasionally, switching back and forth from maintenance to development is a good practice for minimizing occupational boredom.

Conclusions The literature on maintenance organizations is very sparse compared with the literature on development. Although there are

some good books, there are few long-range studies that show application growth, entropy increase, and defect trends over multiple years.

Given that software maintenance is the dominant activity of the software industry in 2009, a great deal more research and study are indicated. Research is needed on data mining of legacy applications to extract business rules; on removing security vulnerabilities from legacy code; on the costs and value of software renovation; and on the application of quality control methods such as inspections, static analysis, and automated testing to legacy code.

Customer Support Organizations

In small companies with few software applications and few customers or application users, support may be carried out on an informal basis by the development team itself. However, as numbers of customers increase and numbers of applications needing support increase, a point will soon be reached where a formal customer support organization will be needed.

Informal rules of thumb for customer support indicate that customer support staffing is dependent on three variables:

1. Number of customers
2. Number of latent bugs or defects in released software
3. Application size measured in terms of function points or lines of code

One full-time customer support person would probably be needed for applications that meet these criteria: 150 customers, 500 latent bugs in the software (75 serious bugs), and 10,000 function points or 500,000 source code statements in a language such as Java.

The most effective known method for improving customer support is to achieve much better application quality levels than are typical today in 2009. Every reduction of about 220 latent defects at delivery can reduce customer support staffing needs by one person. This is based on the assumption that customer support personnel speak to about 30 customers per day, and each released defect is encountered by 30 customers. Therefore, each released defect occupies one day for one customer support staff member, and there are 220 working days per year.

Some companies attempt to reduce customer support costs by charging for support calls, even to report bugs in the applications! This is an extremely bad business practice that primarily offends customers without benefiting the companies. Every customer faced with a charge for customer support is an unhappy customer who is actively in search of a more sensible competitive product.

Also, since software is routinely delivered with hundreds of serious bugs, and since customer reports of those bugs are valuable to software vendors, charging for customer support is essentially cutting off a valuable resource that can be used to lower maintenance costs. Few companies that charge for support have many happy customers, and many are losing market shares.

Unfortunately, customer support organizations are among the most difficult of any kind of software organization to staff and organize well. There are several reasons for this. The first is that unless a company charges for customer support (not a recommended practice), the costs can be high. The second is that customer-support work tends to have limited career opportunities, and this makes it difficult to attract and keep personnel.

As a result, customer support was one of the first business activities to be outsourced to low-cost offshore providers. Because customer support is labor intensive, it was also among the first business activities to attempt to automate at least some responses. To minimize the time required for discussions with live support personnel, there are a variety of frequently asked questions (FAQ) and other topics that users can access by phone or e-mail prior to speaking with a real person.

Unfortunately, these automated techniques are often frustrating to users because they require minutes of time dealing with sometimes arcane voice messages before reaching a real person. Even worse, these automated voice messages are almost useless for the hard of hearing.

That being said, companies in the customer support business have made some interesting technical innovations with voice response systems and also have developed some fairly sophisticated help-desk packages that keep track of callers or e-mails, identify bugs or defects that have been previously reported, and assist with other administrative functions.

Because calls and e-mail from customers contain a lot of potentially valuable information about deep bugs and security flaws, prudent companies want to capture this information for analysis and to use it as part of their quality and security improvement programs.

At a sociological level, an organization called the Service and Support Professionals Association (SSPA) not only provides useful information for support personnel, but also evaluates the customer support of various companies and issues awards and citations for excellence. The SSPA group also has conferences and events dealing with customer support. (The SSPA web site is www.thesspa.com.)

SSPA has an arrangement with the well-known J.D. Power and Associates to evaluate customer service in order to motivate companies

by issuing various awards. As an example, the SSPA web site mentions the following recent awards as of 2009:

- ProQuest Business Solutions—Most improved
- IBM Rochester—Sustained excellence for three consecutive years
- Oracle Corporation—Innovative support
- Dell—Mission critical support
- RSA Security—Best support for complex systems

For in-house support as opposed to commercial companies that sell software, the massive compendium of information contained in the Information Technology Infrastructure Library (ITIL) spells out great topics such as Help-Desk response time targets, service agreements, incident management, and hundreds of other items of information.

Software customer support is organized in a multitier arrangement that uses automated and FAQ as the initial level, and then brings in more expertise at other levels. An example of such a multitier arrangement might resemble the following:

- Level 0—Automated voice messages, FAQ, and pointers to available downloads
- Level 1—Personnel who know basics of the application and common bugs
- Level 2—Experts in selected topics
- Level 3—Development personnel or top-gun experts

The idea behind the multilevel approach is to minimize the time requirements of developers and experts, while providing as much useful information as possible in what is hopefully an efficient manner.

As mentioned in a number of places in this book, the majority of customer service calls and e-mails are due to poor quality and excessive numbers of bugs. Therefore, more sophisticated development approaches such as using Team Software Process (TSP), formal inspections, static analysis, automated testing, and the like will not only reduce development costs and schedules, but will also reduce maintenance and customer support costs.

It is interesting to consider how one of the J.D. Power award recipients, IBM Rochester, goes about customer support:

“There is a strong focus on support responsiveness, in terms of both time to response as well as the ability to provide solutions. When customers call in, there is a target that within a certain amount of time (a minute or a couple of minutes), the call must be answered. IBM does not want long hold times where customers spend >10 minutes just waiting for the phone to be answered.

When problems/defects are reported, the formal fix may take some time. Before the formal fix is available, the team will provide a temporary solution in as soon as possible, and a key metric used is “time to first relief.” The first-relief temporary repairs may take less than 24 hours for some new problems, and even less if the problem is already known.

When formal fixes are provided, a key metric used by IBM Rochester is the quality of the fixes: percent of defective fixes. The Rochester’s defective fix rate is the lowest among the major platforms in IBM. (Since the industry average for bad-fix injection is about 7%, it is commendable that IBM addresses this issue.)

The IBM Rochester support center also conducts a “trailer survey.” This is a survey of customer satisfaction about the service or fix. These surveys are based on samples of problem records that are closed. IBM Rochester’s trailer survey satisfaction is in the high 90s in terms of percentages of satisfied customers.

Another IBM Rochester factor could be called the “cultural factor.” IBM as a corporation and Rochester as a lab both have a long tradition of focus on quality (e.g., winning the Malcolm Baldrige quality award). Because customer satisfaction correlates directly with quality, the IBM Rochester products have long had a reputation for excellence (IBM system /34, system/36, system /38, AS/400, system i, etc.). IBM and Rochester employees are proud of the quality that they deliver for both products and services.”

For major customer problems, teams (support, development, test, etc.) work together to come up with solutions. Customer feedback has long been favorable for IBM Rochester, which explains their multiyear award for customer support excellence. Often when surveyed customers mention explicitly and favorably the amount of support and problem solving that they receive from the IBM Rochester site.

Demographics In the software world, in-house customer support staffed by actual employees is in rapid decline due to the recession. Probably a few hundred large companies still provide such support, but as layoffs and downsizing continue to escalate, their numbers will be reduced.

However, for small companies that have never employed full-time customer support personnel, no doubt the software engineers will still continue to field customer calls and respond to e-mails. There are probably 10,000 or more U.S. organizations with between 1 and 50 employees where customer support tasks are performed informally by software engineers or programmers.

For commercial software organizations, outsourcing of customer support to specialized support companies is now the norm. While some of these support companies are domestic, there are also dozens of customer

support organizations in other countries with lower labor costs than the United States or Europe. However, as the recession continues, labor costs will decline in the United States, which now has large pools of unemployed software technical personnel. Customer support, maintenance, and other labor-intensive tasks may well start to move back to the United States.

Project size The average size of applications where formal customer support is close to being mandatory is about 10,000 function points. Of course, for any size application, customers will have questions and need to report bugs. But applications in the 10,000–function point range usually have many customers. In addition, these large systems always are released with thousand of latent bugs.

Productivity rates Productivity rates for customer support are not measured using function points, but rather numbers of customers assisted. Typically, one tier-1 customer support person on a telephone support desk can talk to about 30 people per day, which translates into each call taking about 16 minutes.

For tier 2 and tier 3 customer support, where experts are used, the work of talking to customers is probably not full time. However, for problems serious enough to reach tier 2, expect each call to take about 70 minutes. For problems that reach tier 3, there will no doubt be multiple calls back and forth and probably some internal research. Expect tier 3 calls to take about 240 minutes.

If a customer is reporting a new bug that has not been identified or fixed, then days or even weeks may be required. (The author worked as an expert witness in a lawsuit where the time required to fix one bug in a financial application was more than nine calendar months. In the course of fixing this bug, the first four attempts each took about two months. They not only failed to fix the original bug, but added new bugs in each fix.)

Schedules The primary schedule issue for customer support is the wait or hold time before speaking to a live support person. Today, in 2009, reaching a live person can take between 10 minutes and more than 60 minutes of hold time. Needless to say, this is very frustrating to clients. Improving quality should also reduce wait times. Assuming constant support staffing, every reduction of ten severity 1 or 2 defects released in software should reduce wait times by about 30 seconds.

Quality Customer support calls are directly proportional to the number of released defects or bugs in software. It is theoretically possible that releasing software with zero defects might reduce the number of customer support calls to zero, too. In today's world, where defect removal

efficiency only averages 85 percent and hundreds or thousands of serious bugs are routinely still present when software is released, there will be hundreds of customer support calls and e-mails.

It is interesting that some open-source and freeware applications such as Linux, Firefox, and Avira seem to have better quality levels than equivalent applications released by established vendors such as Microsoft and Symantec. In part this may be due to the skills of the developers, and in part it may be due to routinely using tools such as static analysis prior to release.

Specialization The role of tier-1 customer support is very specialized. Effective customer support requires a good personality when dealing with crabby customers plus fairly sophisticated technical skills. Of these two, the criterion for technical skill is easier to fill than the criterion for a good personality when dealing with angry or outraged customers. That being said, there are few studies to date that deal with either personality or technical skills in support organizations.

In addition to customer support provided by vendors of software, some user associations and nonprofit groups provide customer support on a volunteer basis. Many freeware and open-source applications have user groups that can answer technical questions. Even for commercial software, it is sometimes easier to get an informed response to a question from an expert user than it is from the company that built the software.

Cautions and counter indications The main caution about customer support work is that it tends to lock personnel into narrow careers, sometimes limited to discussing a single application such as Oracle or SAP for a period of years. There is little chance of career growth or knowledge expansion.

Another caution is that improving customer support via automation and expert systems is technically feasible, but many existing patents cover such topics. As a result, attempts to develop improved customer support automation may require licensing of intellectual property.

Conclusions The literature on customer support is dominated by two very different forms of information. The Information Technology Infrastructure Library (ITIL) contains more than 30 volumes and more than 5,000 pages of information on every aspect of customer support. However, the ITIL library is aimed primarily at in-house customer support and is not used very much by commercial software vendors.

For commercial software customer support, some trade books are available, but the literature tends to be dominated by white papers and monographs published by customer support outsource companies.

Although these tend to be marketing texts, some of them do provide useful information about the mechanics of customer support. There are also interesting reports available from companies that provide customer-support automation, which is both plentiful and seems to cover a wide range of features.

Given the fact that customer support is a critical activity of the software industry in 2009, a great deal more research and study are indicated. Research is needed on the relationship between quality and customer support, on the role of user associations and volunteer groups, and on the potential automation that might improve customer support. In particular, research is needed on providing customer support for deaf and hard-of-hearing customers, blind customers, and those with other physical challenges.

Software Test Organizations

There are ten problems with discussing software test organizations that need to be highlighted:

1. There are more than 15 different kinds of software testing.
2. Many kinds of testing can be performed either by developers, by in-house test organizations, by outsource test organizations, or by quality assurance teams based on company test strategies.
3. With Agile teams and with hierarchical organizations, testers will probably be embedded with developers and not have separate departments.
4. Matrix organizations testers would probably be in a separate testing organization reporting to a skill manager, but assigned to specific projects as needed.
5. Some test organizations are part of quality assurance organizations and therefore have several kinds of specialists besides testing.
6. Some quality assurance organizations collect data on test results, but do no testing of their own.
7. Some testing organizations are called “quality assurance” and perform only testing. These may not perform other QA activities such as moderating inspections, measuring quality, predicting quality, teaching quality, and so on.
8. For any given software application, the number of separate kinds of testing steps ranges from a low of 1 form of testing to a high of 17 forms of testing based on company test strategies.

9. For any given software application, the number of test and/or quality assurance organizations that are part of its test strategy can range from a low of one to a high of five, based on company quality strategies.
10. For any given defect removal activity, including testing, as many as 11 different kinds of specialists may take part.

As can perhaps be surmised from the ten points just highlighted, there is no standard way of testing software applications in 2009. Not only is there no standard way of testing, but there are no standard measures of test coverage or defect removal efficiency, although both are technically straightforward measurements.

The most widely used form of test measurement is that of test coverage, which shows the amount of code actually executed by test cases. Test coverage measures are fully automated and therefore easy to do. This is a useful metric, but much more useful would be to measure defect removal efficiency as well.

Defect removal efficiency is more complicated and not fully automated. To measure the defect removal efficiency of a specific test stage such as *unit test*, all defects found by the test are recorded. After unit test is finished, all other defects found by all other tests are recorded, as are defects found by customers in the first 90 days. When all defects have been totaled, then removal efficiency can be calculated.

Assume unit test found 100 defects, function test and later test stages found 200 defects, and customers reported 100 defects in the first 90 days of use. The total number of defects found was 400. Since unit test found 100 out of 400 defects, in this example, its efficiency is 25 percent, which is actually not far from the 30 percent average value of defect removal efficiency for unit test.

(A quicker but less reliable method for determining defect removal efficiency is that of defect seeding. For example, if 100 known bugs were seeded into the software discussed in the previous paragraph and 25 were found, then the defect removal efficiency level of 25 percent could be calculated immediately. However, there is no guarantee that the “tame” bugs that were seeded would be found at exactly the same rate as “wild” bugs that are made by accident.)

It is an unfortunate fact that most forms of testing are not very efficient and find only about 25 percent to 40 percent of the bugs that are actually present, although the range is from less than 20 percent to more than 70 percent.

It is interesting that there is much debate over *black box testing*, which lacks information on internals; *white box testing*, with full visibility of internal code; and *gray box testing*, with visibility of internals, but testing is at the external level.

So far as can be determined, the debate is theoretical, and few experiments have been performed to measure the defect removal efficiency levels of black, white, or gray box testing. When measures of efficiency are taken, white box testing seems to have higher levels of defect removal efficiency than black box testing.

Because many individual test stages such as unit test are so low in efficiency, it can be seen why several different kinds of testing are needed. The term *cumulative defect removal efficiency* refers to the overall efficiency of an entire sequence of tests or defect removal operations.

As a result of lack of testing standards and lack of widespread testing effectiveness measurements, testing by itself does not seem to be a particularly cost-effective approach for achieving high levels of quality. Companies that depend purely upon testing for defect removal almost never top 90 percent in cumulative defect removal, and often are below 75 percent.

The newer forms of testing such as test-driven development (TDD) use test cases as a form of specification and create the test cases first, before the code itself is created. As a result, the defect removal efficiency of TDD is higher than many forms of testing and can top 85 percent. However, even with TDD, bad-fix injection needs to be factored into the equation. About 7 percent of attempts to fix bugs accidentally include new bugs in the fixes.

If TDD is combined with other approaches such as formal inspection of the test cases and static analysis of the code, then defect removal efficiency can top 95 percent.

There is some ambiguity in the data that deals with automatic testing versus manual testing. In theory, automatic testing should have higher defect removal efficiency than manual testing in at least 70 percent of trials. For example, manual unit testing averages about 30 percent in terms of defect removal efficiency, while automatic testing may top 50 percent. However, testing skills vary widely among software engineers and programmers, and automatic testing also varies widely. More study of this topic is indicated.

The poor defect removal efficiency of normal testing brings up an important question: *If testing is not very effective in finding and removing bugs, what is effective?* This is an important question, and it is also a question that should be answered in a book entitled *Software Engineering Best Practices*.

The answer to the question of “What is effective in achieving high levels of quality?” is that a combination of defect prevention and multiple forms of defect removal is needed for optimum effectiveness.

Defect prevention refers to methods and techniques that can lower defect potentials from U.S. averages of about 5.0 per function point.

Examples of methods that have demonstrated effectiveness in terms of defect prevention include the higher levels of the capability maturity model integration (CMMI), joint application design (JAD), quality function deployment (QFD), root-cause analysis, Six Sigma for software, the Team Software Process (TSP), and also the Personal Software Process (PSP).

For small applications, the Agile method of having an embedded user as part of the team can also reduce defect potentials. (The caveat with embedded users is that for applications with more than about 50 users, one person cannot speak for the entire set of users. For applications with thousands of users, having a single embedded user is not adequate. In such cases, focus groups and surveys of many users are necessary.)

As it happens, formal inspections of requirements, design, and code serve double duty and are very effective in terms of defect prevention as well as being very effective in terms of defect removal. This is because participants in formal inspections spontaneously avoid making the same mistakes that are found during the inspections.

The combination of methods that have been demonstrated to raise defect removal efficiency levels includes formal inspections of requirements, design, code, and test materials; static analysis of code prior to testing; and then a test sequence that includes at least eight forms of testing: (1) unit test, (2) new function test, (3) regression test, (4) performance test, (5) security test, (6) usability test, (7) system test, and (8) some form of external test with customers or clients, such as beta test or acceptance test.

Such a combination of pretest inspections, static analysis, and at least eight discrete test stages will usually approach 99 percent in terms of cumulative defect removal efficiency levels. Not only does this combination raise defect removal efficiency levels, but it is also very cost-effective.

Projects that top 95 percent in defect removal efficiency levels usually have shorter development schedules and lower costs than projects that skimp on quality. And, of course, they have much lower maintenance and customer support costs, too.

Testing is a teachable skill, and there are a number of for-profit and nonprofit organizations that offer seminars, classes, and several flavors of certification for test personnel. While there is some evidence that certified test personnel do end up with higher levels of defect removal efficiency than uncertified test personnel, the poor measurement and benchmark practices of the software industry make that claim somewhat anecdotal. It would be helpful if test certification included a learning segment on how to measure defect removal efficiency.

Following in Table 5-3 are examples of a number of different forms of software inspection, static analysis, and testing, with the probable organization that performs each activity indicated.

TABLE 5-3 Forms of Software Defect Removal Activities

Pretest Removal Inspections		Performed by
1.	Requirements	Analysts
2.	Design	Designers
3.	Code	Programmers
4.	Test plans	Testers
5.	Test cases	Testers
6.	Static analysis	Programmers
General Testing		
7.	Subroutine test	Programmers
8.	Unit test	Programmers
9.	New function test	Testers or programmers
10.	Regression test	Testers or programmers
11.	System test	Testers or programmers
Special Testing		
12.	Performance testing	Performance specialists
13.	Security testing	Security specialists
14.	Usability testing	Human factors specialists
15.	Component testing	Testers
16.	Integration testing	Testers
17.	Nationalization testing	Foreign language experts
18.	Platform testing	Platform specialists
19.	SQA validation testing	Software quality assurance
20.	Lab testing	Hardware specialists
External Testing		
21.	Independent testing	External test company
22.	Beta testing	Customers
23.	Acceptance testing	Customers
Special Activities		
24.	Audits	Auditors, SQA
25.	Independent verification and validation (IV&V)	IV&V contractors
26.	Ethical hacking	Hacking consultants

Table 5-3 shows 26 different kinds of defect removal activity carried out by a total of 11 different kinds of internal specialists, 3 specialists from outside companies, and also by customers. However, only very large and sophisticated high-technology companies would have such a rich mixture of specialization and would utilize so many different kinds of defect removal.

Smaller companies would either have the testing carried out by software engineers or programmers (who often are not well trained), or they would

have a testing group staffed primarily by testing specialists. Testing can also be outsourced, although as of 2009, this activity is not common.

At this point, it is useful to address three topics that are not well covered in the testing literature:

1. How many testers are needed for various kinds of testing?
2. How many test cases are needed for various kinds of testing?
3. What is the defect removal efficiency of various kinds of testing?

Table 5-4 shows the approximate staffing levels for the 17 forms of testing that were illustrated in Table 5-3. Note that this information is only approximate, and there are wide ranges for each form of testing.

Because testing executes source code, the information in Table 5-4 is based on source code counts rather than on function points. With more than 700 programming languages ranging from assembly through

TABLE 5-4 Test Staffing for Selected Test Stages

Application language	Java	
Application code size	50,000	
Application KLOC	50	
Function points	1,000	
General Testing	Assignment Scope	Test Staff
1. Subroutine test	10,000	5.00
2. Unit test	10,000	5.00
3. New function test	25,000	2.00
4. Regression test	25,000	2.00
5. System test	50,000	1.00
Special Testing		
6. Performance testing	50,000	1.00
7. Security testing	50,000	1.00
8. Usability testing	25,000	2.00
9. Component testing	25,000	2.00
10. Integration testing	50,000	1.00
11. Nationalization testing	1,50,000	0.33
12. Platform testing	50,000	1.00
13. SQA validation testing	75,000	0.67
14. Lab testing	50,000	1.00
External Testing		
15. Independent testing	7,500	6.67
16. Beta testing	25,000	2.00
17. Acceptance testing	25,000	2.00

modern languages such as Ruby and E, the same application illustrated in Table 5-4 might vary by more than 500 percent in terms of source code size. Java is the language used in Table 5-4 because it is one of the most common languages in 2009.

The column labeled “Assignment Scope” illustrates the amount of source code that one tester will probably be responsible for testing. Note that there are very wide ranges in assignment scopes based on the experience levels of test personnel, on the cyclomatic complexity of the code, and to a certain extent, on the specific language or combination of languages in the application being tested.

Because the testing shown in Table 5-4 involves a number of different people with different skills who probably would be from different departments, the staffing breakdown for all 17 tests would include 5 developers through unit test; 2 test specialists for integration and system test; 3 specialists for security, nationalization, and usability test; 1 SQA specialist; 7 outside specialists from other companies; and 2 customers: 20 people in all.

Of course, it is unlikely that any small application of 1,000 function points or 50 KLOC (thousands of lines of code) would use (or need) all 17 of these forms of testing. The most probable sequence for a 50-KLOC Java application would be 6 kinds of testing performed by 5 developers, 2 test specialists, and 2 users, for a total of 9 test personnel in all.

In Table 5-5, data from the previous tables is used as the base for staffing, but the purpose of Table 5-5 is to show the approximate numbers of test cases produced for each test stage, and then the total number of test cases for the entire application. Here, too, there are major variations, so the data is only approximate.

The code defect potential for the 50 KLOC code sample of the Java application would be about 1,500 total bugs, which is equal to 1.5 code bugs per function point, or 30 bugs per KLOC. (Note that earlier bugs in requirements and design are excluded and assumed to have been removed before testing begins.)

If all 17 of the test stages were used, they would probably detect about 95 percent of the total bugs present, or 1,425 in all. That would leave 75 bugs latent when the application is delivered. Assuming both the numbers for potential defects and the numbers for test cases are reasonably accurate (a questionable assumption) then it takes an average of 1.98 test cases to find 1 bug.

Of course, since only about 6 out of the 17 test stages are usually performed, the removal efficiency would probably be closer to 75 percent, which is why additional nontest methods such as inspections and static analysis are needed to achieve really high levels of defect removal efficiency.

If even this small 50-KLOC example uses more than 2,800 test cases, it is obvious that corporations with hundreds of software applications

TABLE 5-5 Test Cases for Selected Test Stages

Application language	Java			
Application code size	50,000			
Application KLOC	50			
Function points	1,000			
	Test Staff	Test Cases Per KLOC	Total Test Cases	Test Cases Per Person
General Testing				
1. Subroutine test	5.00	12.00	600	120.00
2. Unit test	5.00	10.00	500	100.00
3. New function test	2.00	5.00	250	125.00
4. Regression test	2.00	4.00	200	100.00
5. System test	1.00	3.00	150	150.00
Special Testing				
6. Performance testing	1.00	1.00	50	50.00
7. Security testing	1.00	3.00	150	150.00
8. Usability testing	2.00	3.00	150	75.00
9. Component testing	2.00	1.50	75	37.50
10. Integration testing	1.00	1.50	75	75.00
11. Nationalization testing	0.33	0.50	25	75.76
12. Platform testing	1.00	2.00	100	100.00
13. SQA validation testing	0.67	1.00	50	74.63
14. Lab testing	1.00	1.00	50	50.00
External Testing				
15. Independent testing	6.67	4.00	200	29.99
16. Beta testing	2.00	2.00	100	50.00
17. Acceptance testing	2.00	2.00	100	50.00
TOTAL TEST CASES			2825	
TEST CASES PER KLOC			56.50	
TEST CASES PER PERSON (20 TESTERS)			141.25	

will eventually end up with millions of test cases. Once created, test cases have residual value for regression test purposes. Fortunately, a number of automated tools can be used to store and manage test case libraries.

The existence of such large test libraries is a necessary overhead of software development and maintenance. However, this topic needs additional study. Creating reusable test cases would seem to be of value. Also, there are often errors in test cases, which is why inspections of test plans and test cases are useful.

With hundreds of different people creating test cases in large companies and government agencies, there is a good chance that duplicate tests will accidentally be created. In fact, this does occur, and a study at

IBM noted about 30 percent redundancy or duplicates in one software lab’s test library.

The final Table 5-6 in this section shows defect removal efficiency levels against six sources of error: requirements defects, design defects, coding defects, security defects, defects in test cases, and performance defects.

Table 5-6 is complicated by the fact that not every defect removal method is equally effective against each type of defect. In fact, many

TABLE 5-6 Defect Removal Efficiency by Defect Type

Pretest Removal Inspections:	Req. defects	Des. defects	Code defects	Sec. defects	Test defects	Perf. defects
1. Requirements	85.00%					
2. Design		85.00%		25.00%		
3. Code			85.00%	40.00%		15.00%
4. Test plans					85.00%	
5. Test cases					85.00%	
6. Static analysis		30.00%	87.00%	25.00%		20.00%
General Testing						
7. Subroutine test			35.00%			10.00%
8. Unit test			30.00%			10.00%
9. New function test		15.00%	35.00%			10.00%
10. Regression test			15.00%			
11. System test	10.00%	20.00%	25.00%	7.00%		25.00%
Special Testing						
12. Performance testing	5.00%	10.00%				70.00%
13. Security testing				65.00%		
14. Usability testing	10.00%	10.00%				
15. Component testing		10.00%	25.00%			
16. Integration testing		10.00%	30.00%			
17. Nationalization testing	3.00%					
18. Platform testing		10.00%				
19. SQA validation testing	5.00%	5.00%	15.00%			
20. Lab testing	10.00%	10.00%	10.00%			20.00%
External Testing						
21. Independent testing		5.00%	30.00%	5.00%	5.00%	10.00%
22. Beta testing	30.00%		25.00%	10.00%		15.00%
23. Acceptance testing	30.00%		20.00%	5.00%		15.00%
Special Activities						
24. Audits	15.00%	10.00%				
25. Independent verification and validation (IV&V)	10.00%	10.00%		10.00%		
26. Ethical hacking				85.00%		

forms of defect removal have 0 percent efficiency against security flaws. Coding defects are the easiest type of defect to remove; requirements defects, security defects, and defects in test materials are the most difficult to eliminate.

Historically, formal inspections have the highest levels of defect removal efficiency against the broadest range of defects. The more recent method of static analysis has a commendably high level of defect removal efficiency against coding defects, but currently operates only on about 15 programming languages out of more than 700.

The data in Table 5-6 has a high margin of error, but the table itself shows the kind of data that needs to be collected in much greater volume to improve software quality and raise overall levels of defect removal efficiency across the software industry. In fact, *every* software application larger than 1,000 function points in size should collect this kind of data.

One important source of defects is not shown in Table 5-6 and that is bad-fix injection. About 7 percent of bug repairs contain a fresh bug in the repair itself. Assume that unit testing found and removed 100 bugs in an application. But there is a high probability that 7 new bugs would be accidentally injected into the application due to errors in the fixes themselves. (Bad-fix injections greater than 25 percent may occur with error-prone modules.)

Bad-fix injection is a very common source of defects in software, but it is not well covered either in the literature on testing or in the literature on software quality assurance.

Another quality issue that is not well covered is that of error-prone modules. As mentioned elsewhere in this book, bugs are not randomly distributed, but tend to clump in a small number of very buggy modules.

If an application contains one or more error-prone modules, then defect removal efficiency levels against those modules may be only half of the values shown in Table 5-6, and bad-fix injection rates may top 25 percent. This is why error-prone modules can seldom be repaired, but need to be surgically removed and replaced by a new module.

In spite of the long history of testing and the large number of test personnel employed by the software industry, a great deal more research is needed. Some of the topics that need research are automatic generation of test cases from specifications, developing reusable test cases, better predictions of test case numbers and removal efficiency, and much better measurement of test results in terms of defect removal efficiency levels.

Demographics In the software world, testing has long been one of the major development activities, and test personnel are among the largest software occupation groups. But to date there is no accurate census of

test personnel, due in part to the fact that so many different kinds of specialists get involved in testing.

Because testing is on the critical path for releasing software, there is a tendency for software project managers or even senior executives to put pressure on test personnel to truncate testing when schedules are slipping. By having test organizations reporting to separate skill managers, as opposed to project or application managers, this adds a measure of independence.

However, testing is such an integral part of software development that test personnel need to be involved essentially from the first day that development begins. Whether testers report to skill managers or are embedded in project teams, they need early involvement during requirement and design. This is especially true with test-driven development (TDD), where test cases are an integral part of the requirements and design processes.

Project size The minimum size of applications where formal testing is mandatory is about 100 function points. As a rule, the larger the application, the more kinds of pretest defect removal activities and more kinds of testing are needed to be successful or even to finish the application at all.

For large systems less than 10,000 function points, inspections, static analysis, security analysis, and about ten forms of testing are needed to achieve high levels of defect removal efficiency. Unfortunately, many companies skimp on testing and nontest activities, so U.S. average results are embarrassingly bad: 85 percent cumulative defect removal efficiency. These results have been fairly flat or constant from 1996 through 2009.

Productivity rates There are no effective productivity rates for testing. There are no effective size metrics for test cases. At a macro level, testing productivity can be measured by using “work hours per function point” or the reciprocal “function points per staff month,” but those measures are abstract and don’t really capture the essence of testing.

Measures such as “test cases created per month” or “test cases executed per month” send the wrong message, because they might encourage extra testing simply to puff up the results and not raise defect removal efficiency.

Measures such as “defects detected per month” are unreliable, because for really top-gun developers, there may not be very many defects to find. The “cost per defect” metric is also unreliable for the same reason. Testers will still run many test cases whether an application has any bugs or not. As a result, cost per defect rises as defect quantities go down; hence the cost per defect metric penalizes quality.

Schedules The primary schedule issues for test personnel are those of test case creation and test case execution. But testing schedules depend more upon the number of bugs found and the time it takes to repair the bugs than on test cases.

One factor that is seldom measured but also delays test schedules is bugs or defects in test cases themselves. A study done some years ago by IBM found more bugs in test cases than in the applications being tested. This topic is not well covered by the testing literature. (This was the same study that had found about 30 percent redundant or duplicate test cases in test libraries.) Running duplicate test cases adds to testing costs and schedules, but not to defect removal efficiency levels.

When testing starts on applications with high volumes of defects, the entire schedule for the project is at risk, because testing schedules will extend far beyond their planned termination. In fact, testing delays due to excessive defect volumes is the main reason for software schedule delays.

The most effective way to minimize test schedules is to have very few defects present because pretest inspections and static analysis found most of them before testing began. Defect prevention such as TSP or joint application design (JAD) can also speed up test schedules.

For the software industry as a whole, delays in testing due to excessive bugs is a major cause of application cost and schedule overruns and also of project cancellations. Because long delays and cancellation trigger a great deal of litigation, high defect potentials and low levels of defect removal efficiency are causative factors in breach of contract lawsuits.

Quality Testing by itself has not been efficient enough in finding bugs to be the only form of defect removal used on major software applications. Testing alone almost never tops 85 percent defect removal efficiency, with the exception of the newer test-driven development (TDD), which can hit 90 percent.

Testing combined with formal inspections and static analysis achieves higher levels of defect removal efficiency, shorter schedules, and lower costs than testing alone. Moreover, these savings not only benefit development, but also lower the downstream costs of customer support and maintenance.

Readers who are executives and qualified to sign contracts are advised to consider 95 percent as the minimum acceptable level of defect removal efficiency. Every outsource contract, every internal quality plan, and every license with a software vendor should require proof that the development organization will top 95 percent in defect removal efficiency.

Specialization Testing specialization covers a wide range of skills. However, for many small companies with a generalist philosophy, software developers may also serve as software testers even though they may not be properly trained for the role.

For large companies, a formal testing department staffed by testing specialists will give better results than development testing by itself. For very large multinational companies and for companies that build systems and embedded software, test and quality assurance specialists will be numerous and have many diverse skills.

There are several forms of test certification available. Testers who go to the trouble of achieving certification are to be commended for taking their work seriously. However, there is not a great deal of empirical data that compares the defect removal efficiency levels of tests carried out by certified testers versus the same kind of testing performed by uncertified testers.

Cautions and counter indications The main caution about testing is that it does not find very many bugs or defects. For more than 50 years, the software industry has routinely delivered large software applications with hundreds of latent bugs, in spite of extensive testing.

A second caution about testing is that testing cannot find requirements errors such as the famous Y2K problem. Once an error becomes embedded in requirements and is not found via inspections, quality function deployment (QFD), or some other nontest approach, all that testing will accomplish is to confirm the error. This is why correct requirements and design documents are vital for successful testing. This also explains why formal inspections of requirements and design documents raise testing efficiency by about 5 percent per test stage.

Conclusions The literature on testing is extensive but almost totally devoid of quantitative data that deals with defect removal efficiency, with testing costs, with test staffing, with test specialization, with return on investment (ROI), or with the productivity of test personnel. However, there are dozens of books and hundreds of web sites with information on testing.

Several nonprofit organizations are involved with testing, such as the Association for Software Testing (AST) and the American Society for Quality (ASQ). There is also a Global Association for Software Quality (GASQ).

There are local and regional software quality organizations in many cities. There are also for-profit test associations that hold a number of conferences and workshops, and also offer certification exams.

Given the central role of testing over the past 50 years of software engineering, the gaps in the test literature are surprising and dismaying.

A technical occupation that has no clue about the most efficient and cost-effective methods for preventing or removing serious errors is not qualified to be called “engineering.”

Some of the newer forms of testing such as test-driven development (TDD) are moving in a positive direction by shifting test case development to earlier in the development cycle, and by joining test cases with requirements and design. These changes in test strategy result in higher levels of defect removal efficiency coupled with lower costs as well.

But to achieve really high levels of quality in a cost-effective manner, testing alone has always been insufficient and remains insufficient in 2009. A synergistic combination of defect prevention and a multiphase suite of defect removal activities that combine inspections, static analysis, automated testing, and manual testing provide the best overall results.

For the software industry as a whole, defect potentials have been far too high, and defect removal efficiency far too low for far too many years. This unfortunate combination has raised development costs, stretched out development schedules, caused many failures and also litigation, and raised maintenance and customer support costs far higher than they should be.

Defect prevention methods such as Team Software Process (TSP), quality function deployment (QFD), Six Sigma for software, joint application design (JAD), participation in inspections, and certified reusable components have the theoretical potential of lowering defect potentials by 80 percent or more compared with 2009. In other words, defect potentials could drop from about 5.0 per function point down to about 1.0 per function point or lower.

Defect removal combinations that include formal inspections, static analysis, test-driven development, using both automatic and manual testing, and certified reusable test cases could raise average defect removal efficiency levels from today’s approximate average of about 85 percent in 2009 up to about 97 percent. Levels that approach 99.9 percent could even be achieved in many cases.

Effective combinations of defect prevention and defect removal activities are available in 2009 but seldom used except by a few very sophisticated organizations. What is lacking is not so much the technologies that improve quality, but awareness of how effective the best combinations really are. Also lacking is awareness of how ineffective testing alone can be. It is lack of widespread quality measurements and lack of quality benchmarks that are delaying improvements in software quality.

Also valuable are predictive estimating tools that can predict both defect potentials and the defect removal efficiency levels of any combination of review, inspection, static analysis, automatic test stage, and manual test stage. Such tools exist in 2009 and are marketed by

companies such as Software Productivity Research (SPR), SEER, Galorath, and Price Systems. Even more sophisticated tools that can predict the damages that latent defects cause to customers exist in prototype form.

The final conclusion is that until the software industry can routinely top 95 percent in average defect removal efficiency levels, and hit 99 percent for critical software applications, it should not even pretend to be a true engineering discipline. The phrase “software engineering” without effective quality control is a hoax.

Software Quality Assurance (SQA) Organizations

The author of this book worked for five years in IBM’s Software Quality Assurance organizations in Palo Alto and Santa Teresa, California. As a result, the author may have a residual bias in favor of SQA groups that function along the lines of IBM’s SQA groups.

Within the software industry, there is some ambiguity about the role and functions of SQA groups. Among the author’s clients (primarily Fortune 500 companies), following is an approximate distribution of how SQA organizations operate:

- In about 50 percent of companies, SQA is primarily a testing organization that performs regression tests, performance tests, system tests, and other kinds of testing that are used for large systems as they are integrated. The SQA organization reports to a vice president of software engineering, to a CIO, or to local development managers and is not an independent organization. There may be some responsibility for measuring quality, but testing is the main focus. These SQA organizations tend to be quite large and may employ more than 25 percent of total software engineering personnel.
- In about 35 percent of companies, SQA is a focal point for estimating and measuring quality and ensuring adherence to local and national quality standards. But the SQA group is separate from testing organizations, and performs only limited and special testing such as standards adherence. To have an independent view, the SQA organization reports to its own vice president of quality and is not part of the development or test organizations. (This is the form of SQA that IBM had when the author worked there.) These organizations tend to be fairly small and employ between 1 percent and 3 percent of total software engineering personnel.
- About 10 percent of companies have a testing organization but no SQA organization at all. The testing group usually reports to the CIO or to a vice president or senior software executive. In such situations, testing is the main focus, although there may be some measurement

of quality. While the testing organization may be large, the staffing for SQA is zero.

- In about 5 percent of companies, there is a vice president of SQA and possibly one or two assistants, but nobody else. In this situation, SQA is clearly nothing more than an act that can be played when customers visit. Such organizations may have testing groups that report to various development managers. The so-called SQA organizations where there are executives but no SQA personnel employ less than one-tenth of one percent of total software engineering personnel.

Because software quality assurance (SQA) is concerned with more than testing, it is interesting to look at the activities and roles of “traditional” SQA groups that operate independently from test organizations.

1. Collecting and measuring software quality during development and after release, including analyzing test results and test coverage. In some organizations such as IBM, defect removal efficiency levels are also calculated.
2. Predicting software quality levels for major new applications, including construction of special quality estimating tools.
3. Performing statistical studies of quality or carrying out root-cause analysis.
4. Examining and teaching quality methods such as quality function deployment (QFD) or Six Sigma for software.
5. Participating in software inspections as moderators or recorders, and also teaching inspections.
6. Ensuring that local, national, and international quality standards are followed. SQA groups are important for achieving ISO 9000 certification, for example.
7. Monitoring the activities associated with the various levels of the capability maturity model integration (CMMI). SQA groups play a major part in software process improvements and ascending to the higher levels of the CMMI.
8. Performing specialized testing such as standards adherence.
9. Teaching software quality topics to new employees.
10. Acquiring quality benchmark data from external organizations such as the International Software Benchmarking Standards Group (ISBSG).

A major responsibility of IBM’s SQA organization was determining whether the quality level of new applications was likely to be good

enough to ship the application to customers. The SQA organization could stop delivery of software that was felt to have insufficient quality levels.

Development managers could appeal an SQA decision to stop the release of questionable software, and the appeal would be decided by IBM's president or by a senior vice president. This did not happen often, but when it did, the event was taken very seriously by all concerned. The fact that the SQA group was vested with this power was a strong incentive for development managers to take quality seriously.

Obviously, for SQA to have the power to stop delivery of a new application, the SQA team had to have its own chain of command and its own senior vice president independent of the development organization. If SQA had reported to a development executive, then threats or coercion might have made the SQA role ineffective.

One unique feature of the IBM SQA organization was a formal "SQA research" function, which provided time and resources for carrying out research into topics that were beyond the state of the art currently available. For example, IBM's first quality estimation tool was developed under this research program. Researchers could submit proposals for topics of interest, and those selected and approved would be provided with time and with some funding if necessary.

Several companies encourage SQA and other software engineering personnel to write technical books and articles for outside journals such as *CrossTalk* (the U.S. Air Force software journal) or some of the IEEE journals.

One company, ITT, as part of its software engineering research lab, allowed articles to be written during business hours and even provided assistance in creating camera-ready copy for books. It is a significant point that authors should be allowed to keep the royalties from the technical books that they publish.

It is an interesting phenomenon that almost every company with defect removal efficiency levels that average more than 90 percent has a formal and active SQA organization. Although formal and active SQA groups are associated with better-than-average quality, the data is not sufficient to assert that SQA is the primary cause of high quality.

The reason is that most organizations that have low software quality don't have any measurements in place, and their poor quality levels only show up if they commission a special assessment, or if they are sued and end up in court.

It would be nice to say that *organizations with formal SQA teams average greater than 90 percent in defect removal efficiency* and that similar companies doing similar software that lack formal SQA teams *average less than 80 percent in defect removal efficiency*. But the unfortunate fact is that only the companies with formal SQA teams are likely to know

what their defect removal efficiency levels are. In fact, quality measurement practices are so poor that even some companies that do have an SQA organization do not know their defect removal efficiency levels.

Demographics In the software world, SQA is not large numerically, but has been a significant source of quality innovation. There are perhaps 5,000 full-time SQA personnel employed in the United States as of 2009.

SQA organizations are very common in companies that build systems software, embedded software, or commercial software, such as SAP, Microsoft, Oracle, and the like. SQA organizations are less common in IT groups such as banks and finance companies, although they do occur within the larger companies.

Many cities have local SQA organizations, and there are also national and international equality associations as well.

There is one interesting anomaly with SQA support of software applications. Development teams that use the Team Software Process (TSP) have their own internal equivalent of SQA and also collect extensive data on bugs and quality. Therefore, TSP teams normally do not have any involvement from corporate SQA organizations. They of course provide data to the SQA organization for corporate reporting purposes, but they don't have embedded SQA personnel.

Project size Normally, SQA involvement is mandatory for large applications above about 2,500 function points. While SQA involvement might be useful for smaller applications, they tend to have better quality than large applications. Since SQA resources are limited, concentrating on large applications is perhaps the best use of SQA personnel.

Productivity rates There are no effective productivity rates for SQA groups. However, it is an interesting and important fact that productivity rates for software applications that do have SQA involvement, and which manage to top 95 percent in defect removal efficiency, are usually much better than applications of the same size that lack SQA.

Even if SQA productivity itself is ambiguous, measuring the quality and productivity of the applications that are supported by SQA teams indicates that SQA has significant business value.

Schedules The primary schedule issues for SQA teams are the overall schedules for the applications that they support. As with productivity and quality, there is evidence that an SQA presence on an application tends to prevent schedule delays.

Indeed if SQA is successful in introducing formal inspections, schedules can even be shortened.

The most effective way to shorten software development schedules is to have very few defects due to defect prevention, and to remove most of them prior to testing due to pretest inspections and static analysis. Since SQA groups push hard for both defect prevention and early defect removal, an effective SQA group will benefit development schedules—and especially so for large applications, which typically run late.

For the software industry as a whole, delays due to excessive bugs are a major cause of application cost and schedule overruns and also of project cancellations. Effective SQA groups can minimize the endemic problems.

It is a proven fact that an effective SQA organization can lead to significant cost reductions and significant schedule improvements for software projects. Yet because the top executives in many companies do not understand the economic value of high quality and regard quality as a luxury rather than a business necessity, SQA personnel are among the first to be let go during a recession.

Quality The roles of SQA groups center on quality, including quality measurement, quality predictions, and long-range quality improvement. SQA groups also have a role in ISO standards and the CMMI. SQA organizations also teach quality courses and assist in the deployment of methods such as quality function deployment (QFD) and Six Sigma for software. In fact, it is not uncommon for many SQA personnel to be Six Sigma black belts.

There is some uncertainty in 2009 about the role of SQA groups when test-driven development (TDD) is utilized. Because TDD is fairly new, the intersection of TDD and SQA is still evolving.

As already mentioned in the testing section of this chapter, readers who are executives and qualified to sign contracts are advised to consider 95 percent as the minimum acceptable level of defect removal efficiency. Every outsource contract, every internal quality plan, and every license with a software vendor should require proof that the development organization will top 95 percent in defect removal efficiency.

There is one troubling phenomenon that needs more study. Large systems above 10,000 function points are often released with hundreds of latent bugs in spite of extensive testing and sometimes in spite of large SQA teams. Some of these large systems ended up in lawsuits where the author happened to an expert witness. It usually happened that the advice of the SQA teams was not taken, and that the project manager skimped on quality control in a misguided attempt to compress schedules.

Specialization SQA specialization covers a wide range of skills that can include statistical analysis, function point analysis, and also testing. Other special skills include Six Sigma, complexity analysis, and root-cause analysis.

Cautions and counter indications The main caution about SQA is that it is there to help, and not to hinder. Dogmatic attitudes are counterproductive for effective cooperation with development and testing groups.

Conclusions An effective SQA organization can benefit not only quality, but also schedules and costs. Unfortunately, during recessions, SQA teams are among the first to be affected by layoffs and downsizing. As the recession of 2009 stretches out, it causes uncertainty about the future of SQA in U.S. business.

Because quality benefits costs and schedules, it is urgent for SQA teams to take positive steps to include measures of defect removal efficiency and measures of the economic value of quality as part of their standard functions. If SQA could expand the number of formal quality benchmarks brought in to companies, and collect data for submission to benchmark groups, the data would benefit both companies and the software industry.

Several nonprofit organizations are involved with SQA, such as the American Society for Quality (ASQ). There is also a Global Association for Software Quality (GASQ).

Local and regional software quality organizations are in many cities. Also, for-profit SQA associations such as the Quality Assurance Institute (QAI) hold a number of conferences and workshops, and also offer certification exams.

SQA needs to assist in introducing a synergistic combination of defect prevention and a multiphase suite of defect removal activities that combine inspections, static analysis, automated testing, and manual testing. There is no silver bullet for quality, but fusions of a variety of quality methods can be very effective. SQA groups are the logical place to provide information and training for these effective hybrid methods.

Effective combinations of defect prevention and defect removal activities are available in 2009, but seldom used except by a few very sophisticated organizations. As mentioned in the testing section of this chapter, what is lacking is not so much the technologies that improve quality, but awareness of how effective the best combinations really are. It is lack of widespread quality measurements and lack of quality benchmarks that are delaying improvements in software quality.

Also valuable are predictive estimating tools that can predict both defect potentials and the defect removal efficiency levels of any combination of review, inspection, static analysis, automatic test stage, and manual test stage. Normally, SQA groups will have such tools and use them frequently. In fact, the industry's first software quality prediction tool was developed by the IBM SQA organization in 1973 in San Jose, California.

The final conclusion is that SQA groups need to keep pushing until the software industry can routinely top 95 percent in average defect removal efficiency levels, and hit 99 percent for critical software applications. Any results less than these are insufficient and unprofessional.

Summary and Conclusions

Fred Brooks, one of the pioneers of software at IBM, observed in his classic book *The Mythical Man Month* that software was strongly affected by organization structures. Not long after Fred published, the author of this book, who also worked at IBM, noted that large systems tended to be decomposed to fit existing organization structures. In particular, some major features were artificially divided to fix standard eight-person departments.

This book only touches the surface of organizational issues. Deeper study is needed on the relative merits of small teams versus large teams. In addition, the “average” span of control of eight employees reporting to one manager may well be in need of revision. Studies of the effectiveness of various team sizes found that raising the span of control from 8 up to 12 would allow marginal managers to return to technical work and would minimize managerial disputes, which tend to be endemic. Further, since software application sizes are increasing, larger spans of control might be a better match for today's architecture.

Another major topic that needs additional study is that of really large software teams that may include 500 or more personnel and dozens of specialists. There is very little empirical data on the most effective methods for dealing with such large groups with diverse skills. If such teams are geographically dispersed, that adds yet another topic that is in need of additional study.

More recently Dr. Victor Basili, Nachiappan Nagappan, and Brendan Murphy studied organization structures at Microsoft and concluded that many of the problems with Microsoft Vista could be traced back to organizational structure issues.

However, in 2009, the literature on software organization structures and their impact is sparse compared with other topics that influence software engineering such as methods, tools, programming languages, and testing.

Formal organization structures tend to be territorial because managers are somewhat protective of their spheres of influence. This tends to narrow the focus of teams. Newer forms of informal organizations that support cross-functional communication are gaining in popularity. Cross-functional contacts also increase the chances of innovation and problem solving.

Software organization structures should be dynamic and change with technology, but unfortunately, they often are a number of years behind where they should be.

As the recession of 2009 continues, it may spur additional research into organizational topics. For example, new subjects that need to be examined include wiki sites, virtual departments that communicate using virtual reality, and the effectiveness of home offices to minimize fuel consumption.

A very important topic with almost no literature is that of dealing with layoffs and downsizing in the least disruptive way. That topic is discussed in Chapters 1 and 2 of this book, but few additional citations exist. Because companies tend to get rid of the wrong people, layoffs often damage operational efficiency levels for years afterwards.

Another important topic that needs research, given the slow development schedules for software, would be a study of global organizations located in separate time zones eight hours apart, which would allow software applications and work products to be shifted around the globe from team to team, and thus permit 24-hour development instead of 8-hour development.

A final organizational topic that needs additional study are the optimum organizations that can create reusable modules and other reusable deliverables, and then construct software applications from reusable components rather than coding them on a line-by-line basis.

Readings and References

- Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1995.
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Crosby, Philip B. *Quality is Free*. New York: New American Library, Mentor Books, 1979.
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- DeMarco, Tom. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1999.
- Glass, Robert L. *Software Creativity*, Second Edition. Atlanta: *books, 2006.
- Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989.
- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Humphrey, Watts. *TSP – Leading a Development Team*. Boston: Addison Wesley, 2006.
- Humphrey, Watts. *Winning with Software: An Executive Strategy*. Boston: Addison Wesley, 2002.

- Jones, Capers. *Applied Software Measurement*, Third Edition. New York: McGraw-Hill, 2008.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston: Addison Wesley Longman, 2003.
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press, 1996.
- Nagappan, Nachiappan, B. Murphy, and V. Basili. *The Influence of Organizational Structure on Software Quality*. Microsoft Technical Report MSR-TR-2008-11. Microsoft Research, 2008.
- Pressman, Roger. *Software Engineering – A Practitioner’s Approach*, Sixth Edition. New York: McGraw-Hill, 2005.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.
- Weinberg, Gerald M. *Becoming a Technical Leader*. New York: Dorset House, 1986.
- Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
- Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.
- Yourdon, Ed. *Death March – The Complete Software Developer’s Guide to Surviving “Mission Impossible” Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.

Project Management and Software Engineering

Introduction

Project management is a weak link in the software engineering chain. It is also a weak link in the academic education curricula of many universities. More software problems and problems such as cost and schedule overruns can be attributed to poor project management than to poor programming or to poor software engineering practices. Because poor project management is so harmful to good software engineering, it is relevant to a book on best practices.

Working as an expert witness in a number of cases involving cancelled projects, quality problems, and other significant failures, the author observed bad project management practices on almost every case. Not only were project management problems common, but in some lawsuits, the project managers and higher executives actually interfered with good software engineering practices by canceling inspections and truncating testing in the mistaken belief that such actions would shorten development schedules.

For example, in a majority of breach of contract lawsuits, project management issues such as inadequate estimating, inadequate quality control, inadequate change control, and misleading or even false status tracking occur repeatedly.

As the recession continues, it is becoming critical to analyze every aspect of software engineering in order to lower costs without degrading operational efficiency. Improving project management is on the critical path to successful cost reduction.

Project management needs to be defined in a software context. The term *project management* has been artificially narrowed by tool vendors so that it has become restricted to the activities of critical path analysis and the production of various scheduling aids such as PERT and Gantt charts. For successful software project management, many other activities must be supported.

Table 6-1 illustrates 20 key project management functions, and how well they are performed circa 2009, based on observations within about 150 companies. The scoring range is from –10 for very poor performance to +10 for excellent performance.

Using this scoring method that runs from +10 to –10, the midpoint or average is 0. Observations made over the past few years indicate that project management is far below average in far too many critical activities.

The top item in Table 6-1, reporting “red flag” items, refers to notifying clients and higher managers that a project is in trouble. In almost every software breach of contract lawsuit, problems tend to be concealed or ignored, which delays trying to solve problems until they grow too serious to be cured.

TABLE 6-1 Software Project Management Performance Circa 2009

	Project Management Functions	Score	Definition
1.	Reporting “red flag” problems	–9.5	Very poor
2.	Defect removal efficiency measurements	–9.0	Very poor
3.	Benchmarks at project completion	–8.5	Very poor
4.	Requirements change estimating	–8.0	Very poor
5.	Postmortems at project completion	–8.0	Very poor
6.	Quality estimating	–7.0	Very poor
7.	Productivity measurements	–6.0	Poor
8.	Risk estimating	–3.0	Poor
9.	Process improvement tracking	–2.0	Poor
10.	Schedule estimating	1.0	Marginal
11.	Initial application sizing	2.0	Marginal
12.	Status and progress tracking	2.0	Marginal
13.	Cost estimating	3.0	Fair
14.	Value estimating	4.0	Fair
15.	Quality measurements	4.0	Fair
16.	Process improvement planning	4.0	Fair
17.	Quality and defect tracking	5.0	Good
18.	Software assessments	6.0	Good
19.	Cost tracking	7.0	Very good
20.	Earned-value tracking	8.0	Very good
	Average	–0.8	Poor

The main reason for such mediocre performance by software project managers is probably lack of effective curricula at the university and graduate school level. Few software engineers and even fewer MBA students are taught anything about the economic value of software quality or how to measure defect removal efficiency levels, which is actually the most important single measurement in software engineering.

If you examine the tools and methods for effective software project management that are available in 2009, a much different profile would occur if software managers were trained at state-of-the-art levels.

Table 6-2 makes the assumption that a much improved curricula for project managers could be made available within ten years, coupled with the assumption that project managers would then be equipped with modern sizing, cost estimating, quality estimating, and measurement tools. Table 6-2 shows what software project managers could do if they were well trained and well equipped.

Instead of jumping blindly into projects with poor estimates and inadequate quality plans, Table 6-2 shows that it is at least theoretically possible for software project managers to plan and estimate with

TABLE 6-2 Potential Software Project Management Performance by 2019

	Project Management Functions	Score	Definition
1.	Reporting “red flag” problems	10.0	Excellent
2.	Benchmarks at project completion	10.0	Excellent
3.	Postmortems at project completion	10.0	Excellent
4.	Status and progress tracking	10.0	Excellent
5.	Quality measurements	10.0	Excellent
6.	Quality and defect tracking	10.0	Excellent
7.	Cost tracking	10.0	Excellent
8.	Defect removal efficiency measurements	9.0	Excellent
9.	Productivity measurements	9.0	Very good
10.	Software assessments	9.0	Very good
11.	Earned-value tracking	9.0	Very good
12.	Quality estimating	8.0	Very good
13.	Initial application sizing	8.0	Very good
14.	Cost estimating	8.0	Very good
15.	Risk estimating	7.0	Good
16.	Schedule estimating	7.0	Good
17.	Process improvement tracking	6.0	Good
18.	Value estimating	6.0	Good
19.	Process improvement planning	6.0	Good
20.	Requirements change estimating	5.0	Good
	Average	8.4	Very good

high precision, measure with even higher precision, and create benchmarks for every major application when it is finished. Unfortunately, the technology of software project management is much better than actual day-to-day performance.

As of 2009, the author estimates that only about 5 percent of U.S. software projects create benchmarks of productivity and quality data at completion. Less than one-half of 1 percent submit benchmark data to a formal benchmark repository such as that maintained by the International Software Benchmarking Standards Group (ISBSG), Software Productivity Research (SPR), the David Consulting Group, Quality and Productivity Management Group (QPMG), or similar organizations.

Every significant software project should prepare formal benchmarks at the completion of the project. There should also be a postmortem review of development methods to ascertain whether improvements might be useful for future projects.

As of 2009, many independent software project management tools are available, but each only supports a portion of overall software project management responsibilities. A new generation of integrated software project management tools is approaching, which has the promise of eliminating the gaps in current project management tools and improving the ability to share information from tool to tool. New classes of project management tools such as methodology management tools have also joined the set available to the software management community.

Software project management is one of the most demanding jobs of the 21st century. Software project managers are responsible for the construction of some of the most expensive assets that corporations have ever attempted to build. For example, large software systems cost far more to build and take much longer to construct than the office buildings occupied by the companies that have commissioned the software. Really large software systems in the 100,000–function point range can cost more than building a domed football stadium, a 50-story skyscraper, or a 70,000-ton cruise ship.

Not only are large software systems expensive, but they also have one of the highest failure rates of any manufactured object in human history. The term *failure* refers to projects that are cancelled without completion due to cost or schedule overruns, or which run later than planned by more than 25 percent.

For software failures and disasters, the great majority of blame can be assigned to the management community rather than to the technical community. Table 6-3 is derived from one of the author's older books, *Patterns of Software System Failure and Success*, published by the International Thomson Press. Note the performance of software managers on successful projects as opposed to their performance associated with cancellations and severe overruns.

TABLE 6-3 Software Management Performance on Successful and Unsuccessful Projects

Activity	Successful Projects	Unsuccessful Projects
Sizing	Good	Poor
Planning	Very good	Fair
Estimating	Very good	Very poor
Tracking	Good	Poor
Measurement	Good	Very Poor
Quality control	Excellent	Poor
Change control	Excellent	Poor
Problem resolutions	Good	Poor
Risk analysis	Good	Very poor
Personnel management	Good	Poor
Supplier management	Good	Poor
Overall Performance	Very good	Poor

As mentioned in Chapter 5 of this book, the author's study of project failures and analysis of software lawsuits for breach of contract reached the conclusion that project failures correlate more closely to the number of managers involved with software projects than they do with the number of software engineers.

Software projects with more than about six first-line managers tend to run late and over budget. Software projects with more than about 12 first-line managers tend to run very late and are often cancelled.

As can easily be seen, deficiencies of the software project management function are a fundamental root cause of software disasters. Conversely, excellence in project management can do more to raise the probability of success than almost any other factor, such as buying better tools, or changing programming languages. (This is true for larger applications above 1000 function points. For small applications in the range of 100 function points, software engineering skills still dominate results.)

On the whole, improving software project management performance can do more to optimize software success probabilities and to minimize failure probabilities than any other known activity. However, improving software project management performance is also one of the more difficult improvement strategies. If it were easy to do, the software industry would have many more successes and far fewer failures than in fact occur.

A majority of the failures of software projects can be attributed to failures of project management rather than to failures of the technical staff. For example, underestimating schedules and resource requirements is associated with more than 70 percent of all projects that are cancelled due to overruns. Another common problem of project management is

ignoring or underestimating the work associated with quality control and defect removal. Yet another management problem is failure to deal with requirements changes in an effective manner.

Given the high costs and significant difficulty associated with software system construction, you might think that software project managers would be highly trained and well equipped with state-of-the-art planning and estimating tools, with substantial analyses of historical software cost structures, and with very thorough risk analysis methodologies. These are natural assumptions to make, but they are false. Table 6-4 illustrates patterns of project management tool usage of leading, average, and lagging software projects.

Table 6-4 shows that managers on leading projects not only use a wider variety of project management tools, but they also use more of the features of those tools.

In part due to the lack of academic preparation for software project managers, most software project managers are either totally untrained or at best partly trained for the work at hand. Even worse, software project managers are often severely under-equipped with state-of-the-art tools.

TABLE 6-4 Numbers and Size Ranges of Project Management Tools
(Size data expressed in terms of function point metrics)

Project Management	Lagging	Average	Leading
Project planning	1,000	1,250	3,000
Project cost estimating			3,000
Statistical analysis			3,000
Methodology management		750	3,000
Benchmarks			2,000
Quality estimation			2,000
Assessment support		500	2,000
Project measurement			1,750
Portfolio analysis			1,500
Risk analysis			1,500
Resource tracking	300	750	1,500
Value analysis		350	1,250
Cost variance reporting		500	1,000
Personnel support	500	500	750
Milestone tracking		250	750
Budget support		250	750
Function point analysis		250	750
Backfiring: LOC to FP			750
Function point subtotal	1,800	5,350	30,250
Number of tools	3	10	18

From data collected from consulting studies performed by the author, less than 25 percent of U.S. software project managers have received any formal training in software cost estimating, planning, or risk analysis; less than 20 percent of U.S. software project managers have access to modern software cost-estimating tools; and less than 10 percent have access to any significant quantity of validated historical data from projects similar to the ones they are responsible for.

The comparatively poor training and equipment of project managers is troubling. There are at least a dozen commonly used software cost-estimating tools such as COCOMO, KnowledgePlan, Price-S, SEER, SLIM, and the like. Of a number of sources of benchmark data, the International Software Benchmarking Standards Group (ISBSG) has the most accessible data collection.

By comparison, the software technical personnel who design and build software are often fairly well trained in the activities of analysis, design, and software development, although there are certainly gaps in topics such as software quality control and software security.

The phrase “project management” has unfortunately been narrowed and misdefined in recent years by vendors of automated tools for supporting real project managers. The original broad concept of project management included all of the activities needed to control the outcome of a project: sizing deliverables, estimating costs, planning schedules and milestones, risk analysis, tracking, technology selection, assessment of alternatives, and measurement of results.

The more narrow concept used today by project management tool vendors is restricted to a fairly limited set of functions associated with the mechanics of critical path analysis, work breakdown structuring, and the creation of PERT charts, Gantt charts, and other visual scheduling aids. These functions are of course part of the work that project managers perform, but they are neither the only activities nor even the most important ones for software projects.

The gaps and narrow focus of conventional project management tools are particularly troublesome when the projects in question are software related. Consider a very common project management question associated with software projects: *What will be the results to software schedules and costs from the adoption of a new development method such as Agile development or the Team Software Process (TSP)?*

Several commercial software estimating tools can predict the results of both Agile and TSP development methods, but not even one standard project management tool such as Microsoft Project has any built-in capabilities for automatically adjusting its assumptions when dealing with alternative software development approaches.

The same is also true for other software-related technologies such as the project management considerations of dealing with the formal

inspections in addition to testing, static analysis, the ISO 9000-9004 standards, the SEI maturity model, reusable components, ITIL, and so forth.

The focus of this chapter is primarily on the activities and tasks associated with *software project management*. Project managers also spend quite a bit of time dealing with personnel issues such as hiring, appraisals, pay raises, and staff specialization. Due to the recession, project managers will probably also face tough decisions involving layoffs and downsizing.

Most software project managers are also involved with departmental and corporate issues such as creating budgets, handling travel requests, education planning, and office space planning. These are important activities, but are outside the scope of what managers do when they are involved specifically with project management.

The primary focus of this chapter is on the tools and methods that are the day-to-day concerns of software project managers, that is, sizing, estimating, planning, measurement and metrics, quality control, process assessments, technology selection, and process improvement.

There are 15 basic topics that project managers need to know about, and each topic is a theme of some importance to professional software project managers:

1. Software sizing
2. Software project estimating
3. Software project planning
4. Software methodology selection
5. Software technology and tool selection
6. Software quality control
7. Software security control
8. Software supplier management
9. Software progress and problem tracking
10. Software measurements and metrics
11. Software benchmarking
12. Software risk analysis
13. Software value analysis
14. Software process assessments
15. Software process improvements

These 15 activities are not the only topics of concern to software project managers, but they are critical topics in terms of the ability to control

major software projects. Unless at least 10 of these 13 are performed in a capable and competent manner, the probability of the project running out of control or being cancelled will be alarmingly high.

Because the author's previous books on *Estimating Software Costs* (McGraw-Hill, 2007) and *Applied Software Measurement* (McGraw-Hill, 2008) dealt with many managerial topics, this book will cover only 3 of the 15 management topics:

1. Software sizing
2. Software progress and problem tracking
3. Software benchmarking

Sizing is the precursor to estimating. Sizing has many different approaches, but several new approaches have been developed within the past year.

Software *progress tracking* is among the most critical of all software project management activities. Unfortunately, based on depositions and documents discovered during litigation, software progress tracking is seldom performed competently. Even worse, when projects are in trouble, tracking tends to conceal problems until it is too late to solve them.

Software *benchmarking* is underreported in the literature. As this book is in production, the ISO standards organization is preparing a new ISO standard on benchmarking. It seems appropriate to discuss how to collect benchmark data and what kinds of reports constitute effective benchmarks.

Software Sizing

The term *sizing* refers to methods for predicting the volume of various deliverable items such as source code, specifications, and user manuals. Software bugs or defects should also be included in sizing, because they cost more money and take more time than any other software “deliverable.” Bugs are an accidental deliverable, but they are always delivered, like it or not, so they need to be included in sizing. Because requirements are unstable and grow during development, changes and growth in application requirements should be sized, too.

Sizing is the precursor to cost estimating and is one of the most critical software project management tasks. Sizing is concerned with predicting the volumes of major kinds of software deliverables, including but not limited to those shown in Table 6-5.

As can be seen from the list of deliverables, the term *sizing* includes quite a few deliverables. Many more things than source code need to be predicted to have complete size and cost estimates.

TABLE 6-5 Software Deliverables Whose Sizes Should Be Quantified

Paper documents	
Requirements	Text requirements Function requirements (features of the application) Nonfunctional requirements (quality and constraints) Use-cases User stories Requirements change (new features) Requirements churn (changes that don't affect size)
Architecture	External architecture (SOA, client-server, etc.) Internal architecture (data structure, platforms, etc.)
Specifications and design	External Internal
Planning documents	Development plans Quality plans Test plans Security plans Marketing plans Maintenance and support plans
User manuals	Reference manuals Maintenance manuals Translations into foreign languages
Tutorial materials	
Translations of tutorial materials	
Online HELP screens	
Translations of HELP screens	
Source code	
New source code	
Reusable source code from certified sources	
Reusable source code from uncertified sources	
Inherited or legacy source code	
Code added to support requirements change and churn	

TABLE 6-5 Software Deliverables Whose Sizes Should Be Quantified (*continued*)

Test cases	
	New test cases
	Reusable test cases
Bugs or defects	
	Requirements defects (original)
	Requirements defects (in changed requirements)
	Architectural defects
	Design defects
	Code defects
	User documentation defects
	“Bad fixes” or secondary defects
	Test case defects

Note that while bugs or defects are accidental deliverables, there are always latent bugs in large software applications and they have serious consequences. Therefore, estimating defect potentials and defect removal efficiency levels are critical tasks of software application sizing.

This section discusses several methods of sizing software applications, which include but are not limited to:

1. Traditional sizing by analogy with similar projects
2. Traditional sizing using “lines of code” metrics
3. Sizing using story point metrics
4. Sizing using use-case metrics
5. Sizing using IFPUG function point metrics
6. Sizing using other varieties of function point metrics
7. High-speed sizing using function point approximations
8. High-speed sizing legacy applications using backfiring
9. High-speed sizing using pattern matching
10. Sizing application requirements changes

Accurate estimation and accurate schedule planning depend on having accurate size information, so sizing is a critical topic for successful software projects. Size and size changes are so important that a new management position called “scope manager” has come into existence over the past few years.

New methods for formal size or scope control have been created. Interestingly, the two most common methods were developed in very distant locations from each other. A method called *Southern Scope* was developed in Australia, while a method called *Northern Scope* was developed in Finland. Both of these scope-control methods focus on change controls and include formal sizing, reviews of changes, and other techniques for quantifying the impact of growth and change. While other size control methods exist, the Southern Scope and Northern Scope methods both appear to be more effective than leaving changes to ordinary practices.

Because thousands of software applications exist circa 2009, careful forensic analysis of existing software should be a good approach for predicting the sizes of future applications. As of 2009, many “new” applications are replacements of existing legacy applications. Therefore, historical data would be useful, if it were reliable and accurate.

Size is a useful precursor for estimating staffing, schedules, effort, costs, and quality. However, size is not the only factor that needs to be known. Consider an analogy with home construction. You need to know the number of square feet or square meters in a house to perform a cost estimate. But you also need to know the specifics of the site, the construction materials to be used, and any local building codes that might require costly additions such as hurricane-proof windows or special septic systems.

For example, a 3000-square-foot home to be constructed on a flat suburban lot with ordinary building materials might be constructed for \$100 per square foot, or \$300,000. But a luxury 3000-square-foot home built on a steep mountain slope that requires special support and uses exotic hardwoods might cost \$250 per square foot or \$750,000.

Similar logic applies to software. An embedded application in a medical device may cost twice as much as the same size application that handles business data. This is because the liabilities associated with software in medical devices require extensive verification and validation compared with ordinary business applications.

(Author’s note: Prior to the recession, one luxury home was built on a remote lake so far from civilization that it needed a private airport and its own electric plant. The same home featured handcrafted windows and wall panels created on site by artists and craftspeople. The budgeted cost was about \$40 million, or more than \$6,000 per square foot. Needless to say, this home was built before the Wall Street crash since the owner was a financier.)

Three serious problems have long been associated with software sizing: (1) Most of the facts needed to create accurate sizing of software deliverables are not known until after the first cost estimates are required; (2) Some sizing methods such as function point analysis are time-consuming

and expensive, which limits their utility for large applications; and (3) Software deliverables are not static in size and tend to grow during development. Estimating growth and change is often omitted from sizing techniques. Let us now consider a number of current software sizing approaches.

Traditional Sizing by Analogy

The traditional method of sizing software projects has been that of analogy with older projects that are already completed, so that the sizes of their deliverables are known. However, newer methods are available circa 2009 and will be discussed later in this chapter.

The traditional sizing-by-analogy method has not been very successful for a variety of reasons. It can only be used for common kinds of software projects where similar projects exist. For example, sizing by analogy works fairly well for compilers, since there are hundreds of compilers to choose from. The analogy method can also work for other familiar kinds of applications such as accounting systems, payroll systems, and other common application types. However, if an application is unique, and no similar applications have been constructed, then sizing by analogy is not useful.

Because older legacy applications predate the use of story points or use-case points, or sometimes even function points, not every legacy application is helpful in terms of providing size guidance for new applications. For more than 90 percent of legacy applications, their size is not known with precision, and even code volumes are not known, due to “dead code” and calls to external routines. Also, many of their deliverables (i.e., requirements, specifications, plans, etc.) have long since disappeared or were not updated, so their sizes may not be available.

Since legacy applications tend to grow at an annual rate of about 8 percent, their current size is not representative of their initial size at their first release. Very seldom is data recorded about requirements growth, so this can throw off sizing by analogy.

Even worse, a lot of what is called “historical data” for legacy applications is very inaccurate and can’t be relied upon to predict future applications. Even if legacy size is known, legacy effort and costs are usually incomplete. The gaps and missing elements in historical data include unpaid overtime (which is almost never measured), project management effort, and the work of part-time specialists who are not regular members of the development team (database administration, technical writers, quality assurance, etc.). The missing data on legacy application effort, staffing, and costs is called *leakage* in the author’s books. For small applications with one or two developers, this leakage from historical data is minor.

But for large applications with dozens of team members, leakage of missing effort and cost data can top 50 percent of total effort.

Leakage of effort and cost data is worse for internal applications developed by organizations that operate as cost centers and that therefore have no overwhelming business need for precision in recording effort and cost data. Outsource applications and software built under contract is more accurate in accumulating effort and cost data, but even here unpaid overtime is often omitted.

It is an interesting point to think about, but one of the reasons why IT projects seem to have higher productivity rates than systems or embedded software is that IT project historical data “leaks” a great deal more than systems and embedded software. This leakage is enough by itself to make IT projects look at least 15 percent more productive than systems or embedded applications of the same size in terms of function points. The reason is that most IT projects are created in a cost-center environment, while systems and embedded applications are created in a profit-center environment.

The emergence of the International Software Benchmarking Standards Group (ISBSG) has improved the situation somewhat, since ISBSG now has about 5000 applications of various kinds that are available to the software engineering community. All readers who are involved with software are urged to consider collecting and providing benchmark data. Even if the data cannot be submitted to ISBSG for proprietary or business reasons, keeping such data internally will be valuable.

The ISBSG questionnaires assist by collecting the same kinds of information for hundreds of applications, which facilitates using the data for estimating purposes. Also, companies that submit data to the ISBSG organization usually have better-than-average effort and cost tracking methods, so their data is probably more accurate than average.

Other benchmark organizations such as Software Productivity Research (SPR), the Quality and Productivity Management Group (QPMG), the David Consulting Group, and a number of others have perhaps 60,000 projects, but this data has limited distribution to specific clients. This private data is also more expensive than ISBSG data. A privately commissioned set of benchmarks with a comparison to similar relevant projects may cost between \$25,000 and \$100,000, based on the number of projects examined. Of course, the on-site private benchmarks are fairly detailed and also correct common errors and omissions, so the data is fairly reliable.

What would be useful for the industry is a major expansion in software productivity and quality benchmark data collection. Ideally, all development projects and all major maintenance and enhancement projects would collect enough data so that benchmarks would become standard practices, rather than exceptional activities.

For the immediate project under development, the benchmark data is valuable for showing defects discovered to date, effort expended to date, and ensuring that schedules are on track. In fact, similar but less formal data is necessary just for status meetings, so a case can be made that formal benchmark data collection is close to being free since the information is needed whether or not it will be kept for benchmark purposes after completion of the project.

Unfortunately, while sizing by analogy should be useful, flaws and gaps with software measurement practices have made both sizing by analogy and also historical data of questionable value in many cases.

Timing of sizing by analogy If there are benchmarks or historical size data from similar projects, this form of sizing can be done early, even before the requirements for the new application are fully known. This is one of the earliest methods of sizing. However, if historical data is missing, then sizing by analogy can't be done at all.

Usage of sizing by analogy There are at least 3 million existing software applications that might, in theory, be utilized for sizing by analogy. However, from visits to many large companies and government agencies, the author hypothesizes that fewer than 100,000 existing legacy applications have enough historical data for sizing by analogy to be useful and accurate. About another 100,000 have partial data but so many errors that sizing by analogy would be hazardous. About 2.8 million legacy applications either have little or no historical data, or the data is so inaccurate that it should not be used. For many legacy applications, no reliable size data is available in any metric.

Schedules and costs This form of sizing is quick and inexpensive, assuming that benchmarks or historical data are available. If neither size nor historical data is available, the method of sizing by analogy cannot be used. In general, benchmark data from an external source such as ISBSG, the David Consulting Group, QPMG, or SPR will be more accurate than internal data. The reason for this is that the external benchmark organizations attempt to correct common errors, such as omitting unpaid overtime.

Cautions and counter indications The main counter indication is that sizing by analogy does not work at all if there is neither historical data nor accurate benchmarks. A caution about this method is that historical data is usually incomplete and leaves out critical information such as unpaid overtime. Formal benchmarks collected for ISBSG or one of the other benchmark companies will usually be more accurate than most internal historical data, which is of very poor reliability.

Traditional Sizing Based on Lines of Code (LOC) Metrics

When the “lines of code” or LOC metric originated in the early 1960s, software applications were small and coding composed about 90 percent of the effort. Today in 2009, applications are large, and coding composes less than 40 percent of the effort. Between the 1960s and today, the usefulness of LOC metrics degraded until that metric became actually harmful. Today in 2009, using LOC metrics for sizing is close to professional malpractice. Following are the reasons why LOC metrics are now harmful.

The first reason that LOC metrics are harmful is that after more than 60 years of usage, there are no standard counting rules for source code! LOC metrics can be counted using either physical lines or logical statements. There can be more than a 500 percent difference in apparent size of the same code segment when the counting method switches between physical lines and logical statements.

In the first edition of the author’s book *Applied Software Measurement* in 1991, formal rules for counting source code based on logical statements were included. These rules were used by Software Productivity Research (SPR) for backfiring when collecting benchmark data. But in 1992, the Software Engineering Institute (SEI) issued their rules for counting source code, and the SEI rules were based on counts of physical lines. Since both the SPR counting rules and the SEI counting rules are widely used, but totally different, the effect is essentially that of having no counting rules.

(The author did a study of the code-counting methods used in major software journals such as *IEEE Software*, *IBM Systems Journal*, *CrossTalk*, the *Cutter Journal*, and so on. About one-third of the articles used physical lines, one-third used logical statements, and the remaining third used LOC metrics, but failed to mention whether physical lines or logical statements (or both) were used in the article. This is a serious lapse on the part of both the authors and the referees of software engineering journals. You would hardly expect a journal such as *Science* or *Scientific American* to publish quantified data without carefully explaining the metrics used to collect and analyze the results. However, for software engineering journals, poor measurements are the norm rather than the exception.)

The second reason that LOC metrics are hazardous is because they penalize high-level programming languages in direct proportion to the power of the language. In other words, productivity and quality data expressed using LOC metrics looks better for assembly language than for Java or C++.

The penalty is due to a well-known law of manufacturing economics, which is not well understood by the software community: *When a manufacturing process has a large number of fixed costs and there is a decline in the number of units manufactured, the cost per unit must go up.*

A third reason is that LOC metrics can't be used to size or measure noncoding activities such as requirements, architecture, design, and user documentation. An application written in the C programming language might have twice as much source code as the same application written in C++. But the requirements and specifications would be the same size.

It is not possible to size paper documents from source code without adjusting for the level of the programming language. For languages such as Visual Basic that do not even have source code counting rules available, it is barely possible to predict source code size, much less the sizes of any other deliverables.

The fourth reason the LOC metrics are harmful is that circa 2009, more than 700 programming languages exist, and they vary from very low-level languages such as assembly to very high-level languages such as ASP.NET. More than 50 of these languages have no known counting rules.

The fifth reason is that most modern applications use more than a single programming language, and some applications use as many as 15 different languages, each of which may have unique code counting rules. Even a simple mix of Java and HTML makes code counting difficult.

Historically, the development of Visual Basic and its many competitors and descendants changed the way many modern programs are developed. Although "visual" languages do have a procedural source code portion, much of the more complex programming uses button controls, pull-down menus, visual worksheets, and reusable components.

In other words, programming is being done without anything that can be identified as a "line of code" for sizing, measurement, or estimation purposes. By today in 2009, perhaps 60 percent of new software applications are developed using either object-oriented languages or visual languages (or both). Indeed, sometimes as many as 12 to 15 different languages are used in the same applications.

For large systems, programming itself is only the fourth most expensive activity. The three higher-cost activities cannot be measured or estimated effectively using the lines of code metric. Also, the fifth major cost element, project management, cannot easily be estimated or measured using the LOC metric either. Table 6-6 shows the ranking in descending order of software cost elements for large applications.

The usefulness of a metric such as lines of code, which can only measure and estimate one out of the five major software cost elements of software projects, is a significant barrier to economic understanding.

Following is an excerpt from the 3rd edition of the author's book *Applied Software Measurement* (McGraw-Hill, 2008), which illustrates the economic fallacy of KLOC metrics. Here are two case studies showing both the LOC results and function point results for the same application

TABLE 6-6 Rank Order of Large System Software Cost Elements

1.	Defect removal (inspections, static analysis, testing, finding, and fixing bugs)
2.	Producing paper documents (plans, architecture, specifications, user manuals)
3.	Meetings and communication (clients, team members, managers)
4.	Programming
5.	Project management

in two languages: basic assembly and C++. In Case 1, we will assume that an application is written in assembly. In Case 2, we will assume that the same application is written in C++.

Case 1: Application written in the assembly language Assume that the assembly language program required 10,000 lines of code, and the various paper documents (specifications, user documents, etc.) totaled to 100 pages. Assume that coding and testing required ten months of effort, and writing the paper documents took five months of effort. The entire project totaled 15 months of effort, and so has a productivity rate of 666 LOC per month. At a cost of \$10,000 per staff month, the application cost \$150,000. Expressed in terms of cost per source line, the cost is \$15 per line of source code.

Case 2: The same application written in the C++ language Assume that the C++ version of the same application required only 1000 lines of code. The design documents probably were smaller as a result of using an object-oriented (OO) language, but the user documents are the same size as the previous case: assume a total of 75 pages were produced. Assume that coding and testing required one month, and document production took four months. Now we have a project where the total effort was only five months, but productivity expressed using LOC has dropped to only 200 LOC per month. At a cost of \$10,000 per staff month, the application cost \$50,000 or only one-third as much as the assembly language version. The C++ version is a full \$100,000 cheaper than the assembly version, so clearly the C++ version has much better economics. But the cost per source line for this version has jumped to \$50.

Even if we measure only coding, we still can't see the value of high-level languages by means of the LOC metric: the coding rates for both the assembly language and C++ versions were both identical at 1000 LOC per month, even though the C++ version took only one month as opposed to ten months for the assembly version.

Since both the assembly and C++ versions were identical in terms of features and functions, let us assume that both versions were 50 function points in size. When we express productivity in terms of function points per staff month, the assembly version had a productivity rate of

3.33 function points per staff month. The C++ version had a productivity rate of 10 function points per staff month. When we turn to costs, the assembly version had a cost of \$3000 per function point, while the C++ version had a cost of \$1000 per function point. Thus, function point metrics clearly match the assumptions of standard economics, which define productivity as *goods or services produced per unit of labor or expense*.

Lines of code metrics, on the other hand, do not match the assumptions of standard economics and in fact show a reversal. Lines of code metrics distort the true economic picture by so much that their use for economic studies involving more than one programming language might be classified as professional malpractice.

Timing of sizing by lines of code Unless the application being sized is going to replace an existing legacy application, this method is pure guesswork until the code is written. If code benchmarks or historical code size data from similar projects exist, this form of sizing can be done early, assuming the new language is the same as the former language. However, if there is no history, sizing using lines of code, or the old language is not the same as the new, this can't be done with accuracy, and it can't be done until the code is written, which is far too late. When either the new application or the old application (or both) use multiple languages, code counting becomes very complicated and difficult.

Usage of lines of code sizing As of 2009, at least 3 million legacy applications still are in use, and another 1.5 million are under development. However, of this total of about 4.5 million applications, the author estimates that more than 4 million use multiple programming languages or use languages for which no effective counting rules exist. Of the approximate total of 500,000 applications that use primarily a single language where counting rules do exist, no fewer than 500 programming languages have been utilized. Essentially, code sizing is inaccurate and hazardous, except for applications that use a single language such as assembler, C, dialects of C, COBOL, Fortran, Java, and about 100 others.

In today's world circa 2009, sizing using LOC metrics still occurs in spite of the flaws and problems with this metric. The Department of Defense and military software are the most frequent users of LOC metrics. The LOC metric is still widely used by systems and embedded applications. The older waterfall method often employed LOC sizing, as does the modern Team Software Process (TSP) development method.

Schedules and costs This form of sizing is quick and inexpensive, assuming that automated code counting tools are available. However, if the application has more than two programming languages, automated code

counting may not be possible. If the application uses some modern language, code counting is impossible because there are no counting rules for the buttons and pull-down menus used to “program” in some languages.

Cautions and counter indications The main counter indication is that lines of code metrics penalize high-level languages. Another indication is that this method is hazardous for sizing requirements, specifications, and paper documents. Also, counts of physical lines of code may differ from counts of logical statements by more than 500 percent. Since the software literature and published productivity data is ambiguous as to whether logical or physical lines are used, this method has a huge margin of error.

Sizing Using Story Point Metrics

The Agile development method was created in part because of a reaction against the traditional software cost drivers shown in Table 6-6. The Agile pioneers felt that software had become burdened by excessive volumes of paper requirements and specifications, many of which seemed to have little value in actually creating a working application.

The Agile approach tries to simplify and minimize the production of paper documents and to accelerate the ability to create working code. The Agile philosophy is that the goal of software engineering is the creation of working applications in a cost-effective fashion. In fact, the goal of the Agile method is to transform the traditional software cost drivers into a more cost-effective sequence, as shown in Table 6-7.

As part of simplifying the paper deliverables of software applications, a method for gathering the requirements for Agile projects is that of *user stories*. These are very concise statements of specific requirements that consist of only one or two sentences, which are written on 3"×5" cards to ensure compactness.

An example of a basic user story for a software cost-estimating tool might be, *The estimating tool should include currency conversion between dollars, euros, and yen.*

Once created, user stories are assigned relative weights called *story points*, which reflect their approximate difficulty and complexity compared

TABLE 6-7 Rank Order of Agile Software Cost Elements

-
- | | |
|----|---|
| 1. | Programming |
| 2. | Meetings and communication (clients, team members, managers) |
| 3. | Defect removal (inspections, static analysis, testing, finding and fixing bugs) |
| 4. | Project management |
| 5. | Producing paper documents (plans, architecture, specifications, user manuals) |
-

with other stories for the same application. The currency conversion example just shown is quite simple and straightforward (except for the fact that currencies fluctuate on a daily basis), so it might be assigned a weight of 1 story point. Currency conversion is a straightforward mathematical calculation and also is readily available from online sources, so this is not a difficult story or feature to implement.

The same cost-estimating application will of course perform other functions that are much harder and more complex than currency conversion. An example of a more difficult user story might be, *The estimating tool will show the effects of CMMI levels on software quality and productivity.*

This story is much harder to implement than currency conversion, because the effects of CMMI levels vary with the size and nature of the application being developed. For small and simple applications, CMMI levels have hardly any impact, but for large and complex applications, the higher CMMI levels have a significant impact. Obviously, this story would have a larger number of story points than currency conversion, and might be assigned a weight of 5, meaning that it is at least five times as difficult as the previous example.

The assignment of story point weights for a specific application is jointly worked out between the developers and the user representative. Thus, for specific applications, there is probably a high degree of mathematical consistency between story point levels; that is, levels 1, 2, 3, and so on, probably come close to capturing similar levels of difficulty.

The Agile literature tends to emphasize that story points are units of size, not units of time or effort. However, that being said, story points are in fact often used for estimating team velocity and even for estimating the overall schedules for both sprints and even entire applications.

However, user stories and therefore story points are very flexible, and there is no guarantee that Agile teams on two different applications will use exactly the same basis for assigning story point weights.

It may be that as the Agile approach gains more and more adherence and wider usage, general rules for determining story point weights will be created and utilized, but this is not the case circa 2009.

It would be theoretically possible to develop mathematical conversion rules between story points and other metrics such as IFPUG function points, COSMIC function points, use-case points, lines of code, and so forth. However, for this to work, story points would need to develop guidelines for consistency between applications. In other words, quantities such as 1 story point, 2 story points, and so on, would have to have the same values wherever they were applied.

From looking at samples of story points, there does not seem to be a strict linear relation between user stories and story points in terms of effort. What might be a useful approximation is to assume that for each

increase of 1 in terms of story points, the IFPUG function points needed for the story would double. For example:

Story Points	IFPUG Function Points
1	2
2	4
3	8
4	16
5	32

This method is of course hypothetical, but it would be interesting to carry out trials and experiments and create a reliable conversion table between story points and function points.

It would be useful if the Agile community collected valid historical data on effort, schedules, defects, and other deliverables and submitted them to benchmarking organizations such as ISBSG. Larger volumes of historical data would facilitate the use of story points for estimating purposes and would also speed up the inclusion of story points in commercial estimating tools such as COCOMO, KnowledgePlan, Price-S, SEER, SLIM, and the like.

A few Agile projects have used function point metrics in addition to story points. But as this book is written in 2009, no Agile projects have submitted formal benchmarks to ISBSG or to other public benchmark sources. Some Agile projects have been analyzed by private benchmark organizations, but the results are proprietary and confidential.

As a result, there is no reliable quantitative data circa 2009 that shows either Agile productivity or Agile quality levels. This is not a sign of professional engineering, but it is a sign of how backwards “software engineering” is compared with more mature engineering fields.

Timing of sizing with story points While Agile projects attempt an overview of an entire application at the start, user stories occur continuously with every sprint throughout development. Therefore, user stories are intended primarily for the current sprint and don’t have much to say about future sprints that will occur downstream. As a result, story points are hard to use for early sizing of entire applications, although useful for the current sprint.

Usage of story point metrics Agile is a very popular method, but it is far from being the only software development method. The author estimates that circa 2009, about 1.5 million new applications are being developed. Of these perhaps 200,000 use the Agile method and also use story points. Story points are used primarily for small to mid-sized IT software applications between about 250 and 5000 function points.

Story points are not used often for large applications greater than 10,000 function points, nor are they often used for embedded, systems, and military software.

Schedules and costs Since story points are assigned informally by team consensus, this form of sizing is quick and inexpensive. It is possible to use collections of story cards and function points, too. User stories could be used as a basis for function point analysis. But Agile projects tend to stay away from function points. It would also be possible to use some of the high-speed function point methods with Agile projects, but as this book is written, there is no data that shows this being done.

Cautions and counter indications The main counter indication for story points is that they tend to be unique for specific applications. Thus, it is not easy to compare benchmarks between two or more different Agile applications using story points, because there is no guarantee that the applications used the same weights for their story points.

Another counter indication is that story points are useless for comparisons with applications that were sized using function points, use-case points, or any other software metric. Story points can only be used for benchmark comparisons against other story points, and even here the results are ambiguous.

A third counter indication is that there are no large-scale collections of benchmark data that are based on story points. For some reason, the Agile community has been lax on benchmarks and collecting historical data. This is why it is so hard to ascertain if Agile has better or worse productivity and quality levels than methods such as TSP, iterative development, or even waterfall development. The shortage of quantitative data about Agile productivity and quality is a visible weakness of the Agile approach.

Sizing Using Use-Case Metrics

Use-cases have been in existence since the 1980s. They were originally discussed by Ivar Jacobsen and then became part of the unified modeling language (UML). Use-cases are also an integral part of the Rational Unified Process (RUP), and Rational itself was acquired by IBM. Use-cases have both textual and several forms of graphical representation. Outside of RUP, use-cases are often used for object-oriented (OO) applications. They are sometimes used for non-OO applications as well.

Use-cases describe software application functions from the point of view of a user or *actor*. Use-cases can occur in several levels of detail, including “brief,” “casual,” and “fully dressed,” which is the most detailed. The fully dressed use-cases are of sufficient detail that they can be used for function point analysis and also can be used to create use-case points.

Use-cases include other topics besides actors, such as preconditions, postconditions, and several others. However, these are well defined and fairly consistent from application to application.

Use-cases and user stories have similar viewpoints, but use-cases are more formal and often much larger than user stories. Because of the age and extensive literature about use-cases, they tend to be more consistent from application to application than user stories do.

Some criticisms are aimed at use-cases for not dealing with nonfunctional requirements such as security and quality. But this same criticism could be aimed with equal validity at any design method. In any case, it is not difficult to append quality, security, and other nonfunctional design issues to use-cases.

Use-case points are based on calculations and logic somewhat similar to function point metrics in concept but not in specific details. The factors that go into use-case points include technical and environmental complexity factors. Once calculated, use-case points can be used to predict effort and costs for software development. About 20 hours of development work per use-case has been reported, but the activities that go into this work can vary.

Use-case diagrams and supporting text can be used to calculate function point metrics as well as use-case metrics. In fact, the rigor and consistency of use-cases should allow automatic derivation of both use-case points and function points.

The use-case community tends to be resistant to function points and asserts that use-cases and function points look at different aspects, which is only partly true. However, since both can yield information on work hours per point, it is obvious that there are more similarities than the use-case community wants to admit to.

If you assume that a work month consists of 22 days at 8 hours per day, there are about 176 hours in a work month. Function point productivity averages about 10 function points per staff month, or 17.6 work hours per function point.

Assuming that use-case productivity averages about 8.8 use-cases per month, which is equivalent to 20 hours per use-case, it can be seen that use-case points and IFPUG function points yield results that are fairly close together.

Other authors and benchmark organizations such as the David Consulting Group and ISBSG have published data on conversion ratios between IFPUG function point metrics and use-case points. While the other conversion ratios are not exactly the same as the ones in this chapter, they are quite close, and the differences are probably due to using different samples.

There may be conversion ratios between use-case points and COSMIC function points, Finnish function points, or other function point variants,

but the author does not use any of the variants and has not searched their literature.

Of course, productivity rates using both IFPUG function points and use-case points have wide ranges, but overall they are not far apart.

Timing of sizing with use-case points Use-cases are used to define requirements and specifications, so use-case points can be calculated when use-cases are fairly complete; that is, toward the end of the requirements phase. Unfortunately, formal estimates are often needed before this time.

Usage of use-case points RUP is a very popular method, but it is far from being the only software development method. The author estimates that circa 2009, about 1.5 million new applications are being developed. Of these, perhaps 75,000 use the RUP method and also use-case points. Perhaps another 90,000 projects are object-oriented and utilize use-cases, but not RUP. Use-case points are used for both small and large software projects. However, the sheer volume of use-cases becomes cumbersome for large applications.

Schedules and costs Since use-case points have simpler calculations than function points, this form of sizing is somewhat quicker than function point analysis. Use-case points can be calculated at a range of perhaps 750 per day, as opposed to about 400 per day for function point analysis. Even so, the cost for calculating use-case points can top \$3 per point if manual sizing is used. Obviously, automatic sizing would be a great deal cheaper and also faster. In theory, automatic sizing of use-case points could occur at rates in excess of 5000 use-case points per day.

Cautions and counter indications The main counter indication for use-case points is that there are no large collections of benchmark data that use them. In other words, use-case points cannot yet be used for comparisons with industry databases such as ISBSG, because function point metrics are the primary metric for benchmark analysis.

Another counter indication is that use-case points are useless for comparisons with applications that were sized using function points, story points, lines of code, or any other software metric. Use-case points can only be used for benchmark comparisons against other use-case points, and even here the results are sparse and difficult to find.

A third counter indication is that supplemental data such as productivity and quality is not widely collected for projects that utilize use-cases. For some reason, both the OO and RUP communities have been lax on benchmarks and collecting historical data. This is why it

is so hard to ascertain if RUP or OO applications have better or worse productivity and quality levels than other methods. The shortage of quantitative data about RUP productivity and quality compared with other methods such as Agile and TSP productivity and quality is a visible weakness of the use-case point approach.

Sizing Based on IFPUG Function Point Analysis

Function point metrics were developed by A.J. Albrecht and his colleagues at IBM in response to a directive by IBM executives to find a metric that did not distort economic productivity, as did the older lines of code metric. After research and experimentation, Albrecht and his colleagues developed a metric called “function point” that was independent of code volumes.

Function point metrics were announced at a conference in 1978 and put into the public domain. In 1984, responsibility for the counting rules of function point metrics was transferred from IBM to a nonprofit organization called the International Function Point Users Group (IFPUG).

Sizing technologies based on function point metrics have been possible since this metric was introduced in 1978. Function point sizing is more reliable than sizing based on lines of code because function point metrics support all deliverable items: paper documents, source code, test cases, and even bugs or defects. Thus, function point sizing has transformed the task of sizing from a very difficult kind of work with a high error rate to one of acceptable accuracy.

Although the counting rules for function points are complex today, the essence of function point analysis is derived by a weighted formula that includes five elements:

1. Inputs
2. Outputs
3. Logical files
4. Inquiries
5. Interfaces

There are also adjustments for complexity. The actual rules for counting function points are published by the International Function Point Users Group (IFPUG) and are outside the scope of this section.

The function point counting items can be quantified by reviewing software requirements and specifications. Note that conventional paper specifications, use-cases, and user stories can all be used for function point analysis. The counting rules also include complexity adjustments.

The exact rules for counting function points are outside the scope of this book and are not discussed.

Now that function points are the most widely used software size metric in the world, thousands of projects have been measured well enough to extract useful sizing data for all major deliverables: paper documents such as plans and manuals, source code, and test cases. Here are a few examples from all three sizing domains. Table 6-8 illustrates typical document volumes created for various kinds of software.

Table 6-8 illustrates only a small sample of the paperwork and document sizing capabilities that are starting to become commercially available. In fact, as of 2009, more than 90 kinds of document can be sized using function points, including translations into other national languages such as Japanese, Russian, Chinese, and so on.

Not only can function points be used to size paper deliverables, but they can also be used to size source code, test cases, and even software bugs or defects. In fact, function point metrics can size the widest range of software deliverables of any known metric.

For sizing source code volumes, data now is available on roughly 700 languages and dialects. There is also embedded logic in several commercial software estimating tools for dealing with multiple languages in the same application.

Since the function point total of an application is known at least roughly by the end of requirements, and in some detail by the middle of the specification phase, it is now possible to produce fairly accurate size estimates for any application where function points are utilized. This form of sizing is now a standard function for many commercial software estimating tools such as COCOMO II, KnowledgePlan, Price-S, SEER, SLIM, and others.

The usefulness of IFPUG function point metrics has made them the metric of choice for software benchmarks. As of 2009, benchmarks based on function points outnumber all other metrics combined. The ISBSG

TABLE 6-8 Number of Pages Created Per Function Point for Software Projects

	Systems Software	MIS Software	Military Software	Commercial Software
User requirements	0.45	0.50	0.85	0.30
Functional specifications	0.80	0.55	1.75	0.60
Logic specifications	0.85	0.50	1.65	0.55
Test plans	0.25	0.10	0.55	0.25
User tutorial documents	0.30	0.15	0.50	0.85
User reference documents	0.45	0.20	0.85	0.90
Total document set	3.10	2.00	6.15	3.45

benchmark data currently has about 5000 projects and is growing at a rate of perhaps 500 projects per year.

The proprietary benchmarks by companies such as QPMG, the David Consulting Group, Software Productivity Research, Galorath Associates, and several others total perhaps 60,000 software projects using function points and grow at a collective rate of perhaps 1000 projects per year. There are no other known metrics that even top 1000 projects.

Over the past few years, concerns have been raised that software applications also contain “nonfunctional requirements” such as performance, quality, and so on. This is true, but the significance of these tends to be exaggerated.

Consider the example of home construction. A major factor in the cost of home construction is the size of the home, measured in terms of square feet or square meters. The square footage, the amenities, and the grade of construction materials are user requirements. But in the author’s state (Rhode Island), local building codes add significant costs due to nonfunctional requirements. Homes built near a lake, river, or aquifer require special hi-tech septic systems, which cost about \$30,000 more than standard septic systems. Homes built within a mile of the Atlantic Ocean require hurricane-proof windows, which cost about three times more than standard windows.

These government mandates are not user requirements. But they would not occur without a home being constructed, so they can be dealt with as subordinate cost elements. Therefore, estimates and measures such as “cost per square foot” are derived from the combination of functional user requirements and government building codes that force mandated nonfunctional requirements on homeowners.

Timing of IFPUG function point sizing IFPUG function points are derived from requirements and specifications, and can be quantified by the time initial requirements are complete. However, the first formal cost estimates usually are needed before requirements are complete.

Usage of IFPUG function points While the IFPUG method is the most widely used form of function point analysis, none of the function point methods are used widely. Out of an approximate total of perhaps 1.5 million new software applications under development circa 2009, the author estimates that IFPUG function point metrics are currently being used on about 5000 applications. Function point variants, back-firing, and function point approximation methods are probably in use on another 2500 applications. Due to limitations in the function point method itself, IFPUG function points are seldom used for applications greater than 10,000 function points and can’t be used at all for small updates less than 15 function points in size.

Schedules and costs This form of sizing is neither quick nor inexpensive. Function point analysis is so slow and expensive that applications larger than about 10,000 function points are almost never analyzed.

Normal function point analysis requires a certified function point analysis to be performed with accuracy (uncertified counts are highly inaccurate). Normal function point analysis proceeds at a rate of between 400 and 600 function points per day. At a daily average consulting fee of \$3000, the cost is between \$5.00 and \$7.50 for every function point counted.

Assuming an average cost of \$6.00 per function point counted, counting a 10,000-function point application would cost \$60,000. This explains why normal function point analysis is usually only performed for applications in the 1000-function point size range.

Later in this section, various forms of high-speed function point approximation are discussed. It should be noted that automatic function point counting is possible when formal specification methods such as use-cases are utilized.

Cautions and counter indications The main counter indication with function point analysis is that it is expensive and fairly time-consuming. While small applications less than 1000 function points can be sized in a few days, large systems greater than 10,000 function points would require weeks. No really large systems greater than 100,000 function points have ever been sized with function points due to the high costs and the fact that the schedule for the analysis would take months.

Another counter indication is that from time to time, the counting rules change. When this occurs, historical data based on older versions of the counting rules may change or become incompatible with newer data. This situation requires conversion rules from older to newer counting rules. If nonfunctional requirements are indeed counted separately from functional requirements, such a change in rules would cause significant discontinuities in historical benchmark data.

Another counter indication is that there is a lower limit for function point analysis. Small changes less than 15 function points can't be sized due to the lower limits of the adjustment factors. Individually, these changes are trivial, but within large companies, there may be thousands of them every year, so their total cost can exceed several million dollars.

A caution is that accurate function point analysis requires certified function point counters who have successfully passed the certification examination offered by IFPUG. Uncertified counters should not be used because the counting rules are too complex. As with tax regulations, the rules change fairly often.

Function point analysis is accurate and useful, but slow and expensive. As a result, a number of high-speed function point methods have been developed and will be discussed later in this section.

Sizing Using Function Point Variations

The success of IFPUG function point metrics led to a curious situation. The inventor of function point metrics, A.J. Albrecht, was an electrical engineer by training and envisioned function points as a general-purpose metric that could be used for information technology projects, embedded software, systems software, and military software, and even games and entertainment software. However, the first published results that used function point metrics happened to be information technology applications such as accounting and financial software.

The historical accident that function point metrics were first used for IT applications led some researchers to conclude that function points *only* worked for IT applications. As a result, a number of function point variations have come into being, with many of them being aimed at systems and embedded software. These function point variations include but are not limited to:

1. COSMIC function points
2. Engineering function points
3. 3-D function points
4. Full function points
5. Feature points
6. Finnish function points
7. Mark II function points
8. Netherlands function points
9. Object-oriented function points
10. Web-object function points

When IFPUG function points were initially used for systems and embedded software, it was noted that productivity rates were lower for these applications. This is because systems and embedded software tend to be somewhat more complex than IT applications and really are harder to build, so productivity will be about 15 percent lower than for IT applications of the same size.

However, rather than accepting the fact that some embedded and systems applications are tougher than IT applications and will therefore have lower productivity rates, many function point variants were developed that increased the apparent size of embedded and systems applications so that they appear to be about 15 percent larger than when measured with IFPUG function points.

As mentioned earlier, it is an interesting point to think about, but one of the reasons why IT projects seem to have higher productivity rates

than systems or embedded software is that IT project historical data leaks a great deal more than historical data systems and embedded software. This is because IT applications are usually developed by a cost center, but systems and embedded software are usually developed by a profit center. This leakage is enough by itself to make IT projects look at least 15 percent more productive than systems or embedded applications of the same size in terms of function points. It is perhaps a coincidence that the size increases for systems and embedded software predicted by function point variants such as COSMIC are almost exactly the same as the leakage rates from IT application historical data.

Not all of the function point variants are due to a desire to puff up the sizes of certain kinds of software, but many had that origin. As a result now, in 2009, the term *function point* is extremely ambiguous and includes many variations. It is not possible to mix these variants and have a single unified set of benchmarks. Although some of the results may be similar, mixing the variants into the same benchmark data collection would be like mixing yards and meters or statute miles and nautical miles.

The function point variations all claim greater accuracy for certain kinds of software than IFPUG function points, but what this means is that the variations produce larger counts than IFPUG for systems and embedded software and for some other types of software. This is not the same thing as “accuracy” in an objective sense.

In fact, there is no totally objective way of ascertaining the accuracy of either IFPUG function points or the variations. It is possible to ascertain the differences in results between certified and uncertified counters, and between groups of counters who calculate function points for the same test case. But this is not true accuracy: it’s only the spread of human variation.

With so many variations, it is now very difficult to use any of them for serious estimating and planning work. If you happen to use one of the variant forms of function points, then it is necessary to seek guidance from the association or group that controls the specific counting rules used.

As a matter of policy, inventors of function point variants should be responsible for creating conversion rules between these variants and IFPUG function points, which are the oldest and original form of functional measurement. However, with few exceptions, there are no really effective conversion rules. There are some conversion rules between IFPUG and COSMIC and also between several other variations such as the Finnish and Netherlands functional metrics.

The older feature point metric was jointly developed by A.J. Albrecht and the author, so it was calibrated to produce results that matched IFPUG function points in over 90 percent of cases; for the other 10 percent, the counting rules created more feature points than function points, but the two could be converted by mathematical means.

There are other metrics with multiple variations such as statute miles and nautical miles, Imperial gallons and U.S. gallons, or temperature measured using Fahrenheit or Celsius. Unfortunately, the software industry has managed to create more metric variations than any other form of “engineering.” This is yet another sign that software engineering is not yet a true engineering discipline, since it does not yet know how to measure results with high precision.

Timing of function point variant sizing Both IFPUG function points and the variations such as COSMIC are derived from requirements and specifications, and can be quantified by the time initial requirements are complete. However, the first formal cost estimates usually are needed before requirements are complete.

Usage of function point variations The four function point variations that are certified by the ISO standards organization include the IFPUG, COSMIC, Netherlands, and Finnish methods. Because IFPUG is much older, it has more users. The COSMIC, Netherlands, and Finnish methods probably have between 200 and 1000 applications currently using them. The older Mark II method probably had about 2000 projects mainly in the United Kingdom. The other function point variations have perhaps 50 applications each.

Schedules and costs IFPUG, COSMIC, and most variations require about the same amount of time. These forms of sizing are neither quick nor inexpensive. Function point analysis of any flavor is so slow and expensive that applications larger than about 10,000 function points are almost never analyzed.

Normal function point analysis for all of the variations requires a certified function point analysis to be performed with accuracy (uncertified counts are highly inaccurate). Normal function point analysis proceeds at a rate of between 400 and 600 function points per day. At a daily average consulting fee of \$3000, the cost is between \$5.00 and \$7.50 for every function point counted.

Assuming an average cost of \$6 per function point counted for the major variants, counting a 10,000–function point application would cost \$60,000. This explains why normal function point analysis is usually only performed for applications in the 1000-function point size range.

Cautions and counter indications The main counter indication with function point analysis for all variations is that it is expensive and fairly time-consuming. While small applications less than 1000 function points can be sized in a few days, large systems greater than 10,000 function points would require weeks. No really large systems greater

than 100,000 function points have ever been sized using either IFPUG or the variations such as COSMIC due to the high costs and the fact that the schedule for the analysis would take months.

Another counter indication is that there is a lower limit for function point analysis. Small changes less than 15 function points can't be sized due to the lower limits of the adjustment factors. This is true for all of the variations such as COSMIC, Finnish, and so on. Individually, these changes are trivial, but large companies could have thousands of them every year at a total cost exceeding several million dollars.

A caution is that accurate function point analysis requires a certified function point counter who has successfully passed the certification examination offered by the function point association that controls the metric. Uncertified counters should not be used, because the counting rules are too complex. As with tax regulations, the rules change fairly often.

Function point analysis is accurate and useful, but slow and expensive. As a result, a number of high-speed function point methods have been developed and will be discussed later in this section.

High-Speed Sizing Using Function Point Approximations

The slow speed and high costs of normal function point analysis were noted within a few years of the initial development of function point metrics. Indeed, the very first commercial software cost-estimating tool that supported function point metrics, SPQR/20 in 1985, supported a method of high-speed function point analysis based on approximation rather than actual counting.

The term *approximation* refers to developing a count of function points without having access to, or knowledge of, every factor that determines function point size when using normal function point analysis.

The business goal of the approximation methods is to achieve function point totals that would come within about 15 percent of an actual count by a certified counter, but achieve that result in less than one day of effort. Indeed, some of the approximation methods operate in only a minute or two. The approximation methods are not intended as a full substitute for function point analysis, but rather to provide quick estimates early in development. This is because the initial cost estimate for most projects is demanded even before requirements are complete, so there is no way to carry out formal function point analysis at that time.

There are a number of function point approximation methods circa 2009, but the ones that are most often used include

1. Unadjusted function points
2. Function points derived from simplified complexity adjustments

3. Function points “light”
4. Function points derived from data mining of legacy applications
5. Function points derived from questions about the application
6. Function points derived from pattern matching (discussed later in this section)

The goal of these methods is to improve on the average counting speed of about 400 function points per day found with normal function point analysis. That being said, the “unadjusted” function point method seems to achieve rates of about 700 function points per day. The method using simplified complexity factors achieves rates of about 600 function points per day. The function point “light” method achieves rates of perhaps 800 function points per day.

The function point light method was developed by David Herron of the David Consulting Group, who is a certified function point counter. His light method is based on simplifying the standard counting rules and especially the complexity adjustments.

The method based on data mining of legacy applications is technically interesting. It was developed by a company called Relativity Technologies (now part of Micro Focus). For COBOL and other selected languages, the Relativity function point tool extracts hidden business rules from source code and uses them as the basis for function point analysis.

The technique was developed in conjunction with certified function point analysts, and the results come within a few percentage points of matching standard function point analysis. The nominal speed of this approach is perhaps 1000 function points per minute (as opposed to 400 per day for normal counts). For legacy applications, this method can be very valuable for retrofitting function points and using them to quantify maintenance and enhancement work.

There are several methods of approximation based on questions about the application. Software Productivity Research (SPR) and Total Metrics both have such tools available. The SPR approximation methods are embedded in the KnowledgePlan estimation tool. The Total Metrics approximation method is called *Function Point Outline* and deals with some interesting external attributes of software applications, such as the size of the requirements or functional specifications.

As noted earlier in this chapter, function points have long been used to measure and predict the size of requirements and specifications. The FP Outlook approach merely reversed the mathematics and uses known document sizes to predict function points, which is essentially another form of backfiring. Of course, document size is only one of the questions asked, but the idea is to create function point approximations based on easily available information.

The speed of the FP Outlook tool and the other question-based function point approximation methods seems to be in the range of perhaps 4000 function points per day, as opposed to the 400 function points per day of normal function point analysis.

Timing of function point approximation sizing The methods based on questions about applications can be used earlier than standard function points. Function points “light” can be used at the same time as standard function points; that is, when the requirements are known. The data mining approach requires existing source code and hence is used primarily for legacy applications. However, the approximation methods that use questions about software applications can be used very early in requirements: several months prior to when standard function point analysis might be carried out.

Usage of function point approximations The function point approximation methods vary in usage. The Relativity method and the Total Metrics method were only introduced in 2008, so usage is still growing: perhaps 250 projects each. The older approximation methods may have as many as 750 projects each.

Schedules and costs The main purpose of the approximation methods is to achieve faster function point counts and lower costs than IFPUG, COSMIC, or any other standard method of function point analysis. Their speed of operation ranges between about twice that of standard function points up to perhaps 20 times standard function point analysis. The cost per function point counted runs from less than 1 cent up to perhaps \$3, but all are cheaper than standard function point analysis.

Cautions and counter indications The main counter indication with function point approximation is accuracy. The Relativity method matches standard IFPUG function points almost exactly. The other approximation methods only come within about 15 percent of manual counts by certified counters. Of course, coming within 15 percent three months earlier than normal function points might be counted, with a cost of perhaps one-tenth normal function point analysis, are both significant business advantages.

Sizing Legacy Applications Based on “Backfiring” or LOC to Function Point Conversion

The concept of *backfiring* is nothing more than reversing the direction of the equations used when predicting source code size from function points. The technology of backfiring or direct conversion of LOC data

into the equivalent number of function points was pioneered by Allan Albrecht, the inventor of the function point metric. The first backfire data was collected within IBM circa 1975 as a part of the original development of function point metrics.

The first commercial software estimating tool to support backfiring was SPQR/20, which came out in 1985 and supported bi-directional sizing for 30 languages. Today, backfiring is a standard function for many commercial software estimating tools such as the ones already mentioned earlier in this section.

From 30 languages in 1985, the number of languages that can be sized or backfired has now grown to more than 450 circa 2009, when all dialects are counted. Of course, for the languages where no counting rules exist, backfiring is not possible. Software Productivity Research publishes an annual table of conversion ratios between logical lines of code and function points, and the current edition circa 2009 contains almost 700 programming languages and dialects. Similar tables are published by other consulting organizations such as Gartner Group and the David Consulting Group.

There are far too many programming languages to show more than a few examples in this short subsection. Note also that the margin of error when backfiring is rather large. Even so, the results are interesting and now widely utilized. Following are examples taken from the author's *Table of Programming Languages and Levels*, which is updated several times a year by Software Productivity Research (Jones, 1996). This data indicates the ranges and median values in the number of source code statements required to encode one function point for selected languages. The counting rules for source code are based on logical statements and are defined in an appendix of the author's book *Applied Software Measurement* (McGraw-Hill, 2008). Table 6-9 shows samples of the ratios of logical source code statements to function points. A full table for all 2,500 or so programming languages would not fit within the book.

Although backfiring is usually not as accurate as actually counting function points, there is one special case where backfiring is more accurate: very small modifications to software applications that have fewer than 15 function points. For changes less than 1 function point, backfiring is one of only two current approaches for deriving function points. (The second approach is pattern matching, which will be discussed later in this section.)

While backfiring is widely used and also supported by many commercial software cost-estimating tools, the method is something of an "orphan," because none of the function point user groups such as IFPUG, COMIC, and the like have ever established committees to evaluate backfiring or produced definitive tables of backfire data.

TABLE 6-9 Ratios of Logical Source Code Statements to Function Points for Selected Programming Languages

Language	Nominal Level	Source Statements Per Function Point		
		Low	Mean	High
1st Generation	1.00	220	320	500
Basic assembly	1.00	200	320	450
Macro assembly	1.50	130	213	300
C	2.50	60	128	170
BASIC (interpreted)	2.50	70	128	165
2nd Generation	3.00	55	107	165
FORTRAN	3.00	75	107	160
ALGOL	3.00	68	107	165
COBOL	3.00	65	107	150
CMS2	3.00	70	107	135
JOVIAL	3.00	70	107	165
PASCAL	3.50	50	91	125
3rd Generation	4.00	45	80	125
PL/I	4.00	65	80	95
MODULA 2	4.00	70	80	90
ADA 83	4.50	60	71	80
LISP	5.00	25	64	80
FORTH	5.00	27	64	85
QUICK BASIC	5.50	38	58	90
C++	6.00	30	53	125
Ada 9X	6.50	28	49	110
Data base	8.00	25	40	75
Visual Basic (Windows)	10.00	20	32	37
APL (default value)	10.00	10	32	45
SMALLTALK	15.00	15	21	40
Generators	20.00	10	16	20
Screen painters	20.00	8	16	30
SQL	27.00	7	12	15
Spreadsheets	50.00	3	6	9

One potential use of backfiring would be to convert historical data for applications that used story points or use-case points into function point form. This would only require deriving logical code size and then using published backfire ratios.

It would also be fairly trivial for various kinds of code analyzers such as complexity analysis tools or static analysis tools to include backfire algorithms, as could compilers for that matter.

Even though the function point associations ignore backfiring, many benchmark organizations such as Software Productivity Research (SPR),

the David Consulting Group, QPMG, Gartner Group, and so on, do publish tables of backfire conversion ratios.

While many languages in these various tables have the same level from company to company, other languages vary widely in the apparent number of source code statements per function point based on which company's table is used. This is an awkward problem, and cooperation among metrics consulting groups would be useful to the industry, although it will probably not occur.

Somewhat surprisingly, as of 2009, all of the published data on backfiring relates to standard IFPUG function point metrics. It would be readily possible to generate backfiring rules for COSMIC function points, story point, use-case points, or any other metric, but this does not seem to have happened, for unknown reasons.

Timing of backfire function point sizing Since backfiring is based on source code, its primary usage is for sizing legacy applications so that historical maintenance data can be expressed in terms of function points. A secondary usage for backfiring is to convert historical data based on lines of code metrics into function point data so it can be compared against industry benchmarks such as those maintained by ISBSG.

Usage of backfire function points The backfire method was created in part by A.J. Albrecht as a byproduct of creating function point metrics. Therefore, backfiring has been in continuous use since about 1975. Because of the speed and ease of backfiring, more applications have been sized with this method than almost any other. Perhaps as many as 100,000 software applications have been sized via backfiring.

Schedules and costs If source code size is known, the backfiring form of sizing is both quick and inexpensive. Assuming automated code counting, rates of more than 10,000 LOC per minute can be converted into function point form. This brings the cost down to less than 1 cent per function point, as opposed to about \$6 per function point for normal manual function point analysis. Backfiring does not require a certified counter. Of course, the accuracy is not very high.

Cautions and counter indications The main counter indication for backfiring is that it is not very accurate. Due to variations in programming styles, individual programmers can vary by as much as 6-to-1 in the number of lines of code used to implement the same functionality. Therefore, backfiring also varies widely. When backfiring is used for hundreds of applications in the same language, such as COBOL, the average value of about 106 code statements in the procedure and data

division yield reasonably accurate function point totals. But for languages with few samples, the ranges are very wide.

A second caution is that there are no standard methods for counting lines of code. The backfire approach was originally developed based on counts of logical statements. If backfiring is used on counts of physical lines, the results might vary by more than 500 percent from backfiring the same samples using logical statements.

Another counter indication is that backfiring becomes very complicated for applications coded in two or more languages. There are automated tools that can handle backfire conversions for any number of languages, but it is necessary to know the proportions of code in each language for the tools to work.

A final caution is that the published rules that show conversion ratios between lines of code and function points vary based on the source. The published rules by the David Consulting Group, Gartner Group, the Quality and Productivity Management Group (QPMG), and Software Productivity Research (SPR) do not show the same ratios for many languages. Since none of the function point associations such as IFPUG have ever studied backfiring, nor have any universities, there is no overall authoritative source for validating backfire assumptions.

Backfiring remains popular and widely used, even though of questionable accuracy. The reason for its popularity is because of the high costs and long schedules associated with normal function point analysis.

Sizing Based on Pattern Matching

The other sizing methods in this section are in the public domain and are available for use as needed. But sizing based on pattern matching has had a patent application filed, so the method is not yet generally available.

The pattern-matching method was not originally created as a sizing method. It was first developed to provide an unambiguous way of identifying applications for benchmark purposes. After several hundred applications had been measured using the taxonomy, it was noted that applications with the same patterns on the taxonomy were of the same size.

Pattern matching is based on the fact that thousands of legacy applications have been created, and for a significant number, size data already exists. By means of a taxonomy that captures the nature, scope, class, and type of existing software applications, a *pattern* is created that can be used to size new software applications.

What makes pattern-matching work is a taxonomy that captures key elements of software applications. The taxonomy consists of seven topics: (1) nature, (2) scope, (3) class, (4) type, (5) problem complexity, (6) code complexity, and (7) data complexity. Each topic uses numeric values for identification.

In comparing one software project against another, it is important to know exactly what kinds of software applications are being compared. This is not as easy as it sounds. The industry lacks a standard taxonomy of software projects that can be used to identify projects in a clear and unambiguous fashion other than the taxonomy that is used with this invention.

The author has developed a multipart taxonomy for classifying projects in an unambiguous fashion. The taxonomy is copyrighted and explained in several of the author's previous books including *Estimating Software Costs* (McGraw-Hill, 2007) and *Applied Software Measurement* (McGraw-Hill, 2008). Following is the taxonomy:

When the taxonomy is used for benchmarks, four additional factors from public sources are part of the taxonomy:

Country code	=	1	(United States)
Region code	=	06	(California)
City code	=	408	(San Jose)
NAIC industry code	=	1569	(Telecommunications)

These codes are from telephone area codes, ISO codes, and the North American Industry Classification (NAIC) codes of the Department of Commerce. These four codes do not affect the size of applications, but provide valuable information for benchmarks and international economic studies. This is because software costs vary widely by country, geographic region, and industry. For historical data to be meaningful, it is desirable to record all of the factors that influence costs.

The portions of the taxonomy that are used for estimating application size include the following factors:

PROJECT NATURE: _____

- 1. New program development
- 2. Enhancement (new functions added to existing software)
- 3. Maintenance (defect repair to existing software)
- 4. Conversion or adaptation (migration to new platform)
- 5. Reengineering (re-implementing a legacy application)
- 6. Package modification (revising purchased software)

PROJECT SCOPE: _____

- 1. Algorithm
- 2. Subroutine
- 3. Module
- 4. Reusable module

5. Disposable prototype
6. Evolutionary prototype
7. Subprogram
8. Stand-alone program
9. Component of a system
10. Release of a system (other than the initial release)
11. New departmental system (initial release)
12. New corporate system (initial release)
13. New enterprise system (initial release)
14. New national system (initial release)
15. New global system (initial release)

PROJECT CLASS: _____

1. Personal program, for private use
2. Personal program, to be used by others
3. Academic program, developed in an academic environment
4. Internal program, for use at a single location
5. Internal program, for use at a multiple locations
6. Internal program, for use on an intranet
7. Internal program, developed by external contractor
8. Internal program, with functions used via time sharing
9. Internal program, using military specifications
10. External program, to be put in public domain
11. External program to be placed on the Internet
12. External program, leased to users
13. External program, bundled with hardware
14. External program, unbundled and marketed commercially
15. External program, developed under commercial contract
16. External program, developed under government contract
17. External program, developed under military contract

PROJECT TYPE: _____

1. Nonprocedural (generated, query, spreadsheet)
2. Batch application

3. Web application
4. Interactive application
5. Interactive GUI applications program
6. Batch database applications program
7. Interactive database applications program
8. Client/server applications program
9. Computer game
10. Scientific or mathematical program
11. Expert system
12. Systems or support program including “middleware”
13. Service-oriented architecture (SOA)
14. Communications or telecommunications program
15. Process-control program
16. Trusted system
17. Embedded or real-time program
18. Graphics, animation, or image-processing program
19. Multimedia program
20. Robotics, or mechanical automation program
21. Artificial intelligence program
22. Neural net program
23. Hybrid project (multiple types)

PROBLEM COMPLEXITY: _____

1. No calculations or only simple algorithms
2. Majority of simple algorithms and simple calculations
3. Majority of simple algorithms plus a few of average complexity
4. Algorithms and calculations of both simple and average complexity
5. Algorithms and calculations of average complexity
6. A few difficult algorithms mixed with average and simple
7. More difficult algorithms than average or simple
8. A large majority of difficult and complex algorithms
9. Difficult algorithms and some that are extremely complex
10. All algorithms and calculations are extremely complex

CODE COMPLEXITY: _____

1. Most “programming” done with buttons or pull-down controls
2. Simple nonprocedural code (generated, database, spreadsheet)
3. Simple plus average nonprocedural code
4. Built with program skeletons and reusable modules
5. Average structure with small modules and simple paths
6. Well structured, but some complex paths or modules
7. Some complex modules, paths, and links between segments
8. Above average complexity, paths, and links between segments
9. Majority of paths and modules are large and complex
10. Extremely complex structure with difficult links and large modules

DATA COMPLEXITY: _____

1. No permanent data or files required by application
2. Only one simple file required, with few data interactions
3. One or two files, simple data, and little complexity
4. Several data elements, but simple data relationships
5. Multiple files and data interactions of normal complexity
6. Multiple files with some complex data elements and interactions
7. Multiple files, complex data elements and data interactions
8. Multiple files, majority of complex data elements and interactions
9. Multiple files, complex data elements, many data interactions
10. Numerous complex files, data elements, and complex interactions

As most commonly used for either measurement or sizing, users will provide a series of integer values to the factors of the taxonomy, as follows:

PROJECT NATURE	1
PROJECT SCOPE	8
PROJECT CLASS	11
PROJECT TYPE	15
PROBLEM COMPLEXITY	5
DATA COMPLEXITY	6
CODE COMPLEXITY	2

Although integer values are used for nature, scope, class, and type, up to two decimal places can be used for the three complexity factors. The algorithms will interpolate between integer values. Thus, permissible values might also be

PROJECT NATURE	1
PROJECT SCOPE	8
PROJECT CLASS	11
PROJECT TYPE	15
PROBLEM COMPLEXITY	5.25
DATA COMPLEXITY	6.50
CODE COMPLEXITY	2.45

The combination of numeric responses to the taxonomy provides a unique “pattern” that facilitates both measurement and sizing. The fundamental basis for sizing based on pattern matching rests on two points:

1. Observations have demonstrated that software applications that have identical patterns in terms of the taxonomy are also close to being identical in size expressed in function points.
2. The seven topics of the taxonomy are not equal in their impacts. The second key to pattern matching is the derivation of the relative weights that each factor provides in determining application size.

To use the pattern-matching approach, mathematical weights are applied to each parameter. The specific weights are defined in the patent application for the method and are therefore proprietary and not included here. However, the starting point for the pattern-matching approach is the average sizes of the software applications covered by the “scope” parameter. Table 6-10 illustrates the unadjusted average values prior to applying mathematical adjustments.

As shown in Table 6-10, an initial starting size for a software application is based on user responses to the *scope* parameter. Each answer is assigned an initial starting size value in terms of IFPUG function points. These size values have been determined by examination of applications already sized using standard IFPUG function point analysis. The initial size values represent the mode of applications or subcomponents that have been measured using function points.

The scope parameter by itself only provides an approximate initial value. It is then necessary to adjust this value based on the other parameters of class, type, problem complexity, code complexity, and data complexity. These adjustments are part of the patent application for sizing based on pattern matching.

From time to time, new forms of software will be developed. When this occurs, the taxonomy can be expanded to include the new forms.

TABLE 6-10 Initial Starting Values for Sizing by Pattern Matching

Value	APPLICATION SCOPE PARAMETER	
	Definition	Size in Function Points
1.	Algorithm	1
2.	Subroutine	5
3.	Module	10
4.	Reusable module	20
5.	Disposable prototype	50
6.	Evolutionary prototype	100
7.	Subprogram	500
8.	Stand-alone program	1,000
9.	Component of a system	2,500
10.	Release of a system	5,000
11.	New Departmental system	10,000
12.	New Corporate system	50,000
13.	New Enterprise system	100,000
14.	New National system	250,000
15.	New Global system	500,000

The taxonomy can be used well before an application has started its requirements. Since the taxonomy contains information that should be among the very first topics known about a future application, it is possible to use the taxonomy months before requirements are finished and even some time before they begin.

It is also possible to use the taxonomy on legacy applications that have been in existence for many years. It is often useful to know the function point totals of such applications, but normal counting of function points may not be feasible since the requirements and specifications are seldom updated and may not be available.

The taxonomy can also be used with commercial software, and indeed with any form of software including classified military applications where there is sufficient public or private knowledge of the application to assign values to the taxonomy tables.

The taxonomy was originally developed to produce size in terms of IFPUG function points and also logical source code statements. However, the taxonomy could also be used to produce size in terms of COSMIC function points, use-case points, or story points. To use the taxonomy with other metrics, historical data would need to be analyzed.

The sizing method based on pattern matching can be used for any size application ranging from small updates that are only a fraction of a function point up to massive defense applications that might top 300,000 function points. Table 6-11 illustrates the pattern-matching

TABLE 6-11 Sample of 150 Applications Sized Using Pattern Matching

Note 1: IFPUG rules version 4.2 are assumed.

Note 2: Code counts are based on logical statements; not physical lines

Application	Size in Function Points (IFPUG 4.2)	Language Level	Total Source Code	Lines per Function Point
1. Star Wars missile defense	352,330	3.50	32,212,992	91
2. Oracle	310,346	4.00	24,827,712	80
3. WWMCCS	307,328	3.50	28,098,560	91
4. U.S. Air Traffic control	306,324	1.50	65,349,222	213
5. Israeli air defense system	300,655	4.00	24,052,367	80
6. SAP	296,764	4.00	23,741,088	80
7. NSA Echelon	293,388	4.50	20,863,147	71
8. North Korean border defenses	273,961	3.50	25,047,859	91
9. Iran's air defense system	260,100	3.50	23,780,557	91
10. Aegis destroyer C&C	253,088	4.00	20,247,020	80
11. Microsoft VISTA	157,658	5.00	10,090,080	64
12. Microsoft XP	126,788	5.00	8,114,400	64
13. IBM MVS	104,738	3.00	11,172,000	107
14. Microsoft Office Professional	93,498	5.00	5,983,891	64
15. Airline reservation system	38,392	2.00	6,142,689	160
16. NSA code decryption	35,897	3.00	3,829,056	107
17. FBI Carnivore	31,111	3.00	3,318,515	107
18. Brain/Computer interface	25,327	6.00	1,350,757	53
19. FBI fingerprint analysis	25,075	3.00	2,674,637	107
20. NASA space shuttle	23,153	3.50	2,116,878	91
21. VA patient monitoring	23,109	1.50	4,929,910	213
22. F115 avionics package	22,481	3.50	2,055,438	91
23. Lexis-Nexis legal analysis	22,434	3.50	2,051,113	91
24. Russian weather satellite	22,278	3.50	2,036,869	91
25. Data warehouse	21,895	6.50	1,077,896	49
26. Animated film graphics	21,813	8.00	872,533	40
27. NASA Hubble controls	21,632	3.50	1,977,754	91
28. Skype	21,202	6.00	1,130,759	53
29. Shipboard gun controls	21,199	3.50	1,938,227	91
30. Natural language translation	20,350	4.50	1,447,135	71
31. American Express billing	20,141	4.50	1,432,238	71
32. M1 Abrams battle tank	19,569	3.50	1,789,133	91
33. Boeing 747 avionics package	19,446	3.50	1,777,951	91
34. NASA Mars rover	19,394	3.50	1,773,158	91

TABLE 6-11 Sample of 150 Applications Sized Using Pattern Matching (*continued*)

Note 1: IFPUG rules version 4.2 are assumed.

Note 2: Code counts are based on logical statements; not physical lines

Application	Size in Function Points (IFPUG 4.2)	Language Level	Total Source Code	Lines per Function Point
35. Travelocity	19,383	8.00	775,306	40
36. Apple iPhone	19,366	12.00	516,432	27
37. Nuclear reactor controls	19,084	2.50	2,442,747	128
38. IRS income tax analysis	19,013	4.50	1,352,068	71
39. Cruise ship navigation	18,896	4.50	1,343,713	71
40. MRI medical imaging	18,785	4.50	1,335,837	71
41. Google search engine	18,640	5.00	1,192,958	64
42. Amazon web site	18,080	12.00	482,126	27
43. Order entry system	18,052	3.50	1,650,505	91
44. Apple Leopard	17,884	12.00	476,898	27
45. Linux	17,505	8.00	700,205	40
46. Oil refinery process control	17,471	3.50	1,597,378	91
47. Corporate cost accounting	17,378	3.50	1,588,804	91
48. FedEx shipping controls	17,378	6.00	926,802	53
49. Tomahawk cruise missile	17,311	3.50	1,582,694	91
50. Oil refinery process control	17,203	3.00	1,834,936	107
51. ITT System 12 telecom	17,002	3.50	1,554,497	91
52. Ask search engine	16,895	6.00	901,060	53
53. Denver Airport luggage	16,661	4.00	1,332,869	80
54. ADP payroll application	16,390	3.50	1,498,554	91
55. Inventory management	16,239	3.50	1,484,683	91
56. eBay transaction controls	16,233	7.00	742,072	46
57. Patriot missile controls	15,392	3.50	1,407,279	91
58. Second Life web site	14,956	12.00	398,828	27
59. IBM IMS database	14,912	1.50	3,181,283	213
60. America Online (AOL)	14,761	5.00	944,713	64
61. Toyota robotic mfg.	14,019	6.50	690,152	49
62. Statewide child support	13,823	6.00	737,226	53
63. Vonage VOIP	13,811	6.50	679,939	49
64. Quicken 2006	11,339	6.00	604,761	53
65. ITMPI web site	11,033	14.00	252,191	23
66. Motor vehicle registrations	10,927	3.50	999,065	91
67. Insurance claims handling	10,491	4.50	745,995	71
68. SAS statistical package	10,380	6.50	511,017	49
69. Oracle CRM features	6,386	4.00	510,878	80

(Continued)

TABLE 6-11 Sample of 150 Applications Sized Using Pattern Matching (continued)

Note 1: IFPUG rules version 4.2 are assumed.

Note 2: Code counts are based on logical statements; not physical lines

Application	Size in Function Points (IFPUG 4.2)	Language Level	Total Source Code	Lines per Function Point
70. DNA analysis	6,213	9.00	220,918	36
71. Enterprise JavaBeans	5,877	6.00	313,434	53
72. Software renovation tool suite	5,170	6.00	275,750	53
73. Patent data mining	4,751	6.00	253,400	53
74. EZ Pass vehicle controls	4,571	4.50	325,065	71
75. U.S. patent applications	4,429	3.50	404,914	91
76. Chinese submarine sonar	4,017	3.50	367,224	91
77. Microsoft Excel 2007	3,969	5.00	254,006	64
78. Citizens bank online	3,917	6.00	208,927	53
79. MapQuest	3,793	8.00	151,709	40
80. Bank ATM controls	3,625	6.50	178,484	49
81. NVIDIA graphics card	3,573	2.00	571,637	160
82. Lasik surgery (wave guide)	3,505	3.00	373,832	107
83. Sun D-Trace utility	3,309	6.00	176,501	53
84. Microsoft Outlook	3,200	5.00	204,792	64
85. Microsoft Word 2007	2,987	5.00	191,152	64
86. Artemis Views	2,507	4.50	178,250	71
87. ChessMaster 2007 game	2,227	6.50	109,647	49
88. Adobe Illustrator	2,151	4.50	152,942	71
89. SpySweeper antispysware	2,108	3.50	192,757	91
90. Norton antivirus software	2,068	6.00	110,300	53
91. Microsoft Project 2007	1,963	5.00	125,631	64
92. Microsoft Visual Basic	1,900	5.00	121,631	64
93. Windows Mobile	1,858	5.00	118,900	64
94. SPR KnowledgePlan	1,785	4.50	126,963	71
95. All-in-one printer	1,780	2.50	227,893	128
96. AutoCAD	1,768	4.00	141,405	80
97. Software code restructuring	1,658	4.00	132,670	80
98. Intel Math function library	1,627	9.00	57,842	36
99. Sony PlayStation game controls	1,622	6.00	86,502	53
100. PBX switching system	1,592	3.50	145,517	91
101. SPR Checkpoint	1,579	3.50	144,403	91
102. Microsoft Links golf game	1,564	6.00	83,393	53
103. GPS navigation system	1,518	8.00	60,730	40

TABLE 6-11 Sample of 150 Applications Sized Using Pattern Matching (*continued*)

Note 1: IFPUG rules version 4.2 are assumed.

Note 2: Code counts are based on logical statements; not physical lines

Application	Size in Function Points (IFPUG 4.2)	Language Level	Total Source Code	Lines per Function Point
104. Motorola cell phone	1,507	6.00	80,347	53
105. Seismic analysis	1,492	3.50	136,438	91
106. PRICE-S	1,486	4.50	105,642	71
107. Sidewinder missile controls	1,450	3.50	132,564	91
108. Apple iPod	1,408	10.00	45,054	32
109. Property tax assessments	1,379	4.50	98,037	71
110. SLIM	1,355	4.50	96,342	71
111. Microsoft DOS	1,344	1.50	286,709	213
112. Mozilla Firefox	1,340	6.00	71,463	53
113. CAI APO (original estimate)	1,332	8.00	53,288	40
114. Palm OS	1,310	3.50	119,772	91
115. Google Gmail	1,306	8.00	52,232	40
116. Digital camera controls	1,285	5.00	82,243	64
117. IRA account management	1,281	4.50	91,096	71
118. Consumer credit report	1,267	6.00	67,595	53
119. Laser printer driver	1,248	2.50	159,695	128
120. Software complexity analyzer	1,202	4.50	85,505	71
121. JAVA compiler	1,185	6.00	63,186	53
122. COCOMO II	1,178	4.50	83,776	71
123. Smart bomb targeting	1,154	5.00	73,864	64
124. Wikipedia	1,142	12.00	30,448	27
125. Music synthesizer	1,134	4.00	90,736	80
126. Configuration control	1,093	4.50	77,705	71
127. Toyota Prius engine	1,092	3.50	99,867	91
128. Cochlear implant (internal)	1,041	3.50	95,146	91
129. Nintendo Game Boy DS	1,002	6.00	53,455	53
130. Casio atomic watch	993	5.00	63,551	64
131. Football bowl selection	992	6.00	52,904	53
132. COCOMO I	883	4.50	62,794	71
133. APAR analysis and routing	866	3.50	79,197	91
134. Computer BIOS	857	1.00	274,243	320
135. Automobile fuel injection	842	2.00	134,661	160
136. Antilock brake controls	826	2.00	132,144	160
137. Quick Sizer Commercial	794	6.00	42,326	53

(Continued)

TABLE 6-11 Sample of 150 Applications Sized Using Pattern Matching (continued)

Note 1: IFPUG rules version 4.2 are assumed.
Note 2: Code counts are based on logical statements; not physical lines

Application	Size in Function Points (IFPUG 4.2)	Language Level	Total Source Code	Lines per Function Point
138. CAI APO (revised estimate)	761	8.00	30,450	40
139. LogiTech cordless mouse	736	6.00	39,267	53
140. Function point workbench	714	4.50	50,800	71
141. SPR SPQR/20	699	4.50	49,735	71
142. Instant messaging	687	5.00	43,944	64
143. Golf handicap analyzer	662	8.00	26,470	40
144. Denial of service virus	138	2.50	17,612	128
145. Quick Sizer prototype	30	20.00	480	16
146. ILOVEYOU computer worm	22	2.50	2,838	128
147. Keystroke logger virus	15	2.50	1,886	128
148. MYDOOM computer virus	8	2.50	1,045	128
149. APAR bug report	3.85	3.50	352	91
150. Screen format change	0.87	4.50	62	71
AVERAGE	33,269	4.95	2,152,766	65

sizing method for a sample of 150 software applications. Each application was sized in less than one minute.

Because the pattern-matching approach is experimental and being calibrated, the information shown in Table 6-11 is provisional and subject to change. The data should not be used for any serious business purpose.

Note that the column labeled “language level” refers to a mathematical rule that was developed in the 1970s in IBM. The original definition of “level” was the number of statements in a basic assembly language that would be needed to provide the same function as one statement in a higher-level language. Using this rule, COBOL is a “level 3” language because three assembly statements would be needed to provide the functions of 1 COBOL statement. Using the same rule, Smalltalk would be a level 18 language, while Java would be a level 6 language.

When function point metrics were developed in IBM circa 1975, the existing rules for language level were extended to include the number of logical source code statements per function point.

For both backfiring and predicting source code size using pattern matching, language levels are a required parameter. However, there is

published data with language levels for about 700 programming languages and dialects.

Timing of pattern-matching sizing Because the taxonomy used for pattern matching is generic, it can be used even before requirements are fully known. In fact, pattern matching is the sizing method that can be applied the earliest in software development: long before normal function point analysis, story points, use-case points, or any other known metric. It is the only method that can be used before requirements analysis begins, and hence provide a useful size approximation before any money is committed to a software project.

Usage of pattern matching Because the pattern matching approach is covered by a patent application and still experimental, usage as of 2009 has been limited to about 250 trial software applications.

It should be noted that because pattern matching is based on an external taxonomy rather than on specific requirements, the pattern-matching approach can be used to size applications that are impossible to size using any other method. For example, it is possible to size classified military software being developed by other countries such as Iran and North Korea, neither of whom would provide such information knowingly.

Schedules and costs The pattern-matching approach is embodied in a prototype sizing tool that can predict application size at rates in excess of 300,000 function points per minute. This makes pattern matching the fastest and cheapest sizing method yet developed. The method is so fast and so easy to perform that several size estimates can easily be performed using best-case, expected-case, and worst-case assumptions.

Even without the automated prototype, the pattern-matching calculations can be performed using a pocket calculator or even by hand in perhaps 2 minutes per application.

Cautions and counter indications The main counter indication for pattern matching is that it is still experimental and being calibrated. Therefore, results may change unexpectedly.

Another caution is that the accuracy of pattern matching needs to be examined with a large sample of historical projects that have standard function point counts.

Sizing Software Requirements Changes

Thus far, all of the sizing methods discussed have produced size estimates that are valid only for a single moment. Observations of software projects indicate that requirements grow and change at rates of between

1 percent and more than 2 percent every calendar month during the design and coding phases.

Therefore, if the initial size estimate at the end of the requirements phase is 1000 function points, then this total might grow by 6 percent or 60 function points during the design phase and by 8 percent or 80 function points during the coding phase. When finally released, the original 1000 function points will have grown to 1140.

Because growth in requirements is related to calendar schedules, really large applications in the 10,000-function point range or higher can top 35 percent or even 50 percent in total growth. Obviously, this much growth will have a significant impact on both schedules and costs.

Some software cost-estimating tools such as KnowledgePlan include algorithms that predict growth rates in requirements and allow users to either accept or reject the predictions. Users can also include their own growth predictions.

There are two flavors of requirements change:

Requirements creep These are changes to requirements that cause function point totals to increase and that also cause more source code to be written. Such changes should be sized and of course if they are significant, they should be included in revised cost and schedule estimates.

Requirements churn These are changes that do not add to the function point size total of the application, but which may cause code to be written. An example of churn might be changing the format or appearance of an input screen, but not adding any new queries or data elements. An analogy from home construction might be replacing existing windows with hurricane-proof windows that fit the same openings. There is no increase in the square footage or size of the house, but there will be effort and costs.

Software application size is never stable and continues to change during development and also after release. Therefore, sizing methods need to be able to deal with changes and growth in requirements, and these requirements changes will also cause growth in source code volumes.

Requirements creep has a more significant impact than just growth itself. As it happens, because changing requirements tend to be rushed, they have higher defect potentials than the original requirements. They also tend to be harder to find and eliminate bugs, because if the changes are late, inspections may be skipped and testing will be less thorough.

As a result, creeping requirements on large software projects tend to be the source of many more defects that get delivered than the original requirements. For large systems in the 10,000-function point range, almost 50 percent of the delivered defects can be attributed to requirements changes during development.

Software Progress and Problem Tracking

From working as an expert witness in a number of software lawsuits, the author noted a chronic software project management problem. Many projects that failed or had serious delays in schedules or quality problems did not identify any problems during development by means of normal progress reports.

From depositions and discovery, both software engineers and first-line project managers knew about the problems, but the information was not included in status reports to clients and senior management when the problems were first noticed. Not until very late, usually too late to recover, did higher management or clients become aware of serious delays, quality problems, or other significant issues.

When asked why the information was concealed, the primary reason was that the lower managers did not want to look bad to executives. Of course, when the problems finally surfaced, the lower managers looked very bad, indeed.

By contrast, projects that are successful always deal with problems in a more rational fashion. They identify the problems early, assemble task groups to solve them, and usually bring them under control before they become so serious that they cannot be fixed. One of the interesting features of the Agile method is that problems are discussed on a daily basis. The same is true for the Team Software Process (TSP).

Software problems are somewhat like serious medical problems. They usually don't go away by themselves, and many require treatment by professionals in order to eliminate them.

Once a software project is under way, there are no fixed and reliable guidelines for judging its rate of progress. The civilian software industry has long utilized ad hoc milestones such as completion of design or completion of coding. However, these milestones are notoriously unreliable.

Tracking software projects requires dealing with two separate issues: (1) achieving specific and tangible milestones, and (2) expending resources and funds within specific budgeted amounts.

Because software milestones and costs are affected by requirements changes and "scope creep," it is important to measure the increase in size of requirements changes, when they affect function point totals. However, as mentioned in a previous section in this chapter, some requirements changes do not affect function point totals, which are termed *requirements churn*. Both creep and churn occur at random intervals. Churn is harder to measure than creep and is often measured via "backfiring" or mathematical conversion between source code statements and function point metrics.

As of 2009, automated tools are available that can assist project managers in recording the kinds of vital information needed for milestone reports. These tools can record schedules, resources, size changes, and also issues or problems.

For an industry now more than 60 years of age, it is somewhat surprising that there is no general or universal set of project milestones for indicating tangible progress. From the author’s assessment and benchmark studies, following are some representative milestones that have shown practical value.

Note that these milestones assume an explicit and formal review connected with the construction of every major software deliverable. Table 6-12 shows representative tracking milestones for large software projects. Formal reviews and inspections have the highest defect removal efficiency levels of any known kind of quality control activity, and are characteristic of “best in class” organizations.

The most important aspect of Table 6-12 is that every milestone is based on completing a review, inspection, or test. Just finishing up a document or writing code should not be considered a milestone unless the deliverables have been reviewed, inspected, or tested.

TABLE 6-12 Representative Tracking Milestones for Large Software Projects

1.	Requirements document completed
2.	Requirements document review completed
3.	Initial cost estimate completed
4.	Initial cost estimate review completed
5.	Development plan completed
6.	Development plan review completed
7.	Cost tracking system initialized
8.	Defect tracking system initialized
9.	Prototype completed
10.	Prototype review completed
11.	Complexity analysis of base system (for enhancement projects)
12.	Code restructuring of base system (for enhancement projects)
13.	Functional specification completed
14.	Functional specification review completed
15.	Data specification completed
16.	Data specification review completed
17.	Logic specification completed
18.	Logic specification review completed
19.	Quality control plan completed

TABLE 6-12 Representative Tracking Milestones for Large Software Projects
(continued)

20.	Quality control plan review completed
21.	Change control plan completed
22.	Change control plan review completed
23.	Security plan completed
24.	Security plan review completed
25.	User information plan completed
26.	User information plan review completed
27.	Code for specific modules completed
28.	Code inspection for specific modules completed
29.	Code for specific modules unit tested
30.	Test plan completed
31.	Test plan review completed
32.	Test cases for specific test stage completed
33.	Test case inspection for specific test stage completed
34.	Test stage completed
35.	Test stage review completed
36.	Integration for specific build completed
37.	Integration review for specific build completed
38.	User information completed
39.	User information review completed
40.	Quality assurance sign off completed
41.	Delivery to beta test clients completed
42.	Delivery to clients completed

In the litigation where the author worked as an expert witness, these criteria were not met. Milestones were very informal and consisted primarily of calendar dates, without any validation of the materials themselves.

Also, the format and structure of the milestone reports were inadequate. At the top of every milestone report, problems and issues or “red flag” items should be highlighted and discussed first.

During depositions and reviews of court documents, it was noted that software engineering personnel and many managers were aware of the problems that later triggered the delays, cost overruns, quality problems, and litigation. At the lowest levels, these problems were often included in weekly status reports or discussed at daily team meetings. But for the higher-level milestone and tracking reports that reached clients and executives, the hazardous issues were either omitted or glossed over.

A suggested format for monthly progress tracking reports delivered to clients and higher management would include these sections:

Suggested Format for Monthly Status Reports for Software Projects

1. New “red flag” problems noted this month
2. Status of last month’s “red flag” problems
3. Discussion of “red flag” items more than one month in duration
4. Change requests processed this month versus change requests predicted
5. Change requests predicted for next month
6. Size in function points for this month’s change requests
7. Size in function points predicted for next month’s change requests
8. Change requests that do not affect size in function points
9. Schedule impacts of this month’s change requests
10. Cost impacts of this month’s change requests
11. Quality impacts of this month’s change requests
12. Defects found this month versus defects predicted
13. Defect severity levels of defects found this month
14. Defect origins (requirements, design, code, etc.)
15. Defects predicted for next month
16. Costs expended this month versus costs predicted
17. Costs predicted for next month
18. Earned value for this month’s deliverable (if earned value is used)
19. Deliverables completed this month versus deliverables predicted
20. Deliverables predicted for next month

Although the suggested format somewhat resembles the items calculated using the earned value method, this format deals explicitly with the impact of change requests and also uses function point metrics for expressing costs and quality data.

An interesting question is the frequency with which milestone progress should be reported. The most common reporting frequency is monthly, although an exception report can be filed at any time it is suspected that something has occurred that can cause perturbations. For example, serious illness of key project personnel or resignation of key personnel might very well affect project milestone completions, and this kind of situation cannot be anticipated.

It might be thought that monthly reports are too far apart for small projects that only last six or fewer months in total. For small projects, weekly reports might be preferred. However, small projects usually do not get into serious trouble with cost and schedule overruns, whereas large projects almost always get in trouble with cost and schedule overruns. This article concentrates on the issues associated with large projects. In the litigation where the author has been an expert witness, every project under litigation except one was larger than 10,000 function points.

The simultaneous deployment of software sizing tools, estimating tools, planning tools, and methodology management tools can provide fairly unambiguous points in the development cycle that allow progress to be judged more or less effectively. For example, software sizing technology can now predict the sizes of both specifications and the volume of source code needed. Defect estimating tools can predict the numbers of bugs or errors that might be encountered and discovered. Although such milestones are not perfect, they are better than the former approaches.

Project management is responsible for establishing milestones, monitoring their completion, and reporting truthfully on whether the milestones were successfully completed or encountered problems. When serious problems are encountered, it is necessary to correct the problems before reporting that the milestone has been completed.

Failing or delayed projects usually lack serious milestone tracking. Activities are often reported as finished while work was still ongoing. Milestones on failing projects are usually dates on a calendar rather than completion and review of actual deliverables.

Delivering documents or code segments that are incomplete, contain errors, and cannot support downstream development work is not the way milestones are used by industry leaders.

Another aspect of milestone tracking among industry leaders is what happens when problems are reported or delays occur. The reaction is strong and immediate: corrective actions are planned, task forces assigned, and correction begins. Among laggards, on the other hand, problem reports may be ignored, and very seldom do corrective actions occur.

In more than a dozen legal cases involving projects that failed or were never able to operate successfully, project tracking was inadequate in every case. Problems were either ignored or brushed aside, rather than being addressed and solved.

Because milestone tracking occurs throughout software development, it is the last line of defense against project failures and delays. Milestones should be established formally and should be based on reviews, inspections, and tests of deliverables. Milestones should not be the dates that

deliverables more or less were finished. Milestones should reflect the dates that finished deliverables were validated by means of inspections, testing, and quality assurance review.

An interesting form of project tracking has been developed by the Shoulders Corp for keeping track of object-oriented projects. This method uses a 3-D model of software objects and classes using Styrofoam balls of various sizes that are connected by dowels to create a kind of mobile. The overall structure is kept in a location viewable by as many team members as possible. The mobile makes the status instantly visible to all viewers. Color-coded ribbons indicate status of each component, with different colors indicated design complete, code complete, documentation complete, and testing complete (gold). There are also ribbons for possible problems or delays. This method provides almost instantaneous visibility of overall project status. The same method has been automated using a 3-D modeling package, but the physical structures are easier to see and have proven more useful on actual projects. The Shoulders Corporation method condenses a great deal of important information into a single visual representation that nontechnical staff can readily understand.

A combination of daily status meetings that center on problems and possible delays are very useful. When formal written reports are submitted to higher managers or clients, the data should be quantified. In addition, possible problems that might cause delays or quality issues should be the very first topics in the report because they are more important than any other topics that are included.

Software Benchmarking

As this book is being written in early 2009, a new draft standard on performance benchmarks is being circulated for review by the International Standards Organization (ISO). The current draft is not yet approved. The current draft deals with concepts and definitions, and will be followed by additional standards later. Readers should check with the ISO organization for additional information.

One of the main business uses of software measurement and metric data is that of *benchmarking*, or comparing the performance of a company against similar companies within the same industry, or related industries. (The same kind of data can also be used as a “baseline” for measuring process improvements.)

The term *benchmark* is far older than the computing and software professions. It seemed to have its origin in carpentry as a mark of standard length on workbenches. The term soon spread to other domains. Another early definition of benchmark was in surveying, where it indicated a metal plate inscribed with the exact longitude, latitude, and

altitude of a particular point. Also from the surveying domain comes the term *baseline* which originally defined a horizontal line measured with high precision to allow it to be used for triangulation of heights and distances.

When the computing industry began, the term benchmark was originally used to define various performance criteria for processor speeds, disk and tape drive speeds, printing speeds, and the like. This definition is still in use, and indeed a host of new and specialized benchmarks has been created in recent years for new kinds of devices such as CD-ROM drives, multisynch monitors, graphics accelerators, solid-state flash disks, high-speed modems, and the like.

As a term for measuring the relative performance of organizations in the computing and software domains, the term benchmark was first applied to data centers in the 1960s. This was a time when computers were entering the mainstream of business operations, and data centers were proliferating in number and growing in size and complexity. This usage is still common for judging the relative efficiencies of data center operations.

Benchmark data has a number of uses and a number of ways of being gathered and analyzed. The most common and significant ways of gathering benchmark data are these five:

1. **Internal collection for internal benchmarks** This form is data gathered for internal use within a company or government unit by its own employees. In the United States, the author estimates that about 15,000 software projects have been gathered using this method, primarily by large and sophisticated corporations such as AT&T, IBM, EDS, Microsoft, and the like. This internal benchmark data is proprietary and is seldom made available to other organizations. The accuracy of internal benchmark data varies widely. For some sophisticated companies such as IBM, internal data is very accurate. For other companies, the accuracy may be marginal.
2. **Consultant collection for internal benchmarks** The second form is that of data gathered for internal use within a company or government unit by outside benchmark consultants. The author estimates that about 20,000 software projects have been gathered using this method, since benchmark consultants are fairly numerous. This data is proprietary, with the exception that results may be included in statistical studies without identifying the sources of the data. Outside consultants are used because benchmarks are technically complicated to do well, and specialists generally outperform untrained managers and software engineers. Also, the extensive experience of benchmark consultants helps in eliminating leakage and in finding other problems.

3. **Internal collection for public or ISBSG benchmarks** This form is data gathered for submission to an external nonprofit benchmark organization such as the International Software Benchmarking Standards Group (ISBSG) by a company's own employees. The author estimates that in the United States perhaps 3000 such projects have been submitted to the ISBSG. This data is readily available and can be commercially purchased by companies and individuals. The data submitted to ISBSG is also made available via monographs and reports on topics such as estimating, the effectiveness of various development methods, and similar topics. The questionnaires for such benchmarks are provided to clients by the ISBSG, together with instructions on how to collect the data. This method of gathering data is inexpensive, but may have variability from company to company since answers may not be consistent from one company to another.
4. **Consultant collection for proprietary benchmarks** This form consists of data gathered for submission to an external for-profit benchmark organization such as Gartner Group, the David Consulting Group, Galorath Associates, the Quality and Productivity Management Group, Software Productivity Research (SPR), and others by consultants who work for the benchmark organizations. Such benchmark data is gathered via on-site interviews. The author estimates that perhaps 60,000 projects have been gathered by the for-profit consulting organizations. This data is proprietary, with the exception of statistical studies that don't identify data sources. For example, this book and the author's previous book, *Applied Software Measurement*, utilize corporate benchmarks gathered by the author and his colleagues under contract. However, the names of the clients and projects are not mentioned due to nondisclosure agreements.
5. **Academic benchmarks** This form is data gathered for academic purposes by students or faculty of a university. The author estimates that perhaps 2000 projects have been gathered using this method. Academic data may be used in PhD or other theses, or it may be used for various university research projects. Some of the academic data will probably be published in journals or book form. Occasionally, such data may be made available commercially. Academic data is usually gathered via questionnaires distributed by e-mail, together with instructions for filling them out.

When all of these benchmark sources are summed, the total is about 100,000 projects. Considering that at least 3 million legacy applications exist and another 1.5 million new projects are probably in development, the sum total of all software benchmarks is only about 2 percent of software projects.

When the focus narrows to benchmark data that is available to the general public through nonprofit or commercial sources, the U.S. total is only about 3000 projects, which is only about 0.07 percent. This is far too small a sample to be statistically valid for the huge variety of software classes, types, and sizes created in the United States. The author suggests that public benchmarks from nonprofit sources such as the ISBSG should expand up to at least 2 percent or about 30,000 new projects out of 1.5 million or so in development. It would also be useful to have at least a 1 percent sample of legacy applications available to the public, or another 30,000 projects.

A significant issue with current benchmark data to date is the unequal distribution of project sizes. The bulk of all software benchmarks are for projects between about 250 and 2500 function points. There is very little benchmark data for applications larger than 10,000 function points, even though these are the most expensive and troublesome kinds of applications. There is almost no benchmark data available for small maintenance projects below 15 function points in size, even though such projects outnumber all other sizes put together.

Another issue with benchmark data is the unequal distribution by project types. Benchmarks for IT projects comprise about 65 percent of all benchmarks to date. Systems and embedded software comprise about 15 percent, commercial software about 10 percent, and military software comprises about 5 percent. (Since the Department of Defense and the military services own more software than any other organizations on the planet, the lack of military benchmarks is probably due to the fact that many military projects are classified.) The remaining 5 percent includes games, entertainment, iPhone and iPod applications, and miscellaneous applications such as tools.

Categories of Software Benchmarks

There are a surprisingly large number of kinds of software benchmarks, and they use different metrics, different methods, and are aimed at different aspects of software as a business endeavor.

Benchmarks are primarily collections of quantitative data that show application, phase, or activity productivity rates. Some benchmarks also include application quality data in the form of defects and defect removal efficiency. In addition, benchmarks should also gather information about the programming languages, tools, and methods used for the application.

Over and above benchmarks, the software industry also performs software *process assessments*. Software process assessments gather detailed data on software best practices and on specific topics such as project management methods, quality control methods, development methods,

maintenance methods, and the like. The process assessment method developed by the Software Engineering Institute (SEI) that evaluates an organization's "capability maturity level" is probably the best-known form of assessment, but there are several others as well.

Since it is obvious that assessment data and benchmark data are synergistic, there are also hybrid methods that collect assessment and benchmark data simultaneously. These hybrid methods tend to use large and complicated questionnaires and are usually performed via on-site consultants and face-to-face interviews. However, it is possible to use e-mail or web-based questionnaires and communicate with software engineers and managers via Skype or some other method rather than actual travel.

The major forms of software benchmarks included in this book circa 2009 are

1. International software benchmarks
2. Industry software benchmarks
3. Overall software cost and resource benchmarks
4. Corporate software portfolio benchmarks
5. Project-level software productivity and quality benchmarks
6. Phase-level software productivity and quality benchmarks
7. Activity-level software productivity and quality benchmarks
8. Software outsource versus internal performance benchmarks
9. Software maintenance and customer support benchmarks
10. Methodology benchmarks
11. Assessment benchmarks
12. Hybrid assessment and benchmark studies
13. Earned-value benchmarks
14. Quality and test coverage benchmarks
15. Cost of quality (COQ) benchmarks
16. Six Sigma benchmarks
17. ISO quality standard benchmarks
18. Security benchmarks
19. Software personnel and skill benchmarks
20. Software compensation benchmarks
21. Software turnover or attrition benchmarks
22. Software performance benchmarks

23. Software data center benchmarks
24. Software customer satisfaction benchmarks
25. Software usage benchmarks
26. Software litigation and failure benchmarks
27. Award benchmarks

As can be seen from this rather long list of software-related benchmarks, the topic is much more complicated than might be thought.

International software benchmarks Between the recession and global software competition, it is becoming very important to be able to compare software development practices around the world. International software benchmarking is a fairly new domain, but has already begun to establish a substantial literature, with useful books by Michael Cusumano, Watts Humphries, Howard Rubin, and Edward Yourdon as well as by the author of this book. One weakness with the ISBSG data is that country of origin is deliberately concealed. This policy should be reconsidered in light of the continuing recession.

When performing international benchmarks, many local factors need to be recorded. For example, Japan has at least 12 hours of unpaid overtime per week, while other countries such as Canada and Germany have hardly any. In Japan the workweek is about 44 hours, while in Canada it is only 36 hours. Vacation days also vary from country to country, as do the number of public holidays. France and the EU countries, for example, have more than twice as many vacation days as the United States.

Of course, the most important international topics for the purposes of outsourcing are compensation levels and inflation rates. International benchmarks are a great deal more complex than domestic benchmarks.

Industry benchmarks As the recession continues, more and more attention is paid to severe imbalances among industries in terms of costs and salaries. For example, the large salaries and larger bonuses paid to bankers and financial executives have shocked the world business community. Although not as well-known because the amounts are smaller, financial software executives and financial software engineering personnel earn more than similar personnel in other industries, too. As the recession continues, many companies are facing the difficult question of whether to invest significant amounts of money and effort into improving their own software development practices, or to turn over all software operations to an outsourcing vendor who may already be quite sophisticated. Benchmarks of industry schedules, effort, and costs will become increasingly important.

As of 2009, enough industry data exists to show interesting variations between finance, insurance, health care, several forms of manufacturing, defense, medicine, and commercial software vendors.

Overall software cost and resource benchmarks Cost and resources at the corporate level are essentially similar to the classic data center benchmarking studies, only transferred to a software development organization. These studies collect data on the annual expenditures for personnel and equipment, number of software personnel employed, number of clients served, sizes of software portfolios, and other tangible aspects associated with software development and maintenance. The results are then compared against norms or averages from companies of similar sizes, companies within the same industry, or companies that have enough in common to make the comparisons interesting. These high-level benchmarks are often produced by “strategic” consulting organization such as McKinsey, Gartner Group, and the like. This form of benchmark does not deal with individual projects, but rather with corporate or business-group expense patterns.

In very large enterprises with multiple locations, similar benchmarks are sometimes used for internal comparisons between sites or divisions. The large accounting companies and a number of management consulting companies can perform general cost and resource benchmarks.

Corporate software portfolio benchmarks A corporate portfolio can be as large as 10 million function points and contain more than 5000 applications. The applications can include IT projects, systems software, embedded software, commercial software, tools, outsourced applications, and open-source applications. Very few companies know how much software is in their portfolios. Considering that the total portfolio is perhaps the most valuable asset that the company owns, the lack of portfolio-level benchmarks is troubling.

There are so few portfolio benchmarks because of the huge size of portfolios and the high costs of collecting data on the entire mass of software owned by large corporations.

A portfolio benchmark study in which the author participated for a large manufacturing conglomerate took about 12 calendar months and involved 10 consultants who visited at least 24 countries and 60 companies owned by the conglomerate. Just collecting data for this one portfolio benchmark cost more than \$2 million. However, the value of the portfolio itself was about \$15 billion. That is a very significant asset and therefore deserves to be studied and understood.

Of course, for a smaller company whose portfolio was concentrated in a single data center, such a study might have been completed in a month by only a few consultants. But unfortunately, large corporations

are usually geographically dispersed, and their portfolios are highly fragmented across many cities and countries.

Project-level productivity and quality benchmarks Project-level productivity and quality benchmarks drop down below the level of entire organizations and gather data on specific projects. These project-level benchmark studies accumulate effort, schedule, staffing, cost, and quality data from a sample of software projects developed and/or maintained by the organization that commissioned the benchmark. Sometimes the sample is as large as 100 percent, but more often the sample is more limited. For example, some companies don't bother with projects below a certain minimum size, such as 50 function points, or exclude projects that are being developed for internal use as opposed to projects that are going to be released to external clients.

Project-level productivity and quality benchmarks are sometimes performed using questionnaires or survey instruments that are e-mailed or distributed to participants. This appears to be the level discussed in the new ISO draft benchmark standard. Data at the project level includes schedules, effort in hours or months, and costs. Supplemental data on programming languages and methodologies may be included. Quality data should be included, but seldom is.

To avoid "apples to oranges" comparisons, companies that perform project-level benchmark studies normally segment the data so that systems software, information systems, military software, scientific software, and other kinds of software are compared against projects of the same type. Data is also segmented by application size, to ensure that very small projects are not compared against huge systems. New projects and enhancement and maintenance projects are also segmented.

Although collecting data at the project level is fairly easy to do, there is no convenient way to validate the data or to ensure that "leakage" has not omitted a significant quantity of work and therefore costs. The accuracy of project level data is always suspect.

Phase-level productivity and quality benchmarks Unfortunately, project-level data is essentially impossible to validate and therefore tends to be unreliable. Dropping down to the level of phases provides increased granularity and therefore increased value. There are no standard definitions of *phases* that are universally agreed to circa 2009. However, a common phase pattern includes requirements, design, development, and testing.

When a benchmark study is carried out as a prelude to software process improvement activities, the similar term *baseline* is often used. In this context, the baseline reflects the productivity, schedule, staffing, and/or quality levels that exist when the study takes place. These results can

then be used to measure progress or improvements at future intervals. Benchmarks and baselines collect identical information and are essentially the same. Project-level data is not useful for baselines, so phase-level data is the minimum level of granularity that can show process improvement results.

Phase-level benchmarks are used by the ISBSG and also frequently used in academic studies. In fact, the bulk of the literature on software benchmarks tends to deal with phase-level data. Enough phase-level data is now available to have established fairly accurate averages and ranges for the United States, and preliminary averages for many other countries.

Activity-level productivity and quality benchmarks Unfortunately, measurement that collects only project data is impossible to validate. Phase-level data is hard to validate because many activities such as technical documentation and project management cross phase boundaries.

Activity-based benchmarks are even more detailed than the project-level benchmarks already discussed. Activity-based benchmarks drop down to the level of the specific kinds of work that must be performed in order to build a software application. For example, the 25 activities used by the author since the 1980s include specific sub-benchmarks for requirements, prototyping, architecture, planning, initial design, detail design, design reviews, coding, reusable code acquisition, package acquisition, code inspections, independent verification and validation, configuration control, integration, user documentation, unit testing, function testing, integration testing, system testing, field testing, acceptance testing, independent testing, quality assurance, installation, and management.

Activity-based benchmarks are more difficult to perform than other kinds of benchmark studies, but the results are far more useful for process improvement, cost reduction, quality improvement, schedule improvement, or other kinds of improvement programs. The great advantage of activity-based benchmarks is that they reveal very important kinds of information that the less granular studies can't provide. For example, for many kinds of software projects, the major cost drivers are associated with the production of paper documents (plans, specifications, user manuals) and with quality control (inspections, static analysis, testing). Both paperwork costs and defect removal costs are often more expensive than coding. Findings such as this are helpful in planning improvement programs and calculating returns on investments. But to know the major cost drivers within a specific company or enterprise, it is necessary to get down to the level of activity-based benchmark studies.

Activity-based benchmarks are normally collected via on-site interviews, although today Skype or a conference call might be used. The benchmark

interview typically takes about two hours and involves the project manager and perhaps three team members. Therefore the hours are about eight staff hours plus consulting time for collecting the benchmark itself. If function points are counted by the consultant, they would take additional time.

Software outsource versus internal performance benchmarks One of the most frequent reasons that the author has been commissioned to carry out productivity and quality benchmark studies is that a company is considering outsourcing some or all of their software development work.

Usually the outsource decision is being carried out high in the company at the CEO or CIO levels. The lower managers are alarmed that they might lose their jobs, and so they commission productivity and quality studies to compare in-house performance against both industry data and also data from major outsource vendors in the United States and abroad.

Until recently, U.S. performance measured in terms of function points per month was quite good compared with the outsource countries of China, Russia, India, and others. However, when costs were measured, the lower labor costs overseas gave offshore outsourcers a competitive edge. Within the past few years, inflation rates have risen faster overseas than in the United States, so the cost differential has narrowed. IBM, for example, recently decided to build a large outsource center in Iowa due to the low cost-of-living compared with other locations.

The continuing recession has resulted in a surplus of U.S. software professionals and also lowered U.S. compensation levels. As a result, cost data is beginning to average out across a large number of countries. The recession is affecting other countries too, but since travel costs continue to go up, it is becoming harder or at least less convenient to do business overseas.

Software maintenance and customer support benchmarks As of 2009, there are more maintenance and enhancement software engineers than development software engineers. Yet benchmarks for maintenance and enhancement work are not often performed. There are several reasons for this. One reason is that maintenance work has no fewer than 23 different kinds of update to legacy applications, ranging from minor changes through complete renovation. Another reason is that a great deal of maintenance work involves changes less than 15 function points in size, which is below the boundary level of normal function point analysis. Although individually these small changes may be fast and inexpensive, there are thousands of them, and their cumulative costs in large companies total to millions of dollars per year.

One of the key maintenance metrics that has value is that of *maintenance assignment scope* or the amount of software one person can keep up and running. Other maintenance metrics include number of users supported, rates at which bugs are fixed, and normal productivity rates expressed in terms of function points per month or work hours per function point. Defect potentials and defect removal efficiency level are also important.

One strong caution for maintenance benchmarks: the traditional “cost per defect” metric is seriously flawed and tends to penalize quality. Cost per defect achieves the lowest costs for the buggiest software. It also seems to be cheaper early rather than late, but this is really a false conclusion based on overhead rather than actual time and motion.

The new requirements for service and customer support included in the Information Technology Infrastructure Library (ITIL) are giving a new impetus to maintenance and support benchmarks. In fact, ITIL benchmarks should become a major subfield of software benchmarks.

Methodology benchmarks There are many different forms of software development methodology such as Agile development, extreme programming (XP), Crystal development, waterfall development, the Rational Unified Process (RUP), iterative development, object-oriented development (OO), rapid application development (RAD), the Team Software Process (TSP), and dozens more. There are also scores of hybrid development methods and probably hundreds of customized or local methods used only by a single company.

In addition to development methods, a number of other approaches can have an impact on software productivity, quality, or both. Some of these include Six Sigma, quality function deployment (QFD), joint application design (JAD), and software reuse.

Benchmark data should be granular and complete enough to demonstrate the productivity and quality levels associated with various development methods. The ISBSG benchmark data is complete enough to do this. Also, the data gathered by for-profit benchmark organizations such as QPMG and SPR can do this, but there are logistical problems.

The logistical problems include the following: Some of the popular development methods such as Agile and TSP use nonstandard metrics such as story points, use-case points, ideal time, and task hours. The data gathered using such metrics is incompatible with major industry benchmarks, all of which are based on function point metrics and standard work periods.

Another logistical problem is that very few organizations that use some of these newer methods have commissioned benchmarks by outside consultants or used the ISBSG data questionnaires. Therefore, the effectiveness of many software development methods is ambiguous and uncertain.

Conversion of data to function points and standard work periods is technically possible, but has not yet been performed by the Agile community or most of the other methods that use nonstandard metrics.

Assessment benchmarks Software assessment has been available in large companies such as IBM since the 1970s. IBM-style assessments became popular when Watts Humphrey left IBM and created the assessment method for the Software Engineering Institute (SEI) circa 1986. By coincidence, the author also left IBM and created the Software Productivity Research (SPR) assessment method circa 1984.

Software process assessments received a burst of publicity from the publication of two books. One of these was Watts Humphrey's book *Managing the Software Process* (Addison Wesley, 1989), which describes the assessment method used by the Software Engineering Institute (SEI). A second book on software assessments was the author's *Assessment and Control of Software Risks* (Prentice Hall, 1994), which describes the results of the assessment method used by Software Productivity Research (SPR). Because both authors had been involved with software assessments at IBM, the SEI and SPR assessments had some attributes in common, such as a heavy emphasis on software quality.

Both the SEI and SPR assessments are similar in concept to medical examinations. That is, both assessment approaches try to find everything that is right and everything that may be wrong with the way companies build and maintain software. Hopefully, not too much will be wrong, but it is necessary to know what is wrong before truly effective therapy programs can be developed.

By coincidence, both SPR and SEI utilize 5-point scales in evaluating software performance. Unfortunately, the two scales run in opposite directions. The SPR scale is based on a Richter scale, with the larger numbers indicating progressively more significant hazards. The SEI scale uses "1" as the most primitive score, and moves toward "5" as processes become more rigorous. Following is the SEI scoring system, and the approximate percentages of enterprises that have been noted at each of the five levels.

SEI Scoring System for the Capability Maturity Model (CMM)

Definition	Frequency
1 = Initial	75.0%
2 = Repeatable	15.0%
3 = Defined	7.0%
4 = Managed	2.5%
5 = Optimizing	0.5%

As can be seen, about 75 percent of all enterprises assessed using the SEI approach are at the bottom level, or “initial.” Note also that the SEI scoring system lacks a midpoint or average.

A complete discussion of the SEI scoring system is outside the scope of this book. The SEI scoring is based on patterns of responses to a set of about 150 binary questions. The higher SEI maturity levels require “Yes” answers to specific patterns of questions.

Following is the SPR scoring system, and the approximate percentages of results noted within three industry groups: military software, systems software, and management information systems software.

SPR Assessment Scoring System

Definition	Frequency (Overall)	Military Frequency	Systems Frequency	MIS Frequency
1 = Excellent	2.0%	1.0%	3.0%	1.0%
2 = Good	18.0%	13.0%	26.0%	12.0%
3 = Average	56.0%	57.0%	50.0%	65.0%
4 = Poor	20.0%	24.0%	20.0%	19.0%
5 = Very Poor	4.0%	5.0%	2.0%	3.0%

The SPR scoring system is easier to describe and understand. It is based on the average responses to the 300 or so SPR questions on the complete set of SPR assessment questionnaires.

By inversion and mathematical compression of the SPR scores, it is possible to establish a rough equivalence between the SPR and SEI scales, as follows:

SPR Scoring Range	Equivalent SEI Score	Approximate Frequency
5.99 to 3.00	1 = Initial	80.0%
2.99 to 2.51	2 = Repeatable	10.0%
2.01 to 2.50	3 = Defined	5.0%
1.01 to 2.00	4 = Managed	3.0%
0.01 to 1.00	5 = Optimizing	2.0%

The conversion between SPR and SEI assessment results is not perfect, of course, but it does allow users of either assessment methodology to have an approximate indication of how they might have appeared using the other assessment technique.

There are other forms of assessment too. For example, ISO quality certification uses a form of software assessment, as do the SPICE and TickIT approaches in Europe.

In general, software assessments are performed by outside consultants, although a few organizations do have internal assessment experts.

For SEI-style assessments, a number of consulting groups are licensed to carry out the assessment studies and gather data.

Hybrid assessment and benchmark studies Benchmark data shows productivity and quality levels, but does not explain what caused them. Assessment data shows the sophistication of software development practices, or the lack of same. But assessments usually collect no quantitative data.

Obviously, assessment data and benchmark data are synergistic, and both need to be gathered. The author recommends that a merger of assessment and benchmark data would be very useful to the industry. In fact the author's own benchmarks are always hybrid and gather assessment and benchmark data concurrently.

One of the key advantages of hybrid benchmarks is that the quantitative data can demonstrate the economic value of the higher CMM and CMMI levels. Without empirical benchmark data, the value of ascending the CMMI from level 1 to level 5 is uncertain. But benchmarks do demonstrate substantial productivity and quality levels for CMMI levels 3, 4, and 5 compared with levels 1 and 2.

The software industry would benefit from a wider consolidation of assessment and benchmark data collection methods. The advantage of the hybrid approach is that it minimizes the number of times managers and technical personnel are interviewed or asked to provide information. This keeps the assessment and benchmark data collection activities from being intrusive or interfering with actual day-to-day work.

Some of the kinds of data that need to be consolidated to get an overall picture of software within a large company or government group include

1. Demographic data on team sizes
2. Demographic data on specialists
3. Demographic data on colocation or geographic dispersion of teams
4. Application size using several metrics (function points, story points, LOC, etc.)
5. Volumes of reusable code and other deliverables
6. Rates of requirements change during development
7. Data on project management methods
8. Data on software development methods
9. Data on software maintenance methods
10. Data on specific programming languages
11. Data on specific tool suites used

12. Data on quality-control and testing methods
13. Data on defect potentials and defect removal efficiency levels
14. Data on security-control methods
15. Activity-level schedule, effort, and cost data

Hybrid assessment and benchmark data collection could gather all of this kind of information in a fairly cost-effective and nonintrusive fashion.

Earned-value benchmarks The earned-value method of comparing accumulated effort and costs against predicted milestones and deliverables is widely used on military software applications; indeed, it is a requirement for military contracts. However, outside of the defense community, earned-value calculations are also used by some outsource contracts and occasionally on internal applications.

Earned-value calculations are performed at frequent intervals, usually monthly, and show progress versus expense levels. The method is somewhat specialized and the calculations are complicated, although dozens of tools are available that can carry them out.

The earned-value approach by itself is not a true benchmark because it has a narrow focus and does not deal with topics such as quality, requirements changes, and other issues. However, the data that is collected for the earned-value approach is quite useful for benchmark studies, and could also show correlations with assessment results such as the levels of the capability maturity model integration (CMMI).

Quality and test coverage benchmarks Software quality is poorly represented in the public benchmark data offered by nonprofit organizations such as ISBSG (International Software Benchmarking Standards Group). In fact, software quality is not very well done by the entire software industry, including some major players such as Microsoft.

Companies such as IBM that do take quality seriously measure all defects from requirements through development and out into the field. The data is used to create benchmarks of two very important metrics: defect potentials and defect removal efficiency. The term *defect potentials* refers to the sum total of defects that are likely to be found in software. The term *defect removal efficiency* refers to the percentage of defects found and removed by every single review, inspection, static analysis run, and test stage.

In addition, quality benchmarks may also include topics such as complexity measured using cyclomatic and essential complexity; test coverage (percentage of code actually touched by test cases); and defect severity levels. There is a shortage of industry data on many quality topics, such as bugs or errors in test cases themselves.

In general, the software industry needs more and better quality and test coverage benchmarks. The test literature is very sparse with information such as numbers of test cases, numbers of test runs, and defect removal efficiency levels.

A strong caution about quality benchmarks is that “cost per defect” is not a safe metric to use because it penalizes quality. The author regards this metric as approaching professional malpractice. A better metric for quality economics is that of defect removal cost per function point.

Cost of quality (COQ) benchmarks It is unfortunate that such an important idea as the “cost of quality” has such an inappropriate name. Quality is not only “free” as pointed out by Phil Crosby of ITT, but it also has economic value. The COQ measure should have been named something like the “cost of defects.” In any case, the COQ approach is older than the software and computing industry and derives from a number of pioneers such as Joseph Juran, W. Edwards Deming, Kaoru Ishikawa, Genichi Taguchi, and others.

The traditional cost elements of COQ include prevention, appraisal, and failure costs. While these are workable for software, software COQ often uses cost buckets such as defect prevention, inspection, static analysis, testing, and delivered defect repairs. The ideas are the same, but the nomenclature varies to match software operations.

Many companies perform COQ benchmark studies of both software applications and engineered products. There is a substantial literature on this topic and dozens of reference books.

Six Sigma benchmarks “Six Sigma” is a mathematical expression that deals with limiting defects to no more than 3.4 per 1 million opportunities. While this quantitative result appears to be impossible for software, the philosophy of Six Sigma is readily applied to software.

The Six Sigma approach uses a fairly sophisticated and complex suite of metrics to examine software defect origins, defect discovery methods, defects delivered to customers, and other relevant topics. However, the Six Sigma approach is also about using such data to improve both defect prevention and defect detection.

A number of flavors of Six Sigma exist, but the most important flavor circa 2009 is that of “Lean Six Sigma,” which attempts a minimalist approach to the mathematics of defects and quality analysis.

The Six Sigma approach is not an actual benchmark in the traditional sense of the word. As commonly used, a benchmark is a discrete collection of data points gathered in a finite period, such as collecting data on 50 applications developed in 2009 by a telecommunications company.

The Six Sigma approach is not fixed in time or limited in number of applications. It is a continuous loop of data collection, analysis, and improvement that continues without interruption once it is initiated.

Although the ideas of Six Sigma are powerful and often effective, there is a notable gap in the literature and data when Six Sigma is applied to software. As of 2009, there is not a great deal of empirical data that shows the application of Six Sigma raises defect removal efficiency levels or lowers defect potentials.

The overall U.S. average for defect potentials circa 2009 is about 5.00 bugs per function point, while defect removal efficiency averages about 85 percent. This combination leaves a residue of 0.75 bug per function point when software is delivered to users.

Given the statistical nature of Six Sigma metrics, it would be interesting to compare all companies that use Lean Six Sigma or Six Sigma for software against U.S. averages. If so, one might hope that defect potentials would be much lower (say about 3.00 bugs per function point), while removal efficiency was much higher (say greater than 95 percent). Unfortunately, this kind of data is sparse and not yet available in sufficient quantity for a convincing statistical study.

As it happens, one way of achieving Six Sigma for software would be to achieve a defect removal efficiency rate of 99.999 percent, which has actually never occurred. However, it would seem useful to compare actual levels of defect removal efficiency against this Six Sigma theoretical target.

From a historical standpoint, defect removal efficiency calculations did not originate in the Six Sigma domain, but rather seemed to originate in IBM, when software inspections were being compared with other forms of defect removal activities in the early 1970s.

ISO quality benchmarks Organizations that needed certification for the ISO 9000-9004 quality standards or for other newer relevant ISO standards undergo an on-site examination of their quality methods and procedures, and especially the documentation for quality control approaches. This certification is a form of benchmark and actually is fairly expensive to carry out. However, there is little or no empirical data that ISO certification improves software quality in the slightest.

In other words, neither defect potentials nor defect removal efficiency levels of ISO certified organizations seem to be better than similar uncertified organizations. Indeed there is anecdotal evidence that average software quality for uncertified companies may be slightly higher than for certified companies.

Security benchmarks With the exception of studies by Homeland Security, the FBI, and more recently, the U.S. Congress, there is almost

a total absence of security benchmarks at the corporate level. As the recession lengthens and security attacks increase, there is an urgent need for security benchmarks that can measure topics such as the resistance of software to attack; numbers of attacks per company and per application; costs of security flaw prevention; costs of recovery from security attacks and denial of service attacks; and evaluations of the most effective forms of security protection.

The software journals do include benchmarks for antivirus and anti-spyware applications and firewalls that show ease of use and viruses detected or viruses let slip through. However, these benchmarks are somewhat ambiguous and casual.

So far as can be determined, there are no known benchmarks on topics such as the number of security attacks against Microsoft Vista, Oracle, SAP, Linux, Firefox, Internet Explorer, and the like. It would be useful to have monthly benchmarks on these topics. The lack of effective security benchmarks is a sign that the software industry is not yet fully up to speed on security issues.

Software personnel and skill benchmarks Software personnel and skills inventory benchmarks in the context of software are a fairly new arrival on the scene. Software has become one of the major factors in global business. Some large corporations have more than 50,000 software personnel of various kinds, and quite a few companies have more than 2500. Over and above the large numbers of workers, the total complement of specific skills and occupation groups associated with software is now approaching 90.

As discussed in earlier chapters, large enterprises have many different categories of specialists in addition to their general software engineering populations: For example, quality assurance specialists, integration and test specialists, human factors specialists, performance specialists, customer support specialists, network specialists, database administration specialists, technical communication specialists, maintenance specialists, estimating specialists, measurement specialists, function point counting specialists, and many others.

There are important questions in the areas of how many specialists of various kinds are needed, how they should be recruited, trained, and perhaps certified in their area of specialization. There are also questions dealing with the best way of placing specialists within the overall software organization structures. Benchmarking in this domain involves collecting information on how companies of various sizes in various industries deal with the increasing need for specialization in an era of downsizing and business process reengineering due to the continuing recession.

A new topic of increasing importance due to the recession is the distribution of foreign software workers who are working in the

United States on temporary work-related visas. This topic has recently been in the press when it was noted that Microsoft and Intel were laying off U.S. workers at a faster rate than they were laying off foreign workers.

Software compensation benchmarks Compensation benchmarks have been used for more than 25 years for nonsoftware studies, and they soon added software compensation to these partly open or blind benchmarks.

The way compensation benchmarks work is that many companies provide data on the compensation levels that they pay to various workers using standard job descriptions. A neutral consulting company analyzes the data and reports back to each company. Each report shows how specific companies compare with group averages. In the partly open form, the names of the other companies are identified but of course their actual data is concealed. In the blind form, the number of participating companies is known, but none of the companies are identified. There are legal reasons for having these studies carried out in blind or partly open forms, which involve possible antitrust regulations or conspiracy charges.

Software turnover and attrition benchmarks This form of benchmark was widely used outside of software before software became a major business function. The software organizations merely joined in when they became large enough for attrition to become an important issue.

Attrition and turnover benchmarks are normally carried out by human resource organizations rather than software organizations. They are classic benchmarks that are usually either blind or partly open. Dozens or even hundreds of companies report their attrition and turnover rates to a neutral outside consulting group, which then returns statistical results to each company. Each company's rate is compared with the group, but the specific rates for the other participants are concealed.

There are also internal attrition studies within large corporations such as IBM, Google, Microsoft, EDS, and the like. The author has had access to some very significant data from internal studies. The most important points were that software engineers with the highest appraisal scores leave in the greatest numbers. The most common reason cited for leaving in exit interviews is that good technical workers don't like working for bad managers.

Software performance benchmarks Software execution speed or performance is one of the older forms of benchmark, and has been carried out since the 1970s. These are highly technical benchmarks that consider application throughput or execution speed for various kinds of situations. Almost every personal computer magazine has benchmarks for

topics such as graphics processing, operating system load times, and other performance issues.

Software data center benchmarks This form of benchmark is probably the oldest form for the computing and software industry and has been carried out continuously since the 1960s. Data center benchmarks are performed to gather information on topics such as availability of hardware and software, mean time to failure of software applications, and defect repair intervals. The new Information Technology Infrastructure Library (ITIL) includes a host of topics that need to be examined so they can be included in service agreements.

While data center benchmarks are somewhat separate from software benchmarks, the two overlap because poor data center performance tends to correlate with poor quality levels of installed software.

Customer satisfaction benchmarks Formal customer satisfaction surveys have long been carried out by computer and software vendors such as IBM, Hewlett-Packard, Unisys, Google, and some smaller companies, too. These benchmark studies are usually carried out by the marketing organization and are used to suggest improvements in commercial software packages.

There are some in-house benchmarks of customer satisfaction within individual companies such as insurance companies that have thousands of computer users. These studies may also be correlated to data center benchmarks.

Software usage benchmarks As software becomes an important business and operational tool, it is obvious that software usage tends to improve the performance of various kinds of knowledge work and clerical work. In fact, prior to the advent of computers, the employment patterns of insurance companies included hundreds of clerical workers who handled applications, claims, and other clerical tasks. Most of these were displaced by computer software, and as a result the demographics of insurance companies changed significantly.

Function point metrics can be used to measure consumption of software just as well as they can measure production of software. Although usage benchmarks are rare in 2009, they are likely to grow in importance as the recession continues.

Usage benchmarks of software project managers, for example, indicate that managers who are equipped with about 3000 function points of cost estimating tools and 3000 function points of project management tools have fewer failures and shorter schedules for their projects than managers who attempt estimating and planning by hand.

Usage studies also indicate that many knowledge workers who are well equipped with software outperform colleagues who are not so well equipped. This is true for knowledge work such as law, medicine, and engineering, and also for work where data plays a significant role such as marketing, customer support, and maintenance.

Software consumption benchmark studies are just getting started circa 2009, but are likely to become major forms of benchmarks within ten years, especially if the recession continues.

Software litigation and failure benchmarks In lawsuits for breach of contract, poor quality, fraud, cost overruns, or project failure, benchmarks play a major role. Usually in such cases software expert witnesses are hired to prepare reports and testify about industry norms for topics such as quality control, schedules costs, and the like. Industry experts are also brought in for tax cases if the litigation involves the value or replacement costs of software assets.

The expert reports produced for lawsuits attempt to compare the specifics of the case against industry background data for topics such as defect removal efficiency levels, schedules, productivity, costs, and the like.

The one key topic where litigation is almost unique in gathering data is that of the causes of software failure. Most companies that have internal failures don't go to court. But failures where the software was developed under contract go to court with high frequency. These lawsuits have extensive and thorough discovery and deposition phases, so the expert witnesses who work on such cases have access to unique data that is not available from any other source.

Benchmarks based on litigation are perhaps the most complete source of data on why projects are terminated, run late, exceed their budgets, or have excessive defect volumes after release.

Award benchmarks There are a number of organizations that offer awards for outstanding performance. For example, the Baldrige Award is well known for quality and customer service. The Forbes Annual issue on the 100 best companies to work for is another kind of award. J.D. Power and Associates issues awards for various kinds of service and support excellence. For companies that aspire to "best in class" status, a special kind of benchmark can be carried out dealing with the criteria of the Baldrige Awards.

If a company is a candidate for some kind of award, quite a bit of work is involved in collecting the necessary benchmark information. However, only fairly sophisticated companies that are actually doing a good job are likely to have such expenses.

As of 2009, probably at least a dozen awards are offered by various corporations, government groups, and software journals. There are awards for customer service, for high quality, for innovative applications, and for many other topics as well.

Types of Software Benchmark Studies Performed

There are a number of methodologies used to gather the data for benchmark studies. These include questionnaires that are administered by mail or electronic mail, on-site interviews, or some combination of mailed questionnaires augmented by interviews.

Benchmarking studies can also be “open” or “blind” in terms of whether the participants know who else has provided data and information during the benchmark study.

Open benchmarks In a fully open study, the names of all participating organizations are known, and the data they provide is also known. This kind of study is difficult to do between competitors, and is normally performed only for internal benchmark studies of the divisions and locations within large corporations.

Because of corporate politics, the individual business units within a corporation will resist open benchmarks. When IBM first started software benchmarks, there were 26 software development labs, and each lab manager claimed that “our work is so complex that we might be penalized.” However, IBM decided to pursue open benchmarks, and that was a good decision because it encouraged the business unit to improve.

Partly open benchmarks One of the common variations of an open study is a limited benchmark, often between only two companies. In a two-company benchmark, both participants sign fairly detailed nondisclosure agreements, and then provide one another with very detailed information on methods, tools, quality levels, productivity levels, schedules, and the like. This kind of study is seldom possible for direct competitors, but is often used for companies that do similar kinds of software but operate in different industries, such as a telecommunications company sharing data with a computer manufacturing company.

In partly open benchmark studies, the names of the participating organizations are known, even though which company provided specific points of data is concealed. Partly open studies are often performed within specific industries such as insurance, banking, telecommunications, and the like. In fact, studies of this kind are performed for a variety of purposes besides software topics. Some of the other uses of partly open studies include exploring salary and benefit plans, office

space arrangements, and various aspects of human relations and employee morale.

An example of a partly open benchmark is a study of the productivity and quality levels of insurance companies in the Hartford, Connecticut, area where half a dozen are located. All of these companies are competitors, and all are interested in how they compare with the others. Therefore, a study gathered data from each and reported back on how each company compared with the averages derived from all of the companies. But information on how a company such as Hartford Insurance compared with Aetna or Travelers would not be provided.

Blind benchmarks In blind benchmark studies, none of the participants know the names of the other companies that participate. In extreme cases, the participants may not even know the industries from which the other companies were drawn. This level of precaution would only be needed if there were very few companies in an industry, or if the nature of the study demanded extraordinary security measures, or if the participants are fairly direct competitors.

When large corporations first start collecting benchmark data, it is obvious that the top executives of various business units will be concerned. They all have political rivals, and no executive want his or her business unit to look worse than a rival business unit. Therefore, every executive will want blind benchmarks that conceal the results of specific units. This is a bad mistake, because nobody will take the data seriously.

For internal benchmark and assessment studies within a company, it is best to show every unit by name and let corporate politics serve as an incentive to improve. This brings up the important point that benchmarks have a political aspect as well as a technical aspect.

Since executives and project managers have rivals, and corporate politics are often severe, nobody wants to be measured unless they are fairly sure the results will indicate that they are better than average, or at least better than their major political opponents.

Benchmark Organizations Circa 2009

A fairly large number of consulting companies collect benchmark data of various kinds. However, these consulting groups tend to be competitors, and therefore it is difficult to have any kind of coordination or consolidation of benchmark information.

As it happens, three of the more prominent benchmark organizations do collect activity-level data in similar fashions: The David Consulting Group, Quality and Productivity Management Group (QPMG), and Software Productivity Research (SPR). This is due to the fact the principals for all

TABLE 6-13 Examples of Software Benchmark Organizations

1.	Business Applications Performance Corporation (BAPco)
2.	Construx
3.	David Consulting Group
4.	Forrester Research
5.	Galorath Associates
6.	Gartner Group
7.	Information Technology Metrics and Productivity Institute (ITMPI)
8.	International Software Benchmarking Standards Group (ISBSG)
9.	ITABHI Corporation
10.	Open Standards Benchmarking Collaborative (OSBC)
11.	Process Fusion
12.	Quality and Productivity Management Group (QPMG)
13.	Quality Assurance Institute (QAI)
14.	Quality Plus
15.	Quantitative Software Management (QSM)
16.	Software Engineering Institute (SEI)
17.	Software Productivity Research (SPR)
18.	Standard Performance Evaluation Corporation (SPEC)
19.	Standish Group
20.	Total Metrics

three organizations have worked together in the past. However, although the data collection methods are similar, there are still some differences. But the total volume of data among these three is probably the largest collection of benchmark data in the industry. Table 6-13 shows examples of software benchmark organizations.

For all of these 20 examples of benchmark organizations, IFPUG function points are the dominant metric, followed by COSMIC function points as a distant second.

Reporting Methods for Benchmark and Assessment Data

Once assessment and benchmark data has been collected, two interesting questions are who gets to see the data, and what is it good for?

Normally, assessment and benchmarks are commissioned by an executive who wants to improve software performance. For example, benchmarks and assessments are sometimes commissioned by the CEO of a corporation, but more frequently by the CIO or CTO.

The immediate use of benchmarks and assessments is to show the executive who commissioned the study how the organization compares

against industry data. The topics of interest at the executive level include

Benchmark Contents (standard benchmarks)

Number of projects in benchmark sample

Country and industry identification codes

Application sizes

Methods and tool used

Growth rate of changing requirements

Productivity rates by activity

Net productivity for entire project

Schedules by activity

Net schedule for entire project

Staffing levels by activity

Specialists utilized

Average staff for entire project

Effort by activity

Total effort for entire project

Costs by activity

Total costs for entire project

Comparison to industry data

Suggestions for improvements based on data

Once an organization starts collecting assessment and benchmark data, they usually want to improve. This implies that data collection will be an annual event, and that the data will be used as baselines to show progress over multiple years.

When improvement occurs, companies will want to assemble an annual baseline report that shows progress for the past year and the plans for the next year. These annual reports are produced on the same schedule as corporate annual reports for shareholders; that is, they are created in the first quarter of the next fiscal year.

The contents for such an annual report would include

Annual Software Report for Corporate Executives and Senior Management

CMMI levels by business group

Completed software projects by type

IT applications

- Systems software
- Embedded applications
- Commercial packages
- Other (if any)
- Cancelled software projects (if any)
- Total costs of software in current year
- Unbudgeted costs in current year
 - Litigation
 - Denial of service attacks
 - Malware attacks and recovery
- Costs by type of software
- Costs of development versus maintenance
- Customer satisfaction levels
- Employee morale levels
- Average productivity
- Ranges of productivity
- Average quality
- Discovered defects during development
- Delivered defects reported by clients in 90 days
- Cost of quality (COQ) for current year
- Comparison of local results to ISBSG and other external benchmarks

Most of the data in the annual report would be derived from assessment and benchmark studies. However, a few topics such as those dealing with security problems such as denial of service attacks are not part of either standard benchmarks or standard assessments. They require special studies.

Summary and Conclusions

Between about 1969 and today in 2009, software applications have increased enormously in size and complexity. In 1969, the largest applications were fewer than 1000 function points, while in 2009, they top 100,000 function points in size.

In 1969, programming or coding was the major activity for software applications and constituted about 90 percent of the total effort. Most applications used only a single programming language. The world total

of programming languages was fewer than 25. Almost the only specialists in 1969 were technical writers and perhaps quality assurance workers.

Today in 2009, coding or programming is less than 40 percent of the effort for large applications, and the software industry now has more than 90 specialists. More than 700 programming languages exist, and almost every modern application uses at least two programming languages; some use over a dozen.

As the software industry increased in numbers of personnel, size of applications, and complexity of development, project management fell behind. Today in 2009, project managers are still receiving training that might have been effective in 1969, but it falls short of what is needed in today's more complicated world.

Even worse, as the recession increases in severity, there is an urgent need to lower software costs. Project managers and software engineers need to have enough solid empirical data to evaluate and understand every single cost factor associated with software. Unfortunately, poor measurement practices and a shortage of solid data on quality, security, and costs have put the software industry in a very bad economic position.

Software costs more than almost any other manufactured product; it is highly susceptible to security attacks; and it is filled with bugs or defects. Yet due to the lack of reliable benchmark and quality data, it is difficult for either software engineers or project managers to deal with these serious problems effectively.

The software industry needs better quality, better security, lower costs, and shorter schedules. But until solid empirical data is gathered on all important projects, both software engineers and project managers will not be able to plan effective solutions to industrywide problems. Many process improvement programs are based on nothing more than adopting the methodology *du jour*, such as Agile in 2009, without any empirical data on whether it will be effective. Better measurements and better benchmarks are the keys to software success.

Readings and References

- Abran, Alain and Reiner R. Dumke. *Innovations in Software Measurement*. Aachen, Germany: Shaker-Verlag, 2005.
- Abran, Alain, Manfred Bundschuh, Reiner Dumke, Christof Ebert, and Horst Zuse. *Software Measurement News*, Vol. 13, No. 2, Oct. 2008.
- Boehm, Dr. Barry. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.
- Booch, Grady. *Object Solutions: Managing the Object-Oriented Project*. Reading, MA: Addison Wesley, 1995.
- Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1974, rev. 1995.
- Bundschuh, Manfred and Carol Dekkers. *The IT Measurement Compendium*. Berlin: Springer-Verlag, 2008.

- Capability Maturity Model Integration. Version 1.1. Software Engineering Institute, Carnegie-Mellon Univ., Pittsburgh, PA. March 2003. www.sei.cmu.edu/cmmi/
- Charette, Bob. *Application Strategies for Risk Management*. New York: McGraw-Hill, 1990.
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Cohn, Mike. *Agile Estimating and Planning*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005.
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- Ebert, Christof and Reiner Dumke. *Software Measurement: Establish, Extract, Evaluate, Execute*. Berlin: Springer-Verlag, 2007.
- Ewusi-Mensah, Kwaku. *Software Development Failures*. Cambridge, MA: MIT Press, 2003.
- Galarath, Dan. *Software Sizing, Estimating, and Risk Management: When Performance is Measured Performance Improves*. Philadelphia: Auerbach Publishing, 2006.
- Garmus, David and David Herron. *Function Point Analysis—Measurement Practices for Successful Software Projects*. Boston: Addison Wesley Longman, 2001.
- Garmus, David and David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Harris, Michael, David Herron, and Stacia Iwanicki. *The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results*. Boca Raton, FL: CRC Press (Auerbach), 2008.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989.
- International Function Point Users Group (IFPUG). *IT Measurement—Practical Advice from the Experts*. Boston: Addison Wesley Longman, 2002.
- Johnson, James, et al. *The Chaos Report*. West Yarmouth, MA: The Standish Group, 2000.
- Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Prentice Hall, 1994.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007.
- Jones, Capers. *Patterns of Software System Failure and Success*. Boston: International Thomson Computer Press, December 1995.
- Jones, Capers. *Program Quality and Programmer Productivity*. IBM Technical Report TR 02.764, IBM. San Jose, CA. January 1977.
- Jones, Capers. *Programming Productivity*. New York: McGraw-Hill, 1986.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000.
- Jones, Capers. "Software Project Management Practices: Failure Versus Success." *CrossTalk*, Vol. 19, No. 6 (June 2006): 4–8.
- Jones, Capers. "Why Flawed Software Projects are not Cancelled in Time." *Cutter IT Journal*, Vol. 10, No. 12 (December 2003): 12–17.
- Laird, Linda M. and Carol M. Brennan. *Software Measurement and Estimation: A Practical Approach*. Hoboken, NJ: John Wiley & Sons, 2006.
- McConnell, Steve. *Software Estimating: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Park, Robert E., et al. *Checklists and Criteria for Evaluating the Costs and Schedule Estimating Capabilities of Software Organizations*. Technical Report CMU/SEI 95-SR-005. Pittsburgh, PA: Software Engineering Institute, January 1995.
- Park, Robert E., et al. *Software Cost and Schedule Estimating—A Process Improvement Initiative*. Technical Report CMU/SEI 94-SR-03. Pittsburgh, PA: Software Engineering Institute, May 1994.
- Parthasarathy, M.A. *Practical Software Estimation—Function Point Metrics for Insourced and Outsourced Projects*. Upper Saddle River, NJ: Infosys Press, Addison Wesley, 2007.
- Putnam, Lawrence H. and Ware Myers. *Industrial Strength Software—Effective Management Using Measurement*. Los Alamitos, CA: IEEE Press, 1997.
- Putnam, Lawrence H. *Measures for Excellence – Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1992.

- Roetzheim, William H. and Reyna A. Beasley. *Best Practices in Software Cost and Schedule Estimation*. Saddle River, NJ: Prentice Hall PTR, 1998.
- Stein, Timothy R. *The Computer System Risk Management Book and Validation Life Cycle*. Chico, CA: Paton Press, 2006.
- Strassmann, Paul. *Governance of Information Management: The Concept of an Information Constitution*, Second Edition. (eBook). Stamford, CT: Information Economics Press, 2004.
- Strassmann, Paul. *Information Payoff*. Stamford, CT: Information Economics Press, 1985.
- Strassmann, Paul. *Information Productivity*. Stamford, CT: Information Economics Press, 1999.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.
- Stukes, Sherry, Jason Deshoretz, Henry Apgar, and Ilona Macias. *Air Force Cost Analysis Agency Software Estimating Model Analysis*. TR-9545/008-2. Contract F04701-95-D-0003, Task 008. Management Consulting & Research, Inc. Thousand Oaks, CA. September 30 1996.
- Stutzke, Richard D. *Estimating Software-Intensive Systems*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Symons, Charles R. *Software Sizing and Estimating—Mk II FPA (Function Point Analysis)*. Chichester, UK: John Wiley & Sons, 1991.
- Wellman, Frank. *Software Costing: An Objective Approach to Estimating and Controlling the Cost of Computer Software*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Whitehead, Richard. *Leading a Development Team*. Boston: Addison Wesley, 2001.
- Yourdon, Ed. *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.
- Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005.

Requirements, Business Analysis, Architecture, Enterprise Architecture, and Design

Introduction

Before any code can be created for a software application, it is necessary to define the features, scope, structure, and user interfaces that will be developed. It is also necessary to define the methods of delivery of those features, and the platforms on which the application will operate. In addition, targets and goals for the application must be defined in terms of performance, security, reliability, and a number of other topics. These various issues are spread among a number of documents and plans that include requirements, business analysis, architecture, and design. Each of these can be subset into several topical segments and subdocuments.

Although a number of templates and models exist for each kind of document, no methods have proven to be totally successful. Even after more than 60 years of software, a number of common problems still occur for almost all major software applications:

1. Requirements grow and change at rates in excess of 1 percent per calendar month.
2. Few applications include greater than 80 percent of user requirements in the first release.
3. Some requirements are dangerous or “toxic” and should not be included.

4. Some applications are overstuffed with extraneous features no one asked for.
5. Most software applications are riddled with security vulnerabilities.
6. Errors in requirements and design cause many high-severity bugs.
7. Effective methods such as requirement and design inspections are seldom used.
8. Standard, reusable requirements and designs are not widely available.
9. Mining legacy applications for “lost” business requirements seldom occurs.
10. The volume of paper documents may be too large for human understanding.

These ten problems are endemic to the software industry. Unlike the design of physical structures such as aircraft, boats, buildings, or medical equipment, software does not utilize effective and proven design methods and standard document formats. In other words, if a reader picks up the requirements or specifications for two different software applications, the contents and format are likely to be very different. These differences make validation difficult because without standard and common structures, there are far too many variations to allow easy editing or error identification. Automated verification of requirements and design are theoretically possible, but beyond the state of the art as of 2009. Formal inspections of requirements and other documents are effective, but of course manual inspections are slower than automated verification.

There are also numerous “languages” for representing requirement and design features. These include use-cases, user stories, decision tables, fishbone diagrams, state-change diagrams, entity-relationship diagrams, executable English, normal English, the unified modeling language (UML), and perhaps 30 other flavors of graphical representation (flowcharts, Nassi-Schneiderman charts, data-flow diagrams, HIPO diagrams, etc.). For quality requirements, there are also special diagrams associated with quality function deployment (QFD).

The existence of so many representation techniques indicates that no perfect representation method has yet been developed. If any one of these methods were clearly superior to the others, then no doubt it would become a de facto standard used for all software projects. So far as can be determined, no representation method is used by more than perhaps 10 percent of software applications. In fact, most software applications utilize multiple representation methods because none is fully adequate

for all business and technical purposes. Therefore, combinations of text and graphical representations in the form of use-cases, flowcharts, and other diagrams are the most common approach.

In this chapter, we will be dealing with some of the many variations in methods for handling software requirements, business analysis, architecture, and design.

Software Requirements

If software engineering is to become a true profession rather than an art form, software engineers have a responsibility to help customers define requirements in a thorough and effective manner.

It is the job of a professional software engineer to insist on effective requirements methods such as joint application design (JAD), quality function deployment (QFD), and requirements inspections. It is also the responsibility of software engineers to alert clients to any potentially harmful requirements.

Far too often the literature on software requirements is passive and makes the incorrect assumption that users will be 100 percent effective in identifying requirements. This is a dangerous assumption. User requirements are never complete and they are often wrong. For a software project to succeed, requirements need to be gathered and analyzed in a professional manner, and software engineering is the profession that should know how to do this well.

It should be the responsibility of the software engineers to insist that proper requirements methods be used. These include data mining of legacy applications, joint application design (JAD), quality function deployment (QFD), prototypes, and requirements inspections. Another method that benefits requirements such as embedded users (as with Agile development). Use-cases might also be recommended.

The users of software applications are not software engineers and cannot be expected to know optimal ways of expressing and analyzing requirements. Ensuring that requirements collection and analysis are at state-of-the-art levels devolves to the software engineering team.

Today in 2009, almost half of all major applications are replacements for aging legacy applications, some of which have been in use for more than 25 years. Unfortunately, legacy applications seldom have current specifications or requirements documents available.

Due to the lack of available information about the features and functions of the prior legacy application, a new form of requirements analysis is coming into being. This new form starts by *data mining* of the legacy application in order to extract business rules and algorithms. As it happens, data mining can also be used to gather data for sizing, in terms of both function points and code statements.

Structure and Contents of Software Requirements

Software requirements obviously describe the key features and functions that a software application will contain. But requirements specifications also serve other business purposes. For example, the requirements should also discuss any limits or constraints on the software, such as performance criteria, reliability criteria, security criteria, and the like.

The costs and schedules of building software applications are strongly influenced by the size of the application in terms of the total requirements set that will be implemented. Therefore, requirements are the primary basis of ascertaining software size.

By fortunate coincidence, the structure of the function point metric is a good match to the fundamental issues that should be included in software requirements. In chronological order, these seven fundamental topics should be explored as part of the requirements gathering process:

1. The *outputs* that should be produced by the application
2. The *inputs* that will enter the software application
3. The *logical files* that must be maintained by the application
4. The *entities and relationships* that will be in the logical files of the application
5. The *inquiry types* that can be used with the application
6. The *interfaces* between the application and other systems
7. Key *algorithms* that must be present in the application

Five of these seven topics are the basic elements of the International Function Point Users Group (IFPUG) function point metric.

The fourth topic, “entities and relationships,” is part of the British Mark II function point metric and the newer COSMIC function point.

The seventh topic, “algorithms,” is a standard factor of the feature point metric, which added a count of algorithms to the five basic function point elements used by IFPUG.

The similarity between the topics that need to be examined when gathering requirements and those used by the functional metrics makes the derivation of function point totals during requirements a fairly straightforward task. In fact, automated creation of function point size from requirements has been accomplished experimentally, although this is not yet commonplace.

However, 30 additional topics also need to be explored and decided during the requirements phase. Some of these are nonfunctional requirements, and some are business requirements needed to determine whether

funding should be provided for the application. These additional topics include

1. The *size* of the application in function points and source code
2. The *schedule* of the application from requirements to delivery
3. The *staffing* of the development team, including key specialists
4. The *cost* of the application by activity and also in terms of cost per function point
5. The *business value* of the application and *return on investment* (ROI)
6. The *nonfinancial* value, such as competitive advantages and customer loyalty
7. The major *risks* facing the application, that is, termination, delays, overruns, and so on
8. The features of *competitive applications* by business rivals
9. The *method of delivery*, such as SOA, SaaS, disks, downloads, and so on
10. The *supply chain* of the application, or related applications upstream or downstream
11. The *legacy requirements* derived from older applications being replaced
12. The *laws and regulations* that impact the application (i.e., tax laws; privacy, etc.)
13. The *quality levels* in terms of defects, reliability, and ease of use criteria
14. The *error-handling* features in case of user errors or power outages, and so on
15. The *warranty terms* of the application and responses to warranty claims
16. The *hardware platform(s)* on which the application will operate
17. The *software platform(s)*, such as operating systems and databases
18. The *nationalization* criteria, or the number of foreign language versions
19. The *security criteria* for the application and its companion databases
20. The *performance criteria*, if any, for the application
21. The *training requirements* or form of tutorial materials that may be needed

22. The *installation procedures* for starting and initializing the application
23. The *reuse criteria* for the application in terms of both reused materials going into the application and also whether features of the application may be aimed at subsequent reuse by downstream applications
24. The *use cases or major tasks* users are expected to be able to perform via the application
25. The *control flow* or sequence of information moving through the application
26. Possible *future requirements* for follow-on releases
27. The *hazard levels* of any requirements that might be potentially “toxic”
28. The *life expectancy* of the application in terms of service life once deployed
29. The projected *total cost of ownership* (TCO) of the application
30. The *release frequency* for new features and repairs (annually, monthly, etc.)

The seven primary topics and the 30 supplemental topics are not the only items that need to be examined during requirements, but none of these should be omitted, since they can all significantly affect software projects.

Most of these 37 topics are needed for many different kinds of applications: commercial packages, in-house applications, outsource applications, defense projects, systems software, and embedded applications.

Statistical Analysis of Software Requirements

From analyzing thousands of software applications in hundreds of companies, the author has noted some basic facts about software requirements.

As software applications grow larger, the volume of software requirements also grows larger. However, the growth in requirements cannot keep pace with the growth of the software itself. As a result, the larger the application, the less complete the requirements are.

The fact that software requirements are incomplete for large software applications leads to the phenomenon of continuous requirements change at rates between 1 percent and 3 percent per calendar month.

Requirements may contain hundreds of bugs or defects. These are difficult to remove via testing, but can be found by means of formal requirement inspections.

Requirements are translated into designs, and designs are translated into code. A study by the author at IBM found that at each translation point, 10 percent to 15 percent of the requirements do not make it downstream into the next stage, at least initially.

In addition to *creeping requirements* instituted by users, which presumably have some business value, a surprising number of changes are added by developers, without any formal requirements or even any apparent need on the part of users. For some applications, more than 7 percent of the delivered functions were added by the developers, sometimes without the users even being aware of them. The topic of spontaneous and unsolicited change is seldom discussed in the requirements literature. (When developers were asked why they did this, the most common response was “I thought it might be useful.”)

In aggregate, about 15 percent of initial user requirements are missing from the formal requirements documents and show up as creeping requirements later on. At each translation point from requirements to some other deliverable such as design or code, about 10 percent of the requirements accidentally drop out and have to be added back in later or in subsequent releases. As mentioned, developers spontaneously add features without any user requirements asking for them, and sometimes even without the knowledge of the users. Perhaps 7 percent of delivered features are in the form of unsolicited developer-added features that lack any customer requirements, although some of these may turn out to be useful. In addition to unplanned growth and unplanned loss of requirements, some requirements are toxic or harmful, while many may contain errors ranging from high severity to low severity.

In theory, some kinds of requirements such as executable English could use static analysis or some form of automated validation, but to date this approach is experimental.

Some software requirements may be *toxic* or cause serious harm if they are not removed. A prime example of a toxic requirement is the famous Y2K problem. Another example of a toxic requirement is the file-handling protocol of the Quicken financial application. If backup files are opened instead of being restored, then data integrity can be lost. A very common toxic requirement in many applications is the failure to accommodate people with three names. Yet another toxic requirement is the poor error-handling routines in many software applications, which have become the preferred route for virus and spyware infections. The bottom line is that the traditional definition of quality as “conformance to requirements” is not safe because of the presence of so many serious toxic requirements.

At this point it is interesting to look at information about the size of software requirements, and also about the numbers of bugs or defects that might be in software requirements.

TABLE 7-1 Requirements Pages per Function Point

Function Points	English Text	Exec. English	Use-Cases	UML Diagrams	User Stories	Average
10	0.40	0.35	0.50	1.00	0.35	0.52
100	0.50	0.45	0.60	1.10	0.40	0.61
1,000	0.55	0.50	0.70	1.15	0.45	0.67
10,000	0.40	0.45	0.60	0.80	0.00	0.56
100,000	0.30	0.40	0.50	0.75	0.00	0.49
Average	0.43	0.43	0.58	0.96	0.40	0.56

Table 7-1 shows the approximate size of software requirements in terms of pages per function point. The metric used is that of the International Function Point Users Group (IFPUG), counting rules version 4.2. Five different requirement “languages” are shown in Table 7-1.

Note that for Table 7-1 and the other tables in this chapter, no data is available for “user stories” for applications in the 10,000 to 100,000–function point range. This is because the Agile methods are not used for such large applications, or at least have not reported any results to benchmark organizations.

The most important fact that Table 7-1 reveals is that the size of requirements peaks at about 1000 function points. For large applications, the volume of paper documents would grow too large to read if 100 percent of requirements were documented.

Table 7-2 extends the results from Table 7-1 and shows the approximate total quantity of pages in the requirements for each of the five methods.

As can be seen, large systems have an enormous volume of pages for requirements, and yet they are not complete. In fact, if requirements were 100 percent complete for a large application in the 100,000–function point size range, it would take more than 2500 days, or almost seven years, to read them! It is obvious that such a mass of paper is unmanageable.

Table 7-3 extends the logic derived from Table 7-2 and shows the approximate completeness of software requirements.

TABLE 7-2 Requirement Pages Produced by Application Size

Function Points	English Text	Exec. English	Use-Cases	UML Diagrams	User Stories	Average
10	4	4	5	10	4	5
100	50	45	60	110	40	61
1,000	550	500	700	1,150	450	670
10,000	4,000	4,500	6,000	8,000	0	4,500
100,000	30,000	40,000	50,000	75,000	0	48,750
Average	6,921	9,010	11,353	16,854	165	8,860

TABLE 7-3 Requirements Completeness by Software Size

Function Points	English Text	Exec. English	Use-Cases	UML Diagrams	User Stories	Average
10	98.00%	99.00%	96.00%	99.00%	93.00%	97.00%
100	95.00%	96.00%	95.00%	97.00%	90.00%	94.60%
1,000	90.00%	93.00%	90.00%	95.00%	87.00%	91.00%
10,000	77.00%	90.00%	82.00%	90.00%	0.00%	84.75%
100,000	62.00%	83.00%	74.00%	80.00%	0.00%	74.75%
Average	84.40%	92.20%	87.40%	92.20%	90.00%	88.42%

As can be seen from Table 7-3, completeness of requirements declines as software size goes up. This explains why creeping requirements are endemic within the software industry. It is doubtful if any requirement method or language could really reach 100 percent for large applications.

Table 7-4 shows the approximate numbers of requirements defects per function point observed in applications of various sizes, using various languages.

While the size of software requirement specifications goes down as application size goes up, the same is not true for requirements bugs or defects. The larger the application, the more requirement bugs there are likely to be.

However, note that these tables show only approximate average results. Many defect prevention methods such as joint application design (JAD), prototypes, and participation in formal inspections can lower these typical results by more than 60 percent.

Table 7-5 extends the results of Table 7-4 and shows the approximate numbers of requirements defects that are likely to occur by application size. For large applications, the numbers are alarming and cry out for using state-of-the-art defect prevention and removal methods.

Note that these defects are of all severity levels. Only a small fraction would generate serious problems. But with thousands of latent defects in requirements, it is obvious that formal inspections and other methods of

TABLE 7-4 Requirements Defects per Function Point

Function Points	English Text	Exec. English	Use-Cases	UML Diagrams	User Stories	Average
10	0.52	0.46	0.65	1.30	0.48	0.68
100	0.57	0.50	0.80	1.46	0.53	0.77
1,000	0.60	0.55	0.98	1.61	0.63	0.87
10,000	0.70	0.60	1.20	1.60	0.00	1.03
100,000	0.72	0.65	1.10	1.65	0.00	1.03
Average	0.62	0.55	0.95	1.52	0.55	0.88

TABLE 7-5 Requirements Defects by Application Size

Function Points	English Text	Exec. English	Use-Cases	UML Diagrams	User Stories	Average
10	5	5	7	13	5	7
100	57	50	80	146	53	77
1,000	600	550	980	1,610	630	874
10,000	7,000	6,000	12,000	16,000	0	10,250
100,000	72,000	65,000	110,000	165,000	0	103,000
Average	15,932	14,321	24,613	36,554	229	22,842

requirement defect removal should be standard practices for all applications larger than 1000 function points.

Because the numbers in Table 7-5 are so large and alarming, Table 7-6 shows only the most serious or “toxic” defects that are likely to occur.

The defects shown in Table 7-6 are harmful problems such as the Y2K problem that cause problems for users and that trigger expensive repairs when they finally surface and are identified.

The bottom line is that requirements cannot be complete for large applications above 10,000 function points. At least they never have been complete.

In addition, there will be requirements defects, and a fraction of requirements defects will cause serious harm. Much more study is needed of requirements defects, defect prevention, and defect removal.

One topic requiring additional study is how many people are involved in the requirements process. Customers have “assignment scopes” of about 5000 function points. That reflects the normal quantity of software features that one user knows well enough to define what is needed. The range of user knowledge runs from about 1000 function points up to perhaps 10,000 function points.

The assignment scope of systems or business analysts is larger, and runs up to about 50,000 function points, although average amounts are perhaps 15,000 function points.

TABLE 7-6 Toxic Requirements that Cause Serious Harm

Function Points	English Text	Exec. English	Use-Cases	UML Diagrams	User Stories	Average
10	0	0	0	0	0	0
100	0	0	0	0	0	0
1,000	1	1	2	4	1	2
10,000	15	14	25	40	0	19
100,000	175	150	300	400	0	205
Average	38	33	65	89	0	45

These typical assignment scopes mean that for a large system in the 50,000–function point range, about ten customers will need to be interviewed by one systems analyst. In other words, the ratio of business analysts to customers is about 1-to-10.

These ratios have implications for the Agile approach of embedding users in development teams. Since most Agile projects are small and fewer than 1500 function points, a single user can suffice to express most of the requirements. However, for large applications, more users are necessary.

Another topic that needs more work is the rate at which requirements can be gathered and analyzed. If you assume a typical joint application design (JAD) session contains four user representatives and two business analysts, they can usually discuss and document requirements at a rate of perhaps 1000 function points per day. It should be noted that requirements specifications average perhaps 0.5 page per function point using English text, and perhaps 0.75 page using the UML.

A single user embedded within an Agile development team can explain requirements at a rate of perhaps 200 function points per day. User stories are compact and average about 0.3 page per function point. However, they are not complete, so verbal interchange between the user and the development team is an integral part of Agile requirements.

Creating Taxonomies of Reusable Software Requirements

For purposes of benchmarks, feature analysis, and statistical analysis of productivity and quality, it is useful to record basic information about software applications. Surprisingly, the software industry does not have a standard taxonomy that allows applications to be uniquely identified. To fill this gap, the author has developed a taxonomy that allows software applications to be analyzed statistically with little ambiguity.

For identifying software for statistical purposes and for studying software requirements by industry, it is useful to know certain basic facts such as the country of origin and the industry. To record these facts, standard codes can be used:

Country code	=	1 (United States)
Region code	=	06 (California)
City code	=	408 (San Jose)
Industry code	=	1569 (Telecommunications)
CMMI level	=	3 (Controlled and repeatable)
Starting date	=	04/20/2009
Plan completion date	=	05/10/2011
True completion date	=	09/25/2011

These codes are from telephone area codes, ISO codes, and the North American Industry Classification (NAIC) codes of the Department

of Commerce. They do not affect the sizing algorithms of the invention, but provide valuable information for benchmarks and international economic studies. This is because software costs vary widely by country, geographic region, and industry. For historical data to be meaningful, it is desirable to record all of the factors that influence costs, schedules, requirements, and other factors.

The entry for “CMMI level” refers to the famous Capability Maturity Model Integration developed by the Software Engineering Institute (SEI).

After location and industry identification, the taxonomy consists of seven topics:

1. Project nature
2. Project scope
3. Project class
4. Project type
5. Problem complexity
6. Code complexity
7. Data complexity

In comparing one software project against another, it is important to know exactly what kinds of software applications are being compared. This is not as easy as it sounds. The industry has long lacked a standard taxonomy of software projects that can be used to identify projects in a clear and unambiguous fashion.

By means of multiple-choice questions, the taxonomy shown here condenses more than 35 million variations down to a small number of numeric data items that can easily be used for statistical analysis. The main purpose of a taxonomy is to provide fundamental structures that improve the ability to do research and analysis.

The taxonomy shown here has been in continuous use since 1984. The taxonomy is explained in several of the author’s prior books, including *Estimating Software Costs* (McGraw-Hill, 2007) and *Applied Software Measurement* (McGraw-Hill, 2008), as well as in older editions of the same books and also in monographs. The taxonomy is also embedded in software estimating tools designed by the author. The elements of the taxonomy follow:

PROJECT NATURE: _____

1. New program development
2. Enhancement (new functions added to existing software)

3. Maintenance (defect repair to existing software)
4. Conversion or adaptation (migration to new platform)
5. Reengineering (re-implementing a legacy application)
6. Package modification (revising purchased software)

PROJECT SCOPE: _____

1. Algorithm
2. Subroutine
3. Module
4. Reusable module
5. Disposable prototype
6. Evolutionary prototype
7. Subprogram
8. Stand-alone program
9. Component of a system
10. Release of a system (other than the initial release)
11. New departmental system (initial release)
12. New corporate system (initial release)
13. New enterprise system (initial release)
14. New national system (initial release)
15. New global system (initial release)

PROJECT CLASS: _____

1. Personal program, for private use
2. Personal program, to be used by others
3. Academic program, developed in an academic environment
4. Internal program, for use at a single location
5. Internal program, for use at multiple locations
6. Internal program, for use on an intranet
7. Internal program, developed by external contractor
8. Internal program, with functions used via time sharing
9. Internal program, using military specifications
10. External program, to be put in public domain

11. External program, to be placed on the Internet
12. External program, leased to users
13. External program, bundled with hardware
14. External program, unbundled and marketed commercially
15. External program, developed under commercial contract
16. External program, developed under government contract
17. External program, developed under military contract

PROJECT TYPE: _____

1. Nonprocedural (generated, query, spreadsheet)
2. Batch application
3. Web application
4. Interactive application
5. Interactive GUI applications program
6. Batch database applications program
7. Interactive database applications program
8. Client/server applications program
9. Computer game
10. Scientific or mathematical program
11. Expert system
12. Systems or support program, including “middleware”
13. Service-oriented architecture (SOA)
14. Communications or telecommunications program
15. Process-control program
16. Trusted system
17. Embedded or real-time program
18. Graphics, animation, or image-processing program
19. Multimedia program
20. Robotics, or mechanical automation program
21. Artificial intelligence program
22. Neural net program
23. Hybrid project (multiple types)

PROBLEM COMPLEXITY: _____

1. No calculations or only simple algorithms
2. Majority of simple algorithms and simple calculations
3. Majority of simple algorithms plus a few of average complexity
4. Algorithms and calculations of both simple and average complexity
5. Algorithms and calculations of average complexity
6. A few difficult algorithms mixed with average and simple
7. More difficult algorithms than average or simple
8. A large majority of difficult and complex algorithms
9. Difficult algorithms and some that are extremely complex
10. All algorithms and calculations extremely complex

CODE COMPLEXITY: _____

1. Most “programming” done with buttons or pull-down controls
2. Simple nonprocedural code (generated, database, spreadsheet)
3. Simple plus average nonprocedural code
4. Built with program skeletons and reusable modules
5. Average structure with small modules and simple paths
6. Well structured, but some complex paths or modules
7. Some complex modules, paths, and links between segments
8. Above average complexity, paths, and links between segments
9. Majority of paths and modules are large and complex
10. Extremely complex structure with difficult links and large modules

DATA COMPLEXITY: _____

1. No permanent data or files required by application
2. Only one simple file required, with few data interactions
3. One or two files, simple data, and little complexity
4. Several data elements, but simple data relationships
5. Multiple files and data interactions of normal complexity
6. Multiple files with some complex data elements and interactions
7. Multiple files, complex data elements and data interactions

- 8. Multiple files, majority of complex data elements and interactions
- 9. Multiple files, complex data elements, many data interactions
- 10. Numerous complex files, data elements, and complex interactions

As most commonly used for either measurement or sizing, users will provide a series of integer values to the factors of the taxonomy, as follows:

PROJECT NATURE	1
PROJECT SCOPE	8
PROJECT CLASS	11
PROJECT TYPE	15
PROBLEM COMPLEXITY	5
DATA COMPLEXITY	6
CODE COMPLEXITY	2

Although integer values are used for nature, scope, class, and type, up to two decimal places can be used for the three complexity factors. Thus, permissible values might also be

PROJECT NATURE	1
PROJECT SCOPE	8
PROJECT CLASS	11
PROJECT TYPE	15
PROBLEM COMPLEXITY	5.25
DATA COMPLEXITY	6.50
CODE COMPLEXITY	2.45

The combination of numeric responses to the taxonomy provides a unique “pattern” that facilitates sizing, estimating, measurement, benchmarks, and statistical analysis of features and requirements. The taxonomy makes it easy to predict the outcome of a future project by examining the results of older projects that have identical or similar patterns using the taxonomy. As it happens, applications with identical patterns are usually of the same size in terms of function points (but not source code) and often have similar results.

Not only are applications that share common patterns close to the same size, but they also tend to have very similar feature sets and to have implemented very similar requirements. Therefore, placing an application on a taxonomy such as the one described here could be a step toward creating families of reusable requirements that can serve dozens or even hundreds of applications. The same taxonomy can assist in assembling the feature sets for systems using the service-oriented architecture (SOA).

When demographic information is included, all the factors in the taxonomy are as follows:

COUNTRY CODE	1 (United States)
REGION CODE	06 (California)
CITY CODE	408 (San Jose)
INDUSTRY CODE	1569 (Telecommunications)
CMMI LEVEL	3 (Controlled and repeatable)
STARTING DATE	04/20/2009
PLAN COMPLETION DATE	05/10/2011
TRUE COMPLETION DATE	09/25/2011
SCHEDULE SLIP	4.25 (Calendar months)
INITIAL SIZE	1000 (Function points)
REUSED SIZE	200 (Function points)
UNPLANNED GROWTH	300 (Function points)
DELIVERED SIZE	1500 (Function points)
INITIAL SIZE (SOURCE CODE)	52,000 (Logical statements)
REUSED SIZE	10,400 (Logical statements)
UNPLANNED GROWTH	15,600 (Logical statements)
DELIVERED SIZE (SOURCE CODE)	62,400 (Logical statements)
PROGRAMMING LANGUAGE(S)	65 (Java)
REUSED CODE	65 (Java)
PROJECT NATURE	1 (New application)
PROJECT SCOPE	8 (Stand-alone application)
PROJECT CLASS	11 (Expert system)
PROJECT TYPE	15 (External, unbundled)
PROBLEM COMPLEXITY	5.25 (Mixed, but high complexity)
DATA COMPLEXITY	6.50 (Mixed, but high complexity)
CODE COMPLEXITY	2.45 (Low complexity)

The taxonomy provides an unambiguous pattern that can be used both for classifying historical data and for sizing and estimating software projects. This is because software applications that share the same pattern also tend to be of the same size when measured using IFPUG function point metrics.

When applications that share the same pattern have differences in productivity or quality, that indicates differences in the effectiveness of methods or differences in the abilities of the development team. In any case, the taxonomy makes statistical analysis more reliable because it prevents “apples to oranges” comparisons.

Software applications will not be of the same size using lines of code (LOC) metrics due to the fact that there are more than 700 programming languages in existence. Also, a majority of software applications are coded in more than one programming language.

Software applications of the same size may vary widely in costs and schedules for development due to the varying skills of the development teams, the programming languages used, the development tools and methods utilized, and also the industry and geographic location of the developing organization. Although size is a required starting point for estimating software applications, it is not the only information needed.

The taxonomy can be used well before an application has started its requirements. Since the taxonomy contains information that should be among the very first topics known about a future application, it is possible to use the taxonomy months before requirements are finished and even some time before they begin.

It is also possible to use the taxonomy on legacy applications that have been in existence for many years. It is often useful to know the function point totals of such applications, but normal counting of function points may not be feasible since the requirements and specifications are seldom updated and may not be available.

The taxonomy can also be used with commercial software, and indeed with any form of software, including classified military applications where there is sufficient public or private knowledge of the application to assign values to the taxonomy tables.

In theory, the taxonomy could be extended to include other interesting topics such as development methods, programming languages, tools, defect removal, and many others. However, two problems make this extension difficult:

- 1. New languages, tools, and methods occur every month, so there is no stability.
- 2. A majority of applications use multiple languages, methods, and tools.

However, to show what an extended taxonomy might look like, following is an example of the basic taxonomy extended to include development methods:

COUNTRY CODE	1 (United States)
REGION CODE	06 (California)
CITY CODE	408 (San Jose)
INDUSTRY CODE	1569 (Telecommunications)
CMMI LEVEL	3 (Controlled and repeatable)
STARTING DATE	04/20/2009
PLAN COMPLETION DATE	05/10/2011
TRUE COMPLETION DATE	09/25/2011
SCHEDULE SLIP	4.25 (Calendar months)

INITIAL SIZE	1000 (Function points)
REUSED SIZE	200 (Function points)
UNPLANNED GROWTH	300 (Function points)
DELIVERED SIZE	1500 (Function points)
INITIAL SIZE (SOURCE CODE)	52,000 (Logical statements)
REUSED SIZE	10,400 (Logical statements)
UNPLANNED GROWTH	15,600 (Logical statements)
DELIVERED SIZE (SOURCE CODE)	62,400 (Logical statements)
PROGRAMMING LANGUAGE(S)	65 (Java)
REUSED CODE	65 (Java)
PROJECT NATURE	1 (New application)
PROJECT SCOPE	8 (Stand-alone application)
PROJECT CLASS	11 (Expert system)
PROJECT TYPE	15 (External; unbundled)
PROBLEM COMPLEXITY	5.25 (Mixed but high complexity)
DATA COMPLEXITY	6.50 (Mixed but high complexity)
CODE COMPLEXITY	2.45 (Low complexity)
SIZING METHOD	1 (IFPUG function points)
ESTIMATING METHODS	3 (KnowledgePlan)
MANAGEMENT REPORTING	2 (Automated insight)
RISK ANALYSIS	0 (Not used)
FINANCIAL VALUE ANALYSIS	1 (Used)
INTANGIBLE VALUE ANALYSIS	0 (Not used)
REQUIREMENTS GATHERING	1 (Joint application design)
REQUIREMENTS LANGUAGE(S)	5 (Hybrid: Use-cases, English)
QUALITY REQUIREMENTS	1 (QFD)
SOFTWARE QUALITY ASSURANCE	1 (Formal SQA involvement)
DEVELOPMENT METHOD	3 (Team Software Process)
PRETEST DEFECT REMOVAL	
REQUIREMENTS INSPECTION	1 (Used)
DESIGN INSPECTION	1 (Used)
CODE INSPECTION	0 (Not used)
STATIC ANALYSIS	1 (Used)
SIX SIGMA	0 (Not used)
IV & V	0 (Not used)
AUTOMATED TESTING	0 (Not used)
TEST STAGES	
UNIT TEST	1 (Used)
NEW FUNCTION TEST	1 (Used)
REGRESSION TEST	1 (Used)
COMPONENT TEST	1 (Used)
PERFORMANCE TEST	1 (Used)

(Continued)

SECURITY TEST	0 (Not used)
INDEPENDENT TEST	0 (Not used)
SYSTEM TEST	1 (Used)
ACCEPTANCE TEST	1 (Used)

Although the basic taxonomy has been in continuous use since 1984, the extended taxonomy that shows tools, languages, and methods is hypothetical. It is included because such an extended taxonomy would facilitate estimates, benchmark analysis, statistical studies, and multiple regression analysis to show the effectiveness of various methods and practices.

By converting millions of alternatives into numeric data by means of multiple-choice questions, taxonomies facilitate statistical analysis. Also, various “patterns” among the alternatives can easily be evaluated in terms of improving or degrading productivity and quality, or exploring reusable requirements. The software industry should invest more energy into development of useful taxonomies along the lines used by other sciences such as biology, linguistics, physics, and chemistry.

**Software Requirements Methods
and Practices**

There are numerous variations in how software requirements are collected, analyzed, and converted into software. Following are descriptions and some results noted for a number of common variations. They are discussed in alphabetical order.

Agile requirements with embedded users An interesting idea that has emerged from the Agile methods is that of a full-time user representative as part of the development team. The role of these embedded users is to provide the requirements for new applications in fairly small doses that can immediately be implemented and put to use. Typically, segments between 5 percent and 10 percent of the total requirements are defined and built during each “sprint.” This is equivalent to 40 to 200 function points per sprint.

This method of full-time users has proven to be effective for small applications where one person can actually express the needs of all users. It is not effective for applications such as Microsoft Office with millions of users, because no one can speak for the needs of all users. Neither is this method effective for certain kinds of embedded applications such as fuel-injection controls.

Including users with development teams is an innovative approach that works well once the limits are understood. See also “Focus Groups,” “Data Mining for Legacy Requirements,” and “Joint Application Design (JAD).”

Creeping requirements Changes taking place in requirements after a formal requirements phase is a normal occurrence. Surprisingly, many applications are not effective in dealing with requirements changes. Creeping requirements are calculated by measuring the function point total for an application at the end of the requirements phase, and then doing another function point count when the application is delivered, including all requirements that surfaced after the requirements phase. This form of measurement indicates creeping requirements grow at about 2 percent per calendar month during the subsequent design phase and perhaps 1 percent per calendar month during much of the coding phase. After the midpoint of the coding phase, requirements changes are redirected into future releases.

Typical growth patterns for a “normal” application of 1500 function points would be in the range of 30 function points of creeping requirements per month during design and 15 function points of growth per month during coding. Since design should last two months and coding eight months, total growth in terms of creeping requirements would be 60 function points during design and 120 function points during coding, or 180 function points in all. Thus, the application with 1500 function points defined at the end of the requirements phase would be delivered as an application of 1680 function points.

Note that larger applications with longer schedules obviously have much larger totals of requirements creep.

Considering the same application in an Agile context, each sprint might include 150 to 250 function points. The total size at delivery would still be about 1680 function points, but the application is developed in stages.

The most effective way to deal with requirements creep is to use methods that reduce unplanned creep and also to use methods that validate changes. Joint Application Design (JAD), executable English, and prototypes slow down creep. Requirements inspections and change control boards can validate changes. The Agile method of embedding users with developers increases creep up to 10 percent per month, but this is benign because the Agile teams are geared up for such growth.

There are several problems associated with creeping requirements outside of the Agile domain: (1) they have higher defect potentials than original requirements; (2) they cause schedule delays and cost overruns; (3) they are frequent causes of litigation for applications developed under contract or for outsourced applications.

Data mining for legacy requirements As of 2009, more than half of “new” applications are replacements for aging legacy software applications. Some of these legacy applications may have been in continuous use for more than 25 years. Unfortunately, the software industry is lax in keeping

requirements and design documents up to date, so for a majority of legacy applications, there is no easy way to find out what requirements need to be transferred to the new replacement.

However, some automated tools can examine the source code of legacy applications and extract latent requirements embedded in the code. These hidden requirements can be assembled for use in the replacement application. They can also be used to calculate the size of the legacy application in terms of function points, and thereby can assist in estimating the new replacement application. Latent requirements can also be extracted manually using formal code inspections, but this is much slower than automated data mining.

Executable English Since many business rules can be expressed in terms of English (or other natural languages), it makes sense to attempt to automate a formal dialect of English that facilitates requirements analysis. This is not a new idea, since COBOL was intended to have similar capabilities. An organization called Internet Business Logic, headed by Dr. Adrian Walker, has such a dialect available and automation to support it. Examples and downloads are available to try out the method. The information on executable English occurs in several web sites, but the Microsoft Development Network is perhaps the best known. The URL is <http://msdn.microsoft.com/en-us/library/cc169602.aspx>.

However, additional study and data would be useful. Some unanswered questions exist about using executable English for “toxic” requirements such as the Y2K problem. There are no intrinsic barriers to expressing harmful requirements in executable English. Also, there are no side-by-side comparisons in terms of requirements costs, requirements defects, or requirement productivity rates between executable English and other methods. Finally, hybrid approaches to use a combination of executable English with other methods have not yet been fully examined.

In theory, it would be possible to run static analysis tools against requirements specifications written in executable English, assuming that the static analysis tools had parsers available. If so, finding logical problems and omissions in executable English might add value to static analysis tools such as Coverity, KlocWorks, XTRAN, and the like.

Automatic error detection in requirements and design created from executable English would help to eliminate serious classes of error that have long been difficult to deal with: incomplete and toxic requirements. A future merger of static analysis and executable English holds many interesting prospects for improving the quality of requirements analysis.

Focus groups A *focus group* is an assembly of customers who are asked to participate in group discussions about the features and functions of

new products. Focus groups usually range from perhaps 5 to more than 25 participants based on the demographic needs of the potential product. Focus groups may offer suggestions or even use working models and prototypes.

Focus groups have proven to be effective for products that are aimed at a mixture of diverse interests and many possible kinds of use. Focus groups are older than software and are frequently used for electronic devices, appliances, and other manufactured objects.

In a software context, focus groups are most effective for commercial software applications aimed at hundreds or thousands of users, where diversity is part of the application goals.

Functional and nonfunctional requirements Software requirements come in two flavors: functional requirements and nonfunctional requirements. The term *functional requirement* is defined as a specific feature that a user wants to have included in a software application. Functional requirements add bulk to software applications, and in general every functional requirement can be measured in terms of function point metrics.

Nonfunctional requirements are defined as constraints or limits users care about with software applications, such as performance or reliability. Nonfunctional requirements may require work to achieve, but usually don't add size to the application.

Joint application design (JAD) The concept of joint application design originated in IBM Toronto as a method for gathering the requirements for financial applications. The normal method of carrying out JAD is for a group of stakeholders or users to meet face-to-face with a group of software architects and designers in a formal setting with a moderator. The JAD sessions use standard requirement checklists to ensure that all relevant topics are covered. Often JAD meetings take place in off-site facilities. Between three and ten users meet with a group of between three and ten software architects and designers in a typical JAD event. The meetings usually run from 2 days to more than 15 days, based on the size of the application under discussion.

JAD sessions have more than 35 years of empirical data and rank as one of the most effective methods for gathering requirements for large applications. Use of JAD can lower creeping requirements levels down to perhaps one-half percent per month.

Pattern matching As noted previously in the section of this chapter dealing with taxonomies, many applications are quite similar in terms of functional requirements. For example, consultants who work with many companies within industries such as finance, insurance, health care, and manufacturing quickly realize that every company within specific

industries has the same kinds of software applications. Indeed, the similarity of applications within industries is what caused the creation of the enterprise resource planning (ERP) tools such as those marketed by SAP, Oracle, BAAN, and others.

However, as of 2009, the software industry lacks effective methods for identifying and reusing specific functional requirements between applications. To identify patterns and similarities, it would be desirable to have all functions expressed in standard fashions, and also to have a full taxonomy of major software features.

It would be possible for various kinds of static analysis tools to identify common patterns among multiple applications, and this would facilitate reuse of common features and functions. But so long as requirements are expressed using more than 30 flavors of graphical representation coupled with free-style English, automated pattern matching is difficult or impossible.

Prototypes By definition, a software *prototype* is a partial model of a possible software application, but stripped down to a few key functions and algorithms. As a general rule, prototypes are about 10 percent of the size of completed applications. The reason for the small size of prototypes is that they are intended to be developed quickly. For example, a 10 percent prototype of a 10,000–function point application would amount to 1000 function points, which is fairly difficult to develop quickly.

The optimal size of applications where prototypes give the best results is around 1000 function points. A 10 percent prototype would be only 100 function points, which can be developed quickly.

Prototypes come in two flavors, *disposable* and *evolutionary*. As the name implies, a disposable prototype can be discarded once it has served its purpose. On the other hand, an evolutionary prototype will add more features and gradually evolve into a finished product.

Of the two flavors, disposable prototypes are safer. The shortcuts and poor quality control associated with evolutionary prototypes may lead to downstream security flaws, quality problems, and performance problems with evolutionary prototypes.

Prototypes of both flavors are very successful in reducing creeping requirements. As a rule of thumb, requirements creep for applications that use prototypes is less than one-half percent per calendar month, or less than half the creep of similar applications without prototypes.

Quality function deployment (QFD) Like many effective quality control approaches, QFD originated in Japan. QFD was apparently first used circa 1972 by Mitsubishi for the quality requirements of a large ocean-going tanker. QFD is sometimes called “house of quality” because the QFD diagrams resemble a house with a peaked roof.

Although QFD originated for manufactured products, it has been used with software. Primarily QFD is used for embedded and systems software, such as aircraft and medical instruments. It is also used by computer companies such as Hewlett-Packard and IBM for both software and hardware products.

There are a number of books and reports on QFD. Since learning to use QFD and successfully deploying takes more than a week, additional information is needed before starting a QFD program. A nonprofit QFD institute exists and is one source of additional data. As with the Six Sigma approach, QFD borrows some topics from martial arts and uses a “belt” system to indicate training levels. As with Six Sigma and many martial arts, a black belt is the highest level of achievement. (Of course, true martial arts practitioners object to this approach on the grounds that earning a black belt in a martial art takes years of training and practice. Earning a black belt in Six Sigma or QFD takes only a few months of training and requires very little in the way of hands-on experience.)

Requirements engineering The topic of requirements engineering is a fairly new subset of software engineering. Requirements engineering attempts to add rigor to requirements gathering and analysis by using formal methods of elicitation, analysis, and also by creating models of the application and validating the requirements. That being said, requirements engineering is still evolving and is not yet a fully formed discipline.

Requirements engineering is most likely to be used for systems and embedded software that operates fairly complex physical devices. The reason is that systems and embedded software needs much more rigor and better quality to operate successfully than any other kinds of software.

While empirical data on requirements engineering is sparse in 2009, anecdotal evidence suggests that applications using requirements engineering methods tend to have somewhat lower levels of requirements defects and somewhat higher levels of requirements defect removal efficiency than similar applications with more casual requirements methods. However, organizations using requirements engineering also tend to be at or above level 3 on the CMMI, which by itself could explain the improvements.

Requirements engineering is synergistic with formal methods such as the Rational Unified Process (RUP) and the UML approach. It is also synergistic with the Team Software Process (TSP). Requirements engineering is not normally used with Agile projects because the rigor is antithetical to the Agile approach. It would not be easy to perform formal requirements engineering analysis on short user stories.

Requirement inspections Formal inspections of software deliverables such as requirements originated within IBM in the early 1970s. Inspections are approaching 40 years of continued usage and remain one of the most effective defect removal methods with the highest levels of defect removal efficiency. Formal inspections can top 85 percent in defect removal efficiency and seldom drop below 65 percent. By contrast, most forms of testing are below 35 percent in defect removal efficiency levels and seldom top 50 percent. Inspections are also good for defect prevention, since participants spontaneously avoid the same kinds of defects that the inspections find.

Inspections are team activities with well-defined roles for the moderator, the recorder, the inspectors, and the person whose work is being inspected. Substantial data and books exist on the topic of inspections. A new nonsoftware inspection organization was created in 2009, in place of the former Software Inspection and Review Organization (SIRO) group from the 1980s.

Requirements traceability Once a specific requirement is defined, it must be included in design documents and source code as well. Test cases must also be created to ensure that the requirement has been correctly implemented. Training materials and user reference materials will probably have to be created to explain how to use the requirement. “Requirements traceability” refers to methods that allow requirements to be backtracked from other deliverables such as code and test cases.

In theory, traceability in both forward and backward directions is possible if each explicit requirement is assigned a unique identifier or serial number. Once assigned, the same number is used in specifications, code, test cases, and other deliverables where the same requirement is used.

Traceability is often performed via a matrix where every requirement is listed on one axis, and every document or code segment that contains the requirement is listed on the other axis. The intersection of the two axes indicates that the requirement was either present or not.

In theory, traceability is straightforward, but in practice, requirements traceability is complex and difficult, although a number of automated tools exist that can ease the problems.

Traceability is most often used for defense applications, systems software, and embedded software, because these applications often have serious legal and liability issues associated with them. Traceability is also important for information technology applications in the wake of the Sarbanes-Oxley Act, which enforces penalties for poor governance of financial software constructed by Fortune 500 companies.

However, traceability is seldom used for web applications, entertaining software, applets for devices such as iPhone, and for software that is developed for internal use within a single company.

Much of the literature on requirements traceability deals with traceability problems, which are numerous and severe. In spite of more than 100 tools that assert that they can help in performing requirements traceability, effective traceability remains troublesome and imperfect.

If reusable requirements will be used in multiple applications, it is obvious that traceability will need to encompass cross-application traces as well as single-application traces. This implies a need for 3-D traceability matrixes.

Reusable requirements Many software applications perform very similar functions within an industry. For example, insurance claims processing is very similar from company to company. Order processing and invoicing are very similar within hundreds of companies and thousands of applications. Almost all applications need functions for error handling.

In theory, at least 60 percent to 75 percent of any business application could probably be created from standard reusable parts, assuming those parts are certified to high levels of reliability and are readily available. Unfortunately, what is lacking is an effective catalog of reusable materials that include reusable requirements, design, code, interfaces, and test cases. Obviously, common features also need to be traceable back to their original origins, in case of errors or recalls.

Some catalogs of reusable functions are within specific domains such as defense and avionics software, and these are samples of what is needed. However, there is no overall industrywide catalog available circa 2009.

As it happens, the taxonomy discussed earlier in this chapter could be extended downwards to describe individual or specific reusable requirements or features. This is because almost every function or feature provided by software applications needs to supply similar services and to perform similar actions. The topics that would compose a taxonomy of reusable functions would probably include

1. The *origin* of the function
2. The *creation date* of the function
3. The *version number* of the function
4. The *certification* level of the function
5. The *business purpose* of the function
6. The *name* of the feature
7. The *traceability serial number* of the function
8. The *programming language* of the function
9. The *links to the function's reusable test cases*

10. The *links to the function's reusable documentation*
11. The *links to related functions*
12. The *inputs* to the function
13. The *outputs* from the function
14. The *messages passed* by the function
15. The *messages received* by the function
16. The *entities and relationships* within the function
17. The *logical files* used by the function
18. The *inquiry types* that can be made of the function
19. The *interfaces* with other functions if other than messages
20. The *error-handling* methods of the function
21. The *security* methods of the function
22. The *algorithms* that the function performs

Reusable requirements would obviously extend requirements traceability into another dimension. Not only would requirements have to be traced backwards from the code in a specific application, but if many applications contain the same reusable function, then cross-application traceability would also be needed. This would necessitate using 3-D matrixes.

Security requirements deployment (SRD) As the global recession intensifies, attacks on software applications in the form of worms, viruses, spyware, keystroke loggers, and denial of service attacks are increasing daily. Most software engineers and most quality assurance personnel are not adequately trained in security control techniques to be fully effective. Most software application customers and users are almost helpless.

The idea of security requirements deployment (SRD), which is being introduced in this book, is to apply the same rigor to security requirements as quality function deployment (QFD) applies to quality requirements. However, there is an additional factor that must be addressed for SRD to be effective. It is necessary to bring in at least one top-gun security expert to meet with the development team and the user representatives during the SRD planning sessions.

The topics that are to be addressed during SRD planning sessions include conventional protection methods such as physical security and avoiding the most common security vulnerabilities. However, the urgency of the situation calls for more advanced methods that can actually improve the resistance of source code to outside attack. This implies

getting up to speed with capability logic, restricting permissions, and using languages such as E that create attack-resistant code. Adopting methods such as those used by the Google Caja approach. The word *Caja* is Spanish for “box” and refers to methods developed by Google for encapsulating JavaScript and HTML to prevent outside agents from attacking or modifying them.

In addition, SRD sessions should discuss security inspections, using static-analysis tools that are optimized to find security flaws, and introducing special security test stages. It may also be relevant to consider the employment of “ethical hackers” to attempt to penetrate or gain access to confidential information or seize control of software.

The seriousness of software security flaws in today’s world requires immediate and urgent solutions. A firewall combined with antivirus software and antispyware software is no longer sufficient to provide real protection. In the modern world, the attacks no longer come from malicious amateurs, but some come from well-funded and well-trained foreign governments and from very well-funded organized crime syndicates.

Unified modeling language (UML) The UML modeling language is an integral part of the Rational Unified Process (RUP) that is now owned by IBM. The history of the UML as a merger of the concepts of Grady Booch, James Rumbaugh, and Ivar Jacobsen is well known among the software community. The UML and its predecessors were originally aimed at supporting object-oriented requirements and design, but can actually support almost any form of software.

The UML is a rich and complex set of graphic notations that encompass not only requirements but also architecture, database design, and other software artifacts. In fact, UML 2.0 includes 13 different kinds of diagram. As a result of the richness of the UML constructs, there is a very lengthy learning curve associated with the UML.

As of 2009, scores of commercial tools can facilitate UML diagram construction and management. UML diagrams can easily be inspected using standard protocols for requirements and design inspections. However, it would also be useful to have some form of automated consistency and validity checking tools. What comes to mind would be a kind of superset of static analysis capabilities.

For reusable requirements and reusable features that are likely to be utilized by multiple applications, it would be useful to have some kind of a pattern-matching intelligent agent that could scour UML diagrams and extract similar patterns.

UML is not a panacea, but the Object Management Group (OMG) is continuously working to add useful features and eliminate troublesome elements. Therefore, UML is likely to expand in usefulness in the future.

UML diagrams are normal inputs to standard function point analysis. In theory, it is possible to develop a tool that would automatically create function point totals from parsing various UML diagrams. In fact, such experimental tools have been constructed.

The meta-language underneath UML is amenable to static analysis and other forms of automatic verification. Test suites might also be constructed from the UML meta-language. Finally, size in terms of function points might be calculated using the meta-language.

Use-cases The concept of use-cases originated with Ivar Jacobsen, who is also one of the pioneers working the UML. Although use-cases are associated with the UML, they are also popular as a stand-alone method of gathering requirements. Use-cases are aimed squarely at functional requirements and provide an interesting visual representation of how users invoke, modify, control, and eventually terminate actions by software applications. The application itself is treated as a black box, and use-cases concentrate on how users interact with it to accomplish business functions.

Use-cases have introduced some interesting abstractions into software requirements analysis, such as “actors” and “roles.” These focus attention on essential topics and tend to lead analysts and customers in fruitful directions.

A number of templates provide assistance in thinking through a sequence of user interactions with software. These templates usually include topics such as “goals,” “actors,” “preconditions,” and “triggers,” among others.

As with other features of the UML, many commercial tools are available for drawing and managing use-cases. Use-cases are also amenable to formal requirements and design inspections, and can be used to predict application size via function point analysis. In general, use-cases are among the easiest requirements artifacts for inspection, because the visual representation makes it easy to examine assumptions.

Use-cases are also used in the context of joint application design (JAD) and are sometimes created on-the-fly during JAD sessions.

User stories The Agile methods aim at creating running code as fast as possible, and the Agile community feels that the massive paper document sets associated with the UML and sometimes with use-cases are barriers to progress rather than effective solutions. As a result, the Agile community has developed a flexible and fast method of gathering requirements termed *user stories*. One unique feature of user stories is that they are closely coupled with test cases; in fact, the test cases and the user stories are developed concurrently.

To keep the user stories concise and in keeping with the Agile philosophy of minimizing paper documents, the stories are usually written on 3" × 5" cards rather than standard office paper. Many user stories are only a single sentence, or perhaps a few sentences. An example of such a short user story might be, "I want to withdraw cash from an ATM." However, this means that complicated transactions may take dozens of cards, with each card defining only a single step in the entire process.

While use-cases can be inputs to function point analysis, their conciseness and lack of detail is one of the reasons why function point analysis is not used very often for Agile applications. In fact, an alternative to user stories would be to base function point analysis on the associated test cases, which of necessity must be more complete.

It is a good thing that test cases and user stories are created concurrently, because formal inspections of user stories would not find many defects, since the stories are so abbreviated. However, inspections of the test cases created with the user stories are of potential value.

Another issue with user stories is their longevity. Once the initial release of an application goes to customers, development of the second and future releases may pass to other development teams or be outsourced. How do these follow-on groups know what requirements are in the first release? In other words, are user stories a practical way of transmitting knowledge about requirement over a 10- to 20-year period?

Some Agile organizations use a metric called *story points* for estimation. However, there are no large benchmark collections that use story points. In addition, it is not possible to compare projects whose requirements are derived from story points against similar projects that used other methods such as UML or use-cases.

It is theoretically possible to convert story points into function points, but a better method would be for Agile projects to use one of the high-speed function point sizing methods. Having function points available would allow side-by-side comparisons with other projects and would permit Agile projects to submit data to standard benchmark collections such as that of the International Software Benchmarking Standards Group (ISBSG).

Summary of Software Requirements Circa 2009

Even after 60 years of software development, methods for gathering and analyzing user requirements continue to be troublesome. Creeping requirements still occur, as do requirements errors and also toxic requirements. Requirement inspections are an effective antidote to these problems, but occur for less than 5 percent of U.S. software projects and even fewer on a global basis.

Research into an extended taxonomy for specific features and specific requirements would be valuable to the industry because such a taxonomy would allow similar requirements to be compared and evaluated from multiple applications. This is because applications that share the same “pattern” on the taxonomy usually have similar features and similar requirements.

Also valuable would be elevating the methods of static analysis so that they operated on requirements. Additional research on data mining to extract hidden requirements from source code would be an adjunct to using static analysis on requirements, as would automatic derivation of function point totals.

The eventual goal of requirements engineering should be to create catalogs of standard reusable requirements and associated test and tutorial materials. In theory, more than 50 percent and perhaps more than 75 percent of the features in software applications could eventually come from certified reusable materials.

Business Analysis

The phrase “business analysis” is very similar to the older phrase “systems analysis.” Many corporations employ business analysis specialists who serve as a liaison between the software engineering community and the operating units of the company.

Because of their role as liaison between the technical and business communities, business analysts are involved very early and are key participants even before requirements elicitation starts.

Business analysts continue to be involved during the design and early part of the coding phases, due to having to analyze and deal with creeping requirements that do not taper off until well into the coding phase. After that, additional requirements are shunted into future releases.

The roles of the business analysts are to aid in requirements elicitation, and to ensure that both the information technology side and the customer or stakeholder side communicate clearly and effectively.

The background and training for business analysis specialists is somewhat ambiguous as of 2009. Many are former systems analysts, software engineers, or quality assurance specialists who wanted broader responsibilities.

There is a nonprofit International Institute of Business Analysis (IIBA) that maintains a Business Analysis Body of Knowledge (BABOK) library with substantial volumes of information.

Because business analysts have backgrounds in both software and business topics, they are in a good position to facilitate requirements elicitation and requirements analysis. For example, business analysts are often moderators at joint application design (JAD) sessions.

Business analysts can also participate in requirement inspections, quality function deployment (QFD), and other activities that either collect requirements or analyze them and explain their meaning to the software community.

Some visible gaps in the roles of business analysts often require other kinds of specialists. To illustrate a few of these gaps:

1. Sizing and estimating software projects
2. Scope management of software projects
3. Risk analysis of software projects
4. Tracking and monitoring the progress of software projects
5. Quality control of software projects
6. Security analysis and protection of software projects

The reason for the assertion that these areas represent “gaps” is because problems are very common in all six areas regardless of whether business analysis is part of the requirements process.

Business analysts should know a great deal about corporate and enterprise software issues. In fact, the roles of business analysts and the roles of enterprise architects, to be discussed later in this chapter, overlap.

In the future it would be useful to have a full and complete description of the roles played by business analysts, architects, enterprise architects, scope managers, and project office managers, because they all have some common responsibilities.

One useful service that business analysts could provide for their employers is to collect and summarize benchmark data from a variety of sources. In fact, in 30 kinds of software benchmarks, early knowledge during the requirements phase would be useful. The 30 forms of benchmark include

1. Portfolio benchmarks
2. Industry benchmarks (banks, insurance, defense, etc.)
3. International benchmarks (U.S., UK, Japan, China, etc.)
4. Application class benchmarks (embedded, systems, IT, etc.)
5. Application size benchmarks (1, 10, 100, 1000, function points, etc.)
6. Requirements creep benchmarks (monthly rates of change)
7. Data center and operations benchmarks (availability, MTTF, etc.)
8. Data quality benchmarks
9. Database volume benchmarks
10. Staffing and specialization benchmarks

11. Staff turnover and attrition benchmarks
12. Staff compensation benchmarks
13. Organization structure benchmarks (matrix, small team, Agile, etc.)
14. Development productivity benchmarks
15. Software quality benchmarks
16. Software security benchmarks (cost of prevention, recovery, etc.)
17. Maintenance and support benchmarks
18. Legacy renovation benchmarks
19. Total cost of ownership (TCO) benchmarks
20. Cost of quality (COQ) benchmarks
21. Customer satisfaction benchmarks
22. Methodology benchmarks (Agile, RUP, TSP, etc.)
23. Tool usage benchmarks (project management, static analysis, etc.)
24. Reusability benchmarks (volumes of various reusable deliverables)
25. Software usage benchmarks (by occupation, by function)
26. Outsource benchmarks
27. Schedule slip benchmarks
28. Cost overrun benchmarks
29. Project failure benchmarks (from litigation records)
30. Litigation cost benchmarks

Business analysts are not the only personnel who should be familiar with such benchmark data, but due to their central and important role early in application development, business analysts are in a key position so the more they know, the more valuable their work becomes.

The assignment scope of business analysts runs between 1500 and 50,000 function points. That means that an approximate ratio of business analysts to ordinary software engineers would range from about 1 to 10 up to perhaps 1 to 25. The ratio of business analysts to customers runs from about 1 to 10 up to perhaps 1 to 50.

Software Architecture

In essence, software architecture is concerned with seven topics:

1. The overall structure of a software application
2. The structure of the data used by the software application
3. The interfaces between a software application and the world outside

4. The decomposition of the application into functional components
5. The linkage or transmission of information among the functional components
6. The performance attributes associated with the structure
7. The security attributes associated with the structure

There are other associated topics, but these seven seem to be the fundamental topics of concern.

The roles of both software architects and enterprise architects have been evolving in recent years and will continue to evolve as new topics such as cloud computing, service-oriented architecture (SOA), and virtualization become more widespread.

The importance of software architecture resembles the importance of the architecture of houses and buildings: the larger the structure, the more important good architecture becomes.

By coincidence, the size of a physical building measured in terms of “square feet” and the size of a software application measured in terms of “function points” share identical patterns when it comes to the importance or value of good architecture. Table 7-7 illustrates how architecture goes up in value with physical size.

Using the information shown in Table 7-7, a small iPhone applet with a size of perhaps 5 function points, or 250 Java statements can be successfully implemented without any formal architecture at all, other than the developer’s private knowledge of the value of structured code.

However, a very large system in the size range of Vista, Oracle, SAP, and the like will probably not even be possible without very good architecture and a number of architectural specialists. These massive applications top 100,000 function points, or more than 5 million statements in a language such as Java (probably more than 15 million in actuality).

Both software architecture and the architecture of buildings are concerned largely with structural issues. However, software architecture is even more complicated than building architecture because software

TABLE 7-7 Value of Architecture Increases with Structural Size

Size in Square Feet or Size in Function Points	Importance of Architecture
1	Not possible and not needed
10	Not needed
100	Minimal need for architecture
1,000	Architecture useful
10,000	Architecture important
100,000	Architecture critical

applications are not static once they are constructed. They grow continuously at about 8 percent per year as new features are added. This is much faster than buildings grow once complete. Also, software applications have no value unless they are operating. When they operate, software applications have a very dynamic structure that can change rapidly due to calls and features that open up and are modified during execution. Therefore, software architects have to deal with dynamic and performance-related issues that building architects only encounter occasionally for structures such as drawbridges and transit systems.

Another significant difference between building architecture and software architecture is in the area of security. Of course, for some buildings such as the Pentagon and CIA headquarters, security is a top architectural concern, but security is not usually a major architectural issue for ordinary homes and small office buildings.

For software, applications security is becoming increasingly important for all size levels. As the recession continues, security will become even more important because threats are becoming much more sophisticated. The recent success of the conflicker worm, which affected more than 1.9 million computers, including some in “secure” government agencies in early 2009, provides an urgent wakeup call to the increasing importance of security as an architectural issue for software.

As software engineering gradually evolves from a craft to an engineering field, the importance of architecture will continue to grow. One reason for this is because software architectural styles are rapidly evolving.

Returning to the analogy of the architecture of buildings, various chronological periods are sometimes characterized by the dominant form of architecture employed. There are also regional differences. Thus, many histories of architecture in the United States include discussions of the “Queen Anne” style, the “General Grant Gothic” style, the “Southern Antebellum” style, the “English Tudor” style, the “Frank Lloyd Wright” style, and dozens more.

Software engineering is not yet old enough to have formal histories of the evolution of architectural styles, but they are changing at least as rapidly as the architecture of homes and buildings.

One useful but missing piece of information from software benchmarks would be a description or taxonomy of the architecture that was used for applications. This would facilitate analysis of topics such as quality levels and security vulnerabilities associated with various software architectures.

When applications were small and averaged less than 1000 function points in size, as they did in until the late 1960s, software architecture was not a topic of interest. Edsger Dijkstra and David Parnas first discussed software architecture as a topic of importance circa 1968. Later pioneers such as Mary Shaw and David Garlan continued to stress

that software architecture was a critical factor for the success of large systems.

The reason for the increasing importance of architecture was because of four factors:

1. Software applications were growing rapidly and exceeding 10,000 function points or 1 million source code statements. Today in 2009, sizes may be greater than ten times larger yet again.
2. The volume of data used by software applications has been growing even faster than software itself. The number of automated records increased from thousands to millions to billions and continues to increase. No doubt trillions of records are just over the horizon.
3. Database and data organization schemas have been evolving as fast or faster than software architectural schemas.
4. Software applications were no longer operating all by themselves on one computer.

When large software applications began to be divided into components that could operate in parallel, or operate on separate computers at the same time, architecture became a very important topic.

Software applications that ran alone on a single computer were considered to have a “monolithic” architecture. One of the significant departures from this model was to have some of the functions executing on a host computer (often a mainframe) while other functions operated on personal computers. This method of decomposition was called *client-server* architecture.

In the 1980s and even more in the 1990s, many other architectural approaches emerged. They included but were not limited to event-driven architecture, three-tier architecture (presentation layer, business logic layer, and database layer), N-tier architecture with even more layers, peer-to-peer architecture, model-driven architecture, and of course the more recent pattern-based architecture, service-oriented architecture (SOA), soon followed by cloud computing.

At the same time that software architectures were expanding and evolving, data structures and data volumes were expanding and evolving. Hierarchical data structures were joined by relational data structures and also row-oriented data, column-oriented data, object-oriented data, and a number of others.

Obviously, software architects need to consider the join between the structure of software itself and optimal data organizations to accomplish the purpose of the application. These are not trivial choices, and both experience and special knowledge are required.

Successfully choosing and designing applications using any of these more recent forms of software and data architecture became a job that

required special training and considerable experience. As a result, many large companies such as IBM and Microsoft created new job descriptions and new job titles such as “architect” and “senior architect.”

As the positions of architect began to appear in large companies, several associations emerged so that architects could share information and gain access to the latest thinking. One of these is the International Association of Software Architects (IASA), and another is the World Wide Institute of Software Architects (WWISA). There are also specialized journals such as the *Microsoft Architecture Journal* dealing with architectural topics.

As of 2009, the weight of evidence supports the hypothesis that large companies that build large software applications should employ professional software architects. That can be considered a best practice.

An interesting question is how many architects does a company need? The normal assignment scope for an architect ranges between 5000 and about 100,000 function points. This means that an application of 10,000 function points will need at least one architect. However, a massive application of 150,000 function points might need at least two architects. Total employment of architects even in large companies such as IBM or Microsoft is probably less than 100 architects out of perhaps 50,000 total software engineers.

However, the evolution of specific architectural styles is far too rapid, and the criteria for evaluating architectures is far too hazy to state that using a specific form of architecture for a specific application is a good choice, a questionable choice, or a potentially disastrous choice.

It should be recalled that hundreds of companies jumped onto the client-server bandwagon in the 1980s, only to discover that complexity levels were so high and implementation so difficult that quality and reliability sometime dropped to unusable levels.

As of 2009, *service-oriented* architecture is attracting a huge amount of coverage in the literature and many early converts. But will SOA prove to be a truly successful architectural advance, or only a quantum leap in complexity without too many benefits? Unfortunately, there are not yet enough completed SOA applications to be sure that this theoretically useful architecture will live up to the promises that are being made on its behalf. (Recall that SOA applications are not downloaded into individual computers, but operate remotely from web hosts. This of course requires high bandwidths and transmission speed to be effective. No one has considered whether there is enough bandwidth available if there are thousands of SOA applications attempting to serve millions of clients at the same time.)

Another form of advanced architecture with huge claims is that of *cloud computing*. With this architecture, applications are segmented so that they can run concurrently on literally hundreds of remote computers.

This raises questions of safety and security given the rather poor security protocols that might be found in a cloud computing environment.

The bottom line for architecture as of 2009 is that it is evolving so rapidly that it is worthwhile to employ professional software architects who can stay current with the evolution of software architectural styles.

But selecting a specific architecture for a specific application is not a clear-cut choice with only one correct answer. The choice needs to be made by the architects assigned to the application, based on their knowledge of both architectural principles and also on their knowledge of the purpose and features of the application in question.

Enterprise Architecture

The need for enterprise architecture has grown progressively more important over the past 30 years, due in large part to the way computers and software became embedded in corporate operations.

In the late 1960s, when mainframe computers first began to be applied to business problems, their capabilities were somewhat primitive and limited. As a result, early business applications tended to be very local, to operate on a specific computer in a specific data center, and to serve only a limited number of users in a single business unit.

Corporations have multiple operating units, including manufacturing, marketing, sales, finance, human resources, and a number of others. Large corporations also have multiple business and manufacturing sites scattered through multiple cities and states.

When computers and software first became business tools, it was a common practice for each operating unit to have its own data center and to develop its own software. Often there was little or no communication between operating units as to the features, interfaces, or data that the applications were automating.

By the 1980s, large corporations had developed hundreds or even thousands of software applications, the majority of which served only narrow and local purposes. When corporate officers such as the CEO needed consolidated information from across all business units, time-consuming and expensive work was necessary to extract data from various applications and produce consolidated reports.

This awkward situation triggered the emergence of enterprise architecture as a key discipline to bring data processing consistency across multiple operating units. The same situation also triggered the emergence of an important commercial software market: enterprise resource planning (ERP). The basic concept of ERP applications is that individual applications are so hard to link together that it would be cheaper to replace all of them with a single large system that could serve all operating units

at the same time, and to store data in a consistent format that served corporate and unit needs simultaneously.

From about 2000 onward, numerous instances of corporate fraud and severe accounting errors such as demonstrated by Enron have added another dimension to enterprise architecture. Enterprise architects are also key players in *software governance*, or ensuring that financial data is accurate and that corporate officers take responsibility for its accuracy under threat of severe penalties.

The main difference between architecture as discussed in the previous section and enterprise architecture is the scope of responsibility. Normally, architects work on individual applications, which might range from 10,000 to more than 100,000 function points. Enterprise architects work on corporate portfolios, which may range from about 2 million function points to more than 20 million function points in aggregate size. Corporate portfolios for large companies such as Microsoft, IBM, or Lockheed contain thousands of applications.

Yet another aspect of enterprise architecture is the fact that large corporations create and use many different kinds of software: conventional information technology applications, web applications, embedded applications, and systems software. Some of these applications are built by in-house personnel; some are outsourced; some come from commercial vendors; some are open-source applications; and some come from mergers and acquisitions with other companies. In addition, any large corporation today in 2009 must also interface with the computer systems of other corporations and also with government agencies such as taxation and workers compensation.

The most difficult part of enterprise architecture is probably that of dealing with joining two software portfolios as a result of a merger or acquisition. Usually, at least 80 percent of the applications in both companies perform similar functions, but they may use different data structures, have different interface methods, and have different internal architectures.

Combining portfolios from two different companies in the wake of a merger is one of the most difficult tasks faced by enterprise architects, by architects, by business analysts, and by all other software engineering personnel.

Yet another set of concerns studied by enterprise architects are the communication methods among disparate business units and also the databases and repositories they develop and maintain.

In addition, enterprise architects are also concerned with a host of technology issues including but not limited to hardware platforms, software operating systems, open-source software, COTS packages from external vendors, and emerging topics such as cloud computing and service-oriented architecture that are not yet fully deployed.

Enterprise architecture has the same relationship to architecture that urban planning has to building architecture. With building architecture, an architect is concerned primarily with a single building. But urban planners need to be concerned about thousands of buildings at the same time. Urban planners need to think about what kinds of infrastructure will be needed to support various sectors such as residential, commercial, industrial, and so forth.

Table 7-8 shows the importance of enterprise architecture with increasing numbers of applications owned by the enterprise.

Table 7-8 brings up an interesting question: How many enterprise architects are needed in a large company? Because this is a fairly new occupation, there is no definitive answer. However, given the complexity of the situation, a corporation probably needs one enterprise architect for about every 1000 significant applications in their portfolio. Thus, if a company has 5000 applications in their portfolio, they may need five enterprise architects.

Expressed another way, the assignment scope of an enterprise architect runs from 500,000 up to more than 2 million function points.

For a large corporation such as IBM, Microsoft, or Unisys, a full portfolio might include

- 3,000 in-house information technology applications
- 1,500 web-based applications
- 1,000 tools (project management, testing, etc.)
- 3,500 commercial applications from other companies (ERP, HR, etc.)
- 2,000 commercial applications sold to other companies
- 2,500 systems-software applications
 - 500 embedded applications (security, AC, etc.)
 - 250 open-source applications
- 14,250 total applications

Assuming this total quantity of applications, then about 15 enterprise architects are likely to be employed.

TABLE 7-8 Value of Enterprise Architecture Increases with Applications

Number of Applications Owned by Enterprise	Importance of Enterprise Architecture
10	Enterprise architecture not needed
100	Enterprise architecture useful
1,000	Enterprise architecture important
10,000	Enterprise architecture very important
100,000	Enterprise architecture critical
1,000,000	Enterprise architecture critical but very difficult to achieve

These disparate applications will probably operate on more than a dozen hardware platforms and encompass at least half a dozen operating systems. In other words, the software world of a large corporation is a smorgasbord of diverse applications, platforms, data file structures, communication channels, and other problem areas.

As of 2009, the roles of enterprise architects are evolving under the impact of service-oriented architecture (SOA), cloud computing, the explosion of open-source applications, and also under the emerging criteria for more accurate financial reports mandated by Sarbanes-Oxley legislation.

The global recession will also have a significant but unpredictable impact on enterprise architecture. There are no models or guidelines for what happens to enterprise architecture during periods of massive layoffs, closures of business units, abandonment of unfinished applications, and reduced numbers of development and maintenance personnel. In fact, there is some risk that enterprise architects themselves may be among those who are laid off, because their work is not always perceived as having a direct impact on corporate bottom lines.

Several nonprofit associations support the enterprise architecture domain. One of these is the Association of Enterprise Architects (AEA), whose web site is aeaassociation.org.

Another is the awkwardly named Association of Open Group Enterprise Architects (AOGEA), whose web site is AOGEA.org. This organization and its awkward name are due to a merger between the Open Group organization and the Global Enterprise Architecture Organization (GEAO). The merged group asserts that it has become the largest association of architects in the world.

There is also a journal for enterprise architects, *The Journal of Enterprise Architecture* (JEA), published by the Association of Enterprise Architects.

It is difficult to find information about the specific plans of enterprise architects for corporations, because their work is usually proprietary and not made available to the public. However, many units of the federal government and most state governments do publish or make available information about their enterprise architectures. The Department of Defense is the world's largest user of computer software and is attempting to develop a new and improved enterprise architecture.

The huge increases in hacking, worms, viruses, and denial of service attacks are obviously topics of great concern to enterprise architects. However, security requires special skills, which are rare today, so external consultants on security are needed to buttress the work of enterprise architects until they can catch up.

In terms of best practices, organizations that own more than about 500 software applications should employ at least one enterprise architect.

Large corporations with more than 5000 software applications may need five, as noted before using ratios of applications to enterprise architects.

The roles played by enterprise architects in specific companies vary widely, and it is hard to pin down best practices. Obviously, increasing data sharing among operating units would be a best practice. Eliminating redundant applications and pruning portfolios of aging and unwieldy legacy applications would be another best practice. Other roles, which may or may not be viewed as best practices, might include changing the ratios of COTS applications to in-house software, and perhaps participating in the selection and deployment of enterprise resource planning (ERP) applications. No doubt the work of enterprise architecture will continue to evolve with technical and business changes.

Software Design

Suppose you were asked by the CEO of your company to examine the most recent 250 applications developed internally and to identify candidate features for creating a library of reusable designs, code, and test cases. How could this assignment be carried out?

This would not be an easy assignment given the state of the art of software design circa 2009. About 75 of the smaller applications below 1000 function points would probably have used Agile development and expressed their designs via user stories perhaps augmented by other representation methods. User stories are useful enough for individual applications, but not necessarily useful for identifying common patterns across multiple applications.

About 50 of the larger business applications above 5000 function points would have used more formal design methods; probably the UML with the requirements being elicited via joint application design (JAD). While the UML does capture individual patterns, the large volume of UML diagrams and their many flavors means that scanning through UML for a sample of 50 applications, trying to identify common features, would not be easy or rapid.

An automated tool such as a static analysis tool might parse the meta-language underlying UML and identify common patterns, but this is not readily done circa 2009.

About 25 of the scientific or engineering applications would have used state-change diagrams, modeling languages such as LePus3, Express, and probably quality function deployment (QFD) with “house of quality” diagrams and various architectural meta-language models.

The remaining 100 of the applications might have utilized a wide variety of methods including but not limited to use-cases, the UML, Nassi-Schneiderman charts, Jackson design, flowcharts, decision tables, data-flow diagrams, HIPO diagrams, and probably more as well. Some of

these define patterns, but they are not easy to scan for a sample of 100 projects.

In summary, the 250 most recent applications might have used more than 50 different design languages and methodologies, which, for the most part, are not easily translatable from one to another. Neither are they amenable to automatic verification and error-checking.

As a result of the large variety of fairly incompatible design representations used on the sample of 250 applications in the same company, there is no easy way to pick out features or patterns that are common among several applications using design documents. This makes it difficult to identify candidate features for a library of reusable materials.

Since all of the applications are complete and operating, it might be possible to identify the patterns by means of static-analysis tools on the source code itself, assuming that all of these applications are written in C, C++, Java, or any of the approximately 25 languages where static analysis operates.

Since some of the design methods have underlying meta-languages, static analysis is theoretically possible, but most static analysis tools support programming languages and not meta-languages as of 2009.

It would also be possible to look for patterns using one or more of the legacy renovation tools that parse source code and to display the code in a fashion that makes maintenance and modification easy.

Yet another possibility would be to use some of the more sophisticated complexity analysis tools that examine source code and to calculate cyclomatic and essential complexity and also to identify code patterns.

The bottom line is that as of 2009, it is easier to find and identify patterns in code than it is to identify patterns in design. This is not the way it should be. Design methods should be amenable to automated analysis in order to detect defects and also to look for patterns of reusable elements.

Another issue with software design is that software design errors are the second most numerous form of software error. Design errors average about 1.25 bugs or mistakes per function point, while code averages about 1.75 bugs per function point.

Since design documentation runs between one page and two pages per function point, the implication is that essentially every page of a design specification has at least one bug or error. This is why design inspections are so powerful and effective in reducing software design problems.

Given that the typical error density in software design remains high whether the representation method consists of use-cases, the UML, flowcharts, or any of the other 50 or so representation methods, there is insufficient data to select any current design methods as a best practice. What is more useful, perhaps, is to consider the fundamental topics that need to be part of software designs.

Software Design Views

When considered objectively, software design is a subset of the more general topic of *knowledge representation*. That brings up important questions as to what kinds of knowledge need to be represented when designing a software application. It also brings up questions as to what languages or forms of representation are best for the various topics that are part of software designs.

Because software is not readily visible and also has dynamic attributes, it is somewhat more difficult to enumerate the topics of software that need to be represented than it might be for a static physical object such as a building. Eight general topics are needed to represent software applications:

1. The *external view* of software features visible to users and derived from explicit user requirements. The external view includes screen images, report formats, and responses to user actions as might occur with embedded software. This view might identify features that are shared with other applications and hence potentially reusable. This view also should deal with error-handling for user errors. This view will also discuss the various hardware and software platforms on which the application will operate, and also the various countries and national languages that will be supported. This view is fairly concise and seems to average between 0.5 and 1.0 page per function point.
2. The *algorithm view* of the mathematical formulas or algorithms contained in the application. These might be straightforward calculations such as currency conversions or very complex formulas such as those associated with quantum mechanics. In any case, the major algorithms need to be represented and explained prior to encoding them. This view is very concise and averages below 0.25 page per function point.
3. The *structural view* of software applications includes components and modules and how they are joined together to form a complete application. This view includes the sequence or concurrency with which these modules will execute. Calls or interfaces to external applications are also part of the structural view. This view might also show modules or features that are reused from external sources or custom-built for a specific application. Classes and inheritance using object-oriented methods would also be shown in the structural view. This is the most verbose view and runs between 1.0 and 2.0 pages per function point.
4. The *data view* includes the kinds of information created, used, or manipulated by the software application. This view includes facts

about the data such as whether it consists of business information, symbols, sensor-based information, images, sounds, or something else. For example, the embedded software inside a cochlear implant converts external audio information into electrical signals. Because as of 2009, there is no “data point” metric or any other metric for expressing the size of databases, repositories, and data warehouses, there is no effective way to express the size or volume of data used by software.

5. The *attribute view* or nonfunctional goals and targets for the application once it is deployed. These attributes can include performance in terms of execution speed, reliability in terms of mean time to failure (MTTF), quality in terms of delivered defects, and a number of other attributes as well. This view is also very concise and usually requires less than three pages no matter how large the application itself is.
6. The *security view* or how the application will defend itself against viruses, worms, search bots, denial of service attacks, and other attempts to either interfere with the operation of the software or steal information used by the software. This view is new circa 2009, but quickly needs to become a standard feature of software application design and especially so for financial applications, health-care applications, and any application that deals with valuable or classified information. This view is too new to have any size information available as of 2009. However, it will probably turn out to be fairly concise.
7. The *pattern view*, or the combinations of the other views that are likely to occur in multiple software applications, and hence are candidates for reuse. Typical patterns with reuse potential will contain similar external features, similar algorithms, and similar data structures. Class libraries and inheritance of object-oriented software may also be part of software patterns. This view seems to require about 0.1 to 0.4 page per function point to describe specific patterns, with the size being based on the reusable feature being described.
8. The *logistical view* records certain historical facts about software applications that are often lost or difficult to find. These logistical topics include the date the application was first started, the locations and companies involved in construction, and information on the methods, tools, and practices used in construction. Application size in terms of both function points and logical code statements would be included in the logistical view, along with the various languages utilized. Since applications continue to grow, the logistical view should identify creeping requirements and then later growth

over multiple releases. The logistical view also includes the sources of reusable materials for the application. The logistical view is intended to aid in benchmarking. The logistical view would also be useful for multiple regression analysis to demonstrate the effectiveness of methods such as Agile or TSP. Part of the logistical view would be the placement of the application on a formal taxonomy, such as the one discussed earlier in this chapter. This view is usually less than ten pages, regardless of the size of the application itself.

When all of the eight views are summed together, the average size is about 3.0 pages per function point, and the range runs from less than 1.5 pages per function point to more than 6.0 pages per function point.

From the fairly large sizes associated with software design, it is easy to understand why the creation of paper documents can cost more than the source code for large applications. It is also easy to understand why some of the Agile concepts are in reaction to the large volumes and high costs of normal software design practices.

Given the multiple views that need to be captured during software design, it is obvious that no single language or representation method can deal with all eight kinds of view. Therefore, software design must utilize multiple methods of representing knowledge:

- Natural language text can be used for defining the attribute view, the logistics view, and for some of the external views. Special forms of natural language such as “executable English” may also be used.
- Images may be needed for some aspects of the external view, such as typical screens or samples of outputs.
- Mathematical formulas or other forms of scientific notations are needed for the algorithm view.
- Symbols and diagrams are needed for the structural view. Because of the dynamic nature of software, some form of animation would be preferable to static views. With animation, performance can be modeled during design.

Since automation for verification purposes would be somewhat difficult across multiple representation methods, it would be desirable and useful if the major views could be mapped into a single meta-language. Obviously, most of the views eventually get mapped into source code, but by the time the code is complete, it is too late to verify and validate the design.

Whether a generalized design meta-language is based on some form of Backus-Naur notation, a definite clause grammar (DCG), or something else, it should have the property of being analyzed automatically for verification and validation purposes. Taking verification and validation

one step further, it might also be possible to generate a suite of test cases from the analysis of the meta-language.

The bottom line on software design circa 2009 is that some of the 50 or so representation methods are effective for individual applications. But none are effective for pattern analysis and identification of candidates for reusable features.

Summary and Conclusions

The creation of various paper representations of software applications before the code itself is created has long been troublesome for the software engineering domain. Errors and mistakes are found in every form of paper description of software. Translation from requirements to design and from design to code always manages to leave some features behind, and often manages to add features that no one asked for.

The cost of producing paper documents is often greater than the cost of the source code itself. While paper documents can be inspected for errors, and inspections are quite effective, it is very difficult to carry out automated verification and validation of either text documents or graphic design documents.

In total software requirements, analysis, architecture, and design contribute to about 60 percent of all software bugs or defects and accumulate between 30 percent and 40 percent of software costs. Indeed, the three top cost elements of large software applications are

1. Finding and fixing bugs (many of which originate in paper documents)
2. Producing paper documents including requirements, architecture, and design
3. Creating the source code itself

Because paper documents are simultaneously more defective and more expensive than source code itself, there is a continuing need for software engineering researchers to pay more attention to both the error content of paper documents and also to the economic costs of paperwork.

Hopefully, future studies will enable software patterns to be more easily found and will also permit more effective validation of requirements and design by automated means.

As of 2009, formal inspection of requirements, architecture, and design is the most effective known way of eliminating defects in these important documents. But inspections are somewhat slow and costly. However, neither static analysis nor testing is fully capable of finding and removing requirements and design errors, so manual inspections are critical activities.

Readings and References

Note: Software requirements, business analysis, architecture, enterprise architecture, and design collectively have more than 500 book titles and thousands of journal articles in print. Yet in spite of the huge volume of published information, these areas of software engineering continue to be troublesome and erratic. The titles shown here represent only a small sample of the available literature.

The Cost and Quality Associated with Software Paperwork

- Beck, Kent. *Test-Driven Development*. Boston, MA: Addison Wesley, 2002.
- Cohen, Lou. *Quality Function Deployment—How to Make QFD Work for You*. Upper Saddle River, NJ: Prentice Hall, 1995.
- Cohn, Mike. *Agile Estimating and Planning*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005.
- Garmus, David and David Herron. *Function Point Analysis—Measurement Practices for Successful Software Projects*. Boston, MA: Addison Wesley Longman, 2001.
- Garmus, David and David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Gilb, Tom and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.
- Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Harris, Michael, David Herron, and Stacia Iwanicki. *The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results*. Boca Raton, FL: CRC Press (Auerbach), 2008.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989.
- Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Prentice Hall, 1994.
- Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 2007.
- Jones, Capers. *Patterns of Software System Failure and Success*. Boston, MA: International Thomson Computer Press, 1995.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley Longman, 2000.
- Jones, Capers. "Software Project Management Practices: Failure Versus Success." *CrossTalk*, Vol. 19, No. 6 (June 2006): 4–8.
- Jones, Capers. "Why Flawed Software Projects are not Cancelled in Time." *Cutter IT Journal*, Vol. 10, No. 12 (December 2003): 12–17.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston, MA: Addison Wesley Longman, 2003.
- McConnell, Steve. *Software Estimating: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Radice, Ronald A. *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing, 2002.
- Roetzheim, William H., and Reyna A. Beasley. *Best Practices in Software Cost and Schedule Estimation*. Upper Saddle River, NJ: Prentice Hall PTR, 1998.
- Strassmann, Paul. *Governance of Information Management: The Concept of an Information Constitution*, Second Edition. (eBook). Stamford, CT: Information Economics Press, 2004.
- Strassmann, Paul. *Information Payoff*. Stamford, CT: Information Economics Press, 1985.
- Strassmann, Paul. *Information Productivity*. Stamford, CT: Information Economics Press, 1999.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.

- Wiegars, Karl E. *Peer Reviews in Software—A Practical Guide*. Boston: Addison Wesley Longman, 2002.
- Yourdon, Ed. *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.

Software Requirements

- Artow, J. & I. Neustadt. *UML and the Unified Process*. Boston: Addison Wesley, 2000.
- Booch, Grady, Ivar Jacobsen, and James Rumbaugh. *The Unified Modeling Language User Guide*, Second Edition. Boston: Addison Wesley, 2005.
- Cockburn, Alistair. *Writing Effective Use Cases*. Boston: Addison Wesley, 2000.
- Cohn, Mike. *User Stories Applied: For Agile Software Development*. Boston: Addison Wesley, 2004.
- Fernandini, Patricia L. *A Requirements Pattern. Succeeding in the Internet Economy*. Boston: Addison Wesley, 2002.
- Gottesidner, Ellen. *The Software Requirements Memory Jogger*. Salem, NH: Goal QPC Inc., 2005.
- Inmon, William H., John Zachman, and Jonathan G. Geiger. *Data Stores, Data Warehousing, and the Zachman Framework*. New York: McGraw-Hill, 1997.
- Orr, Ken. *Structured Requirements Definition*. Topeka, KS: Ken Orr and Associates, Inc., 1981.
- Robertson, Suzanne and James Robertson. *Mastering the Requirements Process*, Second Edition. Boston: Addison Wesley, 2006.
- Wiegars, Karl E. *Software Requirements*, Second Edition. Bellevue, WA: Microsoft Press, 2003.
- Wiegars, Karl E. *More About Software Requirements: Thorny Issues and Practical Advice*. Bellevue, WA: Microsoft Press, 2000.

Software Business Analysis

- Carlenord, Barbara A. *Seven Steps to Mastering Business Analysis*. Ft. Lauderdale, FL: J. Ross Publishing, 2008.
- Haas, Kathleen B. *Getting it Right: Business Requirements Analysis Tools and Techniques*. Vienna, VA: Management Concepts, 2007.

Software Architecture

- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Boston: Addison Wesley, 1997.
- Marks, Eric and Michael Bell. *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology*. New York: John Wiley & Sons, 2006.
- Reekie, John and Rohan McAdam. *A Software Architecture Primer*. Angophora Press, 2006.
- Shaw, Mary and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- Taylor, R.N., N. Medvidovic, E.M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ: Wiley, 2009.
- Warnier, Jean-Dominique. *Logical Construction of Systems*. London: Van Nostrand Reinhold, 1978.

Enterprise Architecture

- Bernard, Scott. *An Introduction to Enterprise Architecture*, Second Edition. Philadelphia, PA: Auerbach Publications, 2008.
- Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison Wesley, 2007.

- Lankhorst, Marc. *Enterprise Architecture at Work: Modeling, Communication, and Analysis*. Cologne, DE: Springer, 2005.
- Spewak, Steven H. *Enterprise Architecture Planning: Developing a Blueprint for Data, Applications, and Technology*. Hoboken, NSJ: Wiley, 1993.

Software Design

- Ambler, S. *Process Patterns—Building Large-Scale Systems Using Object Technology*. Cambridge University Press, SIGS Books, 1998.
- Berger, Arnold S. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. Burlington, MA: CMP Books, 2001.
- Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Design*. Boston: Addison Wesley, 1995.
- Martin, James & Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice Hall, 1985.
- Shalloway, Alan & James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Second Edition. Boston, MA: Addison Wesley Professional, 2004.

This page intentionally left blank

Programming and Code Development

Introduction

This chapter has an unusual slant compared with other books on software engineering. Among other topics, it deals with 12 important questions that are not well covered in the software engineering literature:

1. Why do we have more than 2500 programming languages?
2. Why does a new programming language appear more than once a month?
3. How many programming languages are really needed by software engineering?
4. Why do most modern applications use between 2 and 15 different languages?
5. How many applications being maintained are written in “dead” programming languages with few programmers?
6. How many programmers use major languages; how many use minor languages?
7. Should there be a national translation center that maintains compilers and tools for dead programming languages and that can convert antique languages into modern languages?
8. What are the major kinds of bugs found in source code?
9. How effective are debuggers and static analysis tools compared with inspections?

10. How effective are various kinds of testing in terms of bug removal?
11. How effective is reusable code in terms of quality, security, and costs?
12. Why has the “lines of code” metric stopped being effective for software economic studies?

These 12 topics are not the only topics that are important about programming, but they are not discussed often in software engineering journals or books. Following are discussions of the 12 topics.

A Short History of Programming and Language Development

It is interesting to consider the history of programming and the development of programming languages. The early history of mechanical computers driven by gears, cogs, and later punched cards is interesting, but not relevant. However, these devices did embody the essence of *computer programming*, which is to control the behavior of a mechanical device by means of discrete instructions that could be varied in order to change the behavior of the machine.

The pioneers of computer design include Charles Babbage, Ada Lovelace, Hermann Hollerith, Alan Turing, John Von Neumann, Conrad Zuse, J. Presper Eckert, John Mauchly, and a number of others. John Backus, Konrad Zuse, and others contributed to the foundations of programming languages. David Parnas and Edsger Dijkstra contributed to the development of structured programming that minimized the tendency of code branching to form “spaghetti bowls” of so many branches that the code became nearly unreadable.

Ada Lovelace was an associate of Charles Babbage. In 1842 and 1843, she described a method of calculating Bernoulli numbers for use on the Babbage analytical engine. Her work is often cited as the world’s first computer program, although there is some debate about this.

In the years during and prior to World War II, a number of companies in various countries built electro-mechanical computing devices, primarily for special purposes such as calculating trajectories or handling mathematical tasks.

The earliest models were “programmed” in part by changing wire connections or using plug boards. But during World War II, computing devices were developed with memory that could store both data and instructions. The ability to have language instructions stored in memory opened the gates to modern computer programming as we know it today.

Konrad Zuse of Germany built the Z3 computer in 1941 and later designed what seems to be the first high-level language, Plankalkül,

in 1948, although no compiler was created and the language was not used.

The earliest “languages” that were stored in computers were binary codes or machine languages, which obviously were extremely difficult to understand, code, or modify. The difficulty of working with machine codes directly led to languages that were more amenable to human understanding but capable of being translated into machine instructions.

The earliest of these languages were termed *assembly languages* and usually had a one-to-one correspondence between the human-readable instructions (called *source code*) and the executable instructions (called *object code*).

The idea of developing languages that humans could use to describe various algorithms or data manipulation steps proved to be so useful that very shortly a number of more specialized languages were developed.

In these languages the human portions were optimized for certain kinds of problems, and the work of translating the languages into machine code was left to the compilers. Incidentally, the main difference between an *assembler* and a *compiler* is that assemblers tend to have a one-to-one ratio between source code and object code, while compilers have a one-to-many ratio. In other words, one statement in a compiled language might generate a dozen or more machine instructions.

The ability to translate a single source instruction into many object instructions led to the concept of *high-level programming languages*. In general, the higher the level of a programming language, the more object code can be created from a single source code statement.

Both assembly and compilation were handled by special translation programs as batch activities. The source code could not be run immediately. Sometimes translation might be delayed for hours if the computer was being used for other work and other computers were not available. These delays led to another form of code translation. Programming language translators called *interpreters* were soon developed, which allowed source code to be converted into object code immediately.

In the early days of computing and programming, software was used primarily for a narrow range of mathematical calculations. But the speed of digital computers soon gave rise to wider ranges of applications. When computers started to be aimed at business problems and to manipulate text and data, it became obvious that if the source code included some of the language and vocabulary of the problem domain, then programming languages would be easier to learn and use. The use of computers to control physical devices opened up yet another need for languages optimized for dealing with physical objects.

As a result, scores of domain-specific programming languages were developed that were aimed at tasks such as list processing, business applications, astronomy, embedded applications, and a host of others.

Why Do We Have More than 2500 Programming Languages?

The concept of having source code optimized for specific kinds of business or technical problems is one of the factors that led to the enormous proliferation of programming languages.

There are some technical advantages for having programming languages match the vocabulary of various problem domains. For one thing, such languages are easy to learn for programmers who are familiar with the various domains.

It is actually fairly easy to develop a new programming language. As computers began to be used for more and more kinds of problems, the result was more and more programming languages. Developing a new programming language that attracted other programmers also had social and prestige value.

As a result of these technical and social reasons, the software industry developed new programming languages with astonishing frequency. Today, as of 2009, no one really knows the actual number of programming languages and dialects, but the largest published lists of programming languages now contain 2500 languages (The Language List by Bill Kinnersley, <http://people.ku.edu>).

The author's former company, Software Productivity Research, has been keeping a list of common programming languages since 1984, and the current version contains more than 700 programming languages. New programming languages continue to come out at a rate of two or three per calendar month; some months, more than 10 languages have arrived. There is no end in sight.

One reason for the plethora of languages is that a new language can be developed by a single software engineer in only a month or two. In fact, with compiler-compilers, a new programming language can evolve from a vague idea to compiled code in 60 days or less.

In 1984, the author's first commercial software estimating tool was put on the market. The first release of the tool could perform cost and quality estimates for 30 different programming languages, but the tool itself could handle other languages using the same logic and algorithms. Therefore, we made a statement to customers that our tool could support cost estimates for "all known programming languages."

Having made the claim, it was necessary to back it up by assembling a list of all known programming languages and their levels. At the time the claim was made in 1984, the author hypothesized that the list might include 50 languages. However, when the data was collected, it was discovered that the set of "all known programming languages" included about 250 languages and dialects circa 1984.

It was also discovered while compiling the list that new languages were popping up about once a month; sometimes quite a few more.

It became obvious that keeping track of languages was not going to be quick and easy, but would require continuous effort.

Today, as of 2009, the current list of languages maintained by Software Productivity Research has grown to more than 700 programming languages, and the frequency with which new languages come out seems to be increasing from about one new language per month up to perhaps two or even four and occasionally ten new languages per month.)

An approximate chronology of significant programming languages is shown in Table 8-1.

Table 8-1 is only a tiny subset of the total number of programming languages. It is included just to give readers who may not be practicing programmers an idea of the wide variety of languages in existence.

Those familiar with programming concepts can see from the list that programming language design took two divergent paths:

- Specialized languages that were optimal for narrow sets of problems such as FORTRAN, Lisp, ASP, and SQL
- General-purpose languages that could be used for a wide range of problems such as Ada, Objective C, PL/I, and Ruby.

It is of sociological interest that the vast majority of special-purpose languages were developed by individuals or perhaps two individuals. For example, Basic was developed by John Kemeny and Thomas Kurtz; C was developed by Dennis Ritchie; FORTRAN was developed by John Backus; Java was developed by James Gosling; and Objective C was developed by Brad Cox and Tom Love.

The general-purpose languages were usually developed by committees. For example, COBOL was developed by a famous committee with major inputs from Grace Hopper of the U.S. Navy. Other languages developed by committees include Ada and PL/I. However, some general-purpose languages were also developed by individuals or colleagues, such as Ruby and Objective C.

For reasons that are perhaps more sociological than technological, the attempts at building general-purpose languages such as PL/I and Ada have not been as popular with programmers as many of the special-purpose languages.

This is a topic that needs both sociological and technical research, because PL/I and Ada appear to be well designed, robust, and capable of tackling a wide variety of applications with good results.

Another major divergence in programming languages occurred during the late 1970s, although research had started earlier. This is the split between object-oriented languages such as SMALLTALK, C++, and Objective C and languages that did not adopt OO methods and terminology, such as Basic, Visual Basic, and XML.

TABLE 8-1 Chronology of Programming Language Development

1951	Assembly languages
1954	FORTRAN (Formula Translator)
1958	Lisp (List Processing)
1959	COBOL (Common Business-Oriented Language)
1959	JOVIAL (Jules Own Version of the International Algorithmic Language)
1959	RPG (formerly Report Program Generator)
1960	ALGOL (Algorithmic Language)
1962	APL (A Programming Language)
1962	SIMULA
1964	Basic (Beginner's all-purpose symbolic instruction code)
1964	PL/I
1964	CORAL
1967	MUMPS
1970	PASCAL
1970	Prolog
1970	Forth
1972	C
1978	SQL (Structured query language)
1980	CHILL
1980	dBASE II
1982	SMALLTALK
1983	Ada83
1985	Quick Basic
1985	Objective C
1986	C++
1986	Eiffel
1986	JavaScript
1987	Visual Basic
1987	PERL
1989	HTML (Hypertext Markup Language)
1993	AppleScript
1995	Java
1995	Ruby
1999	XML (Extensible Markup Language)
2000	C#
2000	ASP (Active Server Pages)
2002	ASP.NET

Today in 2009, more than 50 percent of active programming languages tend to be in the object-oriented camp, while the other languages are procedural languages, functional languages, or use some other method of operation.

Yet another dichotomy among programming languages is whether they are typed or un-typed. The term *typed* means that operations in a language are restricted to only specific data types. For example, a typed language would not allow mathematical operations against character data. Examples of typed languages include Ruby, SMALLTALK, and Lisp.

The opposite case, or *un-typed* languages, means that operations can be performed against any type of data. Examples of un-typed languages include assembly language and Forth.

The terms type and un-typed are somewhat ambiguous, as are the related terms of strongly typed and weakly typed. Over and above ambiguity, there is some debate as to the virtues and limits of typed versus un-typed languages.

Exploring the Popularity of Programming Languages

There are a number of ways of studying the usage and popularity of programming languages. These include

1. Statistical analysis of web searches for specific languages
2. Statistical analysis of books and articles published about specific languages
3. Statistical analysis of citations in the literature about specific languages
4. Statistical analysis of job ads for programmers that cite language skills
5. Surveys and statistical analysis of languages in legacy applications
6. Surveys and statistical analysis of languages used for new applications

A company called Tiobe publishes a monthly analysis of programming language popularity that ranks 100 different programming languages. Since this section is being written in May 2009, the 20 most popular languages for this month from the Tiobe rankings are listed in Table 8-2.

Older readers may wonder where COBOL, FORTRAN, PL/I, and Ada reside. They are further down the Tiobe list in languages 21 through 40.

Since new languages pop up at a rate of more than one per month, language popularity actually fluctuates rather widely on a monthly basis. As interesting new programming languages appear, their popularity goes up rapidly. But based on their utility or lack of utility over longer periods, they may drop down again just as fast.

TABLE 8-2 Popularity Ranking of Programming Languages as of May 2009

1.	Java
2.	C
3.	C++
4.	PHP
5.	Visual Basic
6.	Python
7.	C#
8.	JavaScript
9.	Perl
10.	Ruby
11.	Delphi
12.	PL/SQL
13.	SAS
14.	PASCAL
15.	RPG (OS/400)
16.	ABAP
17.	D
18.	MATLAB
19.	Logo
20.	Lua

The popularity of programming languages bears a certain resemblance to the popularity of prime-time television shows. Some new shows such as *Two and a Half Men* surface, attract millions of viewers, and may last for a number of seasons. A few shows such as *Seinfeld* become so popular that they go into syndication and continue to be aired long after production stops. But many shows are dropped after a single season.

It is interesting that the life expectancy of programming languages and the life expectancy of television shows are about the same. Many programming languages have active lives that span only a few “seasons” and then disappear. Other languages become standards and may last for many years. However, when all 2500 languages are considered, the average active life of a programming language when it is being used for new development is less than five years. Very few programming languages attract development programmers after more than ten years.

Some of the languages that are in the class of *Seinfeld* or *I Love Lucy* and may last more than 25 years under syndication include

- Ada
- C
- C++

- COBOL
- Java
- Objective C
- PL/I
- SQL
- Visual Basic
- XML

In a programming language context, the term *syndication* means that the language is no longer under the direct control of its originator, but rather control has passed to a user group or to a commercial company, or that the language has been put in the public domain and is available via open-source compilers.

It would be interesting and valuable if there were benchmarks and statistics kept of the numbers of applications written in these long-lived programming languages. No doubt C and COBOL have each been used for more than 1 million applications on a global basis.

In fact, continuing with the analogy of the entertainment business, it might be interesting to have awards for languages that have been used for large numbers of applications. Perhaps “silver” might go for 100,000 applications, “gold” for 1 million applications, and “platinum” for 10 million applications.

If such an award were created, a good name for it might be the “Hopper,” after Admiral Grace Hopper, who did so much to advance programming languages and especially COBOL. In fact, COBOL is probably the first programming language in history to achieve the 1-million-application plateau.

Although the idea of awards for various numbers of applications is interesting, that would mean that statistics were available for ascertaining how many applications were created in specific languages or combinations of languages. As of 2009, the software industry does not keep such data.

The choice of which language should be used for specific kinds of applications is surprisingly subjective. A colleague at IBM was asked in a meeting if he programmed in the APL language. His response was, “No, I’m not of that faith.”

It would be technically possible to develop a standard method of describing and cataloging the features of programming languages. Indeed, with more than 2500 languages in existence, such a catalog is urgently needed. Even if the catalog only started with 100 of the most widely used languages, it would provide valuable information.

The full set of topics included to create an effective taxonomy of programming languages is outside the scope of this book, but might contain factors such as:

1.	Language name	Name of language
2.	Architecture	Object-oriented, functional, procedural, etc.
3.	Origin	Year of creation, names of inventors
4.	Sources	URLs of distributors of language compilers
5.	Current version	Version number of current release; 1, 2, or whatever
6.	Support	URLs or addresses of maintenance organizations
7.	User associations	Names, URLs, and locations of user groups
8.	Tutorial materials	Books and learning sources about the language
9.	Reviews or critiques	Published reviews of language in refereed journals
10.	Legal status	Public domain, licensed, patents, etc.
11.	Language definition	Whether it is formal, informal
12.	Language syntax	Description of syntax
13.	Language typing	Strongly typed, weakly typed, un-typed, etc.
14.	Problem domains	Mathematics, web, embedded, graphics, etc.
15.	Hardware platforms	Hardware language was intended to support
16.	OS platforms	Operating systems language compilers work with
17.	Intended uses	Targeted application types
18.	Known limitations	Performance, security, problem domains, etc.
19.	Dialects	Variations of the basic language
20.	Companion languages	.NET, XML, etc. (languages used jointly)
21.	Extensibility	Commands added by language users
22.	Level	Logical statements relative to assembly language
23.	Backfire level	Logical statements per function point
24.	Reuse sources	Certified modules, uncertified, etc.
25.	Security features	Intrinsic security features, such as in the E language
26.	Debuggers available	Names of debugging tools
27.	Static analysis available	Names of static analysis tools
28.	Development tools available	Names of development tools
29.	Maintenance tools available	Names of maintenance tools
30.	Applications to date	Approximately 100, 1000, 10,000, 100,000, etc.

Given the huge number of programming languages, it is surprising that no standard taxonomy exists. Web searches reveal more than a dozen topics when using search arguments such as “taxonomies of programming languages” or “categories of programming languages.” However,

these vary widely, and some contain more than 50 different descriptive forms, but seem to lack any fundamental organizing principle.

Returning now to the main theme, somewhat alarmingly, the life expectancy of many software applications is longer than the active life of the languages in which they were written. An example of this is the patient-record systems of medical records maintained by the Veterans Administration. It is written in the MUMPS programming language and has far outlived MUMPS itself.

It is obvious to students of software engineering economics that if programming languages have an average life expectancy of only 5 years, but large applications last an average of 25 years, then software maintenance costs are being driven higher than they should be due to the very large number of aging applications that were coded in programming languages that are now dead or dying.

How Many Programming Languages Are Really Needed?

The plethora of programming languages raises basic questions that need to be addressed by the software engineering literature: *How many programming languages does software engineering really need?*

Having thousands of programming languages raises a corollary question: *Is the existence of more than 2500 programming languages a good thing or a bad thing?*

The argument that asserts having thousands of languages is a good thing centers around the fact that languages tend to be optimized for unique classes of problems. As new problems are encountered, they demand new programming languages, or at least that is a hypothesis.

The argument that asserts having thousands of languages is a bad thing centers around economics. Maintenance of legacy applications written in dead languages is an expensive nightmare. The constant need to train development programmers in the latest cult language is expensive. Many useful tools such as static analysis tools and automated test tools support only a small subset of programming languages, and therefore may require expensive modifications for new languages. Accumulating large volumes of certified reusable code is more difficult and expensive if thousands of languages have to be dealt with.

The existence of thousands of programming languages has created a new subindustry within software engineering. This new subindustry is concerned with translating dead or dying languages into new living languages. For example, it is now possible to translate the MUMPS language circa 1967 into the C or Java languages and to do so automatically.

A corollary subindustry is that of *renovation* or periodically performing special maintenance activities on legacy applications to clean out

dead code, remove error-prone modules, and to reduce the inevitable increase in cyclomatic and essential complexity that occurs over time due to repeated small changes.

Linguists and those familiar with natural human languages are aware that translation from one language to another is not perfect. For example, some Eskimo dialects include more than 30 different words for various kinds of snow. It is hard to get an exact translation into a language such as English that developed in a temperate climate and has only a few variations on “snow.”

Since many programming languages have specialized constructs for certain classes of problem, the translation into other languages may lead to awkward constructs that might be difficult for human programmers to understand or deal with during maintenance and enhancement work. Even so, if the translation opens up a dead language to a variety of static analysis and maintenance tools, the effort is probably worthwhile.

To deal with the question of how many programming languages are needed, it is useful to start by considering the universe of problem areas that need to be put onto computers. There seem to be ten discrete problem areas, divided into two different major kinds of processing, as shown in Table 8-3.

These two general categories reflect the major forms of software that actually exist today: (1) software that processes information, and (2) software that controls physical devices or deals with physical properties such as sound or light or music.

These two broad categories might lead to the conclusion that perhaps two programming languages would be the minimum number that would be able to address all problem areas. One language would be optimized for information systems, and another would be optimized for dealing with physical devices and electronic signals. However, the

TABLE 8-3 Problem Domains of Software Applications

Logical and Mathematical Problem Areas	
1.	Mathematical calculations
2.	Logic and algorithmic expressions
3.	Numerical data
4.	Text and string data
5.	Time and dates
Physical Problem Areas	
1.	Sensor-based electronic signals
2.	Audible signals and music
3.	Static images
4.	Dynamic or moving images
5.	Colors

track records of general-purpose languages such as PL/I and Ada have not indicated much success for languages that attempt to do too many things at once.

Few problems are “pure” and deal with only one narrow topic. In fact, most applications deal with hybrid problem domains. This leads to a possible conclusion that programming languages may reflect the permutations of problem areas rather than the problem areas individually.

If the permutations of all ten problem areas were considered, then we might eventually end up with 3,628,800 programming languages. This is even more unlikely to occur than having one “superlanguage” that could tackle all problem areas.

From examining samples of both information processing applications and embedded and systems software applications, a provisional hypothesis is that about four different problem areas occur in typical software applications. The permutation of four topics out of a total of ten topics leads to the hypothesis that the software engineering domain will eventually end up with about 5,040 different programming languages.

Since we already have about 2500 programming languages and dialects in 2009, there may yet be another 2500 languages still to be developed in the future. At the rate new languages are occurring of roughly 100 per year, it can be projected that new languages will proceed at about the same rate for another 25 years. From an economic standpoint, this does not seem to be a very cost-effective engineering solution.

Assuming that the software engineering community does reach 5040 languages, the probable distribution of those languages would be

- 4800 languages would be dead or dying, with few programmers
- 200 languages would be in legacy applications and therefore need maintenance
- 40 languages would be new and gathering increasing numbers of programmers

A technical alternative to churning out another 2500 specialized languages for every new kind of problem that surfaces would be to consider building polymorphic compilers that would support any combination of problem areas.

Creating a National Programming Language Translation Center

When considering alternatives to churning out another 2500 programming languages, it might be of value to create a formal programming language translation center stocked with the language definitions of all known programming languages.

This center could provide guidance in the translation of dead or dying languages into modern active languages. Some companies already perform translation, but out of today's total of 2500 languages, only a few are handled with technical and linguistic accuracy. Automated translation as of 2009 probably only handles 50 languages out of 2500 total languages.

Given the huge number of existing programming languages and the rapid rate of creation of new programming languages, such a translation center would probably require a full-time staff of at least 50 personnel. This would mean that only very large companies such as IBM or Microsoft or large government agencies such as Homeland Security or the Department of Defense would be likely to attempt such an activity.

Over and above translation, the national programming language translation center could also perform careful linguistic analyses of all current languages in order to identify the main strengths and weaknesses of current languages. One obvious weakness of most languages is that they are not very secure.

Another function of the translation center would be to record demographic information about the numbers and kinds of applications that use various languages. For example, the languages used for financial systems, for weapons systems, for medical applications, for taxation systems, and for patient records have economic and even national importance. It would be useful to keep records of the programming languages used for such vital applications. Obviously, maintenance and restoration of these vital applications has major business and national importance.

Table 8-4 is a summary of 40 kinds of software applications that have critical importance to the United States. Table 8-4 also shows the various programming languages used in these 40 kinds of applications. A major function of a code translation center would be to accumulate more precise data on critical applications and the languages used in them.

Both columns of Table 8-4 need additional research. There are no doubt more kinds of critical applications than the 40 listed here. Also, in order to fit on a printed page, the second column of the table is limited to about six or seven programming languages. For many of these critical applications, there may be 50 or more languages in use at national levels.

The North American Industry Classification (NAIC) codes of the Department of Commerce identify at least 250 industries that the author knows create software in substantial volumes. However, the 40 industries shown in Table 8-4 probably contain almost 50 percent of applications critical to U.S. business and government operations.

TABLE 8-4 Programming Languages Used for Critical Software Applications

Critical Software	Programming Languages
1. Air traffic control	Ada, Assembly, C, Jovial, PL/I
2. Antivirus & security	ActiveX, C, C++, Oberon7
3. Automotive engines	C, C++, Forth, Giotto
4. Banking applications	C, COBOL, E, HTML, Java, PL/I, SQL, XML
5. Broadband	C, C++, CESOF, JAVA
6. Cell phones	C, C++, C#, Objective C
7. Credit cards	ASP.NET, C, COBOL, Java, Perl, PHP, PL/I
8. Credit checking	ABAP, COBOL, FORTRAN, PL/I, XML
9. Credit unions	C, COBOL, HTML, PL/I, SQL
10. Criminal records	ABAP, C, COBOL, FORTRAN, Hancock
11. Defense applications	Ada, Assembly, C, CMS2, FORTRAN, Java, Jovial, SPL
12. Electric power	Assembly, C, DCOPEJ, Java, Matpower
13. FBI, CIA, NSA, etc.	Ada, APL, Assembly, C, C++, FORTRAN, Hancock
14. Federal taxation	C, COBOL, Delphi, FORTRAN, Java, SQL
15. Flight controls	Ada, Assembly, C, C++, C#, LabView
16. Insurance	ABAP, COBOL, FORTRAN, Java, PL/I
17. Mail and shipping	COBOL, dBase2, PL/I, Python, SQL
18. Manufacturing	AML, APT, C, Forth, Lua, RLL
19. Medical equipment	Assembly, Basic, C, CO, CMS2, Java
20. Medical records	ABAP, COBOL, MUMPS, SQL
21. Medicare	Assembly, COBOL, Java, PL/I, dBase2, SQL
22. Municipal taxation	C, COBOL, Delphi, Java
23. Navigation	Assembly, C, C++, C#, Lua, Logo, MatLab
24. Oil and Energy	AMPL, C, G, GAMS/MPGSE, SLP,
25. Open-source software	C, C++, JavaScript, Python, Suneido, XUL
26. Operating systems, large	Assembly, C, C#, Objective C, PL/S, VB
27. Operating systems, small	C, C++, Objective C, OSL, SR
28. Pharmaceuticals	C, C++, Java, PASCAL, SAS, Visual Basic
29. Police records	C, COBOL, DBase2, Hancock, SQL
30. Satellites	C, C++, C#, Java, Jovial, PHP, Pluto
31. Securities trading	ABAP, C #, COBOL, DBase2, Java, SQL
32. Social Security	Assembly, COBOL, PL/I, dBase2, SQL
33. State taxation	C, COBOL, Delphi, FORTRAN, Java, SQL
34. Surface transportation	C, C++, COBOL, FORTRAN, HTML, SQL
35. Telephone switching	C, CHILL, CORAL, Erlang, ESPL1, ESTEREL
36. Television broadcasts	C, C++, C#, Java, Forth
37. Voting equipment	Ada, C, C++, Java
38. Weapons systems	Ada, Assembly, C, C++, Jovial
39. Web applications	AppleScript, ASP, CMM, Dylan, E, Perl, PHP, .NET
40. Welfare (State)	ASP.NET, C, COBOL, dBASE2, PL/I, SQL

As a result of the importance of these 40 software application areas to the United States business and to government operations, they probably receive almost 75 percent of cyberattacks in the form of viruses, spyware, search-bots, and denial of service attacks. These 40 industries need to focus on security. Even a cursory examination of the programming languages used by these industries reveals that few of them are particularly resistant to viruses or malware attacks.

For all 40, maintenance is expensive and for many, it is growing progressively more expensive due to the difficulty of simultaneously maintaining applications written in so many different programming languages.

As a technical byproduct of translation from older languages to new languages, one value-added function of a national programming language translation center would be to eliminate security vulnerabilities at the same time the older languages are being translated.

If the language translation center operated as a profit-making business, it might well grow a good-sized company. Assuming the company billed at the same rate as Y2K companies (about \$1.00 per logical statement), a national translation center might clear \$75 million per year, assuming accurate and competent translation technology.

What the author suggests is that rather than continue to develop random programming languages at random but rapid intervals, there is a need to address programming languages at a fundamental linguistic level.

A study team that included linguists, software engineers, and domain specialists might be able to address the problems of the most effective ways of expressing the ten problem areas and their permutations. The goal would be to understand the minimum set of programming languages capable of handling any combination of problem areas.

If economists were added to the study team, they would also be able to address the financial impact of attempting to maintain and occasionally renovate applications written in hundreds of dead and dying programming languages.

Why Do Most Applications Use Between 2 and 15 Programming Languages

A striking phenomenon of software engineering is the presence of multiple programming languages in the same applications. This is not a new trend, and many older applications used combinations such as COBOL and SQL. More recent combinations might include Java and HTML or XML.

A similar phenomenon is the fact that many programming languages are themselves combinations of two or more other programming

languages. For example, the Objective C language combines features from SMALLTALK and C. The Ruby language combines features from Ada, Eiffel, Perl, and Python among others.

Recall that a majority of programming languages are somewhat specialized, and these seem to be more popular than general-purposes languages. A hypothesis that explains why applications use several different programming languages is that the “problem space” of the application is broader than the “solution space” of individual programming languages.

It was mentioned earlier that many applications include at least four of the ten problem areas cited in Table 8-3. However, many programming languages seem to be optimized only for one to three of the problem areas. This creates a situation where multiple programming languages are needed to implement all of the problem areas in the application.

Of course, using any of the more general-purpose languages such as Ada or PL/I would reduce the numbers of languages, but for sociological reasons, these general-purpose languages have not been as popular as the more specialized languages.

The implications of having many different languages in the same application are that development is more difficult, debugging is more difficult, static analysis is more difficult, and code inspection is more difficult. After release, maintenance and enhancement tasks are more difficult.

Table 8-5 illustrates how both development and maintenance costs go up as the number of languages in an application increase. The costs show the rate of increase compared with a single language.

Both development and maintenance costs increase as numbers of programming languages in the same application increase, but maintenance is more severely impacted.

TABLE 8-5 Impact of Multiple Languages on Costs

Languages in Application	Development Costs	Maintenance Costs
1	\$1.00	\$1.00
2	\$1.07	\$1.14
3	\$1.12	\$1.17
4	\$1.13	\$1.20
5	\$1.18	\$1.24
6	\$1.22	\$1.30
7	\$1.23	\$1.35
8	\$1.27	\$1.40
9	\$1.30	\$1.47
10	\$1.34	\$1.55

How Many Programmers Use Various Programming Languages?

There is no real census of either languages used in applications or number of programmers. While the Department of Commerce and the Bureau of Labor Statistics do issue reports on such topics in the United States, their statistics are known to be inaccurate.

A survey done by the author and his colleagues a few years ago found that the human resources organizations in most large corporations did not know how many programmers or software engineers were actually employed. Since government statistics are based on reports from HR organizations, if they don't know, then HR organizations can't provide good data to the government.

Among the reasons government statistics probably understate the numbers of programmers and software engineers is because of ambiguous job titles. For example, some large companies use titles such as "member of the technical staff" as an umbrella title that might include software engineers, hardware engineers, systems analysts, and perhaps another dozen occupations.

Another problem with knowing how many software engineers there are is the fact that many personnel working on embedded applications are not software engineers or computer scientists by training, but rather electrical engineers, aeronautical engineers, telecommunications engineers, or some other type of engineer.

Because the status of these older forms of engineering is higher than the status of software engineering, many people working on embedded software refuse to be called software engineers and insist on being identified by their true academic credentials.

The study carried out by the author and his colleagues was to derive information on the number of software specialists (i.e., quality assurance, database administration, etc.) employed by large software-intensive companies such as IBM, AT&T, Hartford Insurance, and so forth.

The study included on-site visits and discussions with both HR organizations and also local software managers and executives. It was during the discussions with local software managers and executives that it was discovered that not a single HR organization actually had good statistics on software engineering populations.

Based on on-site interviews with client companies and then extrapolation from their data to national levels, the author assumes that the U.S. total of software engineers circa 2009 is about 2.5 million. Government statistics as of 2009 indicate around 600,000 programmers, but these statistics are low for reasons already discussed. Additionally, the government statistics also tend to omit one-person companies and individual programmers who develop applets or single applications.

About 60 percent of these software engineers work in maintenance and enhancement tasks, and 40 percent work as developers on new applications. There are of course variations. For example, many more developers than maintenance personnel work on web applications, because all of these applications are fairly new. But for traditional mainframe business applications and ordinary embedded and systems software applications, maintenance workers outnumber development workers by a substantial margin.

Table 8-6 shows the approximate numbers of software engineers by language for the United States. However, the data in Table 8-6 is hypothetical and not exact. Among the reasons that the data is not exact is that many software engineers know more than one programming language and work with more than one programming language.

However, Table 8-6 does illustrate a key point: The most common languages for software development are not the same as the most common languages for software maintenance. This situation leads to a great deal of trouble for the software industry.

The most obvious problem illustrated by Table 8-6 is that it is difficult to get development personnel to work on maintenance tasks because of the perceived view that older languages are not as glamorous as modern languages.

A second problem is that due to the differences in programming languages between maintenance and new development, two different sets

TABLE 8-6 Estimated Number of Software Engineers by Language

Development Languages	Software Engineers	Maintenance Languages	Software Engineers
Java	175,000	COBOL	575,000
C	150,000	PL/I	125,000
C++	130,000	Ada	100,000
Visual Basic	100,000	Visual Basic	75,000
C#	90,000	RPG	75,000
Ruby	65,000	Basic	75,000
JavaScript	50,000	Assembler	75,000
Perl	30,000	C	75,000
Python	20,000	FORTRAN	65,000
COBOL	15,000	Java	60,000
PHP	15,000	JavaScript	40,000
Objective C	10,000	Jovial	10,000
Others	150,000	Others	150,000
	1,000,000		1,500,000

of tools are likely to be needed. The developers are interested in using modern tools including static analysis, automated testing, and other fairly new innovations.

However, many of these new tools do not support older languages, so the software maintenance community needs to be equipped with maintenance workbenches that include tools with different capabilities. For example, tools that analyze cyclomatic and essential complexity are used more often in maintenance work than in new development. Tools that can trace execution flow are also used more widely in maintenance work than in development. Another new kind of tool that supports maintenance more than development can “mine” legacy code and extract hidden business rules. Yet another kind of tool that supports maintenance work is tools that can parse the code and automatically generate function point totals.

It is fairly easy for programmers to learn new languages, but nobody can possibly learn 2500 programming languages. An average programmer in the U.S. is probably fairly expert in one language and fairly knowledgeable in three others. Some may know as many as ten languages. The plethora of languages obviously introduces major problems in academic training and in ways of keeping programmers current in their skill sets.

The bottom line is that development and maintenance tool suites are often very different, and this is due in large part to the differences in programming languages used for development and for maintenance.

Since the great majority of languages widely used for development today in 2009 will fall out of service in less than ten years, the software industry faces some severe maintenance challenges.

Languages used for new development are surfacing at rates of more than two per month. Most of these languages will be short-lived. However, some of the applications created in these ephemeral languages will last for many years. As a result, the set of programming languages associated with legacy applications that need maintenance is growing larger at rates that sometimes might top 50 languages per year!

A major economic problem associated with having thousands of programming languages is that the plethora of languages is driving up maintenance costs. Ironically, one of the major claims of new programming languages is that “they improve programming productivity.” Assuming that such claims are true at all, they are only true for new development. Every single new language is eventually going to add to the U.S. software maintenance burden. This is because programming languages have shorter life expectancies than the applications created with them. One by one, today’s “new” languages will drop out of use and leave behind hundreds of aging legacy applications with declining numbers of trained programmers, few effective tools, and sometimes not even working compilers.

What Kinds of Bugs or Defects Occur in Source Code?

In 2008 and 2009, a major new study was performed that identified the most common and serious 25 software bugs or defects. The study was sponsored by the SANS Institute, with the cooperation of MITRE and about 30 other organizations.

This new study is deservedly attracting a great deal of attention. In the history of software quality and security, it will no doubt be ranked as a landmark report. Indeed, all software engineering groups should get copies of the report and make it required reading for software engineers, quality assurance personnel, and also for software managers and executives.

Access to the report can be had via either the SANS Institute or MITRE web sites. The relevant URLs are

- www.SANS.org
- www.CWE-MITRE.org

In spite of the fact that software engineering is now a major occupation and millions of applications have been coded, only recently has there been a serious and concentrated effort to understand the nature of bugs and defects that exist in source code. The SANS report is significant because the list of 25 serious problems was developed by a group of some 40 experts from major software organizations. As a result, it is obvious that the problems cited are universal programming problems and not issues for a single company.

Over the years, many large companies such as IBM, AT&T, Microsoft, and Unisys have had very sophisticated defect tracking and monitoring systems. These same companies have also used root-cause analysis. Some of the results of these internal defect tracking systems have been published, but they usually were not perceived as having general applicability.

A number of common problems have long been well understood: buffer overflows, branches to incorrect locations, and omission of error handling are well known and avoided by experienced software engineers. But that is not the same as attempting a rigorous analysis and quantification of coding defects.

The SANS report is a very encouraging example of the kind of progress that can be made when top experts from many companies work together in a cooperative manner to explore common problems. The SANS study group included experts from academia, government, and commercial companies. It is also encouraging that these three kinds of organizations were able to cooperate successfully. The normal relationship among the three is often adversarial rather than cooperative, so

having all three work together and produce a useful report is a fairly rare occurrence.

Hopefully, the current work will serve as a model of future collaboration that will deal with other important software issues. Some of the additional topics that might do well in a collaborative mode include:

1. Defect removal methods
2. Economic analysis of software development
3. Economic analysis of software maintenance
4. Software metrics and measurement
5. Software reusability

Some of the organizations that participated in the SANS study include in alphabetical order:

- Apple
- Aspect Security
- Breach Security
- CERT
- Homeland Security
- Microsoft
- Mitre
- National Security Agency
- Oracle
- Perdue University
- Red Hat
- Tata
- University of California

This is only a partial list, but it shows that the study included academia, commercial software organizations, and government agencies.

The overall list of 25 security problems was subdivided into three larger topical areas. Readers are urged to review the full report, so only a bare list of topics is included here:

Interactions

1. Poor input validation
2. Poor encoding of output
3. SQL query structures

4. Web page structures
5. Operating system command structures
6. Open transmission of sensitive data
7. Forgery of cross-site requests
8. Race conditions
9. Leaks from error messages

Resource Management

10. Unconstrained memory buffers
11. Control loss of state data
12. Control loss of paths and file names
13. Hazardous paths
14. Uncontrolled code generation
15. Reusing code without validation
16. Careless resource shutdown
17. Careless initialization
18. Calculation errors

Defense Leakages

19. Inadequate authorization and access control
20. Inadequate cryptographic algorithms
21. Hard coding and storing passwords
22. Unsafe permission assignments
23. Inadequate randomization
24. Excessive issuance of privileges
25. Client/server security lapses

The complete SANS list contains detailed information about each of the 25 defects and also supplemental information on how the defects are likely to occur, methods of prevention, and other important issues. This is why readers are urged to examine the full SANS list.

As of 2009, these 25 problems may occur in more than 85 percent of all operational software applications. One or more of these 25 problems can be cited in more than 95 percent of all successful malware attacks. Needless to say, the SANS list is a very important document that needs widespread distribution and extensive study.

The SANS report is a valuable resource for companies involved in testing, static analysis, inspections, and quality assurance. It provides a very solid checklist of topics that need to be validated before code can be safely released to the outside world.

Logistics of Software Code Defects

While the SANS report does an excellent job of identifying serious software and code defects, once the defects are present in the code and the code is in the hands of users, some additional issues need discussion. Following is a list of topics that discuss logistical issues associated with software defects:

1. **Defect** A problem caused by human beings that causes a software application to either stop running or to produce incorrect results. Defects can be errors of commission, where developers did something wrong, or errors of omission, where developers failed to anticipate a specific condition.
2. **Defect severity level** (IBM definition) Severity 1, software stops working; Severity 2, major features disabled or incorrect; Severity 3, minor problem; Severity 4, cosmetic error with no operational impact.
3. **Invalid defect** A problem reported as a defect but which upon analysis turns out to be caused by something other than the software itself. Hardware problems, user errors, and operating system errors mistakenly reported as application errors are the most common invalid defects. These total as many as 15 percent of valid defect reports.
4. **Abeyant defect** (IBM term) A defect reported by a specific customer that cannot be replicated on any other version of the software except the one being used by the customer. Usually, abeyant defects are caused by some unique combination of hardware devices and other applications that run at the same time as the software against which the defect was reported. These are rare but very difficult to track down and repair.
5. **False positive** A code segment initially identified by a static analysis tool or a test case as a potential defect. Upon further analysis, the code turns out to be correct.
6. **Secondhand defects** A defect in an application that was not caused by any overt mistakes on the part of the development team itself, but instead was caused by errors in a compiler or tool used by the development team. Errors in code generators and automatic test tools are examples of secondhand defects. The developers used

the tools in good faith, but as a result, bugs were created. An example of a secondhand defect was a compiler error that incorrectly handled an instruction. The code was compiled and executed, but the instruction did not operate as defined in the language specification. It was necessary to review the machine language listings to find this secondhand defect since it was not visible in the source code itself.

7. **Undetected defects** These are similar to secondhand defects, but turn out to be due to either incomplete test coverage or to gaps in static analysis tools. It is widely known that test suites almost never touch 100 percent of the code in any application, and sometimes less than 60 percent of the code in large applications. To minimize the impact of undetected defects and partial test coverage, it is necessary to use test coverage analysis tools. Major gaps in coverage may need special testing or formal inspections.
8. **Data defects** Defects that are not in source code or applications, but which reside in the data that passes through the application. A very common example of a data defect would be an incorrect mailing address. Data errors are numerous and may be severe, and they are also difficult to eliminate. Data defects probably outnumber code defects, and their status in terms of liability is ambiguous. More serious examples of data defects are errors in credit reports, which can lower credit ratings without any legitimate reason and also without any overt defects in software. Data defects are notoriously difficult to repair, in part because there are no effective quality assurance organizations involved with data defects. In fact, there may not even be any reporting channels.
9. **Externally caused defects** A defect that was not originally a defect, but became one due to external changes such as new tax laws, changes in pension plans, and other government mandates that trigger code changes in software applications. An example would be a change in state sales taxes from 6 percent to 7.5 percent, which necessitates changes in many software applications. Any application that does not make the change will end up with a defect even though it may have run successfully for years prior to the external change. Such changes are frequent but unpredictable because they are based on government actions.
10. **Bad fixes** About 7 percent of attempts to repair a software code defect accidentally contain a new defect. Sometimes there are secondary and even tertiary bad fixes. In one lawsuit against a software vendor, four consecutive attempts to fix a bug in a financial application added new defects and did not fix the original defect. The fifth attempt finally got it right.

11. **Legacy defects** These are defects that surface today, but which may have been hidden in software applications for ten years or more. An example of a legacy defect was a payroll application that stopped calculating overtime payments correctly. What happened was that overtime began to exceed \$10.00 per hour, and the field had been defined with \$9.99 as the maximum amount. The problem was more than ten years old when it first occurred and was identified. (The original developers of the application were no longer even employed by the company at the time the problem surfaced.)
12. **Reused defects** Between 15 percent and 50 percent of software applications are based on reused code either acquired commercially or picked up from other applications. Due to the lack of certification of reusable materials, many bugs or errors are in reused code. Whether liability should be assigned to the developer or to the user of reused material is ambiguous as of 2009.
13. **Error-prone modules** (IBM term) Studies of IBM software discovered that bugs or defects were not randomly distributed but tended to clump in a small number of places. For example, in the IMS database product, about 35 modules out of 425 were found to contain almost 60 percent of total customer-reported bugs. Error-prone modules are fairly common in large software applications. As a rule of thumb, about 3 percent of the modules in large systems are candidates for being classified as error-prone modules.
14. **Incident** An *incident* is an abrupt stoppage of a software application for unknown reasons. However, when the software is restarted, it operates successfully. Incidents are not uncommon, but their origins are difficult to pin down. Some may be caused by momentary power surges or power outages; some may be caused by hardware problems or even cosmic rays; and some may be caused by software bugs. Because incidents are usually random in occurrence and cannot be replicated, it is difficult to study them.
15. **Security vulnerabilities** These are code segments that are frequently used by viruses, worms, and hackers to gain access to software applications. Error handling routines and buffer overflows are common examples of vulnerabilities. As of 2009, these are not usually classified as defects because they are only channels for malicious attacks. However, given the alarming increase in such attacks, there may be a need to reevaluate how to classify security vulnerabilities.
16. **Malicious software engineers** From time to time software engineers become disgruntled with their colleagues, their managers, or the companies that they work for. When this situation occurs,

some software engineers deliberately insert malicious code into the applications that they are developing. This situation is most likely to occur in the time interval between a software engineer receiving a layoff notice and the actual day of departure. While only a few software engineers cause deliberate harm, the situation may become more prevalent as the recession deepens and lengthens. In any case, the fact that software engineers can deliberately perform harmful acts is one of the reasons why software engineers who work for the Internal Revenue Service have their tax returns examined manually. Of course, not only malicious code can occur, but also other harmful kinds of coding might be used by software engineering employees, such as diverting funds to personal accounts.

17. **Defect potentials** This term originated in IBM circa 1973 and is included in all of my major books. The term *defect potential* refers to the sum total of possible defects that are likely to be encountered during software development. The total includes five sources of defects: (1) requirements defects, (2) design defects, (3) code defects, (4) document defects, and (5) bad fixes or secondary defects. Current U.S. averages for defect potentials are about 5.0 per function point. A rule of thumb for predicting defect potentials is to raise the size of the application in function points to the 1.25 power. This gives a useful approximation of total defects that are likely to occur for applications between about 100 function points and 10,000 function points.
18. **Defect removal efficiency** This term also originated in IBM circa 1973. It refers to the ratio of defects detected to defects present. If a unit test finds 30 bugs out of a total of 100 bugs, it is 30 percent efficient. Most forms of testing are less than 50 percent efficient. Static analysis and formal inspections top 80 percent in defect removal efficiency.
19. **Cumulative defect removal efficiency** This term also originated in IBM circa 1973. It refers to the aggregate total of defects removed by all forms of inspection, static analysis, and testing. If a series of removal operations that includes requirement, design, and code inspections; static analysis; and unit, new function, regression, performance, and system tests finds 950 defects out of a possible 1000, the cumulative efficiency is 95 percent. Current U.S. averages are only about 85 percent. Cumulative defect removal efficiency is calculated at a fixed point in time, usually 90 days after software is released to customers.
20. **Performance issues** Some applications have stringent performance criteria. An example might be the target-seeking guidance system in a Patriot missile; another example would be the embedded software inside antilock brakes. If the software fails to achieve

its performance targets, it may be unusable or even hazardous. However, performance issues are not usually classified as defects because no incorrect code is involved. What is involved are execution paths that are too long or that include too many calls and branches. Even though there may be no overt errors, there are substantial liabilities associated with performance problems.

21. **Cyclomatic and essential complexity** These are mathematical expressions that provide a quantitative basis for judging the complexity of source code segments. The metrics were invented by Dr. Tom McCabe and are sometimes called *McCabe complexity metrics*. Calculations are based on graph theory, and the general formula is “edges – nodes + 2.” Practically speaking, cyclomatic complexity levels less than ten indicate low complexity when the code is reviewed by software engineers. Cyclomatic complexity levels greater than 20 indicate very complex code. The metrics are significant because of correlations between defect densities and cyclomatic complexity levels. Essential complexity is similar, but uses mathematical techniques to simplify the graphs by removing redundancy.
22. **Toxic requirement** This is a new term introduced in 2009 and derived from the financial phrase *toxic assets*. A toxic requirement is defined as an explicit user requirement that is harmful and will cause serious damages if not removed. Unfortunately, toxic requirements cannot be removed by means of regular testing because once toxic requirements are embedded in requirements and design documents, any test cases created from those documents will confirm the error rather than identify it. Toxic requirements can be removed by formal inspections of requirements, however. An example of a toxic requirement is the famous Y2K problem, which originated as a specific user requirement. A more recent example of a toxic requirement is the file handling of the Quicken financial software application. If a backup file is “opened” instead of being “restored,” then Quicken files can lose integrity.

Summary and Conclusions on Software Defects

As discussed earlier in this book, the current U.S. average for software defect volumes is about 5.0 per function point. (This total includes requirements defects, design defects, coding defects, documentation defects, and bad fixes or secondary defects.)

Cumulative defect removal is only about 85 percent. As a result, software applications are routinely delivered with about 0.75 defect per function point. Note that at the point of delivery, all of the early defects in requirements and design have found their way into the code. In other

words, while the famous Y2K problem originated as a requirements defect, it eventually found its way into source code. No programming language was immune, and therefore the Y2K problem was endemic across thousands of applications written in all known programming languages.

For a typical application of 1000 function points, 0.75 released defect per function point implies about 750 delivered defects. Of these, about 20 percent will be high-severity defects: 150 high-severity defects will probably be in the code when users get the first releases.

Five important kinds of remedial actions can improve this situation:

1. Measurement of defect volumes by 100 percent of software organizations.
2. Measurement of defect removal efficiency for every kind of inspection, static analysis, and test stage used.
3. Reducing defect potentials by means of effective defect prevention methods such as joint application design (JAD) and quality function deployment (QFD), and others.
4. Raising defect removal efficiency levels by means of formal inspections, static analysis, and improved testing.
5. Examining the results of quality on defect removal costs and also on total development costs and schedules, plus maintenance costs.

The combination of these five key activities can lower defect potentials down to less than 3.0 defects per function point and raise defect removal efficiency levels higher than 95 percent on average, with mission-critical applications hitting 99 percent.

An achievable goal for the software industry would be to achieve averages of less than 3.0 defects per function point, defect removal efficiency levels of more than 95 percent, and delivered defect volumes of less than 0.15 defect per function point.

The combined results from better measurement, better defect prevention, and better defect removal would reduce delivered defects for a 1000-function point application from 750 down to only 150. Of these 150, only about 10 percent would be high-severity defects. Thus, instead of 150 high-severity defects that normally occur today, only 15 high-severity defects might occur. This is an improvement of a full order of magnitude.

Even better, empirical data indicates that applications at the high end of the quality spectrum have shorter development schedules, lower development costs, and much lower maintenance costs.

Indeed, the main reason for both schedule slippages and cost overruns is because of excessive defect volumes at the start of testing.

Most projects are on schedule and within budget until testing starts, at which time excessive defects stretch out testing by several hundred percent compared with plans and cost estimates.

The technologies to achieve better quality results actually exist today in 2009, but are not widely deployed. That means that better awareness of quality and the economic value of quality are critical weaknesses of the software industry circa 2009.

Preventing and Removing Defects from Application Source Code

During development of software applications, the approximate average number of defects encountered averages about 1.75 per function point or 17.5 per KLOC for languages where the ratio of lines of code to function points is about 100. As pointed out earlier in this book, defect volumes vary by the level of the programming languages, and they also vary by the experience and skill of the programming team.

The minimum quantity of defects in source code will be about 0.5 per function point or 5 per KLOC, while the maximum quantity will top 3.5 defects per function point or 35 defects per KLOC, assuming the same level of programming language.

However, in spite of wide ranges of potential defects, there are still more coding defects than any other kind of defect. Defect removal efficiency against coding defects is in the range of 80 percent to 99 percent. Some coding defects will slip through even in the best of cases, although it is certainly better to approach 99 percent than it is to lag at 80 percent.

For coding defects as with all other defect sources, two channels need to be included in order to improve code quality:

1. *Defect prevention*, or methods that can lower defect potentials.
2. *Defect removal*, or methods that can seek out, find, and eliminate coding defects.

The available forms of defect prevention for coding defects include certified reusable code modules, use of patterns or standard coding approaches for common situations, use of structured programming methods, use of higher-level programming languages, constructing prototypes prior to formal development, dividing large applications into small segments (as does Agile development), participation in code inspections, test-based development, and usage of static analysis tools. Pair programming is also reported to have some efficacy in terms of defect prevention, but this method has very low usage and very little data.

The available forms of defect removal for coding defects include desk checking, pair programming, debugging tools, code inspections, static analysis tools, and 17 kinds of conventional testing plus automated unit testing and regression testing.

Defect removal by individual software engineers is difficult to study. Desk checking, debugging, and unit testing are usually private activities with no observers and no detailed records kept. Most corporate defect-tracking systems do not start to collect data until public defect removal begins with formal inspections, function tests, and regression tests. What happens before these public events is usually invisible. There are some exceptions, however.

At one point, IBM asked for volunteers who were willing to record the numbers of bugs they found in their own code by themselves. The purpose of the study was to find out what was the actual defect removal efficiency from these normally invisible forms of defect removal. Obviously, the data was not used in any punitive fashion and was kept confidential, other than to produce some statistical reports.

More recently the Personal Software Process (PSP) and Team Software Process (TSP) methods developed by Watts Humphrey have also included defect recording throughout the code development cycle.

Unfortunately, the Agile development method has moved in the other direction and usually does not record private defect removal. Indeed, many Agile projects do not record defect data at all, which is a mistake because it reduces the ability of the Agile method to prove its value in terms of quality.

The public forms of defect removal are discussed in this book in Chapter 9, which deals with quality. The emphasis in this chapter is more on the private forms of defect removal, which are seldom covered in the software engineering literature.

Private defect removal lacks the large volumes of data associated with some of the public forms such as formal inspections, static analysis, and the test stages that involve other players such as test specialists and software quality assurance. But for the sake of completeness, the topics of private defect prevention and private defect removal need to be included.

Before discussing the effectiveness of either defect prevention or defect removal, it should be noted that individual software engineers or programmers vary widely in experience and skills.

In one controlled study at IBM where a number of programmers were asked to implement the same trial example, the quantity of code produced varied by about 6 to 1 between the bulkiest solution and the most concise solution for the same specification.

Similar studies showed about a 10 to 1 variation in the amount of time a sample of programmers needed to code and debug a standard problem statement.

These wide variations in individual performance mean that individual human variations in a population of software engineers probably account for more divergence in results than do methods, tools, or factors that can be studied objectively.

Forms of Programming Defect Prevention

It is much more difficult to measure or quantify defect prevention than it is to measure defect removal. With defect removal, it is possible to accumulate statistics on numbers of defects found and their severity levels.

Once the project is released to customers, defect counts continue. After 90 days of usage, it is possible to combine the internally discovered defects with the customer-reported defects and to calculate defect removal efficiency. If development personnel found 85 defects and customers reported 15 defects, the removal efficiency is 85 percent. Such data is easy to collect, valuable, and fairly accurate, except for some invisible defects found via private removal actions such as desk checking and unit test.

For defect prevention, there is no easy way to measure the *absence* of defects. The methods available for exploring defect prevention require collecting data from a fairly large number of projects, where some of them utilized a specific defect prevention method and others did not.

For example, assume you measure a sample of 50 projects that used structured coding methods and another 50 projects that did not use structured programming methods. Assume the 50 projects that used structured programming averaged 10 coding defects per KLOC or 1 per function point. Assume the 50 projects that did not use structured programming averaged 20 coding defects per KLOC or 2 per function point. This kind of analysis allows you to make a hypothesis that the structured coding prevents about 50 percent of coding defects, but it is still only a hypothesis and not proof.

Further, real-life situations are seldom simple and easy to deal with. There may be numerous other factors at play, such as usage of static analysis, usage of higher-level languages, usage of inspections, variations in programming experience, complexity of the problems, and so forth.

The many different factors that can influence defect prevention mean that exact knowledge of the effectiveness of any specific factor is somewhat subjective at best, and will probably stay that way.

Academic institutions can perform controlled experiments with students where they measure the effectiveness of a single variable, but such studies are fairly rare concerning defect prevention.

However, from long-range observations involving hundreds of software personnel and hundreds of software projects over a multi-year

time span, some objective factors about defect prevention have reasonably strong support:

Code reuse as defect prevention If reusable code is available that has been certified to zero-defect levels, or at least carefully inspected, tested, and subjected to static analysis before being made reusable, this is the best known form of defect prevention. Defect potentials in certified reusable code modules are only a fraction of the 15 per KLOC normally encountered during custom development; sometimes only about 1/100th as many defects are encountered.

However, and this is an important point, using uncertified reusable code can be both hazardous and expensive. If the defect potentials in uncertified reusable code are more than about 1 per KLOC, and the reused code is plugged into more than ten different applications, the combined debugging costs will be so high that this example of reuse would have a negative return on investment.

Although certified reuse is the most effective form of defect prevention and counts as a best practice, it is also the rarest. Uncertified sources of reuse outnumber certified sources by at least 50 to 1. Reuse of certified code and other materials would class as a best practice. But reuse of materials that are uncertified must be classed as a hazardous practice.

It is much harder for software engineers to debug someone else's unfamiliar code than it is to debug their own. Every single time a reused code module is utilized for a new application, there is a good chance that the same errors will be encountered. Thus, uncertified reuse is hazardous and can be more expensive than custom development of the same module—hence, the reason the uncertified reuse can have a significant negative return on investment (ROI).

Code reuse comes from many sources, including commercial vendors, legacy applications, object-oriented class libraries, corporate reuse libraries, public-domain and open-source libraries, and a number of others. While reusable code is fairly plentiful, something that is not plentiful is data on the repair frequencies of reusable materials. (See the section on certifying reusable materials earlier in this book for additional information.)

As mentioned elsewhere in the book, code reuse by itself is only part of the reusability picture. Reusable designs, data structures, test cases, tutorial information, work breakdown structures, and HELP text are also reusable and should be packaged together with the code they support.

Patterns as defect prevention Programmers and software engineers who have developed large numbers of software applications tend to be aware that certain sequences of code occur many times in many applications. Some of these sequences include validating inputs to ensure that error

conditions such as having character data entered into a numeric field is rejected, or that text and numeric strings do not contain more characters than specified by the application's design.

Patterns gained via personal experience are of course reusable even if informal and personal. However, it has become clear that this kind of knowledge occurs so often that it could be written down, illustrated graphically, and then used to train new software engineers as they learn trade craft.

Pattern-based development has the potential of lowering defect potentials of young and inexperienced developers by more than 50 percent. Once standard patterns are widely published and available, they can also serve to facilitate career changes from one kind of software to another. For example, there are very different kinds of patterns associated with embedded applications than with information technology applications.

What is lacking for pattern-based development circa 2009 is an effective taxonomy that can be used to catalog the patterns and aid in selecting the appropriate set of patterns. Also, there is no exact knowledge of how many patterns are likely to be useful and valuable. In the future, pattern usage will no doubt be classed as a best practice, although doing so in 2009 is probably a few years premature.

Individual software engineers working in a narrow range of applications probably utilize from 25 to 50 common patterns centering in input and output validation, error handling, and perhaps security-related topics. But when all types and forms of software are included, such as financial applications, embedded applications, web applications, operating systems, compilers, avionics, and so on, the total number of useful patterns could easily top 1000. This is too large a number to be listed randomly, so patterns need to be organized if they are to become useful tools of the trade.

Inspections as defect prevention Participation in formal inspections turns out to be equally effective as a defect-prevention method and a defect-removal method. Participants in formal inspections spontaneously avoid making the kinds of mistakes that are found during the inspection sessions. Therefore, after participating in a number of inspections, coding defects tend to be reduced by more than 80 percent compared with the volumes encountered prior to starting to use inspections. As a result, formal inspections get double counted as best practices: they are highly effective for both defect prevention and defect removal.

Inspections turn out to be so effective in terms of defect prevention that long-range usage of inspections has a tendency to become boring for the participants due to a lack of interesting bugs or defects after about a year of inspections. (Unfortunately, some companies stop using inspections, so defect volumes begin to creep upwards again.)

One other useful aspect of inspections is that when novices inspect the work of experts, they spontaneously learn improved programming skills. Conversely, when experts inspect the work of novices, they can provide a great deal of useful advice as well as find a great many bugs or defects. Therefore, it is useful to have several experts or top software engineers as participants in inspections.

Automated static analysis as defect prevention Static analysis is a fairly new technology that is distinct from testing. Automated static analysis tools have embedded rules and logic that are set up to discover common forms of defects in source code. These tools are quite effective and have defect removal efficiency levels that top 85 percent. A caveat is that only about 50 languages out of 2500 are supported, and these are primarily modern languages such as C, C#, C++, Java, and the like. Older and obscure languages such as MUMPS, Coral, Chill, and the like are not supported. However, with almost 100 static analysis tools available, there are tools that can handle some older or specialized languages such as ABAP, Ada, COBOL, and PL/I. Some of the tools have extensible rules, so in theory all of the 2500 languages in existence might gain access to static analysis, although this is unlikely to occur.

Because static analysis tools are effective at finding bugs in source code, and the static analysis tools are usually run by programmers, they have a double benefit of also acting as defect prevention agents. In other words, programmers who carefully respond to the defects identified by automated static analysis tools will spontaneously avoid making the same defects in the future.

As of 2009, usage of static analysis counts as a best practice for supported programming languages. The evidence is already significant for defect removal and is increasing for defect prevention.

Static-analysis tools are widely used by the open-source development community with good results. Due to the power and utility of static analysis, usage is expanding and this method should become a standard activity; in fact, static analysis should be included in every programming development and maintenance environment and should be a normal part of all development and maintenance methodologies.

Test-based development (TBD) as defect prevention The extreme programming (XP) method includes developing test cases prior to developing source code. Indeed, the test cases are used as an adjunct to the requirements and design of software applications.

This method of early test-case development focuses attention on quality, and therefore TBD gets double credit as a best practice for both defect prevention and defect removal. Because TBD is fairly new, empirical

data based on large numbers of trials is not yet available. The rather lax measurement practices of the Agile community add to the problem of ascertaining the actual effectiveness of TBD.

However, from anecdotal evidence, it appears that TBD may reduce defect potentials by perhaps 30 percent and raise unit test defect removal efficiency from around 35 percent up to perhaps 50 percent. Both results are steps in the right direction, but additional data on TBD is needed. TBD is a candidate for a best practice and no doubt will be classed as one when additional quantitative data becomes available.

High-level languages as defect prevention One of the claimed advantages of high-level programming languages is that they reduce defect potentials. A related claim is that if defects do occur, they are easier to find. Both claims appear to be valid, but the situation is somewhat complicated, and there are exceptions to general rules about the effectiveness of high-level languages.

Any reduction in source code volumes will obviously reduce chances for errors. If a specific function requires 1000 lines of code in assembly language, but can be done with only 150 Java statements, the odds are good that fewer defects will occur with Java. Even if both versions have a constant ten bugs per KLOC, the larger assembly version might have 10 bugs, while the smaller Java version might have only 1 or 2.

However, some high-level programming languages have fairly complex syntax and therefore make it easy to introduce errors by accident. The APL programming language is an example of a language that is very high level, but also difficult to read and understand, and therefore difficult to debug, and especially so if the person attempting to debug is not the original programmer.

Observations indicate the languages with regular syntax, mnemonic labels, and commands that are amenable to human understanding will have somewhat fewer coding defects than languages of the same level, but with arcane commands and complicated syntax that include many nested commands.

What would be useful and interesting would be controlled studies by academic institutions that measured both defect densities and debugging times for implementing standard problems in various languages. It would be very interesting to see defect volumes and debugging times compared for popular languages such as C, C#, C++, Objective C, Java, JavaScript, Lua, Ruby, Visual Basic, and perhaps 50 more. However, as of 2009, this kind of controlled study does not seem to exist.

As of 2009, the plethora of programming languages and their negative impact on maintenance costs make best practice status for any specific language somewhat questionable.

Prototypes as defect prevention For large and complex applications, it may be necessary to try out a number of alternative code sequences before selecting a best-case alternative for the final versions. Prototypes are useful in reducing defects in the final version by allowing software engineers to experiment with alternatives in a benign fashion.

As a general rule prototypes are created mainly for the most troublesome and complicated pieces of work. As a result, the size of typical prototypes is only about 5 percent to perhaps 10 percent of the size of the total application. This practice of concentrating on the toughest problems makes prototypes useful, and their compact size keeps them from getting to be expensive in their own right.

Prototypes come in two flavors: disposable and evolutionary. As the name implies, disposable prototypes are used to try out algorithms and code sequences and then discarded. Evolutionary prototypes grow into the finished application.

Because prototypes are usually developed at high speed in an experimental fashion, the disposable prototypes are somewhat safer than evolutionary prototypes. Prototypes may contain more bugs or defects than polished work, and attempting to convert them into a finished product may lead to higher than expected bug counts.

Disposable prototypes used to try out alternative solutions or to experiment with difficult programming problems would be defined as best practices. However, evolutionary prototypes that are carelessly developed in the interest of speed are not best practices, but instead somewhat hazardous.

Code structure as defect prevention Professor Edsger Dijkstra published one of the most famous letters in the history of software engineering entitled “Go-to statements considered harmful.” The letter to the editor was published in August 1968 in *The Communications of the ACM*.

The thesis of this letter was that excessive use of branches or “go to” statements made the structure of software applications so complex that errors of incorrect branch sequences might occur that were very difficult to identify and remove.

This letter triggered a revolution in programming style that came to be known as *structured programming*. Under the principles of structured programming, branches were reduced and programmers began to realize that complex loops and clever coding sequences introduced bugs and made the code harder to test and validate.

As it happens another pioneering software engineer, Dr. Tom McCabe, developed a way of measuring code structure that was published in December 1976 in *IEEE Transactions on Software*. The measures developed by Dr. McCabe were those of “cyclomatic complexity” and “essential complexity.”

Cyclomatic complexity is based on graph theory and is a formal way of evaluating the complexity of a graph that describes the flow of control through a software application. The formula for calculating cyclomatic complexity is “edges – nodes + two.”

Essential complexity is also based on graph theory, only it eliminates redundant or duplicate paths through code.

In terms of cyclomatic complexity, a code segment with no branches has a complexity score of 1, which indicates that the code executes in a linear fashion with no branches or go-to statements. From a psychological standpoint, cyclomatic complexity levels of less than 10 are usually perceived as being well structured. However, as cyclomatic complexity levels rise to greater than 20, the code segments become increasingly difficult to understand or to follow from end to end without errors.

There is some empirical evidence that code with cyclomatic complexity levels of less than 10 have only about 40 percent as many errors as code with cyclomatic complexity levels greater than 20. Code with a cyclomatic complexity level of 1 seems to have the fewest errors, if other factors are held constant, such as the programming languages and the experience of the developer.

One interesting study in IBM found a surprising result: that code defects were sometimes higher for the work of senior or experienced programmers compared with the same volume of code written by novices or new programmers. However, the actual cause of this anomaly was that the experts were working on very difficult and complex applications, while the novices were doing only simple routines that were easy to understand. In any case, the study indicated that problem difficulty has a significant impact on defect density levels.

The importance of cyclomatic and essential complexity on code defects led to the development of a number of commercial tools. Many tools available circa 2009 can calculate cyclomatic and essential complexity of code in a variety of languages.

In the 1980s, several tools on the market were aimed primarily at COBOL and not only evaluated code complexity, but also could automatically restructure the code and reduce both cyclomatic and essential complexity. These tools asserted, with some evidence to back up the assertions, that the revised code with low complexity levels could be modified and maintained with less effort than the original code.

Use of structured programming techniques and keeping cyclomatic complexity levels low would both be viewed as best practices. Code with low complexity levels and few branches tends to have fewer defects, and the defects that are present tend to be easier to find. Therefore, structured programming counts as a best practice for defect prevention.

Segmentation as defect prevention More than 50 years of empirical data has proven conclusively that defect potentials correlate almost perfectly

with application size measured using both lines of code and function points. Because size and defects are closely coupled, it is reasonable to ask, *Why not decompose large systems into a number of smaller segments?*

Unfortunately, this is not as easy as it sounds. To make an analogy, since constructing an 80,000-ton cruise ship is known to be expensive, why not decompose the ship into 80,000 small boats that are cheap to build? Obviously, the features and user requirements of 80,000 small boats are not the same as those of one large 80,000-ton cruise ship.

As of 2009, there are no proven and successful methods for segmenting or decomposing large systems into small independent components. As it happens, the Agile method of dividing a system into segments or *sprints* that can be developed sequentially has shown itself to be fairly successful. But most of the Agile applications are below 10,000 function points and are comparatively simple in architecture.

There have not yet been any Agile projects that tackle something of the size of Microsoft Vista at about 150,000 function points or a large ERP package at perhaps 300,000 function points. Indeed, if Agile sprints were used for these applications and team sizes were in the range of average Agile projects (less than ten people) then probably 150 sprints would be needed for Vista and 300 would be needed for an ERP package. Assuming one month per sprint, the schedule would be perhaps 12 years for Vista and 25 years for the ERP package. Multiple teams would speed things up, but interfaces between the code of each team would add complexity and also add defects.

The bottom line is that segmentation into small independent packages or components is effective when it can be done well, but not always possible given the feature sets and architecture of many large systems. Thus best practice status cannot be assigned to segmentation as of 2009, due to the lack of standard and effective methods for segmentation.

For large applications, segmentation is most common for major features, but each of these features may themselves be in the range of 10,000 function points or more. There is not yet any proven way to divide a massive system of 150,000 function points or 15 million lines of code into perhaps 15,000 small independent pieces. About the best that occurs circa 2009 is to divide these massive systems into perhaps ten large segments.

Methodologies and measurements as defect prevention The Personal Software Process (PSP) and Team Software Process (TSP) developed by Watts Humphrey feature careful recording of all defects found during development, including the normally invisible defects found privately via desk checking and unit testing.

The act of recording specific defects tends to embed them in the minds of software engineers and programmers. The result is that after several projects in succession, coding defects decline by perhaps 40 percent since they are spontaneously avoided.

Measurements and methodologies are therefore useful in terms of defect prevention because they tend to focus attention on defects and so trigger reductions over time. The methods that record defects and focus on quality are classed as best practices.

One unusual aspect of TSP is that the results seem to improve with application size. In other words, TSP operates successfully for large systems in excess of 10,000 function points. This is a fairly rare occurrence among development methods.

Pair programming as defect prevention The idea of pair programming is for two software engineers or programmers to share one workstation. They take turns coding, and the other member of the pair observes the code and makes comments and suggestions as the coding takes place. The pair also has discussions on alternatives prior to actually doing the code for any module or segment.

The method of pair programming has some experimental data that suggests it may be effective in terms of both defect removal and defect prevention. However, the pair programming method has so little usage on actual software projects that it is not possible to evaluate these claims as of 2009 on large-scale applications.

On the surface, pair programming would seem to come very close to doubling the effort required to complete any given code segment. Indeed, due to normal human tendencies to chat and discuss social topics, there is some reason to suspect that pair programming would be more than twice as expensive as individual programming.

Until additional information becomes available from actual projects rather than from small experiments, there is not enough data to judge the impact of pair programming in terms of defect removal or defect prevention.

Other methods as defect prevention The methods cited earlier in this chapter have been used enough so that their effectiveness in terms of code defect prevention can be hypothesized. Other methods seem to have some benefits in terms of defect prevention, but they are harder to judge. One of these methods is Six Sigma as it applies to software. The Six Sigma approach does include measurements of defects and analysis of causes. However, Six Sigma is usually a corporate approach that is not applied to specific projects, so it is harder to evaluate. Other code defect prevention techniques that may be beneficial but for which the

author has no solid data include quality function deployment (QFD), root-cause analysis, the Rational Unified Process (RUP), and many of the Agile development variations.

Combinations and synergies among defect prevention methods Although the methods cited earlier may occur individually, they are often used in combinations that sometimes appear synergistic. For example, structured coding is often used with TSP, with inspections, and with static analysis.

The most frequent combination is the pairing of high-level programming languages with the concepts of structured programming. The combination that tends to yield the highest overall levels of defect prevention would be methodologies such as TSP teamed with high-level programming languages, certified reusable code, patterns, prototypes, static analysis, and inspections.

Overriding all other aspects of defect prevention and defect removal, individual experience and skill levels of the software engineers continue to be a dominant factor. However, as of 2009, the software engineering field lacks standard methods for evaluating human performance; it has no licensing or certification, no board specialties, and no methods of judging professional malpractice. Therefore, expertise among software engineers is important but difficult to evaluate.

Summary of Observations on Defect Prevention

Because of the difficulty and uncertainty of measuring defect prevention, the suite of defect prevention methods lacks the large volumes of solid statistical data associated with defect removal.

Personal defect prevention is especially difficult to study because most of the activities are private and therefore seldom have records or statistical information available, other than data kept by volunteers.

Long-range measurements over time and involving hundreds of applications and software engineers give some strong indications of what works in terms of defect prevention, but the results are still less than precise and will probably stay that way.

Forms of Programming Defect Removal

There is very good data available on the public forms of defect removal such as formal inspections, function test, regression test, independent verification and validation, and many others. But private defect removal is another story. The phrase *private defect removal* refers to activities that software engineers or programmers perform by themselves without witnesses and usually without keeping any written records.

The major forms of private defect removal include, but are not limited to:

1. Desk checking
2. Debugging using automated tools
3. Automated static analysis
4. Subroutine testing
5. Unit testing (manual)
6. Unit testing (automated)

Since most of these defect removal methods are used in private, data to judge their effectiveness comes from either volunteers who keep records of bugs found, or from practitioners of methods that include complete records of all defects, such as PSP and TSP.

Automated static analysis is a method that happens to be used both privately by individual programmers on their own code, and also publicly by open-source developers who are working collaboratively on large applications such as Firefox, Linux, and the like. Therefore, static analysis has substantial data available for its public uses, and it can be assumed that private use of static analysis will be equally effective.

Desk checking for defect removal In the early days of programming and computing, the time lag between writing source code and getting it assembled or compiled was sometimes as much as 24 hours. When program source code was punched into cards and the cards were then put in a queue for assembly or compilation, many hours would go by before the code could be executed or tested.

In these early days of programming between the late 1960s and the 1970s, desk checking or carefully reading the listing of a program to look for errors was the most common method of personal defect removal. Desk checking was also a technical necessity because errors in a deck of punch cards could stop the assembly or compilation process and add perhaps another 24 hours before testing could commence.

Today in 2009, code segments can be compiled or interpreted instantly, and can be executed instantly as well. Indeed, they can be executed using programming environments that include debugging tools and automated static analysis. Therefore, desk checking has declined in frequency of usage due to the availability of personal workstations and personal development environments.

Although there is not much in the way of recent data on the effectiveness of desk checking, historical data from 30 years ago indicates about 40 percent to just over 60 percent in terms of defect removal efficiency levels.

Today in 2009, desk checking is primarily reserved for a small subset of very tricky bugs or defects that have not been successfully detected and removed via other methods. These include security vulnerabilities, performance problems, and sometimes toxic requirements that have slipped into source code. These are hard to detect via static analysis or normal testing because they may not involve overt code errors such as branches to incorrect locations or boundary violations.

These special and unique bugs compose only about 5 percent of total numbers of bugs likely to be found in software applications. Desk checking is actually close to 70 percent in dealing with these very troublesome bugs that have eluded other methods. (The reason that desk checking is not higher is because sometimes software engineers don't realize that a particular code practice is wrong. This is why proofreading of manuscripts is needed. Authors cannot always see their own mistakes.)

While these subtle bugs can be detected using formal inspections, formal inspections do not occur on more than about 10 percent of software applications and require between three and eight participants. Desk checking, on the other hand, is a one-person activity that can be performed at any time with no formal preparation or training.

Desk checking in 2009 is a supplemental method that may not be needed for every software project. It is effective for a number of subtle bugs and might be viewed as a best practice on an as-needed basis.

Automated debugging for defect removal Software engineers and programmers circa 2009 have access to hundreds of debugging tools. These tools normally support either specific programming languages such as Java and Ruby or specific operating systems such as Linux, Leopard, Windows Vista, and many others. In any case, a great many debugging tools are available.

The features of debugging tools vary, but all of them allow the execution of code to be stopped at various places; they allow changes to code; and they may include features to look for common problems such as buffer overflows and branching errors. Beyond that, the specialized debugging tools have a number of special features that are relevant to specific languages or operating systems.

Debugging tools are so common that usage is a standard practice and therefore would be classed as a best practice. That being said, none are 100 percent effective, and quite a few bugs can escape. In fact, given the numbers of bugs found later via inspections, static analysis, and testing, the average efficiency of program debugging is only about 30 percent or less.

Automated static analysis for defect removal Static analysis tools examine source code and the paths through the code and look for common errors.

Some of these tools have built-in sets of rules, while others have extensible rule sets.

A keyword search of the Web using “automated static analysis” turns up more than 100 such tools including Axivion, CAST, Coverity, Fortify, GrammaTeck, Klocwork, Lattix, Ounce, Parasoft, ProjectAnalyzer, ReSharper, SoArc, SofCheck, Viva64, Understand, Visual Studio Team System, and XTRAN.

Individually, each static analysis tool supports up to 30 languages. For common languages such as Java and C, dozens of static analysis tools are available; for older languages such as Ada, Jovial, and PL/I, there are only a few static analysis tools. For very specialized languages such as ABAP used for writing code in SAP environments, there are only one or two static analysis tools.

Without doing an exhaustive search, it appears that out of the current total of 2500 programming languages developed to date, static analysis tools are available for perhaps 50 programming languages. However, some of these static analysis tools support extensible rules, so it is theoretically possible to create rules for examining all of the 2500 languages. This is unlikely to occur, due to economic reasons for obscure languages or those not used for business or scientific applications.

As a class, static analysis tools seem to be effective and can find perhaps 85 percent of common programming errors. Therefore, usage of static analysis tools can be viewed as a best practice; rapidly becoming a standard practice, too.

However, static analysis tools only find coding problems and do not find toxic requirements, performance problems, user interface problems, and some kinds of security vulnerabilities. Therefore, additional forms of defect removal are needed.

Some static analysis tools provide additional features besides defect detection. Some are able to assist in translating older languages into newer languages, such as turning COBOL into Java if desired.

It is also possible to raise the level of static analysis and examine the meta-languages underlying several forms of requirements and design documentation such as those created via the unified modeling language (UML). Indeed, it is theoretically possible to use a form of extended static analysis to create test suites.

Because static analysis and formal code inspections usually find many of the same kinds of bugs, normally either one form or the other is utilized, but not both. Static analysis and inspections have roughly the same levels of defect removal efficiency, but static analysis is cheaper and quicker. However, code inspections can find more subtle problems such as performance issues or security vulnerabilities. These are not code “bugs” per se, but they do cause trouble.

If static analysis and code inspections are both utilized, which occurs for mission-critical applications such as some medical instruments and

some kinds of security and military software, static analysis would normally come before code inspections.

A small number of issues identified by static analysis tools turn out to be false positives, or code segments identified as bugs which turn out to be correct. However, a few false positives is a small price to pay for such a high level of defect removal efficiency.

Subroutine testing for defect removal Testing comes in many flavors and covers many different sizes of code volumes. The phrase *subroutine testing* refers to a small collection of perhaps up to ten source code instructions that produces an output or performs an action that needs to be verified. Subroutine testing is usually the lowest level of testing in terms of code volumes.

By contrast, unit testing would normally include perhaps 100 instructions or more, while the “public” forms of testing such as function testing and regression testing may deal with thousands of instructions.

As the volume of source code increases, paths through the code increase, and therefore more and more test cases are needed to actually cover 100 percent of the code. Indeed, for very large systems, 100 percent coverage appears to be impossible, or at least very rare.

Subroutine testing is a standard practice and also a best practice because it eliminates a significant number of problems. However, the defect removal efficiency of subroutine testing is only 30 percent to perhaps 40 percent. This is because the code volumes are too small for detecting many kinds of bugs such as branching errors.

Subroutine testing may or may not use actual formal test cases. The usual mode is to execute the code and check the outputs for validity. Subroutine test cases, if any, are normally disposable.

Manual unit testing for defect removal Unit testing of complete modules is the largest form of testing that is normally private or carried out by individual programmers without the involvement of other personnel such as test specialists or software quality assurance.

Manual unit testing is the first and oldest kind of formal testing. Indeed, in the 1960s and early 1970s, when many applications only contained 100 code statements or so, unit testing was often the only form of testing performed.

The phrase *unit testing* refers to testing a complete module of perhaps 100 code statements that performs a discrete function with inputs, outputs, algorithms, and logic that need to be validated.

Unit testing can combine “black box” testing and “white box” testing. The phrase *black box* means that the internal code of a module is hidden, so only inputs and outputs are visible. Black box testing therefore tests input and output validity. The phrase *white box* means that internal code is revealed, so branches and control flow through

an application can be tested. Combining the two forms of testing should in theory test everything. However, code coverage seldom hits 100 percent, and for large applications that are high in cyclomatic complexity it may drop below 50 percent.

Unit testing tends to look at limits, ranges of values, error-handling, and security-related issues. Unfortunately, unit testing is only in the range of perhaps 30 percent to 50 percent efficient in finding bugs. For example, unit testing is not able to find many performance-related issues because they typically involve longer paths and multiple modules.

For modules that tend to include a number of branches or complex flows, unit testing begins to encounter problems with test coverage. As cyclomatic complexity levels go up, it takes more and more test cases to cover every path. In fact, 100 percent coverage almost never occurs when cyclomatic complexity levels get above 5, even for modules with only 100 code statements.

Unit testing is a standard activity for software engineering and therefore counts as a best practice in spite of the somewhat low defect removal efficiency. Without unit testing, the later stages of testing such as function testing, stress testing, component testing, and system testing would not be possible.

The test cases created for unit testing are normally placed in a formal test library so that they can be used later for regression testing. Since the test cases are going to be long-lived and used repeatedly, they need proper identification as to what applications and features they test, what functions they test, when they were created, and by whom. There will also be accompanying test scripts that deal with invoking and executing the test cases. The specifics of formal test case design are outside the scope of this book, but such topics are covered in many other books.

Unit testing can be used in conjunction with other forms of defect removal such as formal code inspections and static analysis. Usually, static analysis would be performed prior to unit testing, while code inspections would be performed after unit testing. This is because static analysis is quick and inexpensive and finds many bugs that might be found via unit testing. Unit testing is done prior to code inspections for the same reason; it is faster and cheaper. However, code inspections are very effective at finding subtle issues that elude both static analysis and unit testing, such as security vulnerabilities and performance issues.

Using code inspections, static analysis, and unit testing for the same code is a fairly rare occurrence that most often occurs on mission-critical applications such as weapons systems, medical instruments, and other software applications where failure might cause death or destruction.

Manual unit testing was a normal and standard activity for more than 40 years and is still very widespread. However, performance of units varies from “poorly performed” to “extremely good.” Because of the inconsistencies

in methods of carrying out unit testing and in testing results, the ranges are too wide to say that unit testing per se is a best practice. Careful unit testing with both black box and white box test cases and thoughtful consideration to test coverage would be considered a best practice. Careless unit testing with hasty test cases and partial coverage would rank no better than marginally adequate and would not be a best practice.

Testing is a teachable skill, and there are many classes available by both academia and commercial test companies. There are also several forms of certification for test personnel. It would be useful to know if formal test training and certification elevated test defect removal efficiency by significant amounts. There is considerable anecdotal evidence that certification is beneficial, but more large-scale surveys and studies are needed on this topic.

Automated unit testing for defect removal While manual unit testing has been part of software engineering since the 1960s, automated unit testing is newer and started to occur only in the 1980s in response to larger and more complex applications plus the arrival of graphical user interfaces (GUI), which greatly expanded the nature of software inputs and outputs.

The phrase “automated unit testing” is somewhat ambiguous circa 2009. The most common usage of the term implies manual creation of unit test cases combined with a framework or scaffold that allows them to be run automatically on a regular basis without explicit actions by software engineers.

Automated unit testing has been adopted by the Agile and extreme programming (XP) communities together with the corollary idea of creating test cases before creating code. This combination seems to be fairly effective in terms of defect removal and also pays off with improved defect prevention by focusing the attention of software engineers on quality topics.

The phrase *automated unit testing* deals mainly with test case execution and recording of defects that are encountered: most of the test cases are still created by hand. However, it is theoretically possible to envision automated test case creation as well.

Recall from Chapter 7 that during requirements gathering and analysis, seven fundamental topics and 30 supplemental topics need to be considered. As it happens, these same 37 issues also need to be tested. A form of static analysis elevated to execute against requirements and specification meta-languages should, in theory, be able to produce a suite of test cases as a byproduct.

Some forms of test automation are aimed at web applications; others are aimed at embedded applications; and still others are aimed at information technology products. Automated testing is an emerging technology that as of 2009 is still rapidly evolving.

There is a shortage of solid empirical data that compares automated unit testing and manual unit testing in a side-by-side fashion for applications of similar size and complexity. Anecdotal information gives an edge to automated testing for speed and convenience. However, the most critical metric for testing is that of defect removal efficiency. As this book is written, there is not enough solid data that compares automated unit testing to the best forms of manual unit testing to judge whether automated unit tests have higher levels of defect removal efficiency than manual unit tests.

As additional data becomes available, there is a good chance that automatic unit testing will enter the best practice class. As of 2009, the data shows some effort and cost benefits, but defect removal efficiency benefits remain uncertain.

Defect removal for legacy applications About 40 percent of the software engineers in the world are faced with performing maintenance on aging legacy applications that they did not create themselves. Although the legacy applications may be old, they are far from trouble free, and they still contain latent bugs or defects.

This situation brings up a number of questions about defect removal for legacy code where the original developers are gone, the specifications may be missing or out of date, comments may be sparse or incorrect, regression tests are of unknown completeness, and the code itself may be in a dead language or one the current maintenance team has not used.

Fortunately, a number of companies and tools have addressed the issues of maintaining aging legacy code. Some of these companies have developed “maintenance workbenches” that include features such as:

1. Automated static analysis
2. Automated test coverage analysis
3. Automated function point calculations
4. Automated cyclomatic and essential complexity calculations
5. Automated debugging support for many (but not all) languages
6. Automated data mining for business rules
7. Automated translation from dead languages to newer languages

With aging legacy applications being written in as many as 2500 different programming languages, no single tool can provide universal support. However, for legacy code written in the more common languages such as Ada, COBOL, C, PL/I, and the like, a number of maintenance tools are available.

Usage of maintenance workbenches as a class counts as a best practice, but there are too many tools and variations to identify specific

workbenches. Also, these tools are evolving fairly rapidly, and new features occur frequently.

Synergies and combinations of personal defect removal The methods discussed in this section are used in combination rather than alone. Debugging, automated static analysis, and unit testing form the most common combination. The combined effectiveness of these three methods can top 97 percent in terms of defect removal efficiency when performed by experienced software engineers. The combined results can also drop below 85 percent when performed by novices.

Summary and Conclusions on Personal Defect Removal

Although personal defect removal activities are private and therefore difficult to study, they have been the frontline of defense against software defects for more than 50 years. That being said, the fact that software defects emerge and are still present when software is delivered indicates that none of the personal defect removal methods are 100 percent effective.

However, some of the newer defect removal tools such as automated static analysis are improving the situation and adding rigor to the suite of personal defect removal tools and methods.

Since individual software engineers can keep records of the bugs they find, it would be useful and valuable if personal defect removal efficiency levels could be elevated up to more than 90 percent before the public forms of defect removal begin.

Personal defect removal will continue to have a significant role as software engineering evolves from a craft to a true engineering discipline. Knowing the most effective and efficient ways for preventing and removing defects is a sign of software engineering professionalism. Lack of defect measures and unknown levels of defect removal efficiency imply amateurishness; not professionalism.

Economic Problems of the “Lines of Code” Metric

Introduction

Any discussion of programming and code development would be incomplete without considering the famous lines of code (LOC) metric, which has been used to measure both productivity and quality since the dawn of the computer era.

The LOC metric was first introduced circa 1960 and was used for economic, productivity, and quality studies. At first the LOC metric was reasonably effective for all three purposes.

As additional higher-level programming languages were created, the LOC metric began to encounter problems. LOC metrics were not able to measure noncoding activities such as requirements and design, which were becoming increasingly expensive.

These problems became so severe that a controlled study in 1994 that used both LOC metrics and function point metrics for ten versions of the same application coded in ten languages reached an alarming conclusion: LOC metrics violated the standard assumptions of economic productivity so severely that using LOC metrics for studies involving more than one programming language constituted professional malpractice!

Such a strong statement cannot be made without examples and case studies to show the LOC problems. Following is a chronology of the use of LOC metrics that shows when and why the metric began to cease being useful and start being troublesome. The chronology runs from 1960 to the present day, and it projects some ideas forward to 2020.

Lines of Code Metrics Circa 1960

The lines of code (LOC) metric for software projects was first introduced circa 1960 and was used for economic, productivity, and quality studies. The economics of software applications were measured using “dollars per LOC.” Productivity was measured in terms of “lines of code per time unit.” Quality was measured in terms of “defects per KLOC” where “K” was the symbol for 1000 lines of code. The LOC metric was reasonably effective for all three purposes.

When the LOC metric was first introduced, there was only one programming language, basic assembly language. Programs were small and coding effort composed about 90 percent of the total work. Physical lines and logical statements were the same thing for basic assembly language.

In this early environment, the LOC metric was useful for economic, productivity, and quality analyses. The LOC metric worked fairly well for a single language where there was little or no reused code and where there were no significant differences between counts of physical lines and counts of logical statements. But the golden age of the LOC metric, where it was effective and had no rivals, only lasted about ten years.

However, this ten-year span was time enough so that the LOC metric became firmly embedded in the psychology of software engineering. Once an idea becomes firmly fixed, it tends to stay in place until new evidence becomes overwhelming. Unfortunately, as the software industry changed

and evolved rapidly, the LOC metric did not change. As time passed, the LOC metric became less and less useful until by about 1980 it had become extremely harmful without very many people realizing it. Due to cognitive dissonance, the LOC metric was used but not examined critically in the light of changes in other software engineering methods.

Lines of Code Metrics Circa 1970

By 1970, basic assembly had been supplanted by macro-assembly. The first generation of higher-level programming languages such as COBOL, FORTRAN, and PL/I was starting to be used. Usage of basic assembly language was beginning to drop out of use as better alternatives became available. This was perhaps the first instance of a long series of programming languages that died out, leaving a train of aging legacy applications that would be difficult to maintain as programmers and compilers stopped being available who were familiar with the dead languages.

The first known problem with LOC metrics was in 1970, when many IBM publication groups exceeded their budgets for that year. It was discovered (by the author) that technical publication group budgets had been based on 10 percent of the budgets assigned to programming or coding.

The publication projects based on code budgets for assembly language did not overrun their budgets, but manuals for the projects coded in PL/S (a derivative of PL/I) had major overruns. This was because PL/S reduced coding effort by half, but the technical manuals were as big as ever. Therefore, when publication budgets were set at 10 percent of code budgets, and coding costs declined by 50 percent, all of the publication budgets for PL/S projects were exceeded.

The initial solution to this problem at IBM was to give a formal mathematical definition to language levels. The *level* was defined as the number of statements in basic assembly language needed to equal the functionality of 1 statement in a higher-level language. Thus, COBOL was a level 3 language because it took three basic assembly statements to equal one COBOL statement. Using the same rule, SMALLTALK is a level 18 language.

For several years before function points were invented, IBM used “equivalent assembly statements” as the basis for estimating noncode work such as user manuals. (Indeed, a few companies still use equivalent assembly language even in 2009.)

Thus, instead of basing a publication budget on 10 percent of the effort for writing a program in PL/S, the budget would be based on 10 percent of the effort if the code were basic assembly language. This method was crude but reasonably effective. This method recognized that

not all languages required the same number of lines of code to deliver specific functions.

However, neither IBM customers nor IBM executives were comfortable with the need to convert the sizes of modern languages into the size of an antique language for cost-estimating purposes. Therefore, a better form of metric was felt to be necessary.

The documentation problem plus dissatisfaction with the equivalent assembler method were two of the reasons IBM assigned Allan Albrecht and his colleagues to develop function point metrics. Additional very powerful programming languages such as APL were starting to appear, and IBM wanted both a metric and an estimating method that could deal with noncoding work as well as coding in an accurate fashion.

The use of macro-assembly language had introduced code reuse, and this caused measurement problems, too. It raised the issue of how to count reused code in software applications, or how to count any other reused material for economic purposes.

The solution here was to separate productivity into two discrete topics:

1. *Development* productivity
2. *Delivery* productivity

The former, development productivity, dealt with the code and materials that had to be constructed from scratch in the traditional way.

The latter, delivery productivity, dealt with the final application as delivered, including reused material. For example, using macro-assembly language, a productivity rate for *development productivity* might be 300 lines of code per month. But due to reusing code in the form of macro expansions, *delivery productivity* might be as high as 750 lines of code per month.

This is an important business distinction that is not well understood even in 2009. The true goal of software engineering is to improve the rate of delivery productivity. Indeed, it is possible for delivery productivity to rise while development productivity declines!

This might occur by carefully crafting a reusable code module and certifying it to zero-defect quality levels. Assume a 500-line code module is developed for widespread reuse. Assume the module was carefully developed, fully inspected, examined via static analysis, and fully tested. The module was certified to be of zero-defect status.

This kind of careful development and certification might yield a net development productivity rate of only 100 lines of code per month, while normal development for a single-use module would be closer to 500 lines of code per month. Thus, a total of five months instead of a single month of development effort went to creating the module. This is of course a very low rate of *development* productivity.

However, once the module is certified and available for reuse, assume that utilizing it in additional applications can be done in only one hour. Therefore, every time the module is utilized, it saves about one month of custom development!

If the module is utilized in only five applications, it will have paid for its low development productivity. Every time this module is used, its effective delivery productivity rate is equal to 500 lines of code per hour, or about 66,000 lines of code per month!

Thus, while the *development* productivity of the module dropped down to only 100 lines of code per month, the *delivery* productivity rate is equivalent to 66,000 lines of code per month. The true economic value of this module does not reside in how fast it was developed, but rather in how many times it can be delivered in other applications because it is reusable.

To be successful, reused code needs to approach or achieve zero-defect status. It does not matter what the development speed is, if once completed the code can then be used in hundreds of applications.

As service-oriented architecture (SOA) and software as a service (SaaS) approach, their goal is to make dramatic improvements in the ability to deliver software features. Development speed is comparatively unimportant so long as quality approaches zero-defect levels.

Returning to the historical chronology, another issue shared between macro-assembly language and other new languages was the difference between physical lines of code and logical statements. Some languages, such as Basic, allowed multiple statements to be placed on a physical line. Other languages, such as COBOL, divided some logical statements into multiple physical lines. The difference between a count of physical lines and a count of logical statements could differ by as much as 500 percent. For some languages, there would be more physical lines than logical statements, but for other languages, the reverse was true. This problem was never fully resolved by LOC users and remains troublesome even in 2009.

Due to the increasing power and sophistication of high-level programming languages such as C++, Objective C, SMALLTALK, and the like, the percentage of project effort devoted to coding was dropping from 90 percent down to about 50 percent. As coding effort declined, LOC metrics were no longer effective for economic, productivity, or quality studies.

After function point metrics were developed circa 1975, the definition of language level was expanded to include the number of logical code statements equivalent to 1 function point. COBOL, for example, requires about 105 statements per function point in the procedure and data divisions.

This expansion is the mathematical basis for *backfiring*, or direct conversion from source code to function points. Of course, individual

programming styles make backfiring a method with poor accuracy even though it remains widely used for legacy applications where code exists but specifications may be missing.

There are tables available from several consulting companies such as David Consulting, Gartner Group, and Software Productivity Research (SPR) that provide values for source code statements per function point for hundreds of programming languages.

In 1978, A.J. Albrecht gave a public lecture on function point metrics at a joint IBM/SHARE/GUIDE conference in Monterey, California. Soon after this, function points started to be published in the software literature. IBM customers soon began to use function points, and this led to the formation of a function point user's group, originally in Canada.

Lines of Code Metrics Circa 1980

By about 1980, the number of programming languages had topped 50, and object-oriented languages were rapidly evolving. As a result, software reusability was increasing rapidly.

Another issue that surfaced circa 1980 was the fact that many applications were starting to use more than one programming language, such as COBOL and SQL. The trend for using multiple languages in the same application has become the norm rather than the exception. However, the difficulty of counting lines of code with accuracy was increased when multiple languages were used.

About the middle of this decade, function point users organized and created the nonprofit International Function Point Users Group (IFPUG). Originally based in Canada, IFPUG moved to the United States in the mid-1980s. Affiliates in other countries soon were formed, so that by the end of the decade, function point user groups were in a dozen countries.

In 1985, the first commercial software cost-estimating tool based on function points reached the market, SPQR/20. This tool supported estimates for 30 common programming languages and also could be used for combinations of more than one programming language.

This tool included sizing and estimating of paper documents such as requirements, design, and user manuals. It also estimated noncoding tasks including testing and project management.

Because LOC metrics were still widely used, the SPQR/20 tool expressed productivity and quality results using both function points and LOC metrics. Because it was easy to switch from one language to another, it was interesting to compare the results using both function point and LOC metrics when changing from macro-assembly to FORTRAN or Ada or PL/I or Java.

As the level of a programming language goes up, economic productivity expressed in terms of function points per staff month also goes up,

which matches standard economics. But as language levels get higher, productivity expressed in terms of lines of code per month drops down. This reversal by LOC metrics violates all rules of standard economics and is a key reason for asserting that LOC metrics constitute professional malpractice.

It is a well-known law of manufacturing economics that *when a development cycle includes a high percentage of fixed costs, and there is a decline in the number of units manufactured, the cost per unit will go up.*

If line of code is considered to be a manufacturing unit and there is a switch from a low-level language to a high-level language, the number of units will decline. But the paper documents in the form of requirements, specifications, and user documents do not decline. Instead they stay almost constant and have the economic effect of fixed costs. This of course will raise the cost per unit. Because this situation is poorly understood, two examples will clarify the situation.

Case A Suppose we have an application that consists of 1000 lines of code in basic assembly language. (We can also assume that the application is 5 function points.) Assume the development personnel are paid at a rate of \$5000 per staff month.

Assume that coding took 1 staff month and production of paper documents in the form of requirements, specifications, and user manuals also took 1 staff month. The total project took 2 staff months and cost \$10,000. Productivity expressed as LOC per staff month is 500. The cost per LOC is \$10.00. Productivity expressed in terms of function points per staff month is 2.5. The cost per function point is \$2000.

Case B Assume that we are doing the same application using the Java programming language. Instead of 1000 lines of code, the Java version only requires 200 lines of code. The function point total stays the same at 5 function points. Development personnel are also paid at the same rate of \$5000 per staff month.

In Case B suppose that coding took only 1 staff week, but the production of paper documents remained constant at 1 staff month.

Now the entire project took only 1.25 staff months instead of 2 staff months. The cost was only \$6250 instead of \$10,000. Clearly economic productivity has improved, since we did the same job as Case A with a savings of \$3750. We delivered exactly the same functions to users, but with much less code and therefore much less effort, so true economic productivity increased.

When we measure productivity for the entire project using LOC metrics, our rate has dropped down to only 160 LOC per month from the 500 LOC per month shown for Case A!

Our cost per LOC has soared up to \$31.25 per LOC. Obviously, LOC metrics cannot measure true economic productivity. Also obviously, LOC metrics penalize high-level languages. In fact, many studies have proven that the penalty exacted by LOC metrics is directly proportional to the level of the programming language, with the highest-level languages looking the worst!

Since the function point totals of both Case A and Case B versions are the same at 5 function points, Case B has a productivity rate of 4 function points per staff month. The cost per function point is only \$1250. These improvements match the rules of standard economics, because the faster and cheaper version has better results than the slower more expensive version.

What has happened of course is that the paperwork portion of the project did not decline even though the code portion declined substantially. This is why LOC metrics are professional malpractice if applied to compare projects that used different programming languages. They move in the opposite direction from standard economic productivity rates and penalize high-level languages. Table 8-7 summarizes both Case A and Case B.

As can be seen by looking at Cases A and B when they are side by side, LOC metrics actually reverse the terms of the economic equation and make the large, slow, costly version look better than the small, quick, cheap version.

It might be said that the reversal of productivity with LOC metrics is because paperwork was aggregated with coding. But even when only coding by itself is measured, LOC metrics still violate standard economic assumptions.

TABLE 8-7 Comparing Low-Level and High-Level Languages

	Case A	Case B	Difference
Language	Assembly	Java	
Lines of code (LOC)	1000	200	-800
Function points	5.00	5.00	0
Monthly compensation	\$5,000.00	\$5,000.00	\$0.00
Paperwork effort (months)	1.00	1.00	0
Coding effort (months)	1.00	0.25	-0.75
Total effort (months)	2.00	1.25	-0.75
Project cost	\$10,000.00	\$6,250.00	-\$3,750.00
LOC per month	500	160	-340
Cost per LOC	\$10.00	\$31.25	\$21.25
Function points per month	2.50	4.00	1.5
Cost per function point	\$2,000.00	\$1,250.00	-\$750.00

The 1000 LOC of assembly code was done in 1 month at a rate of 1000 LOC per month. The pure coding cost was \$5000 or \$5.00 per LOC.

The 200 LOC of Java code was done in 1 week, or 0.25 month. Converted into a monthly rate, that is only 800 LOC per month. The coding cost for Java was \$1250, so the cost per LOC was \$6.25.

Thus, Java costs more per LOC than assembly, even though Java took only one-fourth the time and one-fourth the cost! When you try and measure the two different languages using LOC, assembly looks better than Java, which is definitely a false conclusion. Table 8-8 shows the comparison between assembly and Java for coding only.

In real economic terms, the Java code only cost \$1250 while the assembly code cost \$5000. Obviously, Java has better economics because the same job was done for a savings of \$3750.

But the Java LOC production rate is lower than assembly, and the cost per LOC has jumped from \$5.00 to \$6.25! From an economic standpoint, variations in LOC per month and cost per LOC are unimportant if there is a major difference in how much code is needed to complete an application.

Unfortunately, LOC metrics end up as professional malpractice no matter how you use them if you are trying to measure economic productivity between unlike programming languages. By contrast, the Java code's cost per function point was \$250, while the assembly code's cost per function point was \$1000, and this matches the assumptions of standard economics.

Function point production for Java was 20 function points per staff month versus only 5 function points per staff month for assembly. Thus, function points match the assumptions of standard economics while LOC metrics violate standard economics.

Returning to the main thread, within a few years, all other commercial software estimating tools would also support function point metrics, so

TABLE 8-8 Comparing Coding for Low-Level and High-Level Languages

	Case A	Case B	Difference
Language	Assembly	Java	
Lines of code (LOC)	1000	200	-800
Function points	5.00	5.00	0
Monthly compensation	\$5,000.00	\$5,000.00	\$0.00
Coding effort (months)	1.00	0.25	-0.75
Coding cost	\$5,000.00	\$1,250.00	-\$3,750.00
LOC per month	1000	800	-200
Cost per LOC	\$5.00	\$6.25	\$1.25
Function points per month	5	20	15
Cost per function point	\$1,000.00	\$250.00	-\$750.00

that CHECKPOINT, COCOMO, KnowledgePlan, Price-S, SEER, SLIM SPQR/20, and others could express estimates in terms of both function points and LOC metrics.

By the end of this decade, coding effort was below 35 percent of total project effort, and LOC was no longer valid for either economic or quality studies. LOC metrics could not quantify requirements and design defects, which now outnumbered coding defects. LOC metrics could not be used to measure any of the noncoding activities such as requirements, design, documentation, or project management.

The response of the LOC users to these problems was unfortunate: they merely stopped measuring anything but code production and coding defects. The bulk of all published reports based on LOC metrics cover less than 35 percent of development effort and less than 25 percent of defects, with almost no data being published on requirements and design defects, rates of requirements creep, design costs, and other modern problems.

The history of the LOC metric provides an interesting example of Dr. Leon Festinger's theory of cognitive dissonance. Once an idea becomes entrenched, the human mind tends to reject all evidence to the contrary. Only when the evidence becomes overwhelming will there be changes of opinion, and such changes tend to occur rapidly.

Lines of Code Metrics Circa 1990

By about 1990, not only were there more than 500 programming languages in use, but some applications were written in 12 to 15 different languages. There were no international standards for counting code, and many variations were used sometimes without being defined.

In 1991, the first edition of the author's book *Applied Software Measurement* included a proposed draft standard for counting lines of code based on counting logical statements. One year later, Bob Park from the Software Engineering Institute (SEI), also published a proposed draft standard, only based on counting physical lines.

A survey of software journals by the author in 1993 found that about one-third of published articles used physical lines, one-third used logical statements, and the remaining third used LOC metrics without even bothering to say how they were counted. Since there is about a 500 percent variance between physical LOC and logical statements for many languages, this was not a good situation.

The technical journals that deal with medical practice and engineering often devote as much as 50 percent of the text to explaining and defining the measurement methods used to derive the results. The software engineering journals, on the other hand, often fail to define the measurement methods at all.

The software journals seldom devote more than a few lines of text to explaining the nature of the measurements used for the results. This is one of several reasons why the term “software engineering” is something of an oxymoron. In fact it is not even legal to use the term “software engineering” in some states and countries, because software development is not a recognized engineering discipline or a licensed engineering discipline.

But there was a worse problem approaching than ambiguity in counting lines of code. The arrival of Visual Basic introduced a class of programming languages where counting lines of code was not even possible. This is because a lot of Visual Basic “programming” was not done with procedural code, but rather with buttons and pull-down menus.

Of the approximate 2500 programming languages and dialects in existence circa 2009, there are only effective published counting rules for about 150. About another 2000 are similar to other languages and could perhaps share the same counting rules. But for at least 50 languages that use graphics or visual means to augment procedural code, there are no code counting rules at all. Unfortunately, some of the languages without code counting rules tend to be most recent languages that are used for web site development.

In 1994, a controlled study was done that used both LOC metrics and function points for ten versions of the same application written in ten different programming languages, including four object-oriented languages.

The study was published in *American Programmer* in 1994. This study found that LOC metrics violated the basic concepts of economic productivity and penalized high-level and OO languages due to the fixed costs of requirements, design, and other noncoding activities. This was the first published study to state that LOC metrics constituted professional malpractice if used for economic studies where more than one programming language was involved.

By the 1990s most consulting studies that collected benchmark and baseline data used function points. There are no large-scale benchmarks based on LOC metrics. The International Software Benchmarking Standards Group (ISBSG) was formed in 1997 and only publishes data in function point form. Consulting companies such as SPR and the David Consulting Group also use function point metrics.

By the end of the decade, some projects were spending less than 20 percent of the total effort on coding, so LOC metrics could not be used for the 80 percent of effort outside the coding domain. The LOC users remained blindly indifferent to these problems and continued to measure only coding, while ignoring the overall economics of complete development cycles that include requirements, analysis, design, user documentation, project management, and many other noncoding tasks.

By the end of the decade, noncoding defects in requirements and design outnumbered coding defects almost 2 to 1. But since noncode defects could not be measured with LOC metrics, the LOC literature simply ignores them.

Indeed, still in 2009, debates occur about the usefulness of the LOC metric, but the arguments unfortunately are not solidly grounded in manufacturing economics. The LOC enthusiasts seem to ignore the impact of fixed costs on software development.

The main argument of the LOC enthusiasts is that development effort has a solid statistical correlation to size measured in terms of lines of code. This is true, but irrelevant in terms of standard economics.

If it takes 1000 lines of C code to deliver ten function points to customers and the cost was \$10,000, then the cost per LOC is \$10.00. Assuming one month of programming effort, the productivity rate using LOC is 1000 LOC per month.

If the same ten function points were delivered to customers in Objective C, there might be only 250 lines of code and the cost might be only \$2500. The effort might take only one week instead of a whole month. But the cost per LOC is unchanged at \$10.00 and the LOC productivity rate is also unchanged at 1000 LOC per month.

With LOC metrics, both versions appear to have identical productivity rates of 1000 LOC per month, but these are *development* rates; not *delivery* rates. Since the functionality is the same for both C and Objective C versions, it is important that the cost per function point for C was \$1000, while for Objective C the cost per function point was only \$250.

Measured in terms of function points per month, the rate for C was 10, while the rate for Objective C increased to 40. Thus, when measured correctly, the economic value of high-level languages and delivery rates are clearly revealed, while the LOC metric does not show either economic or delivery productivity at all.

Lines of Code Metrics Circa 2000

By the end of the century, the number of programming languages had topped 2000 and continues to grow at more than one new programming language per month. Current rates of new programming language development may approach 100 new languages per year.

Web applications are mushrooming, and all of these are based on very high-level programming languages and substantial reuse. The Agile methods are also mushrooming and also tend to use high-level programming languages. Software reuse in some applications now tops 80 percent. LOC metrics cannot be used for most web applications and are certainly not useful for measuring Scrum sessions and other noncoding activities that are part of Agile projects.

Function point metrics had become the dominant metric for serious economic and quality studies. But two new problems appeared that have kept function point metrics from actually becoming the industry standard for both economic and quality studies.

The first problem is that some software applications are now so large (greater than 300,000 function points) that normal function point analysis is too slow and too expensive to be used.

There are gaps at both ends of normal function point analysis. Above 15,000 function points, the costs and schedule for counting function point metrics become so high that large projects are almost never counted. (Function point analysis operates between 400 and 600 function points per day per counter. The approximate cost is about \$6.00 per function point counted.)

At the low end of the scale, the counting rules for function points do not operate below a size of about 15 function points. Thus, small changes and bug repairs cannot be counted. Individually, such changes may be as small as $1/50^{\text{th}}$ of a function point and are rarely larger than 10 function points. But large companies can make 30,000 or more changes per year, with a total size that can top 100,000 function points.

The second problem is that the success of the original function point metric has triggered an explosion of function point clones. As of 2009, there are at least 24 function point variations. This makes benchmark and baseline studies difficult, because there are very few conversion rules from one variation to another.

In addition to standard IFPUG function points, there are also Mark II function points, COSMIC function points, Finnish function points, Netherlands function points, story points, feature points, web-object points, and many others.

Although LOC metrics continue to be used, they continue to have such major errors that they constitute professional malpractice for economic and quality studies where more than one language is involved, or where non-coding issues are significant.

There is also a psychological problem. LOC usage tends to fixate attention on coding and make the other kinds of software work invisible. For large software projects there may be many more noncode workers than programmers. There will be architects, designers, database administrators, quality assurance, technical writers, project managers, and many other occupations. But since none of these can be measured using LOC metrics, the LOC literature ignores them.

Lines of Code Metrics Circa 2010

It would be nice to predict an optimistic future, but the recession has changed the nature of industry and the future is now uncertain.

If current trends continue, within a few more years the software industry will have more than 3000 programming languages, of which about 2900 will be obsolete or nearly dead languages. The industry will have more than 20 variations for counting lines of code, more than 50 variations for counting function points, and probably another 20 unreliable metrics such as story points, use-case points, cost per defect, or using percentages of unknown numbers. (The software industry loves to make claims such as “improve productivity by 10 to 1” without defining either the starting or the ending point.)

Future generations of sociologists will no doubt be interested in why the software industry spends so much energy on creating variations of things, and so little energy on fundamental issues. No doubt large projects will still be cancelled, litigation for failures will still be common, software quality will still be bad, software productivity will remain low, security flaws will be alarming, and the software literature will continue to offer unsupported claims without actually presenting quantified data.

What the software industry needs is actually fairly straightforward:

1. Measures of defect potentials from all sources expressed in terms of function points; that is, requirements defects, design defects, code defects, document defects, and bad fixes.
2. Measures of defect removal efficiency levels for all forms of inspection, static analysis, and testing.
3. Activity-based productivity benchmarks from requirements through delivery and then for maintenance and customer support from delivery to retirement using function points.
4. Certified sources of reusable material near the zero-defect level.
5. Much improved security methods to guard against viruses, spyware, and hacking.
6. Licenses and board-certification for software engineering specialties.

But until measurement becomes both accurate and cost-effective, none of these are likely to occur. An occupation that will not measure its own performance with accuracy is not a true profession.

Lines of Code Circa 2020

If we look forward to 2020, there are best-case and worst-case scenarios to consider.

The best-case scenario for lines of code metrics is that usage diminishes even faster than it has been and that economic productivity based on delivery becomes the industry focus rather than development and

lines of code. For this scenario to occur, the speed of function point analysis needs to increase and the cost per function point counted needs to decrease from about \$6.00 per function point counted to less than \$0.10 per function point counted, which is technically possible and indeed occurs in 2009, although the high-speed methods are not yet widely deployed since they are so new.

If these changes occur, then function point usage will increase at least tenfold, and many new kinds of economic studies can be carried out. Among these will be measurement of entire portfolios that might top 10 million function points. Corporate backlogs could be sized and prioritized, and some of these exceed 1 million function points. Risk/value analyses for major software applications could become both routine and professionally competent. It will also be possible to do economic analyses of interesting new technologies such as the Agile methods, service-oriented architecture (SOA), software as a service (SaaS), and of course total cost of ownership (TCO).

Under the best-case scenario, software engineering would evolve from a craft or art form into a true engineering discipline. Reliable measures of all activities and tasks will lead to greater success rates on large software applications. The goal of software engineering should be to become a true engineering discipline with recognized specialties, board certification, and accurate information on productivity, quality, and costs. But that cannot be accomplished when project failures outnumber successes for large applications.

So long as quality and productivity are ambiguous and uncertain, it is difficult to carry out multiple regression studies and to select really effective tools and methods. LOC metrics have been a major barrier to economic and quality studies for software.

The worst-case scenario is that LOC metrics continue at about the same level as 2009. The software industry will continue to ignore economic productivity and remain fixated on the illusory “lines of code per month” metric. Under the worst-case scenario, “software engineering” will remain an oxymoron. Trial-and-error methods will continue to dominate, in part because effective tools and methodologies cannot even be studied using LOC metrics. Under the worst-case scenario, failures and project disasters will remain common for large software applications.

Function point analysis will continue to serve an important role for economic studies, benchmarks, and baselines, but only for about 10 percent of software applications of medium size. The cost per function point under the worst-case scenario will remain so high that usage above 15,000 function points will continue to be very rare. There will probably be even more function point variations, and the chronic lack of conversion rules from one variation to another will make large-scale international economic studies almost impossible.

Summary and Conclusions

The history of lines of code metrics is a cautionary tale for all people who work in software. The LOC metric started out well and was fairly effective when there was only one programming language and coding was so difficult it constituted 90 percent of the total effort for putting software on a computer.

But the software industry began to develop hundreds of programming languages. Applications started to use multiple programming languages, and that remains the norm today. Applications grew from less than 1000 lines of code up to more than 10 million lines of code. Coding is the major task for small applications, but for large systems, the work shifts to defect removal and production of paper documents in the forms of requirements, specifications, user manuals, test plans, and many others.

The LOC metric was not able to keep pace with either change. It does not work well when there is ambiguity in counting code, which always occurs with high-level languages and multiple languages in the same application. It does not work well for large systems where coding is only a small fraction of the total effort.

As a result, LOC metrics became less and less useful until sometime around 1985 they started to become actually harmful. Given the errors and misunderstandings that LOC metrics bring to economic, productivity, and quality studies, it is fair to say that in many situations usage of LOC metrics can be viewed as professional malpractice if more than one programming language is part of the study or the study seeks to measure real economic productivity.

The final point is that continued usage of LOC metrics is a significant barrier that is delaying the progress of software engineering from a craft to a true engineering discipline. An occupation that cannot even measure its own work with accuracy is hardly qualified to be called engineering.

Readings and References

- Barr, Michael and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools*. Sebastopol, CA: O'Reilly Media, 2006.
- Beck, K. *Extreme Programming Explained: Embrace Change*. Boston, MA: Addison Wesley, 1999.
- Bott, Frank, A. Coleman, J. Eaton, and D. Rowland. *Professional Issues in Software Engineering*, Third Edition. London and New York: Taylor & Francis, 2000.
- Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison Wesley, 2001.
- Cohen, D., M. Lindvall, & P. Costa, "An Introduction to agile methods." *Advances in Computers*. New York: Elsevier Science (2004): 1–66.
- Garmus, David and David Herron. *Function Point Analysis*. Boston: Addison Wesley, 2001.
- Garmus, David and David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.

- Glass, Robert L. *Facts and Fallacies of Software Engineering (Agile Software Development)*. Boston: Addison Wesley, 2002.
- Hans, Professor van Vliet. *Software Engineering Principles and Practices*, Third Edition. London, New York: John Wiley & Sons, 2008.
- Highsmith, Jim. *Agile Software Development Ecosystems*. Boston, MA: Addison Wesley, 2002.
- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Humphrey, Watts. *TSP—Leading a Development Team*. Boston, MA: Addison Wesley, 2006.
- Hunt, Andrew and David Thomas. *The Pragmatic Programmer*. Boston, MA: Addison Wesley, 1999.
- Jeffries, R., et al. *Extreme Programming Installed*. Boston, MA: Addison Wesley, 2001.
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York, NY: McGraw-Hill, 2008.
- Jones, Capers. *Conflict and Litigation Between Software Clients and Developers*, Version 6. Burlington, MA: Software Productivity Research, June 2006. 54 pages.
- Jones, Capers. *Estimating Software Costs*, Second Edition. New York, NY: McGraw-Hill, 2007.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley Longman, 2000.
- Jones, Capers. "The Economics of Object-Oriented Software." *American Programmer Magazine*, October 1994: 29–35.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston, MA: Addison Wesley Longman, 2003.
- Krutchén, Phillippe. *The Rational Unified Process—An Introduction*. Boston, MA: Addison Wesley, 2003.
- Larman, Craig & Victor Basili. "Iterative and Incremental Development—A Brief History." *IEEE Computer Society*, June 2003: 47–55.
- Love, Tom. *Object Lessons*. New York, NY: SIGS Books, 1993.
- Marciniak, John J. (Ed.) *Encyclopedia of Software Engineering*. (2 vols.) New York, NY: John Wiley & Sons, 1994.
- McConnell, Steve. *Code Complete*. Redmond, WA: Microsoft Press, 1993.
- . *Software Estimation—Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Mills, H., M. Dyer, & R. Linger. "Cleanroom Software Engineering." *IEEE Software*, 4, 5 (Sept. 1987): 19–25.
- Morrison, J. Paul. *Flow-Based Programming. A New Approach to Application Development*. New York, NY: Van Nostrand Reinhold, 1994.
- Park, Robert E. *SEI-92-TR-20: Software Size Measurement: A Framework for Counting Software Source Statements*. Pittsburgh, PA: Software Engineering Institute, 1992.
- Pressman, Roger. *Software Engineering—Practitioner's Approach*, Sixth Edition. New York, NY: McGraw-Hill, 2005.
- Putnam, Lawrence and Ware Myers. *Industrial Strength Software—Effective Management Using Measurement*. Los Alamitos, CA: IEEE Press, 1997.
- . *Measures for Excellence—Reliable Software On-Time Within Budget*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1992.
- Sommerville, Ian. *Software Engineering*, Seventh Edition. Boston, MA: Addison Wesley, 2004.
- Stapleton, J. *DSDM—Dynamic System Development Method in Practice*. Boston, MA : Addison Wesley, 1997.
- Stephens M. and D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Berkeley, CA: Apress L.P., 2003.

This page intentionally left blank

Software Quality: The Key to Successful Software Engineering

Introduction

The overall software quality averages for the United States have scarcely changed since 1979. Although national data is flat for quality, a few companies have made major improvements. These happen to be companies that measure quality because they define quality in such a way that both prediction and measurement are possible.

The same companies also use full sets of defect removal activities that include inspections and static analysis as well as testing. Defect prevention methods such as joint application design (JAD) and development methods that focus on quality such as Team Software Process (TSP) are also used, once the importance of quality to successful software engineering is realized.

Historically, large software projects spend more time and effort on finding and fixing bugs than on any other activity. Because software defect removal efficiency only averages about 85 percent, the major costs of software maintenance are finding and fixing bugs accidentally released to customers.

When development defect removal is added to maintenance defect removal, the major cost driver for total cost of ownership (TCO) is that of defect removal. Between 30 percent and 50 percent of every dollar ever spent on software has gone to finding and fixing bugs.

When software projects run late and exceed their budgets, a main reason is excessive defect levels, which slow down testing and force applications into delays and costly overruns.

When software projects are cancelled and end up in court for breach of contract, excessive defect levels, inadequate defect removal, and poor quality measures are associated with every case.

Given the fact that software defect removal costs have been the primary cost driver for all major software projects for the past 50 years, it is surprising that so little is known about software quality.

There are dozens of books about software quality and testing, but very few of these books actually contain solid and reliable quantified data about basic topics such as:

1. How many bugs are going to be present in specific new software applications?
2. How many bugs are likely to be present in legacy software applications?
3. How can software quality be predicted and measured?
4. How effective are ISO standards in improving quality?
5. How effective are software quality assurance organizations in improving quality?
6. How effective is software quality assurance certification for improving quality?
7. How effective is Six Sigma for improving quality?
8. How effective is quality function deployment (QFD) for improving quality?
9. How effective are the higher levels of the CMMI in improving quality?
10. How effective are the forms of Agile development in improving quality?
11. How effective is the Rational Unified Process (RUP) in improving quality?
12. How effective is the Team Software Process (TSP) in improving quality?
13. How effective are the ITIL methods in improving quality?
14. How effective is service-oriented architecture (SOA) for improving quality?
15. How effective are certified reusable components for improving quality?
16. How many bugs can be eliminated by inspections?
17. How many bugs can be eliminated by static analysis?
18. How many bugs can be eliminated by testing?

19. How many different kinds of testing are needed?
20. How many test personnel are needed?
21. How effective are test specialists compared with developers?
22. How effective is automated testing?
23. How many test cases are needed for applications of various sizes?
24. How effective is test certification in improving performance?
25. How many bug repairs will themselves include new bugs?
26. How many bugs will get delivered to users?
27. How much does it cost to improve software quality?
28. How long does it take to improve software quality?
29. How much will we save from improving software quality?
30. How much is the return on investment (ROI) for better software quality?

This purpose of this chapter is to show the quantified results of every major form of quality assurance activity, inspection stage, static analysis, and testing stage on the delivered defect levels of software applications.

Defect removal comes in “private” and “public” forms. The private forms of defect removal include desk checking, static analysis, and unit testing. They are also covered in Chapter 8, because they concentrate on code defects, and that chapter deals with programming and code development.

The public forms of defect removal include formal inspections, static analysis if run by someone other than the software engineer who wrote the code, and many kinds of testing carried out by test specialists rather than the developers.

Both private and public forms of defect removal are important, but it is harder to get data on the private forms because they usually occur with no one else being present other than the person who is doing the desk checking or unit testing. As pointed out in Chapter 8, IBM used volunteers to record defects found via private removal activities. Some development methods such as Watts Humphrey’s Team Software Process (TSP) and Personal Software Process (PSP) also record private defect removal.

This chapter will also explain how to predict the number of bugs or defects that might occur, and how to predict defect removal efficiency levels. Not only code bugs, but also bugs or defects in requirements, design, and documents need to be predicted. In addition, new bugs accidentally included in bug repairs need to be predicted. These are called “bad fixes.” Finally, there are also bugs or errors in test cases themselves, and these need to be predicted, too.

This chapter will discuss the best ways of measuring quality and will caution against hazardous metrics such as “cost per defect” and “lines of code,” which distort results and conceal the real facts of software quality. In this chapter, several critical software quality topics will be discussed:

- Defining Software Quality
- Predicting Software Quality
- Measuring Software Quality
- Software Defect Prevention
- Software Defect Removal
- Specialists in Software Quality
- The Economic Value of Software Quality

Software quality is the key to successful software engineering. Software has long been troubled by excessive numbers of software defects both during development and after release. Technologies are available that can reduce software defects and improve quality by significant amounts.

Carefully planning and selecting an effective combination of defect prevention and defect removal activities can shorten software development schedules, lower software development costs, significantly reduce maintenance and customer support costs, and improve both customer satisfaction and employee morale at the same time. Improving software quality has the highest return on investment of any current form of software process improvement.

As the recession continues, every company is anxious to lower both software development and software maintenance costs. Improving software quality will assist in improving software economics more than any other available technology.

Defining Software Quality

A good definition for software quality is fairly difficult to achieve. There are many different definitions published in the software literature. Unfortunately, some of the published definitions for quality are either abstract or off the mark. A workable definition of software quality needs to have six fundamental features:

1. Quality should be predictable before a software application starts.
2. Quality needs to encompass all deliverables and not just the code.
3. Quality should be measurable during development.

4. Quality should be measurable after release to customers.
5. Quality should be apparent to customers and recognized by them.
6. Quality should continue after release, during maintenance.

Here are some of the published definitions for quality, and explanations of why some of them don't seem to conform to the six criteria just listed.

Quality Definition 1: "Quality means conformance to requirements."

There are several problems with this definition, but the major problem is that requirements errors or bugs are numerous and severe. Errors in requirements constitute about 20 percent of total software defects and are responsible for more than 35 percent of high-severity defects.

Defining quality as conformance to a major source of error is circular reasoning, and therefore this must be considered to be a flawed and unworkable definition. Obviously, a workable definition for quality has to include errors in requirements themselves.

Don't forget that the famous Y2K problem originated as a specific user requirement and not as a coding bug. Many software engineers warned clients and managers that limiting date fields to two digits would cause problems, but their warnings were ignored or rejected outright.

The author once worked (briefly) as an expert witness in a lawsuit where a company attempted to sue an outsource vendor for using two-digit date fields in a software application developed under contract. During the discovery phase, it was revealed that the vendor cautioned the client that two-digit date fields were hazardous, but the client rejected the advice and insisted that the Y2K problem be included in the application. In fact, the client's own internal standards mandated two-digit date fields. Needless to say, the client dropped the suit when it became evident that they themselves were the cause of the problem. The case illustrates that "user requirements" are often wrong and sometimes even dangerous or "toxic."

It also illustrates another point. Neither the corporate executives nor the legal department of the plaintiff knew that the Y2K problem had been caused by their own policies and practices. Obviously, there is a need for better governance of software from the top when problems such as this are not understood by corporate executives.

Using modern terminology from the recession, it is necessary to remove "toxic requirements" before conformance can be safe. The definition of quality as "conformance to requirements" does not lead to any significant quality improvements over time. No more requirements are being met in 2009 than in 1979.

If software engineering is to become a true profession rather than an art form, software engineers have a responsibility to help customers define requirements in a thorough and effective manner. It is the job of a professional software engineer to insist on effective requirements methods such as joint application design (JAD), quality function deployment (QFD), and requirements inspections.

Far too often the literature on software quality is passive and makes the incorrect assumption that users will be 100 percent effective in identifying requirements. This is a dangerous assumption. User requirements are never complete and they are often wrong. For a software project to succeed, requirements need to be gathered and analyzed in a professional manner, and software engineering is the profession that should know how to do this well.

It should be the responsibility of the software engineers to insist that proper requirements methods be used. These include joint application design (JAD), quality function deployment (QFD), and requirements inspections. Other methods that benefit requirements, such as embedded users or use-cases, might also be recommended. The users themselves are not software engineers and cannot be expected to know optimal ways of expressing and analyzing requirements. Ensuring that requirements collection and analysis are at state-of-the-art levels devolves to the software engineering team.

Once user requirements have been collected and analyzed, then conformance to them should of course occur. However, before conformance can be safe and effective, dangerous or toxic requirements have to be weeded out, excess and superfluous requirements should be pointed out to the users, and potential gaps that will cause creeping requirements should be identified and also quantified. The users themselves will need professional assistance from the software engineering team, who should not be passive bystanders for requirements gathering and analysis.

Unfortunately, requirements bugs cannot be removed by ordinary testing. If requirements bugs are not prevented from occurring, or not removed via formal inspections, test cases that are constructed from the requirements will confirm the errors and not find them. (This is why years of software testing never found and removed the Y2K problem.)

A second problem with this definition is that it is not predictable during development. Conformance to requirements can be measured after the fact, but that is too late for cost-effective recovery.

A third problem with this definition is that for brand-new kinds of innovative applications, there may not be any users other than the original inventor. Consider the history of successful software innovation such as the APL programming language, the first spreadsheet, and the early web search engine that later became Google.

These innovative applications were all created by inventors to solve problems that they themselves wanted to solve. They were not created based on the normal concept of “user requirements.” Until prototypes were developed, other people seldom even realized how valuable the inventions would be. Therefore, “user requirements” are not completely relevant to brand-new inventions until after they have been revealed to the public.

Given the fact that software requirements grow and change at measured rates of 1 percent to more than 2 percent every calendar month during the subsequent design and coding phases, it is apparent that achieving a full understanding of requirements is a difficult task.

Software requirements are important, but the combination of toxic requirements, missing requirements, and excess requirements makes simplistic definitions such as “quality means conformance to requirements” hazardous to the software industry.

Quality Definition 2: “Quality means reliability, portability, and many other -ilities.”

The problem with defining quality as a set of words ending with *ility* is that many of these factors are neither predictable before they occur nor easily measurable when they do occur.

While most of the *-ility* words are useful properties for software applications, some don’t seem to have much to do with quality as we would consider the term for a physical device such as an automobile or a toaster. For example, “portability” may be useful for a software vendor, but it does not seem to have much relevance to quality in the eyes of a majority of users.

The use of *-ility* words to define quality does not lead to quality improvements over time. In 2009, the software industry is no better in terms of many of these *-ilities* than it was in 1979. Using modern language from the recession, many of the *-ilities* are “subprime” definitions that don’t prevent serious quality failures. In fact, using *-ilities* rather than focusing on defect prevention and removal slows down progress on software quality control.

Among the many words that are cited when using this definition can be found (in alphabetical order):

1. Augmentability
2. Compatibility
3. Expandability
4. Flexibility
5. Interoperability

6. Maintainability
7. Manageability
8. Modifiability
9. Operability
10. Portability
11. Reliability
12. Scalability
13. Survivability
14. Understandability
15. Usability
16. Testability
17. Traceability
18. Verifiability

Of the words on this list, only a few such as “reliability” and “testability” seem to be relevant to quality as viewed by users. The other terms range from being obscure (such as “survivability”) to useful but irrelevant (such as “portability”). Other terms may be of interest to the vendor or development team, but not to customers (such as “maintainability”).

The *-ility* words seem to have an academic origin because they don’t really address some of the real-world quality issues that bother customers. For example, none of these terms addresses ease or difficulty of reaching customer support to get help when a bug is noted or the software misbehaves. None of the terms deals with the speed of fixing bugs and providing the fix to users in a timely manner.

The new Information Technology Infrastructure Library (ITIL) does a much better job of dealing with issues of quality in the eyes of users, such as customer support, incident management, and defect repairs intervals than does the standard literature dealing with software quality.

More seriously, the list of *-ility* words ignores two of the main topics that have a major impact on software quality when the software is finally released to customers: (1) defect potentials and (2) defect removal efficiency levels.

The term *defect potential* refers to the total quantity of defects that will likely occur when designing and building a software application. Defect potentials include bugs or defects in requirements, design, code, user documents, and bad fixes or secondary defects. The term *defect removal efficiency* refers to the percentage of defects found by any sequence of inspection, static analysis, and test stages.

To reach acceptable levels of quality in the view of customers, a combination of low defect potentials and high defect removal efficiency rates (greater than 95 percent) is needed. The current U.S. average for software quality is a defect potential of about 5.0 bugs per function point coupled with 85 percent defect removal efficiency. This combination yields a total of delivered defects of about 0.75 per function point, which the author regards as unprofessional and unacceptable.

Defect potentials need to drop below 2.5 per function point and defect removal efficiency needs to average greater than 95 percent for software engineering to be taken seriously as a true engineering discipline. This combination would result in a delivered defect total of only 0.125 defect per function point or about one-sixth of today's averages. Achieving or exceeding this level of quality is possible today in 2009, but seldom achieved.

One of the reasons that good quality is not achieved as widely as it might be is that concentrating on the *-ility* topics rather than measuring defects and defect removal efficiency leads to gaps and failures in defect removal activities. In other words, the *-ilities* definitions of quality are a distraction from serious study of software defect causes and the best methods of preventing and removing software defects.

Specific levels of defect potentials and defect removal efficiency levels could be included in outsource agreements. These would probably be more effective than current contracting practices for quality, which are often nonexistent or merely insist on a certain CMMI level.

If software is released with excessive quantities of defects so that it stops, behaves erratically, or runs slowly, it will soon be discovered that most of the *-ility* words fall by the wayside.

Defect quantities in released software tend to be the paramount quality issue with users of software applications, coupled with what kinds of corrective actions the software vendor will take once defects are reported. This brings up a third and more relevant definition of software quality.

Quality Definition 3: "Quality is the absence of defects that would cause an application to stop working or to produce incorrect results."

A software defect is a bug or error that causes software to either stop operating or to produce invalid or unacceptable results. Using IBM's severity scale, defects have four levels of severity:

- Severity 1 means that the software application does not work at all.
- Severity 2 means that major functions are disabled or produce incorrect results.
- Severity 3 means that there are minor issues or minor functions are not working.
- Severity 4 means a cosmetic problem that does not affect operation.

There is some subjectivity with these defect severity levels because they are assigned by human beings. Under the IBM model, the initial severity level is assigned when the bug is first reported, based on symptoms described by the customer or user who reported the defect. However, a final severity level is assigned by the change team when the defect is repaired.

This definition of quality is one favored by the author for several reasons. First, defects can be predicted before they occur and measured when they do occur. Second, customer satisfaction surveys for many software applications appear to correlate more closely to delivered defect levels than to any other factor. Third, many of the *-ility* factors also correlate to defects, or to the absence of defects. For example, reliability correlates exactly to the number of defects found in software. Usability, testability, traceability, and verifiability also have indirect correlations to software defect levels.

Measuring defect volumes and defect severity levels and then taking effective steps to reduce those volumes via a combination of defect prevention and defect removal activities is the key to successful software engineering.

This definition of software quality does lead to quality improvements over time. The companies that measure defect potentials, defect removal efficiency levels, and delivered defects have improved both factors by significant amounts. This definition of quality supports process improvements, predicting quality, measuring quality, and customer satisfaction as measured by surveys.

Therefore, companies that measure quality such as IBM, Dovél Technologies, and AT&T have made progress in quality control. Also, methods that integrate defect tracking and reporting such as Team Software Process (TSP) have made significant progress in reducing delivered defects. This is also true for some open-source applications that have added static-analysis to their suite of defect removal tools.

Defect and removal efficiency measures have been used to validate the effectiveness of formal inspections, show the impact of static analysis, and fine-tune more than 15 kinds of testing. The subjective measures have no ability to deal with such issues.

Every software engineer and every software project manager should be trained in methods for predicting software defects, measuring software defects, preventing software defects, and removing software defects. Without knowledge of effective quality and defect control, software engineering is a hoax.

The full definition of quality suggested by the author includes these nine factors:

1. Quality implies low levels of defects when software is deployed, ideally approaching zero defects.

2. Quality implies high reliability, or being able to run without stop-page or strange and unexpected results or sluggish performance.
3. Quality implies high levels of user satisfaction when users are surveyed about software applications and its features.
4. Quality implies a feature set that meets the normal operational needs of a majority of customers or users.
5. Quality implies a code structure and comment density that minimize bad fixes or accidentally inserting new bugs when attempting to repair old bugs. This same structure will facilitate adding new features.
6. Quality implies effective customer support when problems do occur, with minimal difficulty for customers in contacting the support team and getting assistance.
7. Quality implies rapid repairs of known defects, and especially so for high-severity defects.
8. Quality should be supported by meaningful guarantees and warranties offered by software developers to software users.
9. Effective definitions of quality should lead to quality improvements. This means that quality needs to be defined rigorously enough so that both improvements and degradations can be identified, and also averages. If a definition for quality cannot show changes or improvements, then it is of very limited value.

The 6th, 7th, 8th, and 9th of these quality issues tend to be sparsely covered by the literature on software quality, other than the new ITIL books. Unfortunately, the ITIL coverage is used only for internal software applications and is essentially ignored by commercial software vendors.

The definition of quality as an absence of defects, combined with supplemental topics such as ease of customer support and maintenance speed, captures the essence of quality in the view of many software users and customers.

Consider how the three definitions of quality discussed in this chapter might relate to a well-known software product such as Microsoft Vista. Vista has been selected as an example because it is one of the best-known large software applications in the world, and therefore a good test bed for trying out various quality definitions.

Applying Definition 1 to Vista: “Quality means conformance to requirements.”

The first definition would be hard to use for Vista, since no ordinary customers were asked what features they wanted in the operating system, although focus groups were probably used at some point.

If you compare Vista with XP, Leopard, or Linux, it seems to include a superabundance of features and functions, many of which were neither requested nor ever used by a majority of users. One topic that the software engineering literature does not cover well, or at all, is that of overstuffing applications with unnecessary and useless features.

Most people know that ordinary requirements usually omit about 20 percent of functions that users want. However, not many people know that for commercial software put out by companies such as Microsoft, Symantec, Computer Associates, and the like, applications may have more than 40 percent features that customers don't want and never use.

Feature stuffing is essentially a competitive move to either imitate what competitors do, or to attempt to pull ahead of smaller competitors by providing hundreds of costly but marginal features that small competitors could not imitate. In either case, feature stuffing is not a satisfactory conformance to user requirements.

Further, certain basic features such as security and performance, which users of operating systems do appreciate, are not particularly well embodied in Vista.

The bottom line is that defining quality as conformance to requirements is almost useless for applications with greater than 1 million users such as Vista, because it is impossible to know what such a large group will want or not want.

Also, users seldom are able to articulate requirements in an effective manner, so it is the job of professional software engineers to help users in defining requirements with care and accuracy. Too often the software literature assumes that software engineers are only passive observers of user requirements, when in fact, software engineers should be playing the role of physicians who are diagnosing medical conditions in order to prescribe effective therapies.

Physicians don't just passively ask patients what the problem is and what kind of medicine they want to take. Our job as software engineers is to have professional knowledge about effective requirement gathering and analysis methods (i.e., like medical diagnostic tests) and to also know what kinds of applications might provide effective "therapies" for user needs.

Passively waiting for users to define requirements without assisting them in using joint application design (JAD) or quality function deployment (QFD) or data mining of legacy applications is unprofessional on the part of the software engineering community. Users are not trained in requirements definition, so we need to step up to the task of assisting them.

Applying Definition 2 to Vista: “Quality means adherence to *-ility* terms.”

When Vista is judged by matching its features against the list of *-ility* terms shown earlier, it can be seen how abstract and difficult to apply such a list really is

1.	Augmentability	Ambiguous and difficult to apply to Vista
2.	Compatibility	Poor for Vista; many old applications don't work
3.	Expandability	Applicable to Vista and fairly good
4.	Flexibility	Ambiguous and difficult to apply to Vista
5.	Interoperability	Ambiguous and difficult to apply to Vista
6.	Maintainability	Unknown to users but probably poor for Vista
7.	Manageability	Ambiguous and difficult to apply to Vista
8.	Modifiability	Unknown to users but probably poor for Vista
9.	Operability	Ambiguous and difficult to apply to Vista
10.	Portability	Poor for Vista
11.	Reliability	Originally poor for Vista but improving
12.	Scalability	Marginal for Vista
13.	Survivability	Ambiguous and difficult to apply to Vista
14.	Understandability	Poor for Vista
15.	Usability	Asserted to be good for Vista, but questionable
16.	Testability	Poor for Vista: complexity far too high
17.	Traceability	Poor for Vista: complexity far too high
18.	Verifiability	Ambiguous and difficult to apply to Vista

The bottom line is that more than half of the *-ility* words are difficult or ambiguous to apply to Vista or any other commercial software application. Of the ones that can be applied to Vista, the application does not seem to have satisfied any of them but expandability and usability.

Many of the *-ility* words cannot be predicted nor can they be measured. Worse, even if they could be predicted and measured, they are of marginal interest in terms of serious quality control.

Applying Definition 3 to Vista: “Quality means an absence of defects, plus corollary factors.”

Released defects can and should be counted for every software application. Other related topics such as ease of reporting defects and speed of repairing defects should also be measured.

Unfortunately, for commercial software, not all of these nine topics can be evaluated. Microsoft together with many other software vendors does not publish data on bad-fix injections or even on total numbers

of bugs reported. However, six of the eight factors can be evaluated by means of journal articles and limited Microsoft data.

1. Vista was released with hundreds or thousands of defects, although Microsoft will not provide the exact number of defects found and reported by users.
2. At first Vista was not very reliable, but achieved acceptable reliability after about a year of usage. Microsoft does not report data on mean time to failure or other measures of reliability.
3. Vista never achieved high levels of user satisfaction compared with XP. The major sources of dissatisfaction include lack of printer drivers, poor compatibility with older applications, excessive resource usage, and sluggish performance on anything short of high-end computer chips and lots of memory.
4. The feature set of Vista has been noted as adequate in customer surveys, other than excessive security vulnerabilities.
5. Microsoft does not release statistics on bad-fix injections or on numbers of defect reports, so this factor cannot be known by the general public.
6. Microsoft customer support is marginal and troublesome to access and use. This is a common failing of many software vendors.
7. Some known bugs have remained in Microsoft Vista for several years. Microsoft is marginally adequate in defect repair speed.
8. There is no effective warranty for Vista (or for other commercial applications). Microsoft's end-user license agreement (EULA) absolves Microsoft of any liabilities other than replacing a defective disk.
9. Microsoft's new operating system is not yet available as this book is published, so it is not possible to know if Microsoft has used methods that will yield better quality than Vista. However, since Microsoft does have substantial internal defect tracking and quality assurance methods, hopefully quality will be better. Microsoft has shown some improvements in quality over time.

Based on this pattern of analysis for the nine factors, it cannot be said that Vista is a high-quality application under any of the definitions. Of the three major definitions, defining quality as conformance to requirements is almost impossible to use with Vista because with millions of users, nobody can define what everybody wants.

The second definition of quality as a string of *-ility* words is difficult to apply, and many are irrelevant. These words might be marginally useful for small internal applications, but are not particularly helpful

for commercial software. Also, many key quality issues such as customer support and maintenance repair times are not found in any of the *-ility* words.

The third definition that centers on defects, customer support, defect repairs, and better warranties seems to be the most relevant. The third also has the advantage of being both predictable and measurable, which the first two lack.

Given the high costs of commercial software, the marginal or useless warranties of commercial software, and the poor customer support offered by commercial software vendors, the author would favor mandatory defect reporting that required commercial vendors such as Microsoft to produce data on defects reported by customers, sorted by severity levels.

Mandatory defect reporting is already a requirement for many products that affect human life or safety, such as medicines, aircraft engines, automobiles, and many other consumer products. Mandatory reporting of business and financial information is also required. Software affects human life and safety in critical ways, and it affects business operations in critical ways, but to date software has been exempt from serious study due to the lack of any mandate for measuring and reporting released defect levels.

Somewhat surprisingly, the open-source software community appears to be pulling ahead of old-line commercial software vendors in terms of measuring and reporting defects. Many open-source companies have added defect tracking and static-analysis tools to their quality arsenal, and are making data available to customers that is not available from many commercial software vendors.

The author would also favor a "lemon law" for commercial software similar to the lemon law for automobiles. If serious defects occur that users cannot get repaired when making good-faith effort to resolve the situation with vendors, vendors should be required to return the full purchase or lease price of the offending software application.

A form of lemon law might also be applied to outsource contracts, except the litigation already provides relief for outsource failures that cannot be used against commercial software vendors due to their one-sided EULA agreements, which disclaim any responsibility for quality other than media replacement.

No doubt software vendors would object to both mandatory defect tracking and also to a lemon law. But shrewd and farsighted vendors would soon perceive that both topics offer significant competitive advantages to software companies that know how to control quality. Since high-quality software is also cheaper and faster to develop and has lower maintenance costs than buggy software, there are even more important economic advantages for shrewd vendors.

The author hypothesizes that a combination of mandatory defect reporting by software vendors plus a lemon law would have the effect of improving software quality by about 50 percent every five years for perhaps a 20-year period.

Software quality needs to be taken much more seriously than it has been. Now that the recession is expanding, better software quality control is one of the most effective strategies for lowering software costs. But effective quality control depends on better measures of quality and on proven combinations of defect prevention and defect removal activities.

Quality prediction, quality measurement, better defect prevention, and better defect removal are on the critical path for advancing software engineering to the status of a true engineering discipline instead of a craft or art form as it is today in 2009.

Defining and Predicting Software Defects

If delivered defects are the main quality problem for software, it is important to know what causes these defects, so that they can be prevented from occurring or removed before delivery.

The software quality literature includes a great deal of pedantic bickering about various terms such as “fault,” “error,” “bug,” “defect” and many other terms. For this book, if software stops working, won’t load, operates erratically, or produces incorrect results due to mistakes in its own code, then that is called a “defect.” (This same definition has been used in 14 of the author’s previous books and also in more than 30 journal articles. The author’s first use of this definition started in 1978.)

However, in the modern world, the same set of problems can occur without the developers or the code being the cause. Software infected by a virus or spyware can also stop working, refuse to load, operate erratically, and produce incorrect results. In today’s world, some defect reports may well be caused by outside attacks.

Attacks on software from hackers are not the same as self-inflicted defects, although successful attacks do imply security vulnerabilities.

In this book and the author’s previous books, software defects have five main points of *origin*:

1. Requirements
2. Design
3. Code
4. User documents
5. Bad fixes (new defects due to repairs of older defects)

Because the author worked for IBM when starting research on quality, the IBM severity scale for classifying defect severity levels is used in this book and the author's previous books. There are four *severity levels*:

- Severity 1: Software does not operate at all
- Severity 2: Major features disabled or incorrect
- Severity 3: Minor features disabled or incorrect
- Severity 4: Cosmetic error that does not affect operation

There are other methods of classifying severity levels, but these four are the most common due to IBM introducing them in the 1960s, so they became a de facto standard.

Software defects have seven kinds of *causes*, with the major causes including

Errors of omission:	Something needed was accidentally left out
Errors of commission:	Something needed is incorrect
Errors of ambiguity:	Something is interpreted in several ways
Errors of performance:	Some routines are too slow to be useful
Errors of security:	Security vulnerabilities allow attacks from outside
Errors of excess:	Irrelevant code and unneeded features are included
Errors of poor removal:	Defects that should easily have been found

These seven causes occur with different frequencies for different deliverables. For paper documents such as requirements and design, errors of ambiguity are most common, followed by errors of omission. For source code, errors of commission are most common, followed by errors of performance and security.

The seventh category, "errors of poor removal," would require root-cause analysis for identification. The implication is that the defect was neither subtle nor hard to find, but was missed because test cases did not cover the code segment or because of partial inspections that overlooked the defect.

In a sense, all delivered defects might be viewed as errors of poor removal, but it is important to find out why various kinds of inspection, static analysis, or testing missed obvious bugs. This category should not be assigned for subtle defects, but rather for obvious defects that should have been found but for some reason escaped to the outside world.

The main reason for including errors of poor removal is to encourage more study and research on the effectiveness of various kinds of defect removal operations. More solid data is needed on the removal efficiency levels of inspections, static analysis, automatic testing, and all forms of manual testing.

The combination of defect origins, defect severity, and defect causes provides a useful taxonomy for classifying defects for statistical analysis or root-cause analysis. For example, the Y2K problem was cited earlier

in this chapter. In its most common manifestation, the Y2K problem might have this description using the taxonomy just discussed:

Y2K origin:	Requirements
Y2K severity:	Severity 2 major features disabled
Y2K primary cause:	Error of commission
Y2K secondary cause:	Error of poor removal

Note that this taxonomy allows the use of primary and secondary factors since sometimes more than one problem is behind having a defect in software.

Note also that the Y2K problem did not have the same severity for every application. An approximate distribution of Y2K severity levels for several hundred applications noted that the software stopped in about 15 percent of instances, which are severity 1 problems; it created severity 2 problems in about 50 percent; it created severity 3 problems in about 25 percent; and had no operational consequences in about 10 percent of the applications in the sample.

To know the origin of a defect, some research is required. Most defects are initially found because the code stops working or produces erratic results. But it is important to know if upstream problems such as requirements or design issues are the true cause. Root-cause analysis can find the true causes of software defects.

Several other factors should be included in a taxonomy for tracking defects. These include whether a reported defect is valid or invalid. (Invalid defects are common and fairly expensive, since they still require analysis and a response.) Another factor is whether a defect report is new and unique, or merely a duplicate of a prior defect report.

For testing and static analysis, the category of “false positives” needs to be included. A *false positive* is the mistaken identification of a code segment that initially seems to be incorrect, but which later research reveals is actually correct.

A third factor deals with whether the repair team can make the same problem occur on their own systems, or whether the defect was caused by a unique configuration on the client’s system. When defects cannot be duplicated, they were termed abeyant defects by IBM, since additional information needed to be collected to solve the problem.

Adding these additional topics to the Y2K example would result in an expanded taxonomy:

Y2K origin:	Requirements
Y2K validity:	Valid defect report
Y2K uniqueness:	Duplicate (this problem was reported millions of times)
Y2K severity:	Severity 2 major features disabled
Y2K primary cause:	Error of commission
Y2K secondary cause:	Error of poor removal

When defects are being counted or predicted, it is useful to have a standard metric for normalizing the results. As discussed in Chapter 5, there are at least ten candidates for such a normalizing metric, including function points, story points, use-case points, lines of code, and so on.

In this book and also in the author's previous books, the function point metric defined by the International Function Point Users Group (IFPUG) is used to quantify and normalize data for both defects and productivity.

There are several reasons for using IFPUG function points. The most important reason in terms of measuring software defects is that non-code defects in requirements, design, and documents are major defect sources and cannot be measured using the older "lines of code" metric.

Another important reason is that all of the major benchmark data collections for productivity and quality use function point metrics, and data expressed via IFPUG function points composes about 85 percent of all known benchmarks.

It is not impossible to use other metrics for normalization, but if results are to be compared against industry benchmarks such as those published by the International Software Benchmarking Standards Group (ISBSG), the IFPUG function points are the most convenient. Later in the discussion of defect prediction, examples will be given of using other metrics in addition to IFPUG function points.

It is interesting to combine the origin, severity, and cause factors to examine the approximate frequency of each.

Table 9-1 shows the combination of these factors for software applications during development. Therefore, Table 9-1 shows *defect potentials*, or the probable numbers of defects that will be encountered during development and after release. Only severity 1 and severity 2 defects are shown in Table 9-1.

Data on defect potentials is based on long-range studies of defects and defect removal efficiency carried out by organizations such as the IBM Software Quality Assurance groups, which have been studying software quality for more than 35 years.

TABLE 9-1 Overview of Software Defect Potentials

Defect Origins	Defects per Function Point	Severity 1 Defects	Severity 2 Defects	Most Frequent Defect Cause
Requirements	1.00	11.00%	15.00%	Omission
Design	1.25	15.00%	20.00%	Omission
Code	1.75	70.00%	57.00%	Commission
Documents	0.60	1.00%	1.00%	Ambiguity
Bad fixes	0.40	3.00%	7.00%	Commission
TOTAL	5.00	100.00%	100.00%	Omission

Other corporations such as AT&T, Coverity, Computer Aid Inc. (CAI), Dovél Technologies, Motorola, Software Productivity Research (SPR), Galorath Associates, the David Consulting Group, the Quality and Productivity Management Group (QPMG), Unisys, Microsoft, and the like, also carry out long-range studies of defects and removal efficiency levels.

Most such studies are carried out by corporations rather than universities because academia is not really set up to carry out longitudinal studies that may last more than ten years.

While coding bugs or coding defects are the most numerous during development, they are also the easiest to find and to get rid of. A combination of inspections, static analysis, and testing can wipe out more than 95 percent of coding defects and sometimes top 99 percent. Requirements defects and bad fixes are the toughest categories of defect to eliminate.

Table 9-2 uses Table 9-1 as a starting point, but shows the latent defects that will still be present when the software application is delivered to users. Table 9-2 shows approximate U.S. averages circa 2009. Note the variations in defect removal efficiency by origin.

It is interesting that when the software is delivered to clients, requirements defects are the most numerous, primarily because they are the most difficult to prevent and also the most difficult to find. Only formal requirements-gathering methods combined with formal requirements inspections can improve the situation for finding and removing requirements defects.

If not prevented or removed, both requirements bugs and design bugs eventually find their way into the code. These are not coding bugs per se, such as branching to a wrong address, but more serious and deep-seated kinds of bugs or defects.

It was noted earlier in this chapter that requirements defects cannot be found and removed by means of testing. If a requirements defect is not prevented or removed via inspection, all test cases created using the requirements will confirm the defect and not identify it.

TABLE 9-2 Overview of Delivered Software Defects

Defect Origins	Defects per Function Point	Removal Efficiency	Delivered Defects per Function Point	Most Frequent Defect Cause
Requirements	1.00	70.00%	0.30	Commission
Design	1.25	85.00%	0.19	Commission
Code	1.75	95.00%	0.09	Commission
Documents	0.60	91.00%	0.05	Omission
Bad fixes	0.40	70.00%	0.12	Commission
TOTAL	5.00	85.02%	0.75	Commission

Since Table 9-2 reflects approximate U.S. averages, the methods assumed are those of fairly careless requirements gathering: waterfall development, CMMI level 1, no formal inspections of requirements, design, or code; no static analysis; and using only five forms of testing: (1) unit test, (2) new function test, (3) regression test, (4) system test, and (5) acceptance test.

Note also that during development, requirements will continue to grow and change at rates of 1 percent to 2 percent every calendar month. These changing requirements have higher defect potentials than the original requirements and lower levels of defect removal efficiency. This is yet another reason why requirements defects cause more problems than any other defect origin point.

Software requirements are the most intractable source of software defects. However, methods such as joint application design (JAD), quality function deployment (QFD), Six Sigma analysis, root-cause analysis, embedding users with the development team as practiced by Agile development, prototypes, and the use of formal requirements inspections can assist in bringing requirements defects under control.

Table 9-3 shows what quality might look like if an optimal combination of defect prevention and defect removal activities were utilized. Table 9-3 assumes formal requirements methods, rigorous development such as practiced using the Team Software Process (TSP) or the higher CMMI levels, prototypes and JAD, formal inspections of all deliverables, static analysis of code, and a full set of eight testing stages: (1) unit test, (2) new function test, (3) regression test, (4) performance test, (5) security test, (6) usability test, (7) system test, and (8) acceptance test.

Table 9-3 also assumes a software quality assurance (SQA) group and rigorous reporting of software defects starting with requirements, continuing through inspections, static analysis and testing, and out into the field with multiple years of customer-reported defects, maintenance, and enhancements. Accumulating data such as that shown in Tables 9-1 through 9-3 requires longitudinal data collection that runs for many years.

TABLE 9-3 Optimal Defect Prevention and Defect Removal Activities

Defect Origins	Defects per Function Point	Removal Efficiency	Delivered Defects per Function Point	Most Frequent Defect Cause
Requirements	0.50	95.00%	0.03	Omission
Design	0.75	97.00%	0.02	Omission
Code	0.50	99.00%	0.01	Commission
Documents	0.40	96.00%	0.02	Omission
Bad fixes	0.20	92.00%	0.02	Commission
TOTAL	2.35	96.40%	0.08	Omission

This combination has the effect of cutting defect potentials by more than 50 percent and of raising cumulative defect removal efficiency from today's average of 85 percent up to more than 96 percent.

It might be possible to even exceed the results shown in Table 9-3, but doing so would require additional methods such as the availability of a full suite of certified reusable materials.

Tables 9-2 and 9-3 are oversimplifications of real-life results. Defect potentials vary with the size of the application and with other factors. Defect removal efficiency levels also vary with application size. Bad-fix injections also vary by defect origins. Both defect potentials and defect removal efficiency levels vary by methodology, by CMMI levels, and by other factors as well. These will be discussed later in the section of this chapter dealing with defect prediction.

Because of the many definitions of quality used by the industry, it is best to start by showing what is predictable and measurable and what is not. To sort out the relevance of the many quality definitions, the author has developed a 10-point scoring method for software quality factors.

- If a factor leads to improvement in quality, its maximum score is 3.
- If a factor leads to improvement in customer satisfaction, its maximum score is 3.
- If a factor leads to improvement in team morale, its maximum score is 2.
- If a factor is predictable, its maximum score is 1.
- If a factor is measurable, its maximum score is 1.
- The total maximum score is 10.
- The lowest possible score is 0.

Table 9-4 lists all of the quality factors discussed in this chapter in rank order by using the scoring factor just outlined. Table 9-4 shows whether a specific quality factor is measurable and predictable, and also the relevance of the factor to quality as based on surveys of software customers. It also includes a weighted judgment as to whether the factor has led to improvements in quality among the organizations that use it.

The quality definitions with a score of 10 have been the most effective in leading to quality improvements over time. As a rule, the quality definitions scoring higher than 7 are useful. However, the quality definitions that score below 5 have no empirical data available that shows any quality improvement at all.

While Table 9-4 is somewhat subjective, at least it provides a mathematical basis for scoring the relevance and importance of the rather

TABLE 9-4 Rank Order of Quality Factors by Importance to Quality

	Measurable Property?	Predictable Property?	Relevance to Quality	Score
Best Quality Definitions				
Defect potentials	Yes	Yes	Very high	10.00
Defect removal efficiency	Yes	Yes	Very high	10.00
Defect severity levels	Yes	Yes	Very high	10.00
Defect origins	Yes	Yes	Very high	10.00
Reliability	Yes	Yes	Very high	10.00
Good Quality Definitions				
Toxic requirements	Yes	No	Very high	9.50
Missing requirements	Yes	No	Very high	9.50
Requirements conformance	Yes	No	Very high	9.00
Excess requirements	Yes	No	Medium	9.00
Usability	Yes	Yes	Very high	8.00
Testability	Yes	Yes	High	8.00
Defect causes	Yes	No	Very high	8.00
Fair Quality Definitions				
Maintainability	Yes	Yes	High	7.00
Understandability	Yes	Yes	Medium	6.00
Traceability	Yes	No	Low	6.00
Modifiability	Yes	No	Medium	5.00
Verifiability	Yes	No	Medium	5.00
Poor Quality Definitions				
Portability	Yes	Yes	Low	4.00
Expandability	Yes	No	Low	3.00
Scalability	Yes	No	Low	2.00
Interoperability	Yes	No	Low	1.00
Survivability	Yes	No	Low	1.00
Augmentability	No	No	Low	0.00
Flexibility	No	No	Low	0.00
Manageability	No	No	Low	0.00
Operability	No	No	Low	0.00

vague and ambiguous collection of quality factors used by the software industry. In essence, Table 9-4 makes these points:

1. Conformance to requirements is hazardous unless incorrect, toxic, or dangerous requirements are weeded out. This definition has not demonstrated any improvements in quality for more than 30 years.
2. Most of the *-ility* quality definitions are hard to measure, and many are of marginal significance. Some are not measurable either. None of the *-ility* words tend to lead to tangible quality gains.

3. Quantification of defect potentials and defect removal efficiency levels have had the greatest impact on improving quality and also the greatest impact on customer satisfaction levels.

If software engineering is to evolve from a craft or art form into a true engineering field, it is necessary to put quality on a firm quantitative basis and to move away from vague and subjective quality definitions. These will still have a place, of course, but they should not be the primary definitions for software quality.

Predicting Software Defect Potentials

To predict software quality, it is necessary to measure software quality. Since companies such as IBM have been doing this for more than 40 years, the best available data comes from companies that have full life-cycle quality measurement programs that start with requirements, continue through development, and then extend out to customer-reported defects for as long as the software is used, which may be 25 years or more. The next best source of data comes from benchmark and commercial software estimating tool companies, since they collect historical data on quality as well as on productivity.

Because software defects come from five different sources, the quickest way to get a useful approximation of software defect potentials is to use IFPUG function point metrics.

The basic sizing rule for predicting defect potentials with function point is: *Take the size of a software application in function points and raise it to the 1.25 power. The result will be a useful approximation of software defect potentials for applications between a low of about 10 function points and a high of about 5000 function points.*

The exponent for this rule of thumb would need to be adjusted downwards for the higher CMMI levels, Agile, RUP, and the Team Software Process (TSP). But since the rule is intended to be applied early, before any costs are expended, it still provides a useful starting point. Readers might want to experiment with local data and find an exponent that gives useful results against local quality and defect data.

Table 9-5 shows approximate U.S. averages for defect potentials. Recall that defect potentials are the sum of five defect origins: requirements defects, design defects, code defects, document defects, and bad-fix injections.

As can be seen from Table 9-5, defect potentials increase with application size. Of course, other factors can reduce or increase the potentials, as will be discussed later in the section on defect prevention.

While the total defect potential is useful, it is also useful to know the distribution of defects among the five origins or sources. Table 9-6

TABLE 9-5 U.S. Averages for Software Defect Potentials

Size in FP Function Points	Defects per Function Point	Defect Potentials
1	1.50	2
10	2.34	23
100	3.04	304
1,000	4.62	4,621
10,000	6.16	61,643
100,000	7.77	777,143
1,000,000	8.56	8,557,143
Average	4.86	1,342,983

illustrates typical defect distribution percentages using approximate average values.

Applying the distribution shown in Table 9-6 to a sample application of 1500 function points, Table 9-7 illustrates the approximate defect potential, or the total number of defects that might be found during development and by customers.

These simple overall examples are not intended as substitutes for commercial quality estimation tools such as KnowledgePlan and SEER, which can adjust their predictions based on CMMI levels; development methods such as Agile, TSP, or RUP; use of inspections; use of static analysis; and other factors which would cause defect potentials to vary and also which cause defect removal efficiency levels to vary.

Rules of thumb are never very accurate, but their convenience and ease of use provide value for rough estimates and early sizing. However, such rules should not be used for contracts or serious estimates.

Predicting Code Defects

Using function point metrics as an overall tool for quality prediction is useful because noncoding defects outnumber code defects. That being said, there are more coding defects than any other single source.

TABLE 9-6 Percentages of Defects by Origin

Defect Origins	Defects per Function Point	Percent of Total Defects
Requirements	1.00	20.00%
Design	1.25	25.00%
Source code	1.75	35.00%
User documents	0.60	12.00%
Bad fixes	0.40	8.00%
TOTAL	5.00	100.00%

TABLE 9-7 Defect Potentials for a Sample Application

(Application size = 1500 Function Points)

Defect Origins	Defects per Function Point	Defect Potentials	Percent of Total Defects
Requirements	1.00	1,500	20.00%
Design	1.25	1,875	25.00%
Source code	1.75	2,625	35.00%
User documents	0.60	900	12.00%
Bad fixes	0.40	600	8.00%
TOTAL	5.00	7,500	100.00%

Predicting code defects is fairly tricky for six reasons:

1. More than 2,500 programming languages are in existence, and they are not equal as sources of defects.
2. A majority of modern software applications use more than one language, and some use as many as 15 different programming languages.
3. The measured range of performance by a sample of programmers using the same language for the same test application varies by more than 10 to 1. Individual skills and programming styles create significant variations in the amount of code written for the same problem, in defect potentials, and also in productivity.
4. Lines of code can be counted using either physical lines or logical statements. For some languages, the two counts are identical, but for others, there may be as much as a 500 percent variance between physical and logical counts.
5. For a number of languages starting with Visual Basic, some programming is done by means of buttons or pull-down menus. Therefore, programming is done without using procedural source code. There are no effective rules for counting source code with such languages.
6. Reuse of source code from older applications or from libraries of reusable code is quite common. If the reused code is certified, it will have very few defects compared with new custom code.

To predict coding defects, it is necessary to know the *level* of a programming language. The concept of the level of a language is often used informally in phrases such as “high-level” or “low-level” languages.

Within IBM in the 1970s, when research was first carried out on predicting code defects, it was necessary to give a formal mathematical definition to language levels. Within IBM the *level* was defined as the number of statements in basic assembly language needed to equal the functionality of 1 statement in a higher-level language.

Using this definition, COBOL was a level 3 language, because it took 3 basic assembly statements to equal 1 COBOL statement. Using the same rule, SMALLTALK is a level 15 language.

(For several years before function points were invented, IBM used “equivalent assembly statements” as the basis for estimating non-code work such as user manuals. Thus, instead of basing a publication budget on 10 percent of the effort for writing a program in PL/S, the budget would be based on 10 percent of the effort if the code were basic assembly language. This method was crude but reasonably effective.)

Dissatisfaction with the equivalent assembler method for estimation was one of the reasons IBM assigned Allan Albrecht and his colleagues to develop function point metrics.

Additional programming languages such as APL, Forth, Jovial, and others were starting to appear, and IBM wanted both a metric and estimating methods that could deal with both noncoding and coding work in an accurate fashion. IBM also wanted to predict coding defects.

The use of macro-assembly language had introduced reuse, and this caused measurement problems, too. It raised the issue of how to count reused code in software applications or any other reused material. The solution here was to separate productivity and quality into two topics: (1) development and (2) delivery.

The former dealt with the code and materials that had to be constructed from scratch. The latter dealt with the final application as delivered, including reused material. For example, using macro-assembly language a productivity rate for *development productivity* might be 300 lines of code per month. But due to reusing code in the form of macro expansions, *delivery productivity* might be as high as 750 lines of code per month.

The same distinction affects quality, too. Assume a program had 1000 lines of new code and 1000 lines of reused code. There might be 15 bugs per KLOC in the new code but 0 bugs per KLOC in the reused code.

This is an important business distinction that is not well understood even in 2009. The true goal of software engineering is to improve the rate of delivery productivity and quality rather than development productivity and quality.

After function point metrics were developed circa 1975, the definition of “language level” was expanded to include the number of logical code statements equivalent to 1 function point. COBOL, for example, requires about 105 statements per function point in the procedure and data divisions. (This expansion is the mathematical basis for backfiring, or direct conversion from source code to function points.)

Table 9-8 illustrates how code size and coding defects would vary if 15 different programming languages were used for the same application, which is 1000 function points. Table 9-8 assumes a constant value of 15 potential coding defects per KLOC for all languages. However,

TABLE 9-8 Examples of Defects per KLOC and Function Point for 15 Languages

(Assumes a constant of 15 defects per KLOC for all languages)

Language Level	Sample Languages	Source Code per Function Point	Source Code per 1000 FP	Coding Defects	Defects per Function Point
1.	Assembly	320	320,000	4,800	4.80
2.	C	160	160,000	2,400	2.40
3.	COBOL	107	106,667	1,600	1.60
4.	PL/I	80	80,000	1,200	1.20
5.	Ada95	64	64,000	960	0.96
6.	Java	53	53,333	800	0.80
7.	Ruby	46	45,714	686	0.69
8.	E	40	40,000	600	0.60
9.	Perl	36	35,556	533	0.53
10.	C++	32	32,000	480	0.48
11.	C#	29	29,091	436	0.44
12.	Visual Basic	27	26,667	400	0.40
13.	ASP NET	25	24,615	369	0.37
14.	Objective C	23	22,857	343	0.34
15.	Smalltalk	21	21,333	320	0.32

the 15 languages have levels that vary from 1 to 15, so very different quantities of code will be created for the same 1000 function points.

Note: Language levels are variable and change based on volumes of reused code or calls to external functions. The levels shown in Table 9-8 are only approximate and are not constants.

As can be seen from Table 9-8, in order to predict coding defects, it is critical to know the programming language (or languages) that will be used and also the size of the application using both function point and lines of code metrics.

The situation is even more tricky when combinations of two or more languages are used within the same application. However, this problem is handled by commercial software cost-estimating tools such as KnowledgePlan, which include multilanguage estimating capabilities. Reused code also adds to the complexity of predicting coding defects.

To show the results of multiple languages in the same application, let us consider two case studies.

In Case Study A, there are three different languages and each language has 1000 lines of code, counted using logical statements. In Case Study B, we have the same three languages, but now each language comprises exactly 25 function points each.

For Case A, the total volume of source code is 3000 lines of code; total function points are 73; and total code defect potentials are 45.

Case A: Three Languages with 1000 Lines of Code Each

Languages	Levels	Lines of Code (LOC)	LOC per Function Point	Function Points	Defect Potential
C	2.00	1,000	160	6	15
Java	6.00	1,000	53	19	15
Smalltalk	15.00	1,000	21	48	15
TOTAL		3,000		73	45
AVERAGE	7.76		41		

When we change the assumptions to Case B and use a constant value of 25 function points for each language, the total number of function points only changes from 73 to 75. But the volume of source code almost doubles, as do numbers of defects. This is because of the much greater impact of the lowest-level language, the C programming language.

When considering either Case A or Case B, it is easily seen that predicting either size or quality for a multi language application is a great deal more complicated than for a single-language application.

Case B: Three Languages with 25 Function Points Each

Languages	Levels	Lines of Code (LOC)	LOC per Function Point	Function Points	Defect Potential
C	2.00	4,000	160	25	60
Java	6.00	1,325	53	25	20
Smalltalk	15.00	525	21	25	8
TOTAL		5,850		75	88
AVERAGE	4.10		78		

It is interesting to look at Case A and Case B in a side-by-side format to highlight the differences. Note that in Case B the influence of the lowest-level language, the C programming language, increases both code volumes and defect potentials:

Source Code (Logical statements)	Case A	Case B
C	1,000	4,000
Java	1,000	1,325
Smalltalk	1,000	525
Total lines of code	3,000	5,850
Total KLOC	3.00	5.85
Function Points	73	75
Code Defects	45	88
Defects per KLOC	15	15
Defects per Function Point	0.62	1.17

Cases A and B oversimplify real-life problems because each case study uses constants for data items that in real-life are variable. For example, the constant of 15 defects per KLOC for code defects is really a variable that can range from less than 5 to more than 25 defects per KLOC.

The number of source code statements per function point is also a variable, and each language can vary by perhaps a range of 2 to 1 around the average values shown by the nominal language “level” default values.

These variables illustrate why predicting quality and defect levels depends so heavily upon measuring quality and defect levels. The examples also illustrate why definitions of quality need to be both measurable and predictable.

Other variables can affect the ability to predict size and defects as well. Suppose, for example, that reused code composed 50 percent of the code volume in Case A. Suppose also that the reused code is certified and has zero defects. Now the calculations for defect predictions need to include reuse, which in this example lowers defect potentials by 50 percent.

When the size of the application is used for productivity calculations, it is necessary to decide whether development productivity or delivery productivity, or both, are the figures of interest.

Predicting software defects is possible to accomplish with fairly good accuracy, but the calculations are not trivial, and they need to include a number of variables that can only be determined by careful measurements.

The Quality Impacts of Creeping Requirements

Function point analysis at the end of the requirements phase and then again at application delivery shows that requirements grow and change at rates in excess of 1 percent per calendar month during the design and coding phases. The total growth in creeping requirements ranges from a low of less than 10 percent of total requirements to a high of more than 50 percent. (One unique project had requirements growth in excess of 200 percent.)

As an example, if an application is sized at 1000 function points when the initial requirements phase is over, then every month at least 10 new function points will be added in the form of new requirements. This growth might continue for perhaps six months, and so increase the size of the application from 1000 to 1060 function points. For small projects, the growth of creeping requirements is more of an inconvenience than a serious issue.

Larger applications have longer schedules and usually higher rates of requirements change as well. For an application initially sized at 10,000 function points, new requirements might lead to monthly growth rates of 125 function points for perhaps 20 calendar months. The delivered application might be 12,500 function points rather than 10,000 function points.

As this example illustrates, creeping requirements growth of a full 25 percent can have a major impact on development schedules, costs, and also on quality and delivered defects.

Because new and changing requirements are occurring later in development than the original requirements, they are often rushed. As a result, defect potentials for creeping requirements are about 10 percent greater than for the original requirements. This is true for toxic requirements and design errors. Code bugs may or may not increase, based upon the schedule pressure applied to the software engineering team.

Creeping requirements also tend to bypass formal inspections and also have fewer test cases created for them. As a result, defect removal efficiency is lower against both toxic requirements and also design errors by at least 5 percent. This seems to be true for code errors as well, with the exception that applications coded in C or Java that use static analysis tools will still achieve high levels of defect removal efficiency against code bugs.

The combined results of higher defect potentials and lower levels of defect removal for creeping requirements result in a much greater percentage of delivered defects stemming from changed requirements than any other source of error. This has been a chronic problem for the software industry.

The bottom line is that creeping requirements combined with below optimum levels of defect prevention and defect removal are a primary cause of cancelled projects, schedule delays, and cost overruns.

As will be discussed later in the sections on defect prevention and defect removal, there are technologies available for minimizing the harm from creeping requirements. However, these effective methods, such as formal requirements and design inspections, are not widely used.

Measuring Software Quality

In spite of the fact that defect removal efficiency is a critical topic for successful software projects, measuring defect removal efficiency or software quality in general are seldom done. From visiting over 300 companies in the United States, Europe, and Asia, the author found the following distribution of the frequency of various kinds of quality measures:

No quality measures at all	44%
Measuring only customer-reported defects	30%
Measuring test and customer-reported defects	18%
Measuring inspection, static analysis, test, and customer-reported defects	7%
Using volunteers for measuring personal defect removal	1%
Overall Distribution	100%

The mathematics of measuring defect removal efficiency is not complicated. Twelve steps in the sequence of data collection and calculations are needed to quantify defect removal efficiency levels:

1. Accumulate data on every defect that occurs, starting with requirements.
2. Assign severity levels to each reported defect as it is fixed.
3. Measure how many defects are removed by every defect removal activity.
4. Use root-cause analysis to identify origins of high-severity defects.
5. Measure invalid defects, duplicates, and false positives, too.
6. After the software is released, measure customer-reported defects.
7. Record hours worked for defect prevention, removal, and repairs.
8. Select a fixed point such as 90 days after release for the calculations.
9. Use volunteers to record private defect removal such as desk checking.
10. Calculate cumulative defect removal efficiency for the entire series.
11. Calculate the defect removal efficiency for each step in the series.
12. Use the data to improve both defect prevention and defect removal.

The effort and costs required to measure defect removal efficiency levels are trivial compared with the value of such information. The total effort required to measure each defect and its associated repair work amounts to only about an hour. Of this time, probably half is expended on customer-reported defects, and the other half is expended on internal defect reports.

However, step 4, root-cause analysis, can take several additional hours based on how well requirements and design are handled by the development team.

The value of measuring defect removal efficiency encompasses the following benefits:

- Finding and fixing bugs is the most expensive activity in all of software, so reducing these costs yields a very large return on investment.
- Excessive numbers of bugs constitute the main reason for schedule slippage, so reducing defects in all deliverables will shorten development schedules.
- Delivered defects are the major cost driver of software maintenance for the first two years after release, so improving removal efficiency lowers maintenance costs.

- Customer satisfaction correlates inversely to numbers of delivered defects, so reducing delivered defects will result in happier customers.
- Team morale correlates with both effective defect prevention and effective defect removal.

Later in the section on the economics of quality, these benefits will be quantified to show the overall value of defect prevention and defect removal.

Many companies and government organizations track software defects found during static analysis, testing, and also defects reported by customers. In fact, a number of commercial software defect tracking tools are available.

These tools normally track defect symptoms, applications containing defects, hardware and software platforms, and other kinds of indicative data such as release number, built number, and so on.

However, more sophisticated organizations also utilize formal inspections of requirements, design, and other materials. Such companies often utilize static analysis in addition to testing and therefore measure a wider range of defects than just those found in source code by ordinary testing.

Some additional information is needed in order to use expanded defect data for root-cause analysis and other forms of defect prevention. These additional topics include

Defect discovery point It is important to record information on the point at which any specific defect is found. Since requirements defects cannot normally be found via testing, it is important to try and identify noncode defect discovery points.

Collectively, noncode defects in requirements and design are more numerous than coding defects, and also may be high in severity levels. Defect repair costs for noncode defects are often higher than for coding defects. Note that there are more than 17 kinds of software testing, and companies do not use the same names for various test stages.

Date of defect discovery: _____

Defect Discovery Point:

- Customer defect report
- Quality assurance defect report
- Test stage _____ defect report
- Static analysis defect report
- Code inspection defect report
- Document inspection defect report

- Design inspection defect report
- Architecture inspection defect report
- Requirements inspection defect report
- Other _____ defect report

Defect origin point It is also important to record information on where software defects originate. This information requires careful analysis on the part of the change team, so many companies limit defect origin research to high-severity defects such as Severity 1 and Severity 2.

Date of defect origination: _____

Defect Origin Point:

- Application name
- Release number
- Build number
- Source code (internal)
- Source code (reused from legacy application)
- Source code (reused from commercial source)
- Source code (commercial software package)
- Source code (bad-fix or previous defect repair)
- User manual
- Design document
- Architecture document
- Requirement document
- Other _____ origination point

Ideally, the lag time between defect origins and defect discovery will be less than a month and hopefully less than a week. It is very important that defects that originate within a phase such as the requirements or design phases should also be discovered and fixed during the same phase.

When there is a long gap between origins and discovery, such as not finding a design problem until system test, it is a sign that software development and quality control processes need to improve.

The best solution for shortening the gap between defect origination and defect discovery is that of formal inspections of requirements, design, and other deliverables. Both static analysis and code inspections are also valuable for shortening the intervals between defect origination and defect discovery.

TABLE 9-9 Best-Case Defect Discovery Points

Defect Origins	Optimal Defect Discovery
Requirements	Requirements inspection
Design	Design inspection
Code	Static analysis
Bad fixes	Static analysis
Documentation	Editing
Test cases	Test case inspection

Table 9-9 shows the best-case scenario for defect discovery methods for various defect origins.

Inspections are best at finding subtle and complex bugs and problems that are difficult to find via testing because sometimes no test cases are created for them. The example of the Y2K problem is illustrative of a problem that could be found via testing so long as two-digit dates were mistakenly believed to be acceptable. Code inspections are useful for finding subtle problems such as security vulnerabilities that may escape both testing and even static analysis.

Static analysis is best at finding common coding errors such as branches to incorrect locations, overflow conditions, poor error handling, and the like. Static analysis prior to testing or as an adjunct to testing will lower testing costs.

Testing is best at finding problems that only show up when the code is operating, such as performance problems, usability problems, interface problems, and other issues such as mathematical errors or format errors for screens and reports.

Given the diverse nature of software bugs and defects, it is obvious that all three defect removal methods are important for success: inspections, static analysis, and testing.

Table 9-10 illustrates the fact that long delays between defect origins and defect discovery lead to very troubling situations. Long gaps also raise bad-fix injections, accidentally including new defects in attempts to repair older defects.

TABLE 9-10 Worst-Case Defect Discovery Points

Defect Origins	Latest Defect Discovery
Requirements	Deployment
Design	System testing
Code	New function testing
Bad fixes	Regression testing
Documentation	Deployment
Test cases	Not discovered

In the worst-case scenario, requirements defects are not found until deployment, while design defects are not found until system test, when it is difficult to fix them without extending the overall schedule for the project. Note that in the worst-case scenario, bugs or errors in test cases themselves are never discovered, so they fester on for many releases.

Defect prevention and early defect removal are far more cost-effective than depending on testing alone.

Other quality measures include some or all of the following:

Earned quality value (EQV) Since it is possible to predict defect potentials and also to predict defect removal efficiency levels, some companies such as IBM have used a form of “earned value” where predictions of defects that would probably be found via inspections, static analysis, and testing are compared with actual defect discovery rates. Predicted and actual defect removal costs are also compared.

If fewer defects are found than predicted, then root-cause analysis can be applied to discover if quality is really better than planned or if defect removal is lax. (Usually quality is better when this happens.)

If more defects are found than predicted, then root-cause analysis can be applied to discover if quality is worse than planned or if defect removal is more effective than anticipated. (Usually, quality is worse when this happens.)

Cost of quality (COQ) Collectively, the costs of finding and fixing bugs are the most expensive known activity in the history of software. Therefore, it is important to gather effort and cost data in such a fashion that cost of quality (COQ) calculations can be performed.

However, for software, normal COQ calculations need to be tailored to match the specifics of software engineering. Usually, data is recorded in terms of hours and then converted into costs by applying salaries, burden rates, and other cost items.

- Defect discovery activity: _____
- Defect prevention activities: _____
- Defect effort reported by users
- Defect damages reported by users
- Preparation hours for inspections
- Preparation hours for static analysis
- Preparation hours for testing
- Defect discovery hours
- Defect reporting hours

- Defect analysis hours
- Defect repair hours
- Defect inspection hours
- Defect static analysis hours
- Test stages used for defect
- Test cases created for defect
- Defect test hours

The software industry has long used the “cost per defect” metric without actually analyzing how this metric works. Indeed, hundreds of articles and books parrot similar phrases such as “it costs 100 times as much to fix a bug after release than during coding” or some minor variation on this phrase. The gist of these dogmatic statements is that the cost per defect rises steadily as the later defects are found.

What few people realize is that cost per defect is always cheapest where the most bugs are found and is most expensive where the fewest bugs are found. In fact, as normally calculated, this metric violates standard economic assumptions because it ignores fixed costs. The cost per defect metric actually penalizes quality and achieves the lowest results for the buggiest applications!

Following is an analysis of why cost per defect penalizes quality and achieves its best results for the buggiest applications. The same mathematical analysis also shows why defects seem to be cheaper if found early rather than found later.

Furthermore, when zero-defect applications are reached, there are still substantial appraisal and testing activities that need to be accounted for. Obviously, the cost per defect metric is useless for zero-defect applications.

Because of the way cost per defect is normally measured, as quality improves, cost per defect steadily increases until zero-defect software is achieved, at which point the metric cannot be used at all.

As with the errors in KLOC metrics, the main source of error is that of ignoring fixed costs. Three examples will illustrate how cost per defect behaves as quality improves.

In all three cases, A, B, and C, we can assume that test personnel work 40 hours per week and are compensated at a rate of \$2500 per week or \$62.50 per hour. Assume that all three software features that are being tested are 100 function points.

Case A: Poor Quality

Assume that a tester spent 15 hours writing test cases, 10 hours running them, and 15 hours fixing 10 bugs. The total hours spent was 40,

and the total cost was \$2500. Since 10 bugs were found, the cost per defect was \$250. The cost per function point for the week of testing would be \$25.00.

Case B: Good Quality

In this second case, assume that a tester spent 15 hours writing test cases, 10 hours running them, and 5 hours fixing one bug, which was the only bug discovered. However, since no other assignments were waiting and the tester worked a full week, 40 hours were charged to the project.

The total cost for the week was still \$2500, so the cost per defect has jumped to \$2500. If the 10 hours of slack time are backed out, leaving 30 hours for actual testing and bug repairs, the cost per defect would be \$1875. As quality improves, cost per defect rises sharply.

Let us now consider cost per function point. With the slack removed, the cost per function point would be \$18.75. As can easily be seen, cost per defect goes up as quality improves, thus violating the assumptions of standard economic measures.

However, as can also be seen, testing cost per function point declines as quality improves. This matches the assumptions of standard economics. The 10 hours of slack time illustrate another issue: when quality improves, defects can decline faster than personnel can be reassigned.

Case C: Zero Defects

In this third case, assume that a tester spent 15 hours writing test cases and 10 hours running them. No bugs or defects were discovered. Because no defects were found, the cost per defect metric cannot be used at all.

But 25 hours of actual effort were expended writing and running test cases. If the tester had no other assignments, he or she would still have worked a 40-hour week, and the costs would have been \$2500. If the 15 hours of slack time are backed out, leaving 25 hours for actual testing, the costs would have been \$1562.

With slack time removed, the cost per function point would be \$15.63. As can be seen again, testing cost per function point declines as quality improves. Here, too, the decline in cost per function point matches the assumptions of standard economics.

Time and motion studies of defect repairs do not support the aphorism that it costs 100 times as much to fix a bug after release as before. Bugs typically require between 15 minutes and 4 hours to repair.

Some bugs are expensive; these are called *abeyant defects* by IBM. Abeyant defects are customer-reported defects that the repair center cannot re-create, due to some special combination of hardware and

software at the client site. Abeyant defects constitute less than 5 percent of customer-reported defects.

Because of the fixed or inelastic costs associated with defect removal operations, cost per defect always increases as numbers of defects decline. Because more defects are found at the beginning of a testing cycle than after release, this explains why cost per defect always goes up later in the cycle. It is because the costs of writing test cases, running them, and having maintenance personnel available act as fixed costs.

In any manufacturing cycle with a high percentage of fixed costs, the cost per unit will go up as the number of units goes down. This basic fact of manufacturing economics is why cost per defect metrics are hazardous and invalid for economic analysis of software applications.

What would be more effective is to record the hours spent for all forms of defect removal activity. Once the hours are recorded, the data could be converted into cost data, and also normalized by converting hours into standard units such as hours per function point.

Table 9-11 shows a sample of the kinds of data that are useful in assessing cost of quality and also doing economic studies and effectiveness studies.

Of course, knowing defect removal hours implies that data is also collected on defect volumes and severity levels. Table 9-12 shows the same set of activities as Table 9-11, but switches from hours to defects. Both Tables 9-11 and 9-12 could also be combined into a single large spreadsheet. However, defect counts and defect effort accumulation tend to come from different sources and may not be simultaneously available.

Defect effort and discovered defect counts are important data elements for long-range quality improvements. In fact, without such data, quality improvement is likely to be minimal or not even occur at all.

Failure to record defect volumes and repair effort is a chronic weakness of the software engineering domain. However, several software development methods such as Team Software Process (TSP) and the Rational Unified Process (RUP) do include careful defect measures. The Agile method, on the other hand, is neither strong nor consistent on software quality measures.

For software engineering to become a true engineering discipline and not just a craft as it is in 2009, defect measurements, defect prediction, defect prevention, and defect removal need to become a major focus of software engineering.

Measuring Defect Removal Efficiency

One of the most effective metrics for demonstrating and improving software quality is that of *defect removal efficiency*. This metric is simple in concept but somewhat tricky to apply. The basic idea of this metric is to

TABLE 9-11 Software Defect Removal Effort Accumulation

Removal Stage	Defect Removal Effort (Hours Worked)			TOTAL HOURS
	Preparation Hours	Execution Hours	Repair Hours	
Inspections:				
Requirements				
Architecture				
Design				
Source code				
Documents				
Static analysis				
Test stages:				
Unit				
New function				
Regression				
Performance				
Usability				
Security				
System				
Independent				
Beta				
Acceptance				
Supply chain				
Maintenance:				
Customers				
Internal SQA				

calculate the percentage of software defects found by means of defect removal operations such as inspections, static analysis, and testing.

What makes the calculations for defect removal efficiency tricky is that it includes noncode defects found in requirements, design, and other paper deliverables, as well as coding defects.

Table 9-13 illustrates an example of defect removal efficiency levels for a full suite of removal operations starting with requirements inspections and ending with Acceptance testing.

Table 9-13 makes a number of simplifying assumptions. One of these is the assumption that all delivered defects will be found by customers in the first 90 days of usage. In real life, of course, many latent defects in delivered software will stay hidden for months or even years. However, after 90 days, new releases will usually occur, and they make it difficult to measure defects for prior releases.

TABLE 9-12 Software Defect Severity Level Accumulation

Removal Stage	Defect Severity Levels				TOTAL DEFECTS
	Severity 1 (Critical)	Severity 2 (Serious)	Severity 3 (Minor)	Severity 4 (Cosmetic)	
Inspections:					
Requirements					
Architecture					
Design					
Source code					
Documents					
Static Analysis					
Test stages:					
Unit					
New function					
Regression					
Performance					
Usability					
Security					
System					
Independent					
Beta					
Acceptance					
Supply chain					
Maintenance:					
Customers					
Internal SQA					

It is interesting to see what kind of defect removal efficiency levels occur with less sophisticated series of defect removal steps that do not include either formal inspections or static analysis.

Since noncode defects that originate in requirements and design eventually find their way into the code, the overall removal efficiency levels of testing by itself without any precursor inspections or static analysis are seriously degraded, as shown in Table 9-14.

When comparing Tables 9-13 and 9-14, it can easily be seen that a full suite of defect removal activities is more efficient and effective than testing alone in finding and removing software defects that originate outside of the source code. In fact, inspections and static analysis are also very efficient in finding coding defects and have the additional property of raising testing efficiency and lowering testing costs.

TABLE 9-13 Software Defect Removal Efficiency Levels

(Assumes inspections, static analysis, and normal testing)

Application size (function points)	1,000			
Language	C			
Code size	125,000			
Noncode defects	3,000			
Code defects	2,000			
TOTAL DEFECTS	5,000			
	Defect Removal Efficiency by Removal Stage			
Removal Stage	Noncode Defects	Code Defects	Total Defects	Removal Efficiency
Inspections:				
Requirements	750	0	750	
Architecture	200	0	200	
Design	1,250	0	1,250	
Source code	100	800	900	
Documents	250	0	250	
Subtotal	2,550	800	3,350	67.00%
Static Analysis	0	800	800	66.67%
Test stages:				
Unit	0	50	50	
New function	50	100	150	
Regression	0	25	25	
Performance	0	10	10	
Usability	50	0	50	
Security	0	20	20	
System	25	50	75	
Independent	0	5	5	
Beta	25	15	40	
Acceptance	25	15	40	
Supply chain	25	10	35	
Subtotal	200	300	500	58.82%
Prerelease Defects	2,750	1,900	4,650	93.00%
Maintenance:				
Customers (90 days)	250	100	350	100.00%
TOTAL	3,000	2,000	5,000	
Removal Efficiency	91.67%	95.00%	93.00%	

TABLE 9-14 Software Defect Removal Efficiency Levels

(Assumes normal testing without inspections or static analysis)

Application size	1000
(function points)	
Language	C
Code size	125,000
Noncode defects	3,000
Code defects	2,000
TOTAL DEFECTS	5,000

Removal Stage	Defect Removal Efficiency by Removal Stage			
	Noncode Defects	Code Defects	Total Defects	Removal Efficiency
Inspections:				
Requirements	0	0	0	
Architecture	0	0	0	
Design	0	0	0	
Source code	0	0	0	
Documents	0	0	0	
Subtotal	0	0	0	0.00%
Static Analysis	0	0	0	0.00%
Test stages:				
Unit	200	350	550	
New function	450	600	1,050	
Regression	0	100	100	
Performance	0	50	50	
Usability	200	75	275	
Security	0	50	50	
System	300	200	500	
Independent	50	10	60	
Beta	150	25	175	
Acceptance	175	20	195	
Supply chain	75	20	95	
Subtotal	1,600	1,500	3,100	62.00%
Prerelease Defects	1,600	1,500	3,100	62.00%
Maintenance:				
Customers (90 days)	1,400	500	1,900	100.00%
TOTAL	3,000	2,000	5,000	
Removal Efficiency	53.33%	75.00%	62.00%	

Without pretest inspections and static analysis, testing will find hundreds of bugs, but the overall defect removal efficiency of the full suite of test activities will be lower than if inspections and static analysis were part of the suite of removal activities.

In addition to elevating defect removal efficiency levels, adding formal inspections and static analysis to the suite of defect removal operations also lowers development and maintenance costs. Development schedules are also shortened, because traditional lengthy test cycles are usually the dominant part of software development schedules. Indeed, poor quality tends to stretch out test schedules by significant amounts because the software does not work well enough to be released.

Table 9-15 shows a side-by-side comparison of cost structures for the two examples discussed in this section. Case X is derived from Table 9-13 and uses a sophisticated combination of formal inspections, static analysis, and normal testing.

Case Y is derived from Table 9-14 and uses only normal testing, without any inspections or static analysis being performed.

The costs in Table 9-15 assume a fully burdened compensation structure of \$10,000 per month. The defect-removal costs assume preparation, execution, and defect repairs for all defects found and identified.

In addition to the cost advantages, excellence in quality control also correlates with customer satisfaction and with reliability. Reliability and customer satisfaction both correlate inversely with levels of delivered defects.

The more defects there are at delivery, the more unhappy customers are. In addition, mean time to failure (MTTF) goes up as delivered defects go down. The reliability correlation is based on high-severity defects in the Severity 1 and Severity 2 classes.

Table 9-16 shows the approximate relationship between delivered defects, reliability in terms of mean time to failure (MTTF) hours, and customer satisfaction with software applications.

Table 9-16 uses integer values, so interpolation between these discrete values would be necessary. Also, the reliability levels are only approximate. Table 9-13 deals only with the C programming language, so adjustments in defects per function point would be needed for the 700 other languages that exist. Additional research is needed on the topics of reliability and customer satisfaction and their correlations with delivered defect levels.

However, not only do excessive levels of delivered defects generate negative scores on customer satisfaction surveys, but they also show up in many lawsuits against outsource contractors and commercial software developers. In fact, one lawsuit was even filed by shareholders of a major software corporation who claimed that excessive defect levels were lowering the value of their stock.

TABLE 9-15 Comparison of Software Defect Removal Efficiency Costs

(Case X = inspections, static analysis, normal testing)

(Case Y = normal testing only)

Application size 1,000
(function points)

Language C

Code size 125,000

Noncode defects 3,000

Code defects 2,000

TOTAL DEFECTS 5,000

Removal Stage	Defect Removal Costs by Activity		
	Case X Removal \$	Case Y Removal \$	Difference
Inspections:			
Requirements			
Architecture			
Design			
Source code			
Documents			
Subtotal	\$168,750	\$0	\$168,750
Static Analysis	\$81,250	\$0	\$81,250
Test stages:			
Unit			
New function			
Regression			
Performance			
Usability			
Security			
System			
Independent			
Beta			
Acceptance			
Supply chain			
Subtotal	\$150,000	\$775,000	-\$625,000
Prerelease Defects	\$400,000	\$775,000	-\$375,000
Maintenance:			
Customers (90 days)	\$175,000	\$950,000	-\$775,000
TOTAL COSTS	\$575,000	\$1,725,000	-\$1,150,000
Cost per Defect	\$115.00	\$345.00	-\$230.00
Cost per Function Pt.	\$575.00	\$1,725.00	-\$1,150.00
Cost per LOC	\$4.60	\$13.80	-\$9.20
ROI from inspections, static analysis			\$3.00
Development Schedule (Calendar months)	12.00	16.00	-4.00

TABLE 9-16 Delivered Defects, Reliability, Customer Satisfaction

(Note 1: Assumes the C programming language)

(Note 2: Assumes 125 LOC per function point)

(Note 3: Assumes severity 1 and 2 delivered defects)

Delivered Defects per KLOC	Defects per Function Point	Mean Time to Failure (MTTF hours)	Customer Satisfaction
0.00	0.00	Infinite	Excellent
1.00	0.13	303	Very good
2.00	0.25	223	Good
3.00	0.38	157	Fair
4.00	0.50	105	Poor
5.00	0.63	66	Very poor
6.00	0.75	37	Very poor
7.00	0.88	17	Very poor
8.00	1.00	6	Litigation
9.00	1.13	1	Litigation
10.00	1.25	0	Malpractice

Better quality control is the key to successful software engineering. Software quality needs to be definable, predictable, measurable, and improvable in order for software engineering to become a true engineering discipline.

Defect Prevention

The phrase “defect prevention” refers to methods and techniques that lower the odds of certain kinds of defects occurring at all. The literature of defect prevention is very sparse, and academic research is even sparser. The reason for this is that studying defect prevention is extremely difficult and also somewhat ambiguous at best.

Defect prevention is analogous to vaccination against serious illness such as pneumonia or flu. There is statistical evidence that vaccination will lower the odds of patients contracting the diseases for which they are vaccinated. However, there is no proof that any specific patient would catch the disease whether receiving a vaccination or not. Also, a few patients who are vaccinated might contract the disease anyway, because vaccines are not 100 percent effective. In addition, some vaccines may have serious and unexpected side-effects.

All of these issues can occur with software defect prevention, too. While there is statistical evidence that certain methods such as prototypes, joint application design (JAD), quality function deployment (QFD), and participation in inspections prevent certain kinds of defects

from occurring, it is hard to prove that those defects would definitely occur in the absence of the preventive methodologies.

The way defect prevention is studied experimentally is to have two versions of similar or identical applications developed, with one version using a particular defect prevention method while the other version did not. Obviously, experimental studies such as this must be small in scale.

The easiest experiments in defect prevention are those dealing with formal inspections of requirements, design, and code. Because inspections record all defects, companies that utilize formal inspections soon accumulate enough data to analyze both defect prevention and defect removal.

Formal inspections are so effective in terms of defect prevention that they reduce defect potentials by more than 25 percent per year. In fact, one issue with inspections is that after about three years of continuous usage, so few defects occur that inspections become boring.

The more common method for studying defect prevention is to examine the results of large samples of applications and note differences in the defect potentials among them. In other words, if 100 applications that used prototypes are compared with 100 similar applications that did not use prototypes, are requirements defects lower for the prototype sample? Are creeping requirements lower for the prototype sample?

This kind of study can only be carried out internally by rather sophisticated companies that have very sophisticated defect and quality measurement programs; that is, companies such as IBM, AT&T, Microsoft, Raytheon, Lockheed, and the like. (Consultants who work for a number of companies in the same industry can often observe the effects of defect prevention by noting similar applications in different companies.)

However, the results of such large-scale statistical studies are sometimes published from benchmark collections by organizations such as the International Software Benchmarking Standards Group (ISBSG), the David Consulting Group, Software Productivity Research (SPR), and the Quality and Productivity Management Group (QPMG).

In addition, consultants such as the author who work as expert witnesses in software litigation may have access to data that is not otherwise available. This data shows the negative effects of failing to use defect prevention on projects that ended up in court.

Table 9-17 illustrates a large sample of 30 methods and techniques that have been observed to prevent software defects from occurring. Although the table shows specific percentages of defect prevention efficiency, the actual data is too sparse to support the results. The percentages are only approximate and merely serve to show the general order of effectiveness.

TABLE 9-17 Methods and Techniques that Prevent Defects

	Activities Observed to Prevent Software Defects	Defect Prevention Efficiency
1.	Reuse (certified sources)	-80.00%
2.	Inspection participation	-60.00%
3.	Prototyping-functional	-55.00%
4.	PSP/TSP	-53.00%
5.	Six Sigma for software	-53.00%
6.	Risk analysis (automated)	-50.00%
7.	Joint application design (JAD)	-45.00%
8.	Test-driven development (TDD)	-45.00%
9.	Defect origin measurements	-44.00%
10.	Root cause analysis	-43.00%
11.	Quality function deployment (QFD)	-40.00%
12.	CMM 5	-37.00%
13.	Agile embedded users	-35.00%
14.	Risk analysis (manual)	-32.00%
15.	CMM 4	-27.00%
16.	Poka-yoke	-23.00%
17.	CMM 3	-23.00%
18.	Scrum sessions (daily)	-20.00%
19.	Code complexity analysis	-19.00%
20.	Use-cases	-18.00%
21.	Reuse (uncertified sources)	-17.00%
22.	Security plans	-15.00%
23.	Rational Unified Process (RUP)	-15.00%
24.	Six Sigma (generic)	-12.50%
25.	Clean-room development	-12.50%
26.	Software Quality Assurance (SQA)	-12.50%
27.	CMM 2	-12.00%
28.	Total Quality Management (TQM)	-10.00%
29.	No use of CMM	0.00%
30.	CMM 1	5.00%
	Average	-30.12%

Note that because defect prevention deals with reducing defect potentials, percentages show negative values for methods that lower defects. Positive values indicate methods that raise defect potentials.

The two top-ranked items deserve comment. The phrase “reuse from certified sources” implies formal reusability where specifications, source code, test cases, and the like have gone through rigorous inspection and test stages, and have proven themselves to be reliable in field trials. Certified reusable components may approach zero defects, and in any

case contain very few defects. Reuse of uncertified material is somewhat hazardous by comparison.

The second method, or participation in formal inspections, has more than 40 years of empirical data. Inspections of requirements, design, and other deliverables are very effective and efficient in terms of defect removal efficiency. But in addition, participants in formal inspections become aware of defect patterns and categories, and spontaneously avoid them in their own work.

One emerging form of risk analysis is so new that it lacks empirical data. This new method consists of performing very early sizing and risk analysis prior to starting a software application or spending any money on development.

If the risks for the project are significantly higher than its value, not doing it at all will obviously prevent 100 percent of potential defects. The Victorian state government in Australia has started such a program, and by eliminating hazardous software applications before they start, they have saved millions of dollars.

New sizing methods based on pattern matching can shift the point at which risk analysis can be performed about six months earlier than previously possible. This new approach is promising and needs additional study.

There are other things that also have some impact in terms of defect prevention. One of these is certification of personnel either for testing or for software quality assurance knowledge. Certification also has an effect on defect removal. The defect prevention effects are shown using negative percentages, while the defect removal effects are shown with positive percentages.

Here too the data is only approximate, and the specific percentages are derived from very sparse sources and should not be depended upon. Table 9-18 is sorted in terms of defect prevention.

The data in Table 9-18 should not be viewed as accurate, but only approximate. A great deal more research is needed on the effectiveness of various kinds of certification. Also, the software industry circa 2009 has overlapping and redundant forms of certification. There are multiple testing and quality associations that offer certification, but these separate groups certify using different methods and are not coordinated. In the absence of a single association or certification body, these various nonprofit and for-profit test and quality assurance associations offer rival certificates that use very different criteria.

Yet another set of factors that has an effect in terms of defect prevention are various kinds of metrics and measurements, as discussed earlier in this book.

For metrics and measurements to have an effect, they need to be capable of demonstrating quality levels and measuring changes against

TABLE 9-18 Influence of Certification on Defect Prevention and Removal

	Certificate	Defect Prevention Benefit	Defect Removal Benefit
31.	Six Sigma black belt	-12.50%	10.00%
32.	International Software Testing Quality Board (ISTQB)	-12.00%	10.00%
33.	Certified Software Quality Engineer (CSQE)-ASQ	-10.00%	10.00%
34.	Certified. Software Quality Analyst (CSQA)	-10.00%	10.00%
35.	Certified Software Test Manager (CSTM)	-7.00%	7.00%
36.	Six Sigma green belt	-6.00%	5.00%
37.	Microsoft certification (testing)	-6.00%	6.00%
38.	Certified Software Test Professional (CSTP)	-5.00%	12.00%
39.	Certified Software Tester (CSTE)	-5.00%	12.00%
40.	Certified Software Project Manager (CSPM)	-3.00%	3.00%
	Average	-7.65%	8.50%

quality baselines. Therefore, many of the *-ility* measures and metrics cannot even be included because they are not measurable.

Table 9-19 shows the approximate impacts of various measurements and metrics on defect prevention and defect removal. IFPUG function points are top-ranked because they can be used to quantify defects in

TABLE 9-19 Software Metrics, Measures, and Defect Prevention and Removal

	Metric	Defect Prevention Benefit	Defect Removal Benefit
41.	IFPUG function points	-30.00%	15.00%
42.	Six Sigma	-25.00%	20.00%
43.	Cost of quality (COQ)	-22.00%	15.00%
44.	Root cause analysis	-20.00%	12.00%
45.	TSP/PSP	-20.00%	18.00%
46.	Monthly rate of requirements change	-17.00%	5.00%
47.	Goal-question metrics	-15.00%	10.00%
48.	Defect removal efficiency	-12.00%	35.00%
49.	Use-case points	-12.00%	5.00%
50.	COSMIC function points	-10.00%	10.00%
51.	Cyclomatic complexity	-10.00%	7.00%
52.	Test coverage percent	-10.00%	22.00%
53.	Percent of requirements missed	-7.00%	3.00%
54.	Story points	5.00%	-5.00%
55.	Cost per defect	10.00%	-15.00%
56.	Lines of code (LOC)	15.00%	-12.00%
	Average	-11.25%	9.06%

requirements and design as well as code. IFPUG function points can also be used to measure software defect removal costs and quality economics.

Note that the bottom two metrics, cost per defect and lines of code, are shown as harmful metrics rather than beneficial because they violate the assumptions of standard economics.

Note that the two bottom-ranked measurements from Table 9-16 have a negative impact; that is, they make quality worse rather than better. As commonly used in the software literature, both cost per defect and lines of code are close to being professional malpractice, because they violate the canons of standard economics and distort results.

The lines of code metric penalizes high-level languages and makes both the quality and productivity of low-level languages look better than it really is. In addition, this metric cannot even be used to measure requirements and design defects or any other form of noncode defect.

The cost per defect metric penalizes quality and makes buggy applications look better than applications with few defects. This metric cannot even be used for zero-defect applications. A nominal quality metric that penalizes quality and can't even be used to show the highest level of quality is a good candidate for being professional malpractice.

The final aspect of defect prevention discussed in this chapter is that of the effectiveness of various international standards. Unfortunately, the effectiveness of international standards has very little empirical data available.

There are no known controlled studies that demonstrate if adherence to standards improves quality. There is some anecdotal evidence that at least some standards, such as ISO 9001-9004, degrade quality because some companies that did not use these standards had higher quality on similar applications than companies that had been certified. Table 9-20 shows approximate results, but the table has two flaws. It only shows a small sample of standards, and the rankings are based on very sparse and imperfect information.

In fields outside of software such as medical practice, standards are normally validated by field trials, controlled studies, and extensive analysis. For software, standards are not validated and are based on the subjective views of the standards committees. While some of these committees are staffed by noted experts and the standards may be useful, the lack of validation and field trials prior to publication is a sign that software engineering needs additional evolution before being classified as a full engineering discipline.

Tables 9-17 through 9-20 illustrate a total of 65 defect prevention methods and practices. These are not all used at the same time. Table 9-18 shows the approximate usage patterns observed in several hundred U.S. companies (and in about 50 overseas companies).

TABLE 9-20 International Standards, Defect Prevention and Removal

	Standard or Government Mandate	Defect Prevention Benefit	Defect Removal Benefit
57.	ISO/IEC 10181 Security Frameworks	-25.00%	25.00%
58.	ISO 17799 Security	-15.00%	15.00%
59.	Sarbanes-Oxley	-12.00%	6.00%
60.	ISO/IEC 25030 Software Product Quality Requirements	-10.00%	5.00%
61.	ISO/IEC 9126-1 Software Engineering Product Quality	-10.00%	5.00%
62.	IEEE 730-1998 Software Quality Assurance Plans	-8.00%	5.00%
63.	IEEE 1061-1992 Software Metrics	-7.00%	2.00%
64.	ISO 9000-9003 Quality Management	-6.00%	5.00%
65.	ISO 9001:2000 Quality Management System	-4.00%	7.00%
	Average	-10.78%	8.33%

Table 9-21 is somewhat troubling because the three top-ranked methods have been demonstrated to be harmful and make quality worse rather than better. In fact, of the really beneficial defect prevention methods, only a handful such as prototyping, measuring test coverage, and joint application design (JAD) have more than 50 percent usage in the United States.

Usage of many of the most powerful and effective methods such as inspections or measuring cost of quality (COQ) have less than 33 percent usage or penetration. The data shown in Table 9-18 is not precise, since much larger samples would be needed. However, it does illustrate a severe disconnect between effective methods of defect prevention and day-to-day usage in the United States.

Part of the reason for the dismaying patterns of usage is because of the difficulty of actually measuring and studying defect prevention methods. Only a few large and sophisticated corporations are able to carry out studies of defect prevention. Most universities cannot study defect prevention because they lack sufficient contacts with corporations and therefore have little data available.

In conclusion, defect prevention is sparsely covered in the software literature. There is very little empirical data available, and a great deal more research is needed on this topic.

One way to improve defect prevention and defect removal would be to create a nonprofit foundation or association that studied a wide range of quality topics. Both defect prevention and defect removal would be included. Following is the hypothetical structure and functions of a proposed nonprofit International Software Quality Foundation (ISQF).

TABLE 9-21 Usage Patterns of Software Defect Prevention Methods

	Defect Prevention Method	Percent of U.S. Projects
1.	Reuse (uncertified sources)	90.00%
2.	Cost per defect	75.00%
3.	Lines of code (LOC)	72.00%
4.	Prototyping-functional	70.00%
5.	Test coverage percent	67.00%
6.	No use of CMM	50.00%
7.	Joint application design (JAD)	45.00%
8.	Percent of requirements missed	38.00%
9.	Software quality assurance (SQA)	36.00%
10.	Use-cases	33.00%
11.	IFPUG function points	33.00%
12.	Test-driven development (TDD)	30.00%
13.	Cost of quality (COQ)	29.00%
14.	Scrum sessions (daily)	28.00%
15.	CMM 3	28.00%
16.	Agile embedded users	27.00%
17.	Six Sigma	24.00%
18.	Risk analysis (manual)	22.00%
19.	Rational Unified Process (RUP)	22.00%
20.	Cyclomatic complexity	21.00%
21.	CMM 1	20.00%
22.	Monthly rate of requirements change	20.00%
23.	Code complexity analysis	19.00%
24.	ISO 9001:2000 Quality Management System	19.00%
25.	Microsoft certification (testing)	18.00%
26.	ISO 9000-9003 Quality Management	18.00%
27.	Root cause analysis	17.00%
28.	ISO/IEC 9126-1 Software Engineering Product Quality	17.00%
29.	TSP/PSP	16.00%
30.	ISO/IEC 25030 Software Product Quality Requirements	16.00%
31.	IEEE 1061-1992 Software Metrics	16.00%
32.	Defect origin measurements	15.00%
33.	Root cause analysis	15.00%
34.	IEEE 730-1998 Software Quality Assurance Plans	15.00%
35.	PSP/TSP	14.00%
36.	Six Sigma for software	13.00%
37.	Six Sigma (generic)	13.00%
38.	Story points	13.00%

(Continued)

TABLE 9-21 Usage Patterns of Software Defect Prevention Methods (*continued*)

	Defect Prevention Method	Percent of U.S. Projects
39.	Inspection participation	12.00%
40.	CMM 2	12.00%
41.	Sarbanes-Oxley	12.00%
42.	Six Sigma green belt	11.00%
43.	ISO/IEC 10181 Security Frameworks	11.00%
44.	Six Sigma black belt	10.00%
45.	Defect removal efficiency	10.00%
46.	Use-case points	10.00%
47.	ISO 17799 Security	10.00%
48.	Goal-Question Metrics	9.00%
49.	CMM 4	8.00%
50.	Certified Software Test Professional (CSTP)	8.00%
51.	Security plans	7.00%
52.	Quality function deployment (QFD)	6.00%
53.	Total quality management (TQM)	6.00%
54.	Certified Software Project Manager (CSPM)	6.00%
55.	International Software Testing Quality Board (ISTQB)	4.00%
56.	Certified Software Quality Analyst (CSQA)	4.00%
57.	Certified Software Tester (CSTE)	4.00%
58.	COSMIC function points	4.00%
59.	Certified Software Quality Engineer (CSQE) – ASQ	3.00%
60.	Risk analysis (automated)	2.00%
61.	Certified Software Test Manager (CSTM)	2.00%
62.	Reuse (certified sources)	1.00%
63.	CMM 5	1.00%
64.	Poka-yoke	0.10%
65.	Clean-room development	0.10%

Proposal for a Nonprofit International Software Quality Foundation (ISQF)

The ISQF will be a nonprofit foundation that is dedicated to improving the quality and economic value of software applications. The form of incorporation is to be decided by the initial board of directors. The intent is to incorporate under section 501(c) of the Internal Revenue Code and thereby be a tax-exempt organization that is authorized to receive donations.

The fundamental principles of ISQF are the following:

1. Poor quality has been and is damaging the professional reputation of the software community.

2. Poor quality has been and is causing significant litigation between clients and software development corporations.
3. Significant software quality improvements are technically possible.
4. Improved software quality has substantial economic benefits in reducing software costs and schedules.
5. Improved software quality depends upon accurate measurement of quality in many forms, including, but not limited to, measuring software defects, software defect origins, software defect severity levels, methods of defect prevention, methods of defect removal, customer satisfaction, and software team morale.
6. The major cost of software development and maintenance is that of eliminating defects. ISQF will mount major studies on measuring the economic value of defect prevention, defect removal, and customer satisfaction.
7. Measurement and estimation are synergistic technologies. ISQF will evaluate software quality and reliability estimation methods, and will publish the results of their evaluations. No fees from estimation tool vendors will be accepted. The evaluations will be independent and based on standard benchmarks and test cases.
8. Software defects can originate in requirements, design, coding, user documents, and also in test plans and test cases themselves. In addition, there are secondary defects that are introduced while attempting to repair earlier defects. ISQF will study all sources of software problems and attempt to improve all sources of software defects and user dissatisfaction.
9. ISQF will sponsor research in technical topics that may include, but are not be limited to, inspections, static analysis, test case design, test coverage analysis, test tools, defect reporting, defect tracking tools, bad-fix injections, error-prone module removal, complexity analysis, defect prevention, formal inspections, quality measurements, and quality metrics.
10. The ISQF will also sponsor research to quantify the effects of all social factors that influence software quality, including the effectiveness of software quality assurance organizations (SQA), separate test organizations, separate maintenance organizations, international standards, and the value of certification. Methods of studying software customer satisfaction will also be supported.
11. The service metrics defined in the Information Technology Infrastructure Library (ITIL) are all dependent upon achieving satisfactory levels of quality. ISQF will incorporate principles from the ITIL library, and will also sponsor research studies to show the

correlations between reliability and availability and quality levels in terms of delivered defects.

12. As new technologies appear in the software industry, it is important to stay current with their quality impacts. ISQF will perform or commission studies on the quality results of a variety of new approaches including but not limited to Agile development, cloud computing, crystal development, extreme programming, open-source development, and service-oriented architecture (SOA).
13. ISQF will provide model curricula for university training in software measurement, metrics, defect prevention, defect removal, customer support, customer satisfaction, and the economic value of software quality.
14. ISQF will provide model curricula for MBA programs that deal with the economics of software and the principles of software management. The economics of quality will be a major subtopic.
15. ISQF will provide model curricula for corporate and in-house training in software measurement, metrics, defect prevention, defect removal, customer support, customer satisfaction, and the economic value of software quality.
16. ISQF will provide recommended skill profiles for the occupations of software quality assurance (SQA), software testing, software customer support, and software quality measurement.
17. ISQF will offer examinations and licensing certificates for the occupations of software quality assurance (SQA), software testing, software customer support, and software quality measurement. Of these, software quality measurement has no current certification.
18. ISQF will establish boards of competence to administer examinations and define the state of the art for software quality assurance (SQA), software testing, and software quality measurement. Other boards and specialties may be added at future times.
19. ISQF will define the conditions of professional malpractice as they apply to inadequate methods of software quality control. Examples of such conditions may include failing to keep adequate records of software defects, failing to utilize sufficient test stages and test cases, and failing to perform adequate inspections of critical materials.
20. ISQF will cooperate with other nonprofit organizations that are concerned with similar issues. These organizations include but are not limited to the Global Association for Software Quality (GASQ) in Belgium, the World Quality Conference, the IEEE, the ISO, ANSI, IFPUG, SPIN, and the SEI. IASQ will also cooperate with other organizations such as universities, the Information Technology

Metrics and Productivity Institute (ITMPI), the Project Management Institute (PMI), the Quality Assurance Institute (QAI), software testing societies, and relevant engineering, benchmarking, and professional organizations such as the ISBSG benchmarking group. ISQF will also cooperate with similar quality organizations abroad such as those in China, Japan, India, and Russia. This cooperation might include reciprocal memberships if other organizations are willing to participate in that fashion.

21. ISQF will be governed by a board of five directors, to be elected by the membership. The board of directors will appoint a president or chief executive officer. The president will appoint a treasurer, secretary, and such additional officers as may be required by the terms and place of incorporation. Initially, the board, president, and officers will serve as volunteers on a pro bono basis. To ensure inclusion of fresh information, the term of president will be two calendar years.
22. Funding for the ISQF will be a combination of dues, donations, grants, and possible fund-raising activities such as conferences and events.
23. The ISQF will also have a technical advisory board of five members to be appointed by the president. The advisory board will assist ISQF in staying at the leading edge of research into topics such as testing, inspections, quality metrics, and also availability and reliability and other ITIL metrics.
24. The ISQF will use modern communication methods to expand the distribution of information on quality topics. These methods will include an ISQF web site, webinars, a possible quality Wikipedia, Twitter, blogs, and online newsletters.
25. The ISQF will have several subcommittees that deal with topics such as membership, grants and donations, press liaison, university liaison, and liaison with other nonprofit organizations such as the Global Association of Software Quality in Belgium.
26. To raise awareness of the importance of quality, the ISQF will produce a quarterly journal, with a tentative name of *Software Quality Progress*. This will be a refereed journal, with the referees all coming from the ISQF membership.
27. To raise awareness of the importance of quality, the ISQF will sponsor an annual conference and will solicit nominations for a series of "outstanding quality awards." The initial set of awards will be organized by type of software (information systems, commercial applications, military software, outsourced applications, systems and embedded software, web applications). The awards will be for

lowest numbers of delivered defects, highest levels of defect removal efficiency, best customer service, and highest rankings of customer satisfaction.

28. To raise awareness of the importance of software quality, ISQF members will be encouraged to write and review articles and books on software quality topics. Both technical journals such as *CrossTalk* and mainstream business journals such as the *Harvard Business Review* will be journals of choice.
29. To raise awareness of the importance of software quality, ISQF will begin the collection of a major library of books, journal articles, and monographs on topics and issues associated with software quality.
30. To raise awareness of the importance of software quality, ISQF will sponsor benchmark studies of software defects, defect severity levels, defect removal efficiency, test coverage, inspection efficiency, inspection and test costs, cost of quality (COQ), and software litigation where poor quality was one of the principal complaints by the plaintiffs.
31. To raise awareness of the economic consequences of poor quality, the ISQF will sponsor research on consequential damages, deaths, and property losses associated with poor software quality.
32. To raise awareness of the economic consequences of poor quality, the ISQF will collect public information on the results of software litigation where poor quality was part of the plaintiff's claims. Such litigation includes breach of contract cases, fraud cases, and cases where poor quality damaged plaintiff business operations.
33. To raise awareness of the importance of software quality, ISQF chapters will be encouraged at state and local levels, such as Rhode Island Software Quality Association or a Boston Software Quality Association.
34. To ensure high standards of quality education, the ISQF will review and certify specific courses on software quality matters offered by universities and private corporations as well. Courses will be submitted for certification on a voluntary basis. Minimal fees will be charged for certification in order to defray expenses. Fees will be based on time and material charges and will be levied whether or not a specific course passes certification or is denied certification.
35. To ensure that quality topics are included and are properly defined in contracts and outsource agreements, the ISQF will cooperate with the American Bar Association, state bar associations, the American Arbitration Society, and various law schools on the legal status of software quality and on contractual issues.

36. ISQF members will be asked to subscribe to a code of ethics that will be fully defined by the ISQF technical advisory board. The code of ethics will include topics such as providing full and honest information about quality to all who ask, avoiding conflicts of interest, and basing recommendations about quality on solid empirical information.
37. Because security and quality are closely related, the ISQF will also include security attack prevention and also recovery from security attacks topics as part of the overall mission. However, security is highly specialized and requires additional skills outside the normal training of quality assurance and test personnel.
38. Because of the serious global recession, the ISQF will attempt to rapidly disseminate empirical data on the economic value of quality. High quality for software has been proven to shorten development schedules, lower development costs, improve customer support, and reduce maintenance costs. But few managers and executives have access to the data that supports such claims.

Software engineering and software quality need to be more closely coupled than has been the norm in the past. Better prediction of quality, better measurement of quality, more widespread usage of effective defect prevention methods and defect removal methods are all congruent with advancing software engineering to the status of a true engineering discipline.

Software Defect Removal

Although both defect prevention and defect removal are important, it is easier to study and measure defect removal. This is because counts of defects found by means of inspections, static analysis, and testing provide a quantitative basis for calculating defect removal efficiency levels.

In spite of the fact that defect removal is theoretically easy to study, the literature remains distressingly sparse. For example, testing has an extensive literature with hundreds of books, thousands of journal articles, many professional associations, and numerous conferences. Yet hardly any of the testing literature contains empirical data on the measured numbers of test cases created, actual counts of defects found and removed, data on bad-fix injection rates, or other tangible data points.

Several important topics have almost no citations at all in the testing literature. For example, a study done at IBM found more errors in test cases than in the software that was being tested. The same study

found about 35 percent duplicate or redundant test cases. Yet neither test case errors nor redundant test cases are discussed in the software testing literature.

Another gap in the literature of both testing and other forms of defect removal concerns bad-fix injections. About 7 percent of attempts to repair software defects contain new defects in the repairs themselves. In fact, sometimes there are secondary and even tertiary bad fixes; that is, three consecutive attempts to fix a bug may fail to fix the original bug and introduce new bugs that were not there before!

Another problem with the software engineering literature and also with software professional associations is a very narrow focus. Most testing organizations tend to ignore static analysis and inspections.

As a result of this narrow focus, the synergies among various kinds of defect removal operations are not well covered in the quality or software engineering literature. For example, carrying out formal inspections of requirements and design not only finds defects, but also raises the defect removal efficiency levels of subsequent test stages by at least 5 percent by providing better and more complete source material for constructing test cases.

Running automated static analysis prior to testing will find numerous defects having to do with limits, boundary conditions, and structural problems, and therefore speed up subsequent testing.

Formal inspections are best at finding very complicated and subtle problems that require human intelligence and insight. Formal inspections are also best at finding errors of omission and errors of ambiguity.

Static analysis is best at finding structural and mechanical problems such as boundary conditions, duplications, failures of error-handling, and branches to incorrect routines. Static analysis can also find security flaws.

Testing is best at finding problems that occur when software is executing, such as performance issues, usability issues, and security issues.

Individually, these three methods are useful but incomplete. When used together, their synergies can elevate defect removal efficiency levels and also reduce the effort and costs associated with defect removal activities.

Table 9-22 provides an overview of 80 different forms of software defect removal: static analysis, inspections, many kinds of testing, and some special forms of defect removal associated with software litigation.

Although Table 9-22 shows overall values for defect removal efficiency, the data really deals with removal efficiency against selected defect categories. For example, automated static analysis might find 87 percent of structural code problems, but it can't find requirements omissions or problems such as the Y2K problem that originate in requirements.

TABLE 9-22 Overview of 80 Varieties of Software Defect Removal Activities

DEFECT REMOVAL ACTIVITIES				
Activities		Number of Test Cases per FP	Defect Removal Efficiency	Bad-Fix Injection Percent
STATIC ANALYSIS				
1.	Automated static analysis	0.00	87.00%	2.00%
2.	Requirements inspections	0.00	85.00%	6.00%
3.	External design inspection	0.00	85.00%	6.00%
4.	Use-case inspection	0.00	85.00%	4.00%
5.	Internal design inspection	0.00	85.00%	4.00%
6.	New code inspections	0.00	85.00%	4.00%
7.	Reuse certification inspection	0.00	84.00%	2.00%
8.	Test case inspection	0.00	83.00%	5.00%
9.	Automated document analysis	0.00	83.00%	6.00%
10.	Legacy code inspections	0.00	83.00%	6.00%
11.	Quality function deployment	0.00	82.00%	3.00%
12.	Document proof reading	0.00	82.00%	1.00%
13.	Nationalization inspection	0.00	81.00%	3.00%
14.	Architecture inspections	0.00	80.00%	3.00%
15.	Test plan inspection	0.00	80.00%	5.00%
16.	Test script inspection	0.00	78.00%	4.00%
17.	Test coverage analysis	0.00	77.00%	3.00%
18.	Document editing	0.00	77.00%	2.50%
19.	Pair programming review	0.00	75.00%	5.00%
20.	Six Sigma analysis	0.00	75.00%	3.00%
21.	Bug repair inspection	0.00	70.00%	3.00%
22.	Business plan inspections	0.00	70.00%	8.00%
23.	Root-cause analysis	0.00	65.00%	4.00%
24.	Governance reviews	0.00	63.00%	5.00%
25.	Refactoring of code	0.00	62.00%	5.00%
26.	Error-prone module analysis	0.00	60.00%	10.00%
27.	Independent audits	0.00	55.00%	10.00%
28.	Internal audits	0.00	52.00%	10.00%
29.	Scrum sessions (daily)	0.00	50.00%	2.00%
30.	Quality assurance review	0.00	45.00%	7.00%
31.	Sarbanes-Oxley review	0.00	45.00%	10.00%
32.	User story reviews	0.00	40.00%	10.00%
33.	Informal peer reviews	0.00	40.00%	10.00%
34.	Independent verification and validation	0.00	35.00%	12.00%
35.	Private desk checking	0.00	35.00%	7.00%

(Continued)

TABLE 9-22 Overview of 80 Varieties of Software Defect Removal Activities
(continued)

DEFECT REMOVAL ACTIVITIES				
	Activities	Number of Test Cases per FP	Defect Removal Efficiency	Bad-Fix Injection Percent
36.	Phase reviews	0.00	30.00%	15.00%
37.	Correctness proofs	0.00	27.00%	20.00%
	Average	0.00	66.92%	6.09%
GENERAL TESTING				
38.	PSP/TSP unit testing	3.50	52.00%	2.00%
39.	Subroutine testing	0.25	50.00%	2.00%
40.	XP testing	2.00	40.00%	3.00%
41.	Component testing	1.75	40.00%	3.00%
42.	System testing	1.50	40.00%	7.00%
43.	New function testing	2.50	35.00%	5.00%
44.	Regression testing	2.00	30.00%	7.00%
45.	Unit testing	3.00	25.00%	4.00%
	Average	2.06	41.00%	4.13%
	Sum	16.50		
AUTOMATIC TESTING				
46.	Virus/spyware test	3.50	80.00%	4.00%
47.	System test	2.00	40.00%	8.00%
48.	Regression test	2.00	37.00%	7.00%
49.	Unit test	0.05	35.00%	4.00%
50.	New function test	3.00	35.00%	5.00%
	Average	2.11	45.40%	5.60%
	Sum	10.55		
SPECIALIZED TESTING				
51.	Virus testing	0.70	98.00%	2.00%
52.	Spyware testing	1.00	98.00%	2.00%
53.	Security testing	0.40	90.00%	4.00%
54.	Limits/capacity testing	0.50	90.00%	5.00%
55.	Penetration testing	4.00	90.00%	4.00%
56.	Reusability testing	4.00	88.00%	0.25%
57.	Firewall testing	2.00	87.00%	3.00%
58.	Performance testing	0.50	80.00%	7.00%
59.	Nationalization testing	0.30	75.00%	10.00%
60.	Scalability testing	0.40	65.00%	6.00%
61.	Platform testing	0.20	55.00%	5.00%
62.	Clean-room testing	3.00	45.00%	7.00%
63.	Supply chain testing	0.30	35.00%	10.00%

TABLE 9-22 Overview of 80 Varieties of Software Defect Removal Activities
(continued)

DEFECT REMOVAL ACTIVITIES				
	Activities	Number of Test Cases per FP	Defect Removal Efficiency	Bad-Fix Injection Percent
64.	SOA orchestration	0.20	30.00%	5.00%
65.	Independent testing	0.20	25.00%	12.00%
	Average	1.18	70.07%	5.48%
	Sum	17.70		
USER TESTING				
66.	Usability testing	0.25	65.00%	4.00%
67.	Local nationalization testing	0.40	60.00%	3.00%
68.	Lab testing	1.25	45.00%	5.00%
69.	External beta testing	1.00	40.00%	7.00%
70.	Internal acceptance testing	0.30	30.00%	8.00%
71.	Outsource acceptance testing	0.05	30.00%	6.00%
72.	COTS acceptance testing	0.10	25.00%	8.00%
	Average	0.48	42.14%	5.86%
	Sum	3.35		
LITIGATION ANALYSIS, TESTING				
73.	Intellectual property testing	2.00	80.00%	1.00%
74.	Intellectual property review	0.00	80.00%	3.00%
75.	Breach of contract review	0.00	80.00%	2.00%
76.	Breach of contract testing	2.00	70.00%	2.00%
77.	Tax litigation review	0.00	80.00%	4.00%
78.	Tax litigation testing	1.00	70.00%	4.00%
79.	Fraud code review	0.00	80.00%	2.00%
80.	Embezzlement code review	0.00	80.00%	2.00%
	Average	2.35	77.14%	2.71%
	Sum	5.00		
	TOTAL TEST CASES	53.10		
	PER FUNCTION POINT			

Table 9-22 is sorted in descending order of defect removal efficiency. However, the results shown are maximum values. In real life, the range of measured defect removal efficiency can be less than half of the nominal maximum values shown in Table 9-18.

Although Table 9-22 lists 80 different kinds of software defect removal activities, that does not imply that all of them are used at the same time.

In fact, the U.S. average for defect removal activities includes only six kinds of testing:

U.S. Average Sequence of Defect Removal

1. Unit test
2. New function test
3. Performance test
4. Regression test
5. System test
6. Acceptance or beta test

These six forms of testing, collectively, range between about 70 percent and 85 percent in cumulative defect removal efficiency levels: far below what is needed to achieve high levels of reliability and customer satisfaction. The bottom line is that testing, by itself, is insufficient to achieve professional levels of quality.

An optimum sequence of defect removal activities would include several kinds of pretest inspections, static analysis, and at least eight forms of testing:

Optimal Sequence of Software Defect Removal

Pretest Defect Removal

1. Requirements inspection
2. Architecture inspection
3. Design inspection
4. Code inspection
5. Test case inspection
6. Automated static analysis

Testing Defect Removal

7. Subroutine test
8. Unit test
9. New function test
10. Security test
11. Performance test
12. Usability test

13. System test
14. Acceptance or beta test

This combination of synergistic forms of defect removal will achieve cumulative defect removal efficiency levels in excess of 95 percent for every software project and can achieve 99 percent for some projects.

When the most effective forms of defect removal are combined with the most effective forms of defect prevention, then software engineering should be able to achieve consistent levels of excellent quality. If this combination can occur widely enough to become the norm, then software engineering can be considered a true engineering discipline.

Software Quality Specialists

As noted earlier in the book, more than 115 types of occupations and specialists are working in the software engineering domain. In most knowledge-based occupations such as medicine and law, specialists have extra training and sometimes extra skills that allow them to outperform generalists in selected areas such as in neurosurgery or maritime law.

For software engineering, the literature is sparse and somewhat ambiguous about the roles of specialists. Much of the literature on software specialization is vaporous and merely expresses some kind of bias. Many authors prefer a generalist model where individuals are interchangeable and can handle requirements, design, development, and testing as needed. Other authors prefer a specialist model where key skills such as testing, quality assurance, and maintenance are performed by trained specialists.

In this chapter, we will focus primarily on two basic questions:

1. Do specialized skills lower defect potentials and benefit defect prevention?
2. Do specialized skills raise defect removal efficiency levels?

Not all of the 115 or so specialists will be discussed, but those whose roles have a potential impact on quality levels will be discussed in terms of defect prevention and defect removal.

The 20 specialist categories discussed in this chapter include, in alphabetical order:

1. Architects
2. Business analysts
3. Database analysts
4. Data quality analysts

5. Enterprise architects
6. Estimating specialists
7. Function point specialists
8. Inspection moderators
9. Maintenance specialists
10. Requirements analysts
11. Performance specialists
12. Risk analysis specialists
13. Security specialists
14. Six Sigma specialists
15. Systems analysts
16. Software quality assurance (SQA)
17. Technical writers
18. Testers
19. Usability specialists
20. Web designers

For each of these 20 specialist groups, we will consider the volume of potential defects they face, and whether they have a tangible impact on defect prevention and defect removal activities.

Table 9-23 ranks the specialists in terms of *assignment scope*. This metric represents the number of function points normally assigned to one practitioner. Table 9-23 also shows the volume of defects that the various occupations face as part of their jobs. Table 9-23 then shows the approximate impacts of these specialized occupations on both defect prevention and defect removal.

The top-ranked specialists face large numbers of potential defects that are also capable of causing great damage to entire corporations as well as to the software applications owned by those corporations. Following are short discussions of each of the 20 kinds of specialists.

Risk Analysis Specialists

Assignment scope = 300,000 function points

Defect potentials = 7.00

Defect prevention impact = -75 percent

Defect removal impact = 25 percent

The large assignment scope of 300,000 function points indicates that companies do not need many risk analysts, but the ones they employ need to be very competent and understand both technical and financial risks.

TABLE 9-23 Software Specialization Impact on Software Quality

	Specialized Occupations	Assignment Scope	Defect Potential	Defect Prevention	Defect Removal
1.	Risk analysis specialists	300,000	7.00	75.00%	25.00%
2.	Enterprise architects	250,000	6.00	25.00%	20.00%
3.	Six Sigma specialists	250,000	5.00	25.00%	30.00%
4.	Database analysts	100,000	3.00	15.00%	10.00%
5.	Architects	100,000	3.00	17.00%	12.00%
6.	Usability specialists	100,000	1.00	10.00%	15.00%
7.	Security specialists	50,000	7.00	70.00%	20.00%
8.	Data quality analysts	50,000	5.00	12.00%	15.00%
9.	Business analysts	50,000	3.50	25.00%	10.00%
10.	Estimating specialists	25,000	3.00	20.00%	25.00%
11.	Systems analysts	20,000	6.00	20.00%	20.00%
12.	Performance specialists	20,000	1.00	10.00%	12.00%
13.	Quality assurance (QA)	10,000	5.50	15.00%	40.00%
14.	Web designers	10,000	4.00	15.00%	12.00%
15.	Requirements analysts	10,000	4.00	20.00%	15.00%
16.	Testers	10,000	3.00	15.00%	50.00%
17.	Function point specialists	5,000	4.00	10.00%	10.00%
18.	Technical writers	2,000	1.00	10.00%	10.00%
19.	Maintenance specialists	1,500	3.50	30.00%	20.00%
20.	Inspection moderators	1,000	4.50	27.00%	35.00%
	Average	68,225	4.00	23.30%	20.30%

Given the enormous number of business failures as part of the recession, it is obvious that risk analysis is not yet as sophisticated as it should be; especially for dealing with financial risks.

Risk analysts face more than 100 percent of the potential defects associated with any given software application. Not only do they have to deal with technical risks and quality risks, but they also need to address financial risks and legal risks that are outside the normal realm of software quality and defect measurement.

A formal and careful risk analysis prior to committing funds to a major software application can stop investments in excessively hazardous projects before any serious money is spent. For questionable projects, a formal and careful risk analysis prior to starting the project can introduce better technologies prior to committing funds.

The keys to successful early risk analysis include the ability to do early sizing, early cost estimating, early quality estimating, and knowledge of dozens of potential risks derived from analysis of project failures and successes.

The main role of risk analysts in terms of quality are to stop bad projects before they start, and to ensure that projects that do start utilize state-of-the-art quality methods. Risk analysts also need to understand the main reasons for software failures, and they should be familiar with software litigation results for cases dealing with cancelled projects, breach of contract, theft of intellectual property, patent violations, embezzlement via software, fraud, tax issues, Sarbanes-Oxley issues, and other forms of litigation as well.

Enterprise Architects

Assignment scope = 250,000 function points

Defect potentials = 6.00

Defect prevention impact = -25 percent

Defect removal impact = 20 percent

Enterprise architects are key players whose job is to understand every aspect of corporate business and to match business needs against entire portfolios, which may contain more than 3000 separate applications and total to more than 10 million function points. Not only internal software, but also open-source applications and commercial software packages such as Vista and SAP need to be part of the enterprise architect's domain of knowledge.

The main role of enterprise analysts in terms of quality is to understand the business value of quality to corporate operations, and to ensure that top executives have similar understandings. Both enterprise architects and corporate executives need to push for excellence in order to achieve speed of delivery.

Enterprise architects also play a role in corporate governance, by ensuring that critical mistakes such as the Y2K problem are prevented from occurring in the future.

Six Sigma Specialists

Assignment scope = 250,000 function points

Defect potentials = 5.00

Defect prevention impact = -25 percent

Defect removal impact = 30 percent

The large assignment scope for Six Sigma specialists indicates that their work is corporate in nature rather than being limited to specific applications. The main role of Six Sigma specialists in terms of quality is to provide expert analysis of defect origins and defect causes, and to suggest effective methods of continuous improvement to reduce the major sources of software error.

Database Analysts

Assignment scope = 100,000 function points

Defect potentials = 7.00

Defect prevention impact = -75 percent

Defect removal impact = -25 percent

In today's world of 2009, major corporations and government agencies own even more data than they own software. Customer data, employee data, manufacturing data, total to millions of records scattered over dozens of databases and repositories. This collection of enterprise data is a valuable asset that needs to be accessed for key business decisions, and also protected against hacking, theft, and unauthorized access.

There is a major quality weakness in 2009 in the area of data quality. There are no "data point" metrics that express the size of databases and repositories. As a result, it is very hard to quantify data quality. In fact, for all practical purposes, no literature at all on data quality uses actual counts of errors.

As a result, database analysts and data quality analysts are severely handicapped. They both play key roles in quality, but lack all of the tools they need to do a good job.

The major role played by database analysts in terms of quality is to ensure that databases and repositories are designed and organized in optimal fashions, and that processes are in place to validate the accuracy of all data elements that are added to enterprise data storage.

Architects

Assignment scope = 100,000 function points

Defect potentials = 3.00

Defect prevention impact = -17 percent

Defect removal impact = 12 percent

Architects also have a large assignment scope, and need to be able to envision and deal with the largest known applications of the modern world, such as Vista, ERP packages like SAP and Oracle, air-traffic control, defense applications, and major business applications.

Over the past 50 years, software applications have evolved from running by themselves to running under an operating system to running as part of a multitier network and indeed to running in fragments scattered over a cloud of hardware and software platforms that may be thousands of miles apart.

As a result, the role of architects has become much more complex in 2009 than it was even ten years ago. Architects need to understand modern application practices such as service-oriented architecture (SOA),

cloud computing, and multitier hierarchies. In addition, architects need to know the sources and certification methods of various kinds of reusable material that constitutes more than 50 percent of many large applications circa 2009.

The main role that architects play in terms of quality is to understand the implications of software defects in complex, multitier, highly distributed environments where software components may come from dozens of sources.

Usability Specialists

Assignment scope = 100,000 function points

Defect potentials = 1.00

Defect prevention impact = -10 percent

Defect removal impact = 15 percent

The word “usability” defines what customers need to do to operate software successfully. It also includes what software customers need to do when the software misbehaves.

Usability specialists often have a background in cognitive psychology and are well versed in various kinds of software interfaces: keyboard commands, buttons, touch screens, voice recognitions, and even more.

The main role of usability specialists in terms of quality is to ensure that software applications have interfaces and control sequences that are as natural and intuitive as possible. Usability studies are normally carried out with volunteer clients who use the software while it is under development.

Large computer and software companies such as IBM and Microsoft have usability laboratories where customers can be observed while they are using prerelease versions of software and hardware products. These labs monitor keystrokes, screen touches, voice commands, and other interface methods. Usability specialists also debrief customers after every session to find out what customers like and dislike about interfaces and command sequences.

Security Specialists

Assignment scope = 50,000 function points

Defect potentials = 7.00

Defect prevention impact = -70 percent

Defect removal impact = 20 percent

There is an increasing need for more software security specialists, and also for better training of software security specialists both at the university level and after employment, as security threats evolve and change.

As of 2009, due in part to the recession, attacks and data theft are increasing rapidly in numbers and sophistication. Hacking is rapidly moving from the domain of individual amateurs to organized crime and even to hostile foreign governments.

Software applications are not entirely safe behind firewalls, even with active antivirus and antispyware applications installed. There is an urgent need to raise the immunity levels of software applications by using techniques such as Google's Caja, the E programming language, and changing permission schemas.

Security and quality are not identical, but they are very close together, and both prevention and removal methods are congruent and synergistic. The closeness of quality and security is indicated by the fact that major avenues of attack on software applications are error-handling routines.

The main role of security specialists in terms of quality is to stay current on the latest kinds of threats, and to ensure that both new applications and legacy applications have state-of-the-art security defenses.

Data Quality Analysts

Assignment scope = 50,000 function points

Defect potentials = 5.00

Defect prevention impact = -12 percent

Defect removal impact = 15 percent

As of 2009, data quality analysts are few in number and woefully under-equipped in terms of tools and technology. There is no effective size metric for data volumes (i.e., a data point metric similar to function points). As a result, no empirical data is available on topics such as defect potentials for databases and repositories, effective defect removal methods, defect estimation, or defect measurement.

The theoretical role of data quality analysts is to prevent data errors from occurring, and to recommend effective removal methods. However, given the very large number of apparent data errors in public records, credit scores, accounting, taxes, and so on, it is obvious that data quality lags behind even software quality. In fact, data and software appear to lag behind every other engineering and technical domain in terms of quality control.

Business Analysts

Assignment scope = 50,000 function points.

Defect potentials = 3.5

Defect prevention impact = -25 percent

Defect removal impact = 10 percent

In many information technology organizations, business analysts are the primary connection between the software community and the

community of software users. Business analysts are required to be well versed in both business needs and in software engineering technologies.

The main role that business analysts should play in terms of quality is to convince both the business and technical communities that high levels of software quality will shorten development schedules and lower development costs. Too often, the business clients set arbitrary schedules and then attempt to force the software community to try and meet those schedules by skimping on inspections and truncating testing.

Good business analysts should have data available from sources such as the International Software Benchmarking Standards Group (ISBSG) that shows the relationships between quality, schedules, and costs. Business analysts should also understand the value of methods such as joint application design (JAD), quality function deployment (QFD), and requirements inspections.

Estimating Specialists

Assignment scope = 25,000 function points

Defect potentials = 3.00

Defect prevention impact = -20

Defect removal impact = 25 percent

It is a sign of sophistication when a company employs software estimating specialists. Usually these specialists work in project offices or special staff groups that support line managers, who often are not well trained in estimation.

Estimation specialists should have access to and be familiar with the major software estimating tools that can predict quality, schedules, and costs. Examples of such tools include COCOMO, KnowledgePlan, Price-S, SoftCost, SEER, Slim, and a number of others. In fact, a number of companies utilize several of these tools for the same applications and look for convergence.

The main role of an estimating specialist in terms of quality is to predict quality early. Ideally, quality will be predicted before substantial funds are spent. Not only that, but multiple estimates may be needed to show the effects of variations in development practices such as Agile development, Team Software Process (TSP), Rational Unified Process (RUP), formal inspections, static analysis, and various kinds of testing.

Systems Analysts

Assignment scope = 20,000 function points

Defect potentials = 6.00

Defect prevention impact = -25 percent

Defect removal impact = 20 percent

Software systems analysts are one of the interface points between the software engineering or programming community and end users of software. Systems analysts and business analysts perform similar roles, but the title “systems analyst” occurs more often for embedded and systems software, which are developed for technical purposes rather than to satisfy local business needs.

The main role of systems analysts in terms of quality is to understand that all forms of representation for software (user stories, use-cases, formal specification languages, flowcharts, Nassi-Schneiderman charts, etc.) may contain errors. These errors may not be amenable to discovery via testing, which would be too late in any case. Therefore, a key role of systems analysts is to participate in formal inspections of requirements, internal design documents, and external design documents. If the application is being constructed using test-driven development, systems analysts will participate in test case design and construction. Systems analysts will also participate in activities such as joint application design (JAD) and quality function deployment (QFD).

Performance Specialists

Assignment scope = 20,000 function points

Defect potentials = 1.00

Defect prevention impact = -10 percent

Defect removal impact = 12 percent

The occupation of “performance specialist” is usually found only in very large companies that build very large and complex software applications; that is, IBM, Raytheon, Lockheed, Boeing, SAP, Oracle, Unisys, Google, Motorola, and the like.

The general role of performance specialists is to understand every potential bottleneck in hardware and software platforms that might slow down performance.

Sluggish or poor performance is viewed as a quality issue, so the role of performance specialists is to assist software engineers and software designers in building software that will achieve good performance levels.

In today’s world of 2009, with multitier architectures as the dominant model and with multiple programming languages as the dominant form of development, the work of performance specialists has become much more difficult than it was only ten years ago. Looking ahead, the work of performance specialists will probably become even more difficult ten years from now.

Software Quality Assurance

Assignment scope = 10,000 function points

Defect potentials = 5.50

Defect prevention impact = -15 percent

Defect removal impact = 40 percent

The general title of “quality assurance” is much older than software and has been used by engineering companies for about 100 years. Within the software world, the title of “software quality assurance” has existed for more than 50 years. Today in 2009, software quality specialists average between 2 percent and 6 percent of total software employment in most large companies. The hi-tech companies such as IBM and Lockheed employ more software quality assurance personnel than do lo-tech companies such as insurance and general manufacturing.

A small percentage of software quality assurance personnel have been certified by one or more of the software quality assurance associations.

The roles of software quality assurance vary from company to company, but they usually include these core activities: ensuring that relevant international and corporate quality standards are used and adhered to, measuring defect removal efficiency, measuring cyclomatic and essential complexity, teaching classes in quality, and estimating or predicting quality levels.

A few very sophisticated companies such as IBM have quality assurance research positions, where the personnel can develop new and improved quality control methods. Some of the results of these QA research groups include formal inspections, function point metrics, automated configuration control tools, clean-room development, and joint application design (JAD).

Given the fact that quality assurance positions have existed for more than 50 years and that SQA personnel number in the thousands, why is software quality in 2009 not much better than it was in 1979?

One reason is that in many companies, quality assurance plays an advisory role, but their advice does not have to be followed. In some companies such as IBM, formal QA approval is necessary prior to delivering a product to customers. If the QA team feels that quality methods were deficient, then delivery will not occur. This is a very serious business issue.

In fact, very few projects are stopped from being delivered. But the theoretical power to stop delivery if quality is inadequate is a strong incentive to pursue state-of-the-art quality control methods.

Therefore, a major role of software quality assurance is to ensure that state-of-the-art measures, methods, and tools are used for quality control, with the knowledge that poor quality can lead to delays in delivery.

Web Designers

Assignment scope = 10,000 function points

Defect potentials = 4.00

Defect prevention impact = -15 percent

Defect removal impact = 12 percent

Software web design is a fairly new occupation, but one that is growing faster than almost any other. The fast growth in web design is due to software companies and other businesses migrating to the Web as their main channel for marketing and information.

The role of web design in terms of software quality is still evolving and will continue to do so as web sites move toward virtual reality and 3-D representation. As of 2009, some of the roles are to ensure that all interfaces are fairly intuitive, and that all links and connections actually work.

Unfortunately, due to the exponential increase in hacking, data theft, and denial of service attacks, web quality and web security are now overlapping. Effective quality for web sites must include effective security, and many web design specialists do not yet know enough about security to be fully effective.

Requirements Analysts

Assignment scope = 10,000 function points

Defect potentials = 4.00

Defect prevention impact = -20 percent

Defect removal impact = 15 percent

The work of requirements analysts overlaps the work of systems analysts and business analysts. However, those who specialize in requirements analysis also know topics such as quality function deployment (QFD), joint application design (JAD), requirements inspections, and at least half a dozen requirements representation methods such as use-cases, user stories, and several others.

Because the majority of “new” applications being developed circa 2009 are really nothing more than replacements for legacy applications, requirements analysts should also be conversant with data mining. In fact, the best place to start the requirements analysis for a replacement application is to mine the older legacy application for business rules and algorithms that are hidden in the code. Data mining is necessary because usually the original specifications are either missing completely or long out of date.

The role of requirements analysis in terms of quality is to ensure that toxic requirements defects are removed before they enter the design or find their way into source code. The frequently cited Y2K problem is an example of a toxic requirement.

Because the measured rate at which requirements grow after the requirements phase is between 1 percent and 3 percent per calendar month, another quality role is to ensure that prototypes, embedded users, JAD, or other methods are used that minimize unplanned changes in requirements.

Requirements analysts should also be members of or support change control boards that review and approve requirements changes.

Testers

Assignment scope = 10,000 function points

Defect potentials = 3.00

Defect prevention impact = -15 percent

Defect removal impact = 50 percent

Software testing is one of the specialized occupations where there is some empirical evidence that specialists can outperform generalists.

Not every kind of testing is performed by test specialists. For example, unit testing is almost always carried out by the developers. However, the forms of testing that integrate the work of entire teams of developers need testing specialists for large applications. Such forms of testing include new function testing, regression testing, and system testing among others.

The role of test specialists in terms of quality is to ensure that test coverage approaches 99 percent, that test cases themselves do not contain errors, and that test libraries are effectively maintained and purged of duplicate test cases that add cost but not value.

Although not a current requirement for test case personnel, it would be useful if test specialists also measured defect removal efficiency levels and attempted to raise average testing efficiency from today's average of around 35 percent up to at least 75 percent.

Test specialists should also be pioneers in new testing technologies such as automated testing. Running static analysis tools prior to testing could also be added with some value accruing.

Function Point Specialists

Assignment scope = 5000 function points

Defect potentials = 4.00

Defect prevention impact = -10 percent

Defect removal impact = 10 percent

Because function point metrics are the best choice for normalizing quality data and creating effective benchmarks of quality information, function point specialists are rapidly becoming part of successful quality improvement programs.

However, traditional manual counts of function points are too slow and too costly to be used as standard quality control methods. The average counting speed by a certified function point specialist is only about 400 function points per day. This explains why function point analysis is almost never used for applications larger than about 10,000 function points.

However, new methods have been developed that allow function points to be calculated at least six months earlier than previously possible.

These same methods operate at speeds in excess of 10,000 function points per minute. This makes it possible to use function points for early quality estimation, as well as for measuring quality and producing quality benchmarks.

The role of function point specialists in terms of quality is to create useful size information fast enough and early enough that it can serve for risk analysis, quality prediction, and quality measures.

Technical Writers

Assignment scope = 2000 function points

Defect potentials = 1.00

Defect prevention impact = -10 percent

Defect removal impact = 10 percent

Good writing is a fairly rare skill in the human species. As a result, good software technical manuals are also fairly rare. Many kinds of quality problems are common in software manuals, including ambiguity, missing information, poor organization structures, and incurred data.

There are automated tools available that can analyze the readability of text, such as the FOG index and the Fleisch index. But these are seldom used for software manuals. Editing is useful, as are formal inspections of user documentation.

Another approach, which was actually used by IBM, was to select examples of user documents with the highest user evaluation scores and use them as samples.

The role of technical writers in terms of software quality is make sure that factual data is complete and correct, and that manuals are easy to read and understand.

Maintenance Specialists

Assignment scope = 1,500 function points

Defect potentials = 3.5

Defect prevention impact = -30 percent

Defect removal impact = 20 percent

Maintenance programming in terms of both enhancing legacy software and repairing bugs has been the dominant activity for the software industry for more than 20 years. This should not be a surprise, because for every industry older than 50 years, more people are working on repairs of existing products than are working on new development.

As the recession deepens and lengthens, the U.S. automobile industry is providing a very painful example of this fact: automotive manufacturing is shrinking faster than the polar ice fields, while automotive repairs are increasing.

Aging legacy applications have a number of quality problems, including poor structure, dead code, error-prone modules, and poor or missing comments.

As the recession continues, many companies are considering ways of stretching out the useful lives of legacy applications. In fact, renovation and data mining of legacy software are both growing, even in the face of the recession.

The main role of maintenance programmers in terms of quality is to strengthen the quality of legacy software. The methods available to do this include full renovation using automated tools; complexity measurement and reduction; dead code removal; improving comments; identification and surgical removal of error-prone modules; converting code from orphan languages such as MUMPS or Coral into modern languages such as Java or Ruby, and improving the security flaws of legacy applications.

Inspection Moderators

Assignment scope = 1000 function points

Defect potentials = 4.5

Defect prevention impact = -25 percent

Defect removal impact = 35 percent

Software inspections have a number of standard roles, including the moderator, the recorder, the inspectors, and the person whose work is being inspected. The moderator is the key to a successful inspection. The tasks of the moderator include keeping the discussions on track, minimizing disruptive events, and ensuring that the inspection session starts and ends on time.

The main role of inspection moderators in terms of quality include ensuring the materials to be inspected are delivered in time for pre-inspection review, making sure that the inspectors and other personnel show up on time, keeping the inspection team focused on defect identification (as opposed to repairs), and intervening in potential arguments or disputes.

The inspection recorder plays a key role too, because the recorder keeps notes and fills out the defect reports of all bugs or defects that the inspection identified. This is not as easy as it sounds, because there may be some debate as to whether a particular issue is a defect or a possible enhancement.

Summary and Conclusions on Software Specialization

The overall topic of software specialization is not well covered in the software engineering literature. Considering that there are more than 115 specialists associated with software, this fact is mildly surprising.

When it comes to software quality, some forms of specialization do add value, and this can be shown by analysis of both defect prevention and defect removal. The key specialists who add the most value to software quality include risk analysts, Six Sigma specialists, quality assurance personnel, inspection moderators, maintenance specialists, and professional test personnel.

However, many other specialists such as business analysts, enterprise architects, architects, estimating specialists, and function point specialists also add value.

The Economic Value of Software Quality

The economic value of software quality is not well covered in the software engineering literature. There are several reasons for this problem. One major reason is the rather poor measurement practices of the software engineering domain. Many cost factors such as unpaid overtime are routinely ignored. In addition, there are frequent gaps and omissions in software cost data, such as omission of project management costs and the omission of part-time specialists such as technical writers. In fact, only the effort and costs of coding have fairly good data available. Everything else, such as requirements, design, inspections, testing, quality assurance, project offices, and documentation tend to be underreported or ignored.

As pointed out in other sections, the software engineering literature depends too much on vague and unpredictable definitions of quality such as “conformance to requirements” or adhering to a collection of ambiguous terms ending in *ility*. These unscientific definitions slow down research on software quality economics.

Two other measurement problems also affect quality economic studies. These problems are the usage of two invalid economic measures: cost per defect and lines of code. As discussed earlier in this chapter, cost per defect penalizes quality and achieves its lowest costs for the buggiest applications. Lines of code penalizes high-level programming languages and disguises the value of high-level languages for studying either quality or productivity.

In this section, the economic value of quality will be shown by means of eight case studies. Because the value of software quality correlates to application size, four discrete size ranges will be used: 100 function points, 1000 function points, 10,000 function points, and 100,000 function points.

Applications in the 100–function point range are usually small features for larger systems rather than stand-alone applications. However, this is a very common size range for prototypes of larger applications.

There may be small stand-alone applications in this range such as currency converters or applets for devices such as iPhones.

Applications in the 1000–function point range are normally stand-alone software applications such as fuel-injection controls, atomic watch controls, compilers for languages such as Java, and software estimating tools in the class of COCOMO.

Applications in the 10,000–function point range are normally important systems that control aspects of business, such as insurance claims processing, motor vehicle registration, or child-support applications.

Applications in the 100,000–function point range are normally major systems in the class of large international telephone-switching systems, operating systems in the class of Vista and IBM's MVS, or suites of linked applications such as Microsoft Office. Some enterprise resource planning (ERP) applications are in this size range, and may even top 300,000 function points. Also, large defense applications such as the World Wide Military Command and Control System (WWMCCS) also top 100,000 function points.

To reduce the number of variables, all eight of the examples are assumed to be coded in the C programming language and have a ratio of about 125 code statements per function point.

Because all eight of the applications are assumed to be written in the same programming language, productivity and quality can be expressed using the lines of code metric without distortion. The lines of code metric is invalid for comparisons between unlike programming languages.

For each size plateau, two cases will be illustrated: average quality and excellent quality. The average quality case assumes waterfall development, CMMI level 1, normal testing, and nothing special in terms of defect prevention.

The excellent quality case assumes at least CMMI level 3, formal inspections, static analysis, rigorous development such as the Team Software Process (TSP), and the use of prototypes and joint application design (JAD) for requirements gathering.

(Some readers may wonder why Agile development is not used for the case studies. The main reason is that there are no Agile applications in the 10,000– and 100,000–function point ranges. The Agile method is used primarily for smaller applications in the 1000–function point range.)

Although all of the case studies are derived from actual applications, to make the calculations consistent, a number of simplifying assumptions are used. These assumptions include the following key points:

- All cost data is based on a fully burdened cost of \$10,000 per staff month. A staff month is considered to have 132 working hours. This is equivalent to \$75.75 per hour.

- Work months are assumed to consist of 22 days, and each day consists of 8 hours. Unpaid overtime is not shown nor is paid overtime.
- Defect potentials are the total numbers of defects found in five categories: requirements defects, design defects, code defects, documentation defects, and bad fixes, or secondary defects accidentally included in defect repairs.
- Creeping requirements are not shown. The sizes of the six case studies reflect application size as delivered to clients.
- Software reuse is not shown. All six cases can be assumed to reuse about 15 percent of legacy code. But to simplify assumptions, the defect potentials in the reused code and other materials are assumed to equal defect potentials of new material. Larger volumes of certified reusable material would significantly improve both the quality and productivity of all six case studies, and especially so for the larger systems above 10,000 function points.
- Bad-fix injections are not shown. About 7 percent of attempts to repair bugs accidentally introduce a new bug, but the mathematics of bad-fix injection is complicated since the bugs are not found in the activity where they originate.
- The first year of maintenance is assumed to find 100 percent of latent bugs delivered with the software. In reality, many bugs fester for years, but the examples only show the first year of maintenance.
- The maintenance data only shows defect repairs. Enhancements and adding new features are excluded in order to highlight quality value.
- Maintenance defect repair rates are based on average values of 12 bugs fixed per staff month. In real life, ranges can run from fewer than 4 to more than 20 bugs repaired each month.
- Application staff size is based on U.S. average assignment scopes for all classes of software personnel, which is approximately 150 function points. That is, if you divide application size in function points by the total staffing complement of technical workers plus project managers, the result will be close to 150 function points. This value includes software engineers and also specialists such as quality assurance, technical writers, and test personnel.
- Schedules for the “average” cases are based on raising function point size to the 0.4 power. This rule of thumb provides a fairly good approximation of schedules from start of requirements to delivery in terms of calendar months.
- Schedules for the “excellent” cases are based on raising function point size to the 0.36 power. This exponent works well with object-oriented

software and rigorous development practices. It is also a good fit for Agile projects, except that the lack of data above 10,000 function points for Agile makes the upper level uncertain.

- Data in this section is expressed using the function point metric defined by the International Function Point Users' Group (IFPUG) version 4.2 of the counting rules. Other functional metrics such as COSMIC function points or engineering function points or Mark II function points would yield different results from the values shown here.
- Data on source code in this section is expressed using counts of logical statements rather than counts of physical lines. There can be as much as 500 percent difference in apparent code size based on whether counts are physical or logical lines. The counting rules are those of the author's book *Applied Software Measurement*.

The reason for these simplifying assumptions is to minimize extraneous variations among the eight case studies, so that the data is presented in a consistent fashion for each. Because all of these assumptions vary in real life, readers are urged to try out alternative values based on their own local data or on benchmarks from organizations such as the International Software Benchmarking Standards Group (ISBSG).

The simplifying assumptions serve to make the results consistent, but each of the assumptions can change in either direction by fairly large amounts.

The Value of Quality for Very Small Applications of 100 Function Points

Small applications in this range usually have low defect potentials and fairly high defect removal efficiency levels. This is because such small applications can be developed by a single person, so there are no interface problems between features developed by different individuals or different teams. Table 9-24 shows quality value for very small applications of 100 function points.

Note that cost per defect goes *up* as quality improves; not *down*. This phenomenon distorts economic analysis. As will be shown in the later examples, cost per defect tends to decline as applications grow larger. This is because large applications have many more defects than small ones.

Prototypes or applications in this size range are very sensitive to individual skill levels, primarily because one person does almost all of the work. The measured variations for this size range are about 5 to 1 in how much code gets written for a given specification and about 6 to 1 in terms of productivity and quality levels. Therefore, average values need to be used with caution. Averages are particularly unreliable for applications where one person performs the bulk of the entire application.

TABLE 9-24 Quality Value for 100 Function Point Applications

(Note: 100 function points = 12,500 C statements)

	Average Quality	Excellent Quality	Difference
Defects per function point	3.50	1.50	-2.00
Defect potential	350	150	-200.00
Defect removal efficiency	94.00%	99.00%	5.00%
Defects removed	329	149	-181
Defects delivered	21	2	-20
Cost per defect prerelease	\$379	\$455	\$76
Cost per defect postrelease	\$1,061	\$1,288	\$227
Development schedule (calendar months)	6	5	-1
Development staffing	1	1	0
Development effort (staff months)	6	5	-1
Development costs	\$63,096	\$52,481	-\$10,615
Function points per staff month	15.85	19.05	3.21
LOC per staff month	1,981	2,382	401
Maintenance staff	1	1	0
Maintenance effort (staff months)	2	0	-1.63
Maintenance costs (year 1)	\$17,500	\$1,250	-\$16,250
TOTAL EFFORT	8	5	-3
TOTAL COST	\$80,596	\$53,731	-\$26,865
TOTAL COST PER STAFF MEMBER	\$40,298	\$26,865	-\$13,432
TOTAL COST PER FUNCTION POINT	\$805.96	\$537.31	-\$269
TOTAL COST PER LOC	\$6.45	\$4.30	-\$2.15
AVERAGE COST PER DEFECT	\$720	\$871	\$152

The Value of Quality for Small Applications of 1000 Function Points

For small applications of 1000 function points, quality starts to become very important, but it is also somewhat easier to achieve than it is for large systems. At this size range, teams are small and methods such as Agile development tend to be dominant, other than for systems and embedded software where more rigorous methods such as the Team Software Process (TSP) and the Rational Unified Process (RUP) are more common. Table 9-25 shows the value of quality for small applications in the 1000-function point range.

The bulk of the savings for the Excellent Quality column shown in Table 9-25 would come from shorter testing schedules due to the use of requirements, design, and code inspections. Other changes that added value include the use of Team Software Process (TSP), static analysis prior to testing, and the achievement of higher CMMI levels.

TABLE 9-25 Quality Value for 1000–Function Point Applications

(Note: 1000 function points = 125,000 C statements)

	Average Quality	Excellent Quality	Difference
Defects per function point	4.50	2.50	–2.00
Defect potential	4,500	2,500	–2,000
Defect removal efficiency	93.00%	97.00%	4.00%
Defects removed	4,185	2,425	–1,760
Defects delivered	315	75	–240.00
Cost per defect prerelease	\$341	\$417	\$76
Cost per defect postrelease	\$909	\$1,136	\$227
Development schedule (calendar months)	16	12	–4
Development staffing	7	7	0.00
Development effort (staff months)	106	80	–26
Development costs	\$1,056,595	\$801,510	–\$255,086
Function points per staff month	9.46	12.48	3.01
LOC per staff month	1,183	1,560	376.51
Maintenance staff	2	2	0
Maintenance effort (staff months)	26	6	–20.00
Maintenance costs (year 1)	\$262,500	\$62,500	–\$200,000
TOTAL EFFORT	132	86	–46
TOTAL COST	\$1,319,095	\$864,010	–\$455,086
TOTAL COST PER STAFF MEMBER	\$158,291	\$103,681	–\$54,610
TOTAL COST PER FUNCTION POINT	\$1,319.10	\$864.01	–\$455
TOTAL COST PER LOC	\$10.55	\$6.91	–\$3.64
AVERAGE COST PER DEFECT	\$625	\$776	\$152

In the size range of 1000 function points, numerous methods are fairly effective. For example, both Agile development and extreme programming report good results in this size range as do the Rational Unified Process (RUP) and the Team Software Process (TSP).

The Value of Quality for Large Applications of 10,000 Function Points

When software applications reach 10,000 function points, they are very significant systems that require close attention to quality control, change control, and corporate governance. In fact, without careful quality and change control, the odds of failure or cancellation top 35 percent for this size range.

Note that as application size increases, defect potentials increase rapidly and defect removal efficiency levels decline, even with sophisticated quality control steps in place. This is due to the exponential increase in

the volume of paperwork for requirements and design, which often leads to partial inspections rather than 100 percent inspections. For large systems, test coverage declines and the number of test cases mounts rapidly, but cannot usually keep pace with complexity. Table 9-26 shows the increasing value of quality as size goes up to 10,000 function points.

Cost savings from better quality increase as application sizes increase. The general rule is that the larger the software application, the more valuable quality becomes. The same principle is true for change control, because the volume of creeping requirements goes up with application size.

For large systems, the available methods that demonstrate improvement begin to decline. For example, Agile methods are difficult to apply, and when they are, the results are not always good. For large systems, rigorous methods such as the Rational Unified Process (RUP) or Team Software Process (TSP) yield the best results and have the greatest amount of empirical data.

TABLE 9-26 Quality Value for 10,000–Function Point Applications

(Note: 10,000 function points = 1,250,000 C statements)

	Average Quality	Excellent Quality	Difference
Defects per function point	6.00	3.50	–2.50
Defect potential	60,000	35,000	–25,000
Defect removal efficiency	84.00%	96.00%	12.00%
Defects removed	50,400	33,600	–16,800
Defects delivered	9,600	1,400	–8,200
Cost per defect prerelease	\$341	\$417	\$76
Cost per defect postrelease	\$833	\$1,061	\$227
Development schedule (calendar months)	40	28	–12
Development staffing	67	67	0.00
Development effort (staff months)	2,654	1,836	–818
Development costs	\$26,540,478	\$18,361,525	–\$8,178,953
Function points per staff month	3.77	5.45	1.68
LOC per staff month	471	681	209.79
Maintenance staff	17	17	0
Maintenance effort (staff months)	800	117	–683.33
Maintenance costs (year 1)	\$8,000,000	\$1,166,667	–\$6,833,333
TOTAL EFFORT (STAFF MONTHS)	3,454	1,953	–1501
TOTAL COST	\$34,540,478	\$19,528,191	–\$15,012,287
TOTAL COST PER STAFF MEMBER	\$414,486	\$234,338	–\$180,147
TOTAL COST PER FUNCTION POINT	\$3,454.05	\$1,952.82	–\$1,501.23
TOTAL COST PER LOC	\$27.63	\$15.62	–\$12.01
AVERAGE COST PER DEFECT	\$587	\$739	\$152

The Value of Quality for Very Large Applications of 100,000 Function Points

Software applications in the 100,000–function point range are among the most costly endeavors of modern business. These large systems are also hazardous, because many of them fail, and almost all of them exceed their budgets and planned schedules.

Without excellence in software quality control, the odds of completing a software application of 100,000 function points are only about 20 percent. The odds of finishing it on time and within budget hover close to 0 percent.

Even with excellent quality control and excellent change control, massive applications in the 100,000–function point range are expensive and troublesome. Table 9-27 illustrates the two cases for such massive applications.

TABLE 9-27 Quality Value for 100,000–Function Point Applications

(Note: 100,000 function points = 12,500,000 C statements)

	Average Quality	Excellent Quality	Difference
Defects per function point	7.00	4.00	–3.00
Defect potential	700,000	400,000	–300,000
Defect removal efficiency	81.00%	94.00%	13.00%
Defects removed	567,000	376,000	–191,000
Defects delivered	133,000	24,000	–109,000
Cost per defect prerelease	\$303	\$379	\$76
Cost per defect postrelease	\$758	\$985	\$227
Development schedule (calendar months)	100	63	–37
Development staffing	667	667	0.00
Development effort (staff months)	66,667	42,064	–24,603
Development costs	\$666,666,667	\$420,638,230	–\$246,028,437
Function points per staff month	1.50	2.38	0.88
LOC per staff month	188	297	109.67
Maintenance staff	167	167	0
Maintenance effort (staff months)	11,083	2,000	–9,083
Maintenance costs (year 1)	\$110,833,333	\$20,000,000	–\$90,833,333
TOTAL EFFORT	77,750	44,064	–33686
TOTAL COST	\$777,500,000	\$440,638,230	–\$336,861,770
TOTAL COST PER STAFF MEMBER	\$933,000	\$528,766	–\$404,234
TOTAL COST PER FUNCTION POINT	\$7,775.00	\$4,406.38	–\$3,368.62
TOTAL COST PER LOC	\$62.20	\$352.51	\$290.31
AVERAGE COST PER DEFECT	\$530	\$682	\$152

There are several reasons why defect potentials are so high for massive applications and why defect removal efficiency levels are reduced. The first reason is that for such massive applications, requirements changes will be so numerous that they exceed most companies' ability to control them well.

The second reason is that paperwork volumes tend to rise with application size, and this slows down activities such as inspections of requirements and design. As a result, massive applications tend to use partial inspections rather than 100 percent inspections of major deliverable items.

A third reason, which was worked out mathematically at IBM in the 1970s, is that the number of test cases needed to achieve 90 percent coverage of code rise exponentially with size. In fact, the number of test cases required to fully test a massive system of 100,000 function points approaches infinity. As a result, testing efficiency declines with size, even though static analysis and inspections stay about the same.

A useful rule of thumb for predicting overall number of test cases is to raise application size in function points to the 1.2 power. As can be seen, test case volumes rise very rapidly, and most companies cannot keep pace, so test coverage declines. Automated static analysis is still effective. Inspections are also effective, but for 100,000 function points, partial inspections of key deliverables are the norm rather than 100 percent inspections. This is because paperwork volumes also rise exponentially with size.

Return on Investment in Software Quality

As already mentioned, the value of software quality goes up as application size goes up. Table 9-28 calculates the approximate return on investment for the "excellent" case studies of 100 function points, 1000 function points, 10,000 function points, and 100,000 function points.

Here too the assumptions are simplified to make calculations easy and understandable. The basic assumption is that every software team member needs five days of training to get up to speed in software inspections and the Team Software Process (TSP). These training days are then multiplied by average hourly costs of \$75.75 per employee.

These training expenses are then divided into the total savings figure that includes both development and maintenance savings due to high quality. The final result is the approximate ROI based on dividing value by training expenses. Table 9-28 illustrates the ROI calculations.

The ROI figure reflects the total savings divided by the total training expenses needed to bring team members up to speed in quality technologies.

In real life, these simple assumptions would vary widely, and other factors might also be considered. Even so, high levels of software quality

TABLE 9-28 Return on Investment in Software Quality

Function point size	100	1,000	10,000	100,000
Education hours	80	560	5,360	53,360
Education costs	\$6,060	\$42,420	\$406,020	\$4,042,020
Savings from high quality	\$26,865	\$455,086	\$15,012,287	\$336,861,770
Return on investment (ROI)	\$4.43	\$10.73	\$36.97	\$83.34

have a very solid return on investment due to the reduction in development schedules, development costs, and maintenance costs.

There may be many other topics where software engineers and managers need training, and there may be other cost elements such as the costs of ascending to the higher levels of the capability maturity model. While the savings from high quality are frequently observed, the exact ROI will vary based on the way training and process improvement work is handled under local accounting rules.

If the reduced risks of cancelled projects or major overruns were included in the ROI calculations, the value would be even higher.

Other technologies such as high volumes of certified reusable material would also have a beneficial impact on both quality and productivity. However, as this book is written in 2009, only limited sources are available for certified reusable materials. Uncertified reuse is hazardous and may even be harmful rather than beneficial.

Summary and Conclusions

In spite of the fact that the software industry spends more money on finding and fixing bugs than any other activity, software quality remains ambiguous and poorly covered in the software engineering literature.

There are dozens of books on software quality and testing, but hardly any of them contain quantitative data on defect volumes, numbers of test cases, test coverage, or the costs associated with defect removal activities.

Even worse, much of the literature on quality merely cites urban legends of how “cost per defect rises throughout development and into the field,” without realizing that such a trend is caused by ignoring fixed costs.

Software quality does have value, and the value increases as application sizes get bigger. In fact, without excellence in quality control, even completing a large software application is highly unlikely. Completing it on time and within budget in the absence of excellent quality control is essentially impossible.

Readings and References

- Beck, Kent. *Test-Driven Development*. Boston, MA: Addison Wesley, 2002.
- Chelf, Ben and Raoul Jetley. *Diagnosing Medical Device Software Defects Using Static Analysis*. San Francisco, CA: Coverity Technical Report, 2008.
- Chess, Brian and Jacob West. *Secure Programming with Static Analysis*. Boston, MA: Addison Wesley, 2007.
- Cohen, Lou. *Quality Function Deployment—How to Make QFD Work for You*. Upper Saddle River, NJ: Prentice Hall, 1995.
- Crosby, Philip B. *Quality is Free*. New York, NY: New American Library, Mentor Books, 1979.
- Everett, Gerald D. and Raymond McLeod. *Software Testing*. Hoboken, NJ: John Wiley & Sons, 2007.
- Gack, Gary. Applying Six Sigma to Software Implementation Projects. <http://software.isixsigma.com/library/content/c040915b.asp>.
- Gilb, Tom and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.
- Hallowell, David L. Six Sigma Software Metrics, Part 1. <http://software.isixsigma.com/library/content/c03910a.asp>.
- International Organization for Standards. ISO 9000 / ISO 14000. <http://www.iso.org/iso/en/iso9000-14000/index.html>.
- Jones, Capers. *Software Quality—Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press, 1997.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston, MA: Addison Wesley Longman, 2003.
- Land, Susan K., Douglas B. Smith, John Z. Walz. *Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards*. Los Alamitos, CA: Wiley-IEEE Computer Society Press, 2008.
- Mosley, Daniel J. *The Handbook of MIS Application Software Testing*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1993.
- Myers, Glenford. *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979.
- Nandyal, Raghav. *Making Sense of Software Quality Assurance*. New Delhi: Tata McGraw-Hill Publishing, 2007.
- Radice, Ronald A. *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing, 2002.
- Wieggers, Karl E. *Peer Reviews in Software—A Practical Guide*. Boston, MA: Addison Wesley Longman, 2002.

This page intentionally left blank

A

- abeyant defects, 512
- access control lists (ACLs), 143
- acquisition, circa 2049, 204–207
- activities, 60
- activity-level productivity and quality benchmarks, 416–417
- actors, 373
- administrative access, 143
- administrative rights, 149
- adware, 142–143
- Agile
 - requirements with embedded users, 456
 - self-organizing Agile teams, 289–293
 - taxonomy for software methodology analysis, 66–67
- algorithm view, 481
- analogy, sizing by, 363–365
- appraisals
 - for software personnel, 50–51
 - of technical staff, 45–46
- approval, 90
- approximations, 383
- architects, 623–624
- architecture
 - best practices, 75–77
 - circa 2049, 210–213
 - enterprise, 210–213, 475–479
 - software, 470–475
 - See also* service-oriented architecture (SOA)
- assemblers, 491
- assembly languages, 491
- assessment benchmarks, 419–421

- assignment scope, 620
- attribute view, 482
- attributes, 60
- attrition benchmarks, 426
- authentication, authorization, and access, 143
- Authorization Oriented Architecture* (Hamer-Hodges), 140, 141
- automated debugging, for defect removal, 531
- automated static analysis
 - as defect prevention, 523
 - for defect removal, 531–533
- automated unit testing, for defect removal, 535–536
- award benchmarks, 428–429

B

- back doors, 143–144
- backfiring, 318
 - sizing legacy applications based on, 385–389
- bad fixes, 513
- bad-fix injections, 337
- balanced matrix, 306
 - See also* matrix management
- baselines, best practices, 112–115
- benchmarking, 408–411
- benchmarks
 - academic benchmarks, 410
 - activity-level productivity and quality benchmarks, 416–417
 - assessment benchmarks, 419–421
 - award benchmarks, 428–429

benchmarks (*continued*)

- best practices, 112–115
- blind benchmarks, 430
- categories of, 411–413
- chart of accounts for activity-
level software benchmarks, 68
- consultant collection for internal
benchmarks, 409
- consultant collection for
proprietary benchmarks, 410
- corporate software portfolio
benchmarks, 415
- cost of quality (COQ)
benchmarks, 423
- customer satisfaction
benchmarks, 427
- earned-value benchmarks, 422
- hybrid assessment and
benchmark studies, 421–422
- industry benchmarks, 413–414
- internal collection for internal
benchmarks, 409
- internal collection for public or
ISBSG benchmarks, 410
- international software
benchmarks, 413
- ISO quality benchmarks, 424
- methodology, 418–419
- open benchmarks, 429
- organizations, 430–431
- overall software cost and
resource benchmarks, 414
- partly open benchmarks, 429–430
- phase-level productivity and
quality benchmarks, 415–416
- quality and test coverage
benchmarks, 422–423
- reporting methods for benchmark
and assessment data,
431–433
- security benchmarks, 424–425
- Six Sigma benchmarks,
423–424
- software compensation
benchmarks, 426
- software data center
benchmarks, 427
- software litigation and failure
benchmarks, 428

- software maintenance
and customer support
benchmarks, 417–418
- software outsource vs. internal
performance benchmarks, 417
- software performance
benchmarks, 426–427
- software personnel and skill
benchmarks, 425–426
- software turnover and attrition
benchmarks, 426
- software usage benchmarks,
427–428
- types of benchmark studies
performed, 429–430

best practices, 39–41

- 30 best practices for 1000– and
10,000–function point
projects, 31
- 30 best practices of IT projects
and embedded/systems
projects, 32

appraisals and career planning
for software personnel, 50–51

canceling or turning around
troubled projects, 84–86

certification of reusable
materials, 101–107

certifying methods, tools, and
practices, 64–70

certifying software engineers,
specialists, and managers,
94–97

communication during software
projects, 97–99

configuration control, 119–120

customer support of software
applications, 156–158

defining and evaluating, 7–10

early sizing and scope control of
software applications, 51–53

executive management support of
software applications, 74–75

inspections and static analysis,
124–128

international software standards,
135–136

minimizing harm from layoffs
and downsizing, 41–45

motivation and morale of
 managers and executives,
 47–50
 motivation and morale of
 technical staff, 45–47
 outsourcing software
 applications, 53–58
 programming or coding,
 107–109
 protecting against viruses,
 spyware, and hacking,
 138–153
 protecting intellectual property
 in software, 136–138
 requirements of software
 applications, 70–72
 selecting software methods, tools,
 and practices, 59–64
 selection and hiring of software
 personnel, 50
 software architecture and design,
 75–77
 software benchmarks and
 baselines, 112–115
 software change control before
 release, 117–119
 software change management
 after release, 159–161
 software deployment and
 customization, 154–155
 software maintenance and
 enhancement, 161–164
 software performance analysis,
 134–135
 software project cost estimating,
 79–81
 software project governance,
 109–110
 software project measurements
 and metrics, 110–112
 software project milestone and
 cost tracking, 115–116
 software project organization
 structures, 87–89
 software project planning, 77–78
 software project risk analysis,
 81–83
 software project value analysis,
 83–84

software quality assurance
 (SQA), 120–124
 software reusability, 99–101
 software security analysis and
 control, 132–134
 software warranties and recalls,
 158–159
 terminating or withdrawing
 legacy applications, 166–167
 testing and test library control,
 128–132
 training clients or users of
 software applications,
 155–156
 training managers of software
 projects, 89–91
 training software technical
 personnel, 91–92
 updates and releases of software
 applications, 164–165
 use of software specialists,
 92–94
 user involvement in software
 projects, 72–73
 using contractors and
 management consultants,
 58–59
See also neutral practices; worst
 practices
 black box testing, 128, 329, 533
See also testing
 blacklists, 148
 blind benchmarks, 430
 bohrbug, 135
 books, 258–259, 260–263
 bot herders, 144
 botnets, 144
 browser hijackers, 144
 browsing, 244
 bugs, 509–512
 business analysis, 468–470
 business analysts, 625–626

C

canceling troubled projects, best
 practices, 84–86
 capability-based security, 143
 career planning, for software
 personnel, 50–51

- cautions and counter indications
 - customer support
 - organizations, 327
 - hierarchical organizations, 304
 - matrix organizations, 308
 - one-person projects, 286
 - pair programming, 289
 - self-organizing Agile teams, 293
 - software maintenance
 - organizations, 321
 - software test organizations, 340
 - Team Software Process (TSP)
 - teams, 297
- certification
 - best practices, 94–97
 - circa 2049, 218–220
 - influence of on defect prevention
 - and removal, 604
 - and specialization, 241
- certification of reusable materials,
 - best practices, 101–107
- certification of web sites, 142
- certifying methods, tools, and
 - practices, best practices, 64–70
- change control before release, best
 - practices, 117–119
- change management after release,
 - best practices, 159–161
- chart of accounts for activity-level
 - software benchmarks, 68
- class, defined, 65
- client management, 90
- cloud computing, 474–475
- code complexity, 393, 451
- code inspections, 125
- code reuse, as defect
 - prevention, 521
- code structure, as defect
 - prevention, 525–526
- coding, best practices, 107–109
- colocation, vs. distributed
 - development, 278–281
- commercial education, 250–252
- communication, best practices,
 - 97–99
- compensation benchmarks, 426
- compilers, 491
- complexity of software, 122
- conferences, 254–255
- configuration control, best
 - practices, 119–120
- contractors, best practices, 58–59
- cookie poisoning, 145
- cookies, 144–145
- corporate software portfolio
 - benchmarks, 415
- cost, of learning methods, 230
- cost drivers for software
 - applications, 2
 - revised sequence circa 2019, 3
- cost estimating, best practices,
 - 79–81
- cost of quality control and defect
 - repairs, 122–123
- cost of quality (COQ), 590–591
 - benchmarks, 423
- cost per defect, 17–18
- cost tracking, best practices,
 - 115–116
- cost-estimating tool circa 2049,
 - features, 178–179, 193
- costs, of software development, 4–5
- creeping requirements, 457
 - quality impacts of, 584–585
 - See also* requirements creep
- critical topics, 19–23
- Crosby, Phil, 123
- CrossTalk*, 257
- cumulative defect removal
 - efficiency, 330, 515
- currency, 230
- curricula, proposed, 269–273
- customer satisfaction, 121
 - benchmarks, 427
- customer support
 - benchmarks, 417–418
 - best practices, 156–158
 - circa 2049, 188–190
- customer support organizations,
 - 322–328
- customer training, circa 2049,
 - 190–191
- customization, best practices,
 - 154–155
- Cutter Journal*, 257
- cyberextortion, 145

cyberstalking, 145
cyclomatic complexity, 516, 526

D

data center benchmarks, 427
data complexity, 393, 451–452
data defects, 513
data mining for legacy
 requirements, 457–458
data quality specialists, 625
data view, 481–482
database analysts, 623
defect discovery point, 587–588, 589
defect origin point, 588–590
defect potential, 69, 422, 515, 562
 overview, 573
 predicting, 578–579
 for a sample application, 580
defect prevention, 130–131,
 330–331, 518
 forms of, 520–529
 influence of certification on, 604
 international standards, 606
 methods and techniques, 602
 metrics and measures, 604
 optimal activities, 575
 overview, 600–608
 proposal for a nonprofit
 international software quality
 foundation, 608–613
 usage patterns of defect
 prevention methods,
 607–608
defect quantities and origins,
 121–122
defect removal, 130, 131, 518–520
 effort accumulation, 594
 forms of, 529–537
 forms of software defect removal
 activities, 332
 influence of certification on, 604
 international standards, 606
 for legacy applications, 536–537
 metrics and measures, 604
 optimal activities, 575
 overview, 613–619
 overview of 80 varieties of
 activities, 615–617

 synergies and combinations
 of, 537
defect removal efficiency, 69–70,
 122, 422, 515, 562
costs, 599
cumulative defect removal
 efficiency, 330, 515
by defect type, 336
levels, 596–597
measuring, 593–600
defect repair rates, 314
defect severity levels, 122, 512, 571
 accumulation, 595
defects
 causes, 571
 defined, 512
 defining and predicting,
 570–578
 examples of defects per KLOC
 and function point, 582
 kinds of defects occurring in
 source code, 509–512
 logistics of software code defects,
 512–516
 overview of delivered software
 defects, 574
 percentages of defects by
 origin, 579
 points of origin, 570
 predicting, 579–584
delivered defects by
 application, 122
delivered defects, reliability and
 customer satisfaction, 600
delivery productivity, 540–541, 581
demographics
 customer support organizations,
 325–326
 hierarchical organizations,
 302–303
 matrix organizations, 306
 one-person projects, 284
 pair programming, 288
 self-organizing Agile teams, 291
 software maintenance
 organizations, 319
 software quality assurance (SQA)
 organizations, 345

demographics (*continued*)
 software test organizations,
 337–338
 Team Software Process (TSP)
 teams, 295
 denial of service, 145–146
 deployment
 best practices, 154–155
 circa 2049, 190–191
 paths, 12–14
 quantifying, 16–19
 design, 479–480
 best practices, 75–77
 circa 2049, 182–184
 views, 481–484
 desk checking, for defect removal,
 530–531
 development
 circa 2049, 184–186
 paths, 10–12
 practices by size of
 application, 11
 quantifying, 16–19
 development methodology. *See*
 development process
 development process, 61–62
 development productivity,
 540–541, 581
 disposable prototypes, 460
 distributed development, vs.
 colocation, 278–281
 documentation, circa 2049,
 186–188
 dotted line reporting authority, 305
 downsizing, best practices for
 minimizing harm from, 41–45
 drivers, 286
 due diligence, circa 2049, 216–218

E

earned quality value (EQV), 590
 earned value, 111
 earned-value benchmarks, 422
 e-bombs, 146
 e-books, 246–247, 258–259
 economic value of quality, 123
 education
 commercial, 250–252

graduate university education,
 265–266
 in-house, 248–249
 knowledge areas, 232
 learning methods, 227–230
 proposed curricula, 269–273
 ranking of learning channels in
 2009, 231
 topics software engineers need to
 learn in 2009, 230–233
 undergraduate university
 education, 263–265
 vendor, 252–253
 e-learning, 245–246
 electromagnetic pulse (EMP), 146
 electromagnetic radiation,
 146–147
 electronic books, 246–247, 258–259
 e-mail address harvesting, 150
 EMP. *See* electromagnetic
 pulse (EMP)
 end user license agreements
 (EULAs), 158–159
 enhancements, 103, 104
 best practices, 161–164
 circa 2049, 191–195
 enhancement value of high-
 quality reusable
 materials, 105
 paths, 14–16
 enterprise architects, 622
 enterprise architecture, 475–479
 circa 2049, 210–213
 value of increases with
 applications, 477
 entropy, 315
 error-prone modules, 316–318, 514
 essential complexity, 516, 526
 estimated software security
 costs, 153
 estimating, defined, 78
 estimating specialists, 626
 EULAs. *See* end user license
 agreements (EULAs)
 evaluation, circa 2049, 204–207
 evangelists, 236–240
 evolutionary prototypes, 460
 executable English, 458

- executives
 - management support of software applications, 74–75
 - motivation and morale, 47–50
- external view, 481
- externally caused defects, 513

F

- facilitation, 90
- Fagan, Michael, 124
- failure benchmarks, 428
- failure rate, 5
- false positives, 512, 572
- focus groups, 121, 458–459
- formal inspections, 125
- function point analysis, sizing
 - based on, 376–379
- function point approximations,
 - high-speed sizing using, 383–385
- Function Point Outlook tool, 384–385
- function point specialists, 630–631
- function points
 - number of pages created per function point, 377
 - sizing using function point variations, 380–383
 - See also* micro function points
- functional requirements, 459
- funding, 90

G

- Gilb, Tom, 124, 125
- governance, 476
 - best practices, 109–110
- graduate university education, 265–266
- gray box testing, 128, 329
 - See also* testing

H

- hacking, 147
- hacking protection, best practices, 138–153
- heisenbug, 134–135
- hierarchical organizations, 298–304

- high-level programming languages, 491
 - as defect prevention, 524
- hiring, software personnel, 50
- Hull, Raymond, 301
- hybrid assessment and benchmark studies, 421–422

I

- identity theft, 147
 - insurance, 142
- IEEE Computer*, 257
- IFPUG
 - sizing based on IFPUG function point analysis, 376–379
 - See also* function point analysis; function points
- ility words, 561–563
- incidents, 514
- industry benchmarks, 413–414
- Information Technology Infrastructure Library (ITIL), 196
- information technology (IT)
 - organizations, vs. systems software organizations, 277–278
- Information Technology Metrics and Productivity Institute Journal*, 258
- in-house education, 248–249
- inspection moderators, 632
- inspections
 - best practices, 124–128
 - as defect prevention, 522–523
- instrumentation, 134
- intangible value, 84
- intellectual property protection,
 - best practices, 136–138
- interpreters, 491
- international software
 - benchmarks, 413
- international software quality foundation (ISQF), proposal for, 608–613
- international software standards,
 - best practices, 135–136
- invalid defects, 512

ISO quality benchmarks, 424
 ITIL. *See* Information Technology
 Infrastructure Library (ITIL)

J

Joint Application Design (JAD),
 requirements, 459
 journals, 257–258

K

Kawasaki, Guy, 240
 key practice areas, 120
 keystroke loggers, 147–148
 knowledge areas, circa 2009, 232
 knowledge representation, 481

L

language development
 chronology of, 494
 history of, 490–491
 See also programming languages
 layoff, best practices for minimizing
 harm from, 41–45
 learning effectiveness, 230
 learning efficiency, 230
 learning methods
 commercial education, 250–252
 education channels available in
 1995, 229
 electronic books, 246–247
 evaluating, 229–230
 evolution of learning channels,
 228–230
 gaps in training circa 2009,
 266–267
 graduate university education,
 265–266
 in-house education, 248–249
 live conferences, 254–255
 mentoring, 260
 new directions in software
 learning, 267–268
 omissions from, 227–228,
 266–267
 on-the-job training, 259–260
 professional books, monographs,
 and technical reports, 260–263

proposed curricula, 269–273
 ranking of learning channels in
 2009, 231
 self-study using books, e-books,
 and training material,
 258–259
 self-study with CDs or DVDs,
 249–250
 simulation web sites,
 256–257
 software journals, 257–258
 undergraduate university
 education, 263–265
 vendor education, 252–253
 web browsing, 244
 webinars, podcasts, and
 e-learning, 245–246
 wiki sites, 255–256
See also knowledge areas;
 training
 legacy defects, 514
 licensing
 circa 2049, 218–220
 and specialization, 241
 lines of code (LOC), 17
 circa 1960, 538–539
 circa 1970, 539–542
 circa 1980, 542–546
 circa 1990, 546–548
 circa 2000, 548–549
 circa 2010, 549–550
 circa 2020, 550–551
 overview, 537–538
 sizing based on, 366–370
 litigation
 benchmarks, 428
 circa 2049, 221–225
 live conferences, 254–255
 LOC. *See* lines of code (LOC)
 LOC to function point conversion
 ratios of logical source code
 statements to function
 points, 387
 sizing legacy applications based
 on, 385–389
 logistical view,
 482–483

M

- macro viruses, 151
- maintenance, 103
 - benchmarks, 417–418
 - best practices, 161–164
 - circa 2049, 191–195
 - kinds of maintenance work, 311
 - maintenance value of high-
 - quality reusable materials, 104
 - paths, 14–16
 - quantifying, 16–19
 - software maintenance
 - organizations, 309–322
 - specialists, 631–632
- maintenance assignment scope, 314, 418
- malicious software engineers, 514–515
- malware, 148
- management consultants, best practices, 58–59
- managers
 - motivation and morale, 47–50
 - training, 89–91
- mandelbug, 135
- manual unit testing, for defect removal, 533–535
- matrix management, 304–308
- measurements, as defect prevention, 527–528
- measurements and metrics, best practices, 110–112
- mentoring, 260
- methodologies, as defect prevention, 527–528
- methodologies, practices, and results, 24–29
- methodology benchmarks, 418–419
- Metric Views*, 257
- micro function points, 318
- milestone tracking, best practices, 115–116
- milestones
 - defined, 116
 - tracking milestones for large software projects, 404–405

- monographs, 260–263
- monthly status reports, 406
- motivation and morale
 - of managers and executives, 47–50
 - of technical staff, 45–47

N

- nature, 60
 - defined, 65
- navigators, 286
- neutral practices, 17, 24, 35
 - See also* best practices; worst practices
- nonfunctional requirements, 459
- Northern Scope, 53

O

- object code, 491
- object-oriented (OO) paradigm, 181–182
- observers, 286
- occupation titles, 235–236
- one-person projects, 284–286
- online education, 229
- on-the-job training, 259–260
- open benchmarks, 429
- organization structures
 - best practices, 87–89
 - customer support organizations, 322–328
 - hierarchical organizations, 298–304
 - matrix organizations, 304–308
 - one-person projects, 284–286
 - pair programming, 286–289
 - self-organizing Agile teams, 289–293
 - software maintenance
 - organizations, 309–322
 - software quality assurance (SQA)
 - organizations, 342–348
 - software test organizations, 328–342
 - specialist organizations, 308–309
 - Team Software Process (TSP)
 - teams, 293–298

- outsourcing
 - best practices, 53–58
 - circa 2049, 195–204
 - distribution of outsource results
 - after 24 months, 54
- overall software cost and resource
 - benchmarks, 414

P

- pair programming, 108, 286–289
 - as defect prevention, 528
- partly open benchmarks, 429–430
- pattern matching
 - initial starting values for sizing
 - by, 395
 - requirements, 459–460
 - sizing based on, 389–401
- pattern view, 482
- patterns, 389
 - 150 applications sized used
 - pattern matching, 396–400
 - architectural, 76–77
 - as defect prevention, 521–522
 - staffing patterns for software
 - projects, 88
- payloads, 152
- performance benchmarks, 426–427
- performance issues, 515–516
- performance specialists, 627
- personnel
 - appraisals and career planning,
 - 50–51
 - motivation and morale, 45–47
 - ratio of specialists to general
 - software personnel, 241–243
 - selection and hiring of, 50
 - See also* specialists
- Peter, Lawrence J., 301
- the Peter Principle, 301
- phase-level productivity and
 - quality benchmarks, 415–416
- phishing, 148
 - See also* spear phishing
- physical security, 148
- piracy, 148–149
- planning
 - best practices, 77–78
 - defined, 78
- podcasts, 245–246
- polymorphic viruses, 151
- portfolio analysis, circa 2049,
 - 210–213
- private defect removal, 529–530
- Priven, Lew, 124
- problem complexity, 392, 451
- problem domains of software
 - applications, 500
- problem tracking, 403–408
- process assessments, 411–412
- productivity rates
 - customer support
 - organizations, 326
 - hierarchical organizations, 303
 - matrix organizations, 307
 - one-person projects, 285
- pair programming, 288
- self-organizing Agile teams,
 - 291–292
- software maintenance
 - organizations, 320
- software quality assurance (SQA)
 - organizations, 345
- software test organizations, 338
- Team Software Process (TSP)
 - teams, 296
- professional malpractice
 - defined, 34
 - methods and practices considered
 - professional malpractice, 34
- profilers, 134
- programming
 - best practices, 107–109
 - history of, 490–491
 - pair programming, 108
- programming languages, 492–495
 - chronology of programming
 - language development, 494
 - creating a national programming
 - language translation center,
 - 501–504
 - estimated number of software
 - engineers by language, 507
 - how many needed, 499–501
 - how many programmers use
 - various languages,
 - 506–508

- impact of multiple languages
 - on cost, 505
- multiple languages in the same
 - applications, 504–505
- popularity of, 494–499
- typed vs. un-typed, 494
- used for critical software
 - applications, 503
- progress tracking, 403–408
- project class, 391, 449–450
- project management
 - numbers and size ranges of
 - project management
 - tools, 356
 - overview, 351–359
 - performance circa 2009, 352
 - performance on successful and
 - unsuccessful projects, 355
 - potential performance
 - by 2019, 353
 - See also* sizing of software
 - applications
- project nature, 390, 448–449
- project offices, 78
- project planning, best practices,
 - 77–78
- project risk analysis, best practices,
 - 81–83
- project scope, 390–391, 449
- project size
 - customer support
 - organizations, 326
 - hierarchical organizations, 303
 - matrix organizations, 306–307
 - one-person projects, 284–285
 - pair programming, 288
 - self-organizing Agile
 - teams, 291
 - software maintenance
 - organizations, 319
 - software quality assurance (SQA)
 - organizations, 345
 - software test organizations, 338
 - Team Software Process (TSP)
 - teams, 295–296
- project type, 391–392, 450
- project value analysis, best
 - practices, 83–84

- protecting intellectual property,
 - best practices, 136–138
- prototypes, 460
 - as defect prevention, 525

Q

- quality
 - applying definitions to Vista,
 - 565–570
 - customer support organizations,
 - 326–327
 - defining, 558–565
 - economic value of quality, 633–642
 - hierarchical organizations, 303
 - impact of creeping requirements,
 - 584–585
 - matrix organizations, 307
 - measuring, 585–587
 - one-person projects, 285
 - overview, 555–558
 - pair programming, 289
 - rank order of quality factors by
 - importance, 577
 - return on investment in quality,
 - 641–642
 - self-organizing Agile teams, 292
 - software maintenance
 - organizations, 320–321
 - software quality assurance (SQA)
 - organizations, 346
 - software test organizations, 339
 - Team Software Process (TSP)
 - teams, 296
 - value of for applications of 100
 - function points, 636–637
 - value of for applications of 1000
 - function points, 637–638
 - value of for applications of 10,000
 - function points, 638–639
 - value of for applications of
 - 100,000 function points,
 - 640–641
 - See also* defect prevention; defect
 - removal
- quality benchmarks, 422–423
- quality function deployment (QFD),
 - requirements, 460–461
- quality specialists, 619–632

R

Radice, Ron, 124
 recalls, best practices, 158–159
 refactoring, 162
 releases, best practices, 164–165
 renovation, 14, 313, 499–500
 renovation productivity, 314
 requirements, 439
 Agile requirements with
 embedded users, 456
 completeness by software
 size, 445
 creating taxonomies of reusable
 software requirements,
 447–456
 creeping requirements, 457
 data mining for legacy
 requirements, 457–458
 defects by application size, 446
 defects per function point, 445
 engineering, 461
 executable English, 458
 focus groups, 458–459
 functional and nonfunctional
 requirements, 459
 inspections, 462
 Joint Application Design
 (JAD), 459
 pages per function point, 444
 pages produced by application
 size, 444
 pattern matching, 459–460
 prototypes, 460
 quality function deployment
 (QFD), 460–461
 reusable requirements, 463–464
 security requirements
 deployment (SRD), 464–465
 statistical analysis of software
 requirements, 442–447
 structure and contents of
 software requirements,
 440–442
 toxic requirements that cause
 serious harm, 446
 traceability, 462–463
 unified modeling language
 (UML), 465–466

 use-cases, 466
 user stories, 466–467
 requirements analysis, circa 2049,
 179–182
 requirements analysts, 629–630
 requirements changes, sizing,
 401–402
 requirements churn, 202, 402, 403
 requirements creep, 402, 403
 quality impacts of, 584–585
 requirements of software
 applications, best practices,
 70–72
 reusability, best practices, 99–101
 reusable materials
 certification of, 101–107
 customer support value of
 high-quality reusable
 materials, 105
 development value of
 high-quality reusable
 materials, 103
 enhancement value of
 high-quality reusable
 materials, 105
 maintenance value of
 high-quality reusable
 materials, 104
 total cost of ownership of
 high-quality reusable
 materials, 106
 reusable requirements, 463–464
 reused defects, 514
 reverse appraisals, of technical
 staff, 45–46
 risk analysis, best practices, 81–83
 risk analysis specialists, 620–622
 rogue security sites, 144
 root users, 149
 rootkits, 149

S

SANS report, 509–512
 Sarbanes-Oxley (SOX) Act,
 109–110
 schedules
 customer support
 organizations, 326

- hierarchical organizations, 303
- matrix organizations, 307
- one-person projects, 285
- pair programming, 288
- self-organizing Agile teams, 292
- software maintenance
 - organizations, 320
- software quality assurance (SQA)
 - organizations, 345–346
- software test organizations, 339
- Team Software Process (TSP)
 - teams, 296
- schrodenbug, 135
- scope, defined, 65
- scope control, best practices, 51–53
- scope managers, 53
- Scrum masters, 291
- Scrum sessions, 290–291
- secondhand defects, 512–513
- security analysis and control, best practices, 132–134
- security benchmarks, 424–425
- security costs, 153
- security requirements deployment (SRD), 464–465
- security specialists, 624–625
- security view, 482
- security vulnerabilities, 514
- segmentation, as defect prevention, 526–527
- SEI scoring system for the CMM, 419
- selecting software methods, tools, and practices, best practices, 59–64
- self-organizing Agile teams, 289–293
- self-study
 - using books, e-books, and training material, 258–259
 - using CDs or DVDs, 249–250
- Selling of the Dream* (Kawasaki), 240
- Service and Support Professionals Association (SSPA), 323–324
- service-oriented architecture (SOA), 101, 181–182, 474
- Shoulders Corporation, project tracking method, 116
- simulation web sites, 256–257
- Six Sigma
 - benchmarks, 423–424
 - as defect prevention, 528
 - specialists, 622
- size of application, 60
- sizing of software applications, 359–363
 - based on IFPUG function point analysis, 376–379
 - based on pattern matching, 389–401
 - best practices, 51–53
 - deliverables whose sizes should be quantified, 360–361
 - high-speed sizing using function point approximations, 383–385
 - legacy applications based on backfiring or LOC to function point conversion, 385–389
 - requirements changes, 401–402
 - traditional sizing based on lines of code (LOC) metrics, 366–370
 - traditional sizing by analogy, 363–365
 - using function point variations, 380–383
 - using story point metrics, 370–373
 - using use-case metrics, 373–376
- See also* scope control
- smart card hijacking, 149
- SOA. *See* service-oriented architecture (SOA)
- software architecture, 470–475
 - value of increases with structural size, 471
- Software Assurance (SwA), 139
- software compensation
 - benchmarks, 426
- software design, 479–480
- views, 481–484
- software journals, 257–258

- software learning, circa 2049, 213–216
 - software outsource vs. internal performance benchmarks, 417
 - software package evaluation and acquisition, circa 2049, 204–207
 - software personnel. *See* personnel
 - software personnel and skill benchmarks, 425–426
 - software quality assurance (SQA)
 - best practices, 120–124
 - organizations, 342–348
 - specialists, 627–628
 - Software Security State of the Art Report (SOAR), 139
 - software test organizations, 328–342
 - software turnover and attrition benchmarks, 426
 - solid line reporting authority, 305
 - source code, 491
 - Southern Scope, 53
 - SOX. *See* Sarbanes-Oxley (SOX) Act
 - spam, 149–150
 - span of control, 89, 290
 - spear phishing, 150
 - See also* phishing
 - specialist organizations, 308–309
 - specialists
 - best practices for use of, 92–94
 - the challenge of organizing, 281–284
 - circa 2009, 233–236
 - distribution of specialists for 1000 total software staff, 283
 - occupation groups, 93–94
 - quality specialists, 619–632
 - ratio of specialists to general software personnel, 241–243
 - See also* personnel
 - specialization
 - customer support organizations, 327
 - hierarchical organizations, 304
 - impact on software quality, 621
 - in large software organizations, 237–239
 - matrix organizations, 307–308
 - one-person projects, 285–286
 - pair programming, 289
 - self-organizing Agile teams, 292
 - software maintenance organizations, 321
 - software quality assurance (SQA) organizations, 347
 - software test organizations, 340
 - Team Software Process (TSP) teams, 296–297
 - varieties of circa 2009, 236–241
 - SPR assessment scoring system, 420
 - sprints, 52, 292, 527
 - spyware, 150–151
 - spyware protection, best practices, 138–153
 - SQA. *See* software quality assurance (SQA)
 - SSPA. *See* Service and Support Professionals Association (SSPA)
 - staffing patterns for software projects, 88
 - standards, best practices, 135–136
 - Starr, Paul, 218–219
 - static analysis, best practices, 124–128
 - status reports, 406
 - Stewart, Roger, 124
 - story point metrics, sizing using, 370–373
 - strong matrix, 306
 - See also* matrix management
 - structural view, 481
 - structured programming, 525
 - subroutine testing, for defect removal, 533
 - systems analysis. *See* business analysis
 - systems analysts, 626–627
 - systems software organizations, vs. information technology (IT) organizations, 277–278
- T**
- tangible financial value, 83
 - taxonomies, proposed taxonomy for software methodology analysis, 65–66

TCO. *See* total cost of ownership (TCO)

Team Software Process (TSP)
 taxonomy for software methodology analysis, 67
 teams, 293–298

teams, overview, 275–277

technical reports, 260–263

technical staff
 motivation and morale, 45–47
 skill sets needed, 92
 training best practices, 91–92
See also personnel

technical writers, 631

technology selection, circa 2049, 207–210

technology transfer, circa 2049, 207–210

terminating legacy applications, best practices, 166–167

test case coverage, 122

test coverage benchmarks, 422–423

test-based development (TBD), as defect prevention, 523–524

test-driven development (TDD), 330

testers, 630

testing
 best practices, 128–132
 black box testing, 128, 329, 533
 by customers or users, 129
 by developers, 128
 gray box testing, 128, 329
 software test organizations, 328–342
 test cases for selected test stages, 335
 by test specialists or software quality assurance, 129
 test staffing for selected test stages, 333
 unit tests, 329
 white box testing, 128, 329, 533–534

The Social Transformation of American Medicine (Starr), 218–219

total cost of ownership (TCO), 106

toxic requirements, 516

traceability, 119

tracking progress and problems, 403–408

training
 best practices for training clients or users of software, 155–156
 best practices for training managers, 89–91
 best practices for training technical personnel, 91–92
 gaps in training circa 2009, 266–267
 proposed curricula, 269–273
See also learning methods

training material, 258–259

Trojans, 151

troubled projects, best practices, 84–86

turning around troubled projects, best practices, 84–86

turnover benchmarks, 426

type, 60
 defined, 65

U

UCITA. *See* Uniform Computer Information Transaction Act (UCITA)

undergraduate university education, 263–265

undetected defects, 513

unified modeling language (UML), requirements, 465–466

Uniform Computer Information Transaction Act (UCITA), 158–159

unit tests, 329, 533–536

updates, best practices, 164–165

usability labs, 121

usability specialists, 624

usage benchmarks, 427–428

use-case metrics
 requirements, 466
 sizing using, 373–376

user documentation, circa 2049, 186–188

user groups and forums, 121
 user involvement in software
 projects, best practices, 72–73
 user stories, requirements,
 466–467

V

value analysis
 best practices, 83–84
 intangible value, 84
 tangible financial value, 83
 vendor education, 252–253
 vendor project management, 90
 virtual environments, 97–98
 virus protection, best practices,
 138–153
 viruses, 151

W

war driving, 152
 warranties, best practices, 158–159
 weak matrix, 306
 See also matrix management

web browsing, 244
 web designers, 628–629
 web sites, simulation, 256–257
 webinars, 229, 245–246
 whaling, 152
 white box testing, 128, 329,
 533–535
 See also testing
 wiki sites, 98, 255–256
 wireless security leaks, 152
 withdrawing legacy applications,
 best practices, 166–167
 worms, 152
 worst practices, 17
 See also best practices; neutral
 practices

Z

Zachman, John, 75
 Zachman architectural schema,
 75–76
 zombies, 146