
Software Engineering – Concepts and Implementation

CONTENTS

Lecture 1 THE PRODUCT

1

-
- ⊙ Software
 - ⊙ Software Characteristics
 - ⊙ Software Components
 - ⊙ Software Applications
 - ⊙ Software A Crisis on the Horizon
 - ⊙ Software myths

Lecture 2 PROCESS

13

-
- ⊙ Software Engineering – A Layered Technology
 - ⊙ Process, Methods, and Tools
 - ⊙ A Generic view of Software Engineering
 - ⊙ The Software Process
 - ⊙ Software process models
 - ⊙ Linear Sequential Models
 - ⊙ The Incremental model
 - ⊙ The formal methods Model

Lecture 3 PROJECT MANAGEMENT CONCEPT

26

-
- ⊙ The management Spectrum
 - ⊙ People
 - ⊙ The Problem
 - ⊙ Problem decomposition
 - ⊙ The process
 - ⊙ Process Decomposition
 - ⊙ The Project

Lecture 4 SOFTWARE PROJECT PLANNING – I

40

-
- ⊙ Observations on Estimating
 - ⊙ Project Planning Objectives
 - ⊙ Software Scope
 - ⊙ Obtaining Information Necessary for Scope
 - ⊙ A scoping Example
 - ⊙ Resources
 - ⊙ Human Resources
 - ⊙ Reusable Software Resources
 - ⊙ Environmental Resources

Lecture 5 SOFTWARE PROJECT PLANNING – II	50
---	-----------

- ⊙ Decomposition Techniques
- ⊙ Problem Based Estimations
- ⊙ An example of LOC Based Estimation

Lecture 6 RISK MANAGEMENT	58
----------------------------------	-----------

- ⊙ Reactive Vs. Proactive Risk Strategies
- ⊙ Software Risks
- ⊙ Risk Identification
- ⊙ Risk projection
- ⊙ Risk Mitigation, Monitoring and Management
- ⊙ Safety Risk and Hazards

Lecture 7 PROJECT SCHEDULING AND TRACKING – I	74
--	-----------

- ⊙ Comments on “Lateness”
- ⊙ Basic Principles
- ⊙ The Relationship Between People and Effort
- ⊙ An Empirical Relationship
- ⊙ Effort Distribution
- ⊙ Defining a Task Set for the Software Project
- ⊙ Degree of Rigor
- ⊙ Defining Adaption Criteria
- ⊙ Computing a Task Set Selector Value
- ⊙ Interpreting the TSS value and Selecting the Task Set

Lecture 8 PROJECT SCHEDULING AND TRACKING – II	85
---	-----------

- ⊙ Selecting Software Engineering Tasks
- ⊙ Refinement of major Tasks
- ⊙ Defining a Tasks Network
- ⊙ Scheduling
- ⊙ Timeline Charts
- ⊙ Tracking the Schedule
- ⊙ The Project Plan

Lecture 9 SOFTWARE QUALITY ASSURANCE – I	97
---	-----------

- ⊙ Quality concept
- ⊙ The Quality Movement
- ⊙ Software Quality Assurance
- ⊙ Software Reviews

Lecture 10 SOFTWARE QUALITY ASSURANCE – II	108
---	------------

- ⊙ Formal Techniques Reviews
 - ⊙ Formal Approaches to SQA
-

-
- ⊙ Statistical Quality Assurance
 - ⊙ Software Reliability
 - ⊙ The SQA Plan
 - ⊙ The ISO 9000 Quality Standards

Lecture 11 SOFTWARE CONFIGURATION MANAGEMENT**123**

-
- ⊙ Software Configuration management
 - ⊙ The SCM process
 - ⊙ Identification of Objects in the Software Configuration
 - ⊙ Version Control
 - ⊙ Change Control
 - ⊙ Configuration Audit
 - ⊙ Status Reporting
 - ⊙ SCM Standards

Lecture 12 SYSTEM ENGINEERING – I**139**

-
- ⊙ The System Engineering Hierarchy
 - ⊙ System Modeling
 - ⊙ Information Engineering : An Overview
 - ⊙ Product Engineering : An overview
 - ⊙ Information Engineering

Lecture 13 SYSTEM ENGINEERING – II**150**

-
- ⊙ Information Strategy Planning
 - ⊙ Enterprise modeling
 - ⊙ ‘Business Level Data Modeling
 - ⊙ Business Area Analysis
 - ⊙ Information flow modeling

Lecture 14 SYSTEM ENGINEERING - III**161**

-
- ⊙ Product Engineering
 - ⊙ Modeling The System Architecture
 - ⊙ System modeling and Simulation
 - ⊙ System Specification

Lecture 15 ANALYSIS MODELING – I**175**

-
- ⊙ The Elements of the Analysis model
 - ⊙ Data Modeling
 - ⊙ Data Objects, Attributes and Relationships
 - ⊙ Cardinality and modality
 - ⊙ Entity Relationship Diagrams
 - ⊙ Function modeling and information flow
-

-
- ⊙ Data Flow Diagrams
 - ⊙ Extensions for Real Time Systems
 - ⊙ Ward and Mellor Extension
 - ⊙ Hatley and Pirbhai Extension

Lecture 16 ANALYSIS MODELING –II**192**

-
- ⊙ Behavioral Modeling
 - ⊙ The mechanic of Structured Analysis
 - ⊙ Creating an Entity Relationship Diagram
 - ⊙ Creating a Data flow model
 - ⊙ Creating a Control flow Model
 - ⊙ The Control Specification
 - ⊙ The Process Specification

Lecture 17 ANALYSIS MODELING – III**207**

-
- ⊙ Data Dictionary
 - ⊙ An Overview of other Classical Analysis methods
 - ⊙ Data Structured System Development
 - ⊙ Jackson System Development
 - ⊙ SADT

Lecture 18 DESIGN CONCEPTS AND PRINCIPLES – I**214**

-
- ⊙ Software Design and Software Engineering
 - ⊙ The Design Process
 - ⊙ Design Principle
 - ⊙ Design Concepts

Lecture 19 DESIGN CONCEPTS AND PRINCIPLES – II**230**

-
- ⊙ Functional independence
 - ⊙ Design Heuristics for Effective modularity
 - ⊙ The Design model
 - ⊙ Design Documentation

Lecture 20 DESIGN METHODS – I**241**

-
- ⊙ Architectural design
 - ⊙ The Architectural Design process
 - ⊙ Transform Mapping
 - ⊙ Transaction Mapping
 - ⊙ Sign Postprocessing

Lecture 21 DISCUSSION

Lecture 22 DESIGN METHODS – II	262
---------------------------------------	------------

- ⊗ Architectural Design optimization
- ⊗ Interface Design
- ⊗ Human Computer interface Design
- ⊗ General Interaction
- ⊗ Procedural Design

Lecture 23 DISCUSSION

Lecture 24 DESIGN FOR REAL TIME-I	284
--	------------

- ⊗ Real Time Systems
- ⊗ Integration and performance Issues
- ⊗ Interrupt handling
- ⊗ Real time databases
- ⊗ Real Time Operating Systems
- ⊗ Real Time Languages
- ⊗ Task Synchronization and Communications

Lecture 25 DESIGN FOR REAL TIME – II	293
---	------------

- ⊗ Analysis and Simulation of Real Time Systems
- ⊗ Real Time Design

Syllabus	306
-----------------	------------



Lecture 1

The Product

Objectives

In this lecture you will learn the following

- ✎ Software characteristics
- ✎ Software components
- ✎ Software applications
- ✎ Software myths

Coverage Plan

Lecture 1

- 1.1 Snap shot the evolving role of software
- 1.2 Software
- 1.3 Software Characteristics
- 1.4 Software Components
- 1.5 Software Applications
- 1.6 Software: a crisis on the horizon
- 1.7 Software Myths
- 1.8 Short Summary
- 1.9 Brain Storm

1.1 Snap Shot

Today software takes on a dual role. It is a product and at the same time the vehicle for delivering a product. As a product it delivers the computing potential embodied by computer hardware. Whether it resides within a cellular phone or operates inside a mainframe computer software is an information transformer producing, managing, acquiring modifying, displaying or transmitting information that can be as simple as a single bit or as complex as a multimedia simulation. As the vehicle used to deliver the product software acts as the basis for the control of the computer (operating systems) the communication of information (networks) and the creation and control of other programs (software tools and environments).

Software delivers what many believe will be the most important product of the twenty-first century information. Software transforms personal data (e.g., an individual financial transactions) so that the data can be more useful in a local context it manages business information to enhance competitiveness it provides a gateway to world wide information networks (e.g., the Internet); and it provides the means for acquiring information in all of its forms.

1.2 Software

In 1970s less than 1 percent of the public could have intelligently described what “computer software” meant. Today most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form: Software is (1) instructions (computer programs) that when executed provide desired function and performance (2) data structures that enable the programs to adequately manipulate information and (3) documents that describe the operation and use of the programs. There is no question that other more complete definitions could be offered. But we need more than a formal definition.

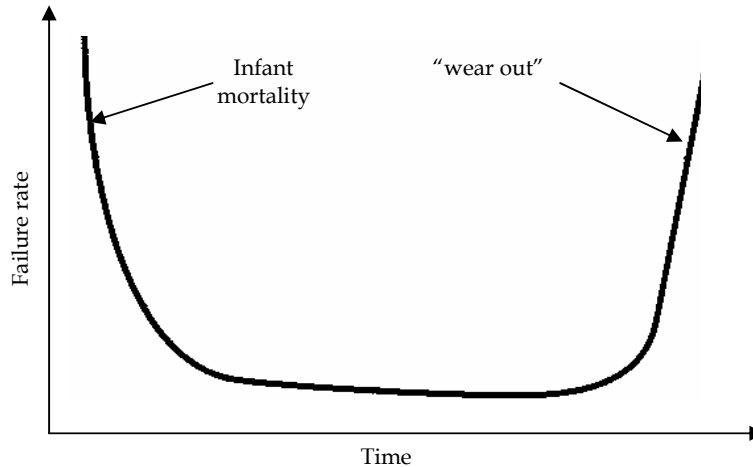
1.3. Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering) it is important to examine the characteristics of software that make it different from other things that humans build. When hardware is built the human creative process (analysis, design, construction) is ultimately translated into a physical form. If we build a new computer our initial sketches formal design drawings and bread boarded prototypes evolve into a physical product (VLSI chips, circuit boards, power supplies etc). Software is a logical rather than a physical system element. Therefore software has characteristics that differ considerably from those of hardware.

Software is developed or engineered, it is not manufactured in the classical sense. Although some similarities exist between software development and hardware manufacture the two activities are fundamentally different. In both activities high quality is achieved through good design but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities depend on people but the relationship between people applied and work accomplished is entirely different.

Software doesn't wear out. Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the “bathtub curve”, indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects) defects are corrected, and the failure rate drops to a steady state level (hopeful, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Figure 1.1 Failure curve for hardware



Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form shown Figure 1.2 Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (hopefully without introducing other) and the curve flattens as shown as shown Figure 1.2 is a gross over simplification of actual failure models for software. However the implication is clear software doesn't wear out. But it does deteriorate!

This seeming contradiction can best be explained by considering Figure 1.3 During its life software will undergo change (maintenance) As changes are made it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.3 Before the curve can return to the originally steady state failure rate, another change is requested causing the curve to spike again Slowly the minimum failure rate level begins to rise the software is deteriorating due to change.

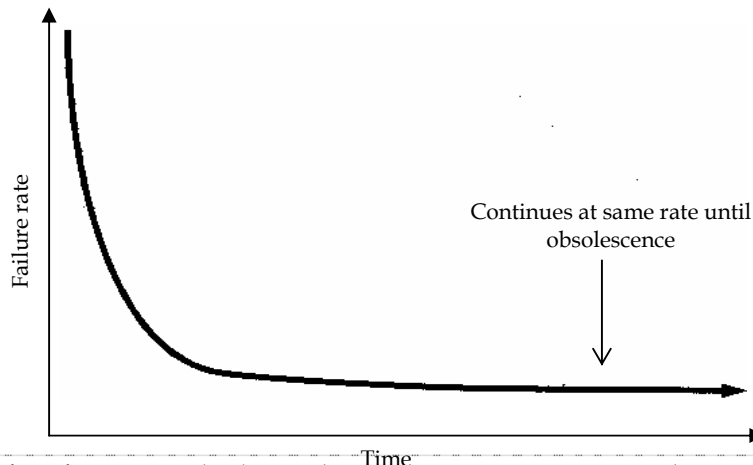


Figure 1.2 Failure curve for software (ideallized)

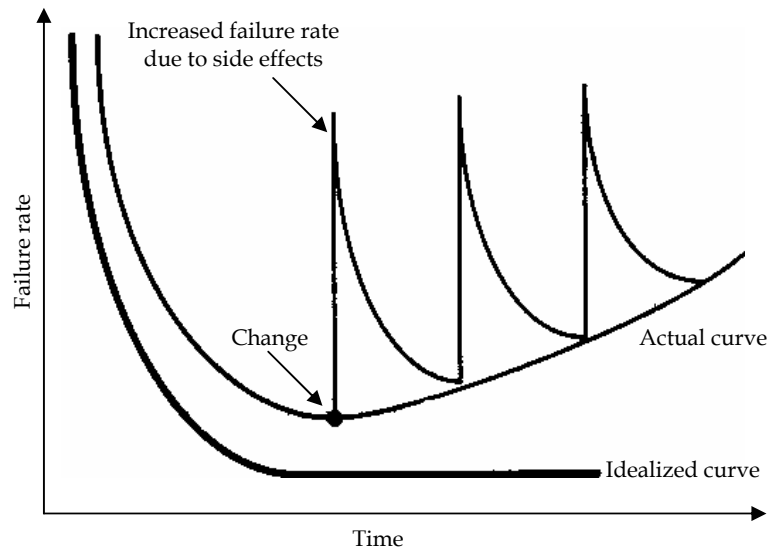


Figure 1.3 Actual failure curve for software

Most software is custom-built rather than being assembled from existing components. Consider the manner in which the control hardware for a microprocessor based product is designed and built. The design engineer draws a simple schematic of the digital circuitry does some fundamental analysis to ensure that proper function will be achieved and then refers to a catalog of digital components. Each integrated circuit (often called an "IC" or a "Chip") has a part number a defined validated function a well-defined interface and a standard set of integration guidelines. After each component is selected it can be ordered off the shelf.

Sadly, software designers are not afforded the luxury described above. With few exceptions there are no catalogs of software components. It is possible to order off-the-shelf software, but only as a complete unit not as components that can be reassembled into new programs. Although much has been written about "software reusability" we are only beginning to see successful implementations of the concept.

1.4 Software Components

As an engineering discipline evolves a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design (i.e., the parts of the design that represent something new). In the hardware world, component reuse is a natural part of the engineering process. In the software world it is something that has yet to be achieved on a broad scale.

Reusability is an important characteristic of a high quality software component. A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner, but had a limited domain of application. Today we have extended our view of reuse components encapsulate both data and the processing that is applied to the data enabling the software engineer to create new application from reusable parts. For example today's interactive interfaces are built using reusable components that enable the creation of graphics windows pull-down menus and a wide variety of interaction mechanism. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

Software components are built using a programming language that has a limited vocabulary an explicitly defined grammar and well formed rules of syntax and semantics. At the lowest level the language mirrors the instruction set of the hardware. At mid-level programming languages such as Ada 95, C or Smalltalk are used to create a procedural description of the program. At the highest level the language uses graphical icons or other symbology to represent the requirements for a solution. Executable instructions are automatically generated.

Machine level language symbolic representation of the CPU instruction set. When a good software developer produces a maintainable well documented program machine level language can made extremely efficient use of memory and "optimize" program execution speed. When a program is poorly designed and has little documentation machine language is a nightmare.

Mid-level languages allow the software developer and the program to be machine-independent. When a more sophisticated translator is used, the vocabulary, grammar, syntax and semantics of a mid-level language can be such more sophisticated than machine-level languages. In fact mid-level language compilers and interpreters produce machine-level language as output.

Although hundreds of programming languages are in use today fewer than ten mid-level programming languages are widely used in the industry. Languages such as COBOL and FORTRAN remain in widespread use more than 30 years after their introduction. More modern programming languages such as Ada95, C, C++, Eiffel, Java and Smalltalk have each gained an enthusiastic following.

Machine code assembly languages and mid-level programming languages are often referred to as the first three generation of computer languages. With any of these languages the programmer must be concerned both with the specification of the information structure and the control of the program itself. Hence languages in the first three generation are termed procedural languages.

Fourth generation languages also called nonprocedural languages move the software developer even further from the computer hardware. Rather than requiring the developer to specify procedural detail, the nonprocedural language implies a program by "specifying the desired result rather than specifying action required to achieve that result" [COB85]. Support software translates the specification of result into a machine executable program.

1.5 Software Applications

Software may be applied in any situation for which a pre-specified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert systems and artificial neural network software). Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information. For example many business applications make use of highly structured input data and produce formatted "reports". Software that controls an automated machine (e.g., numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.

Information determinacy refers to the predictability of the order and timing of information. An engineering analysis program accepts data that have a predefined order, executes the analysis algorithm without interruption and produces resultant data in report or graphical format. Such applications are determinate. A multiuser operating system on the other hand accepts inputs that have varied content and arbitrary timing, executes algorithms that can be interrupted by external conditions and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate.

It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

System Software System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors and file management utilities) processes complex but determinate information structures. Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data. In either case the systems software area is characterized by heavy interaction with computer hardware, heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing and sophisticated process management; complex data structures and multiple external interfaces.

Real-Time Software Programs that monitor/analyze/control real world events as they occur are called real-time software. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment so that real-time response (typically ranging from 1 millisecond to 1 minute) can be maintained. It should be noted that the term "real-time" differs from "interactive" or timesharing". A real-time system must respond within strict time constraints. The response time of an interactive (or time-sharing) system can normally be exceeded without disastrous results.

Business Software Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory, etc.) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operation or management decision making. In addition to conventional data processing applications, business software applications also encompass interactive and client/server computing (e.g., point-of-scale transaction processing).

Engineering and Scientific Software Engineering and Scientific software has been characterized by "number crunching" algorithms. Application ranges from astronomy to volcanology from automotive

stress analysis to space shuttle orbital dynamics and from molecular biology to automated manufacturing. However new applications with the engineering/scientific area are moving away from conventional numerical algorithms. Computer aided design system simulation and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded Software Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., digital functions in an automobile such as fuel control, dashboard displays, braking systems, etc.,).

Personal Computer Software The personal computer software market has burgeoned over the past decade. Word processing, spreadsheets, computer graphics, multimedia entertainment, database management personal and business financial applications and external network or database access are only a few of hundreds of application.

Artificial Intelligence Software Artificial Intelligence (AI) software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. An active AI area is expert systems also called knowledge-based systems. However other application areas for AI software are pattern recognition (image and voice) theorem proving and game playing. In recent years a new branch of AI software called artificial neural networks, has evolved. A neural network simulates the structure of brain processes (the functions of the biological neuron) and may ultimately lead to a new class of software that can recognize complex patterns and learn from past experience.

1.6 Software: A Crisis on the Horizon

Many industry observers have characterized the problems associated with software development as a “crisis” Yet what we really have may be something rather different.

The word “crisis” is defined in Webster’s Dictionary as “a turning point in the course of anything : decisive or crucial time stage or event” Yet for software there has been no “turning point” no “decisive time” only slow evolutionary change. In the software industry we have had a “crisis” that has been with us for close to 30 years and that is a contradiction in terms.

Anyone who looks up the word “crisis” in the dictionary will find another definition: “the turning point in the course of a disease when it becomes clear whether the patient will live or die.” This definition may give us a clue about the real nature of the problems that have plagued software development.

We have yet to reach the stage of crisis in computer software. What we really have is a chronic affliction. The word “affliction” is defined as anything causing pain or distress” But it is the definition of the adjective “chronic” that is the key to our argument: “lasting a long time or recurring often; continuing indefinitely”. It is far more accurate to describe what we have endured for the past three decades as a chronic affliction rather than a crisis. There are no miracle cures, but there are many ways that we can reduce that pain as we strive to discover a cure.

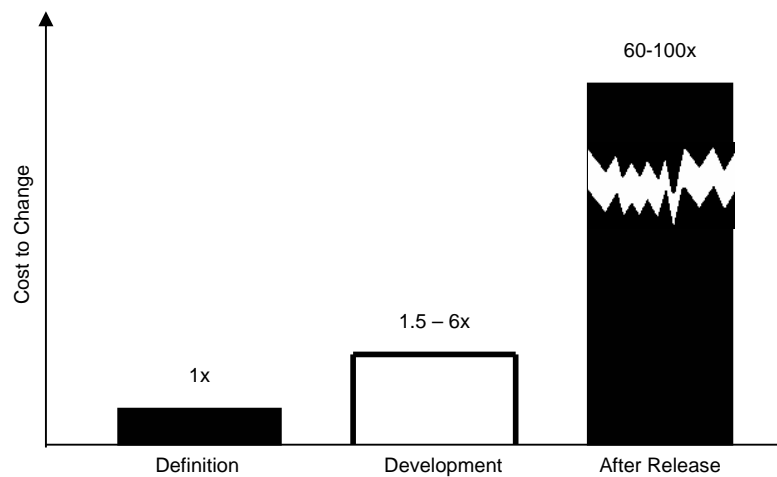


Figure 1.4 The impact of change

Whether we call it a software crisis or a software affliction the term alludes to asset or problems that are encountered in the development of computer software. The problems are not limited to software that “doesn’t function properly” Rather, the affliction encompasses problems associated with how we develop software how we maintain a growing volume of existing software and how we can expect to keep pace with a growing demand for more software. Although reference to a crisis or even an affliction can be criticized for being melodramatic the phrases do serve a useful purpose by denoting real problems that are encountered in all area of software development.

1.7 Software Myths

Many causes of a software affliction can be traced to a mythology that arose during the early history of software development. Unlike ancient myths which often provided human lessons that are well worth heeding software myths propagated misinformation and confusion. Software myths had a number of attributes that made them insidious ; For instance they appeared to be reasonable statements of fact (sometimes containing elements of truth) they had an intuitive feel and they were often promulgated by experienced practitioners who “knew the score”.

Today most knowledgeable professionals recognize myths for what they are— misleading attitudes that have caused serious problems of managers and technical people alike. However old attitudes and habits are difficult to modify and remnants of software myths are still believed.

Management Myths Managers with software responsibility like managers in most disciplines are often under pressure to maintain budgets keep schedules from slipping and improve quality. Like a drowning person who grasps at a straw a software a manager often grasps at belief in a software myth if that belief will lessen the pressure(even temporarily).

Myth: We already have a book that’s full of standard and procedures for building software. Won’t that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software development practice? Is it complete? In many cases the answer to all of these questions is “no”.

Myth: My people do have state of the art software development tools, After all we buy them the newest computers.

Reality: It takes much more than the latest model mainframe workstation or PC to do high quality software development. Computer aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity yet the majority of software developers still do not use them.

Myth: If we get behind schedule we can add more programmers and catch up (sometimes called the “Mongolian horde concept”)

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [BRO75], adding people to a late software project makes it later”. At first this statement may seem counterintuitive. However as new people are added people who were working must spend time educating the newcomers thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Consumer Myths: A customer who requests computer software may be a person at the next desk, a technical group down the hall the marketing /sales department or an outside company that has requested software under contract. In many cases the customer believes myths about software because software responsible managers and practitioners do little to correct misinformation ≥ Myths lead to false expectations (by the consumer) and ultimately dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing programs we can fill in the details later.

Reality: poor up-front definition is the major cause of failed software efforts. A formal and detailed description of information domain, function performance interfaces, design constraints and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change , but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements do change, but the impact of change varies with the time at which it is introduced. Figure 1.4 illustrates the impact of change. If serious attention is given to up-front definition early requests for change can be accommodated easily. The customer can review requirements and recommend modification with relatively little impact on cost . When changes are requested during software design cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification i.e., additional cost, Changes in function, performance interfaces or other characteristics

during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production use can be more than an order of magnitude more expensive than the same change requested earlier.

Practitioner's Myth Myths that are still believed by software practitioners have been fostered by decades of programming culture. As we noted earlier in this chapter during the early days of software programming was viewed as an art form. Old ways and attitudes die hard.

Myth Once we write the program and get it to work our job is done.

Reality Someone once said that “ the sooner you begin ‘writing code ‘, the longer it’ll take you to get done” Industry data indicate that between 50and 70 percent of all effort expended on a program will be expended after it is delivered to the customer for the first time.

Myth Until I get the program “running,” I really have now a of assessing its quality.

Reality One of the most effective software quality assurance mechanisms can be applied from the inception fo a project the formal technical review. Software review are a “quality filter” tha ;t has ebeb found to be more effeetive than testing for finding certain classes of software errors.

Myth : The only deliverable for a successful project is the working program.

Reality : A working program is only one part of a software configuration that includes programs, documents and data. Documentation forms the foundatin for successful developmetn and more important provides guidance for the software maintenance task.

Many software professionals recognize the fallacy fo the myths described above. Regrettably habitual attitudes and mehtods foster poor management and technical practices even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of pratical solutions for software development.

1.8 Short Summary

- Software has become the key element in the evolution of computer based systems and products. Over the past four decades, software has evolved from a specialized problem-solving and informaiton analysis tool to an industry in itself. But early ‘ programming ‘ culture and hsitory have created a set of problems that persists today. Software has become a limiting factor in the evolution of computer based systems.
- Software is composed of programs, data, and ddocuments. Each of these items comprises a configuration that is created as part of the software engineering process. The intent of software engineering is to provide a framework for building software with higher quality.

1.9 Brain Storm

1. Write a note on software components.
2. write a note on software applicaitons.



Lecture 2

Process

Objectives

In this lecture you will learn the following

- ✧ Software Engineering - a layered Technology
- ✧ Software Process
- ✧ Software Process Model
- ✧ The linear Sequential Model

Coverage Plan

Lecture 2

- 2.1 Snap Shot
- 2.2 Software engineering – Layered technology
- 2.3 Process, Methods, and Tools
- 2.4 A Generic View of Software Engineering
- 2.5 The Software Process.
- 2.6 Software Process Models
- 2.7 Linear Sequential Model
- 2.8 Evolutionary Software Process Model
- 2.9 The Incremental Model
- 2.10 The Formal Methods Model
- 2.11 Short Summary
- 2.12 Brain Storm

2.1 Snap Shot

Software engineering is performed by creative, knowledgeable people who should work within a defined and mature software process. The intent of this lecture is to provide a survey of the current state of the software process and to provide pointers to more detailed discussion of management.

2.2 Software Engineering – Layered Technology

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer at the seminal conference on the subject still serves as a basis for discussion:

“Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.”

Almost every reader will be tempted to add to this definition. It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of a mature process. And yet, Bauer’s definition provides us with a baseline. What are the “sound engineering principles” that can be applied to computer software development? How do we “economically” build software so that it is “reliable”? What is required to create computer programs that work “efficiently” on not one but many different “real machines”? These are the questions that continue to challenge software engineers.

The IEEE [IEE93] has developed a more comprehensive definition when it states;

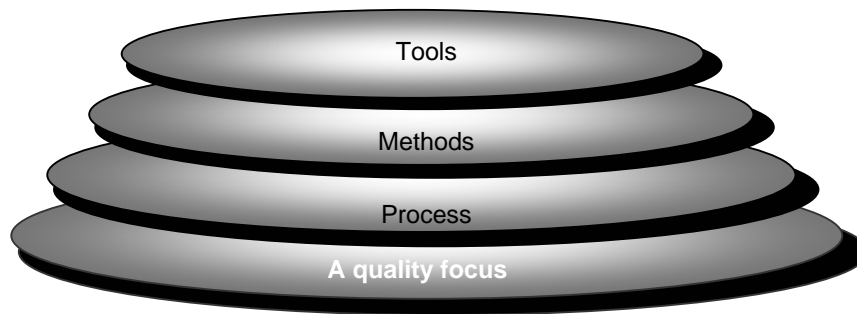
“Software Engineering (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software that is, the application of engineering to software. (2) The study of approaches as in (1) “.

2.3 Process, Methods, and Tools

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a focus on quality.

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed.

Figure 2.1 software engineering layers



Software engineering methods provide the technical “how to’s” for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and maintenance. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established. CASE combines software, hardware, and a software engineering database to create a software engineering environment that is analogous to CAD/CAE for hardware.

2.4 A Generic View of Software Engineering

Engineering is the analysis, design, construction, verification, and management of technical entities. Regardless of the entity that is to be engineered, the following questions must be asked and answered:

- What is the problem to be solved?
- What are the characteristics of the entity that is used to solve the problem?
- How will the entity be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity?

Throughout this book we focus on a single entity computer software. To engineer software adequately, a software development process must be defined. In this section the generic characteristics of the software process are considered. Later in this chapter, specific process models are addressed.

The work that is associated with software engineering can be categorized in to three generic phases, regardless of application area, project size, or complexity. Each phase addresses one or more of the questions noted above.

The definition phase focuses on what. That is, during definition, the software developer attempts to identify what information is to be processed what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the

software are identified. Although the methods applied during the definition phase will vary depending upon the software engineering paradigm (or combination of paradigms) that is applied, three major tasks will occur in some form. System or information engineering software project, planning and requirements analysis.

The development phase focuses on how. That is during development a software engineer attempts to define how data are to be structured, how function is to be implemented as a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language) and how testing will be performed. The methods applied during the development phase will vary, but three specific technical tasks should always occur software design code generation and software testing.

The maintenance phase focuses on change that is associated with error correction, adaptations required as the software's environment evolves and changes due to enhancements brought about by changing customer requirements. The maintenance phase reapplies the steps of the definition and development phases, but does so in the context of existing software. Four types of change are encountered during maintenance phase.

Correction. Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software, corrective maintenance changes the software to correct defects.

Adaptation. Over time, the original environment (e.g CPU, operating system, business rules external product characteristics) for which the software was developed is likely to change. adaptive maintenance results in modification to the software to accommodate changes to its external environment.

Enhancement. As software is used, the customer/user will recognize additional functions that will provide benefit, perfect maintenance extends the software beyond its original functional requirements.

Prevention. Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

Today, the "aging software plant" is forcing many companies to pursue software reengineering strategies. In a global sense, software reengineering is often considered as part of business process reengineering [STR 95]

The phases and related steps described in our generic view of software engineering are complemented by a number of umbrella activities. Typical activities in this category include.

- ◆ Software project tracking and control
- ◆ Formal technical reviews
- ◆ Software quality assurance
- ◆ Software configuration management
- ◆ Document preparation and production
- ◆ Reusability management
- ◆ Measurement

- ◆ Risk management

Umbrella activities are applied throughout the software process and are discussed in parts Two and Five of this book.

2.5 The Software Process.

A software process can be characterized as shown in fig 2.2. A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of task sets each a collection of software engineering work tasks, project milestones, software work products and deliverables, and quality assurance points enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally umbrella activities such as software quality assurance, software configuration management, and measurement overlay the process model. Umbrella activities are independent of any one-framework activity and occur throughout the process.

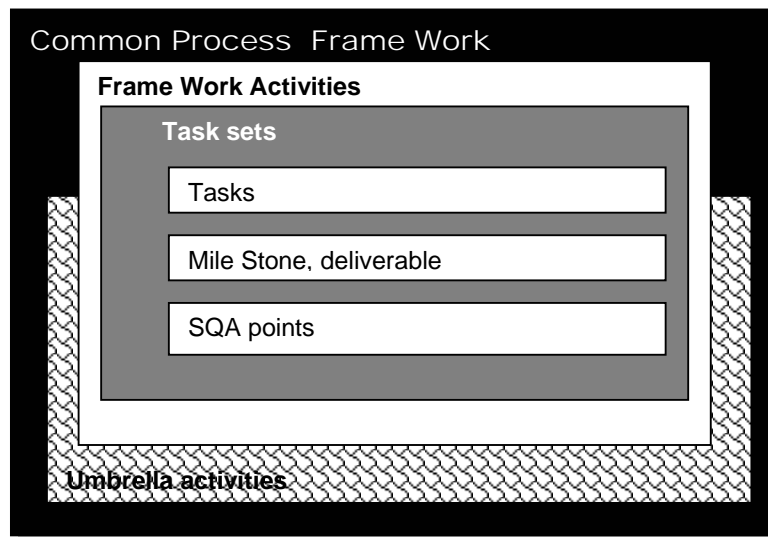


Figure 2.2 The software process

In recent years, there has been a significant emphasis on “process maturity”[PAU 93]. The Software Engineering Institute (SEI) has developed a comprehensive model that is predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity. To determine an organization’s current state of process maturity, the SEI uses an assessment questionnaire and a five-point grading scheme. The grading scheme determines compliance with a capability maturity model [PAU93] that defines key activities required at different levels of process maturity. The SEI approach provides a measure of the global effectiveness of a company’s software engineering practices and established five process maturity levels, which are defined in the following manner.

Level 1: Initial The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: Repeatable – Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: Defined – The software process for both management and engineering activities is documented, standardized and integrated into an organization wide software process. All projects use a documents and approved version of the organization's process for developing and maintaining software. This level includes all characteristics defined for level 3.

Level 4: Managed – Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

Level 5: Optimizing – Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

The five levels defined by the SEI are derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization's process maturity.

The SEI has associated key process areas with each of the maturity levels. The KPAs describe those software engineering functions (e.g software project planning, requirements management) that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics.

- Goals the overall objectives that the KPA must achieve
- Commitments requirements (imposed on the organization) that must be met achieve the goals, and that provide proof of intent to comply with the goals
- Abilities those things that must be in place (organizationally and technically) that will enable that organization to meet the commitments
- Activities the specific tasks that are required to achieve the KPA function.
- Methods for monitoring implementation – the manner in which the activities are monitored as they are put into place.
- Methods for verifying implementation – the manner in which proper practice for the KPA can be verified. Eighteen KPAs (each described using the structure noted above) are defined across the maturity model and are mapped into different levels of process maturity. The following KPAs should be achieved at each process maturity level:

Process maturity level 2

- ⊥ Software configuration management
- ⊥ Software quality assurance
- ⊥ Software subcontract management
- ⊥ Software project tracking and oversight
- ⊥ Software project planning
- ⊥ Requirement management

Process maturity level 3

- ⊥ Peer reviews
- ⊥ Inter group coordination
- ⊥ Software product engineering
- ⊥ Integrated software management
- ⊥ Training program
- ⊥ Organization process definition
- ⊥ Organization process focus

Process maturity level 4

- ⊥ Software quality management
- ⊥ Quantitative process management
- ⊥ Process maturity level 5
- ⊥ Process change management
- ⊥ Technology change management
- ⊥ Defect prevention

Each of the KPAs is defined by a set of key practices that contribute to satisfying its goals. The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted. The SEI defines key indicators as “those key practices or components of key practices that offer the greatest insight into whether the goals of a key process area have been achieved.” assessment questions are designed to probe for the existence (or lack thereof) of a key indicator.

2.6 Software Process Models

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers described in previous section and the generic phases also discussed in previous section. This strategy is often referred to as a process model or a software engineering paradigm. A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. In an intriguing paper on the nature of the software process, L.B.S. raccoon [RAC95] uses fractals as the basis for a discussion of the true nature of the software process.

All software development can be characterized as a problem solving loop (figure 2.3a) in which four distinct stages are encountered; status quo, problem definition, technical development, and solution integration. Status quo “represents the current state of affairs”; problem definition identifies the specific problem to be solved; technical development solves the problem through the application of some technology, and solution integration delivers the results who requested the solution in the first place.

The problem solving loop described above applies to software engineering work at many different levels of resolution. It can be used at the macro level when the entire application is considered; at a middle level when program components are being engineered, and even at the line of code level. Therefore, a figure 2.3b, each stage in the problem solving loop contains an identical problem solving loop, which contains still another problem solving loop (this continues to some rational boundary; for software, a line of code);

Realistically, it is difficult to compartmentalize activities as neatly as figure 2.3b implies because cross talk occurs within and across stages, yet this simplified view leads to a very important idea; regardless of the process model that is chosen for a software project, all of the stages – status quo, problem definition,

technical development, and solution integration-coexist simultaneously at some level of detail. Given the recursive nature figure 2.3b the four stages discussed above apply equally to the analysis of a complete application and to the generation of a small segment of code.

Raccoon suggests a “Chaos model”; that describes “ software development a continuum from the user to the developer to the technology”. As work progresses toward a complete system, the stages described above are applied recursively to user needs and the developer’s technical specification of the software.

2.7 Linear Sequential Model

Figure 2.4 illustrates that linear sequential model for software engineering. Sometimes called the “classic life cycle” or the “waterfall model”, the linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design coding, testing, and maintenance. Modeled after the conventional engineering cycle, the linear sequential model encompasses the following activities.

System / information engineering and modeling. Because software always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interface with other elements such as hardware, people, and gathering at the system level with a small amount of top level analysis and design. Information engineering encompasses requirements gathering a the strategic business level and at the business area level.

Software requirements analysis. The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer must understand the information domain for the software, as well as required function, behavior, performance, and interfacing requirements for both the system and the software are documented and reviewed with the customer.

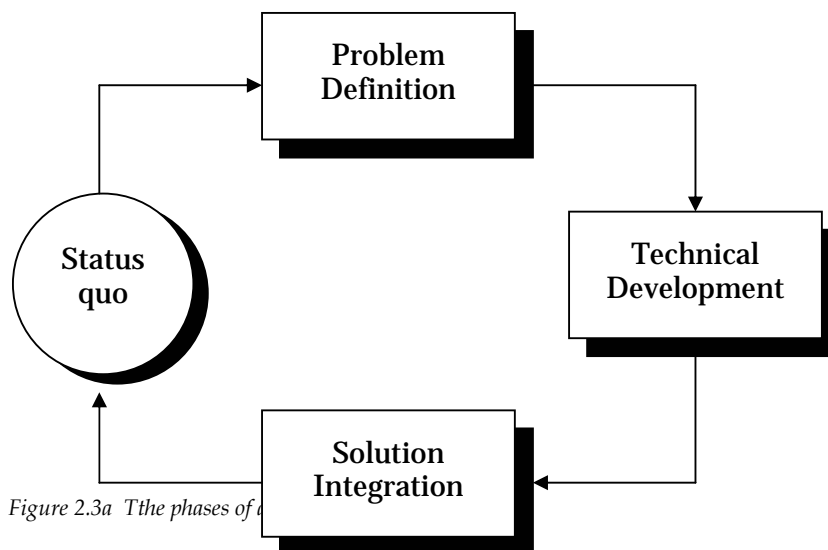
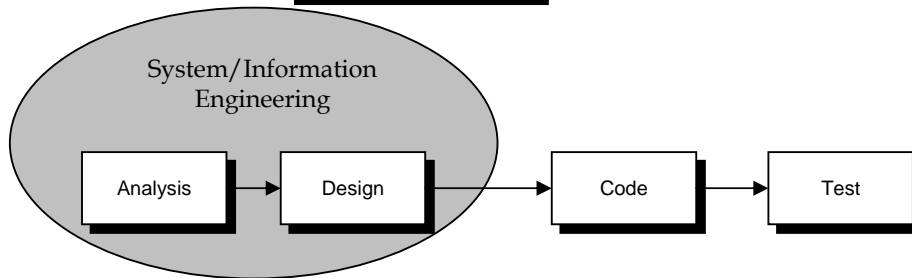
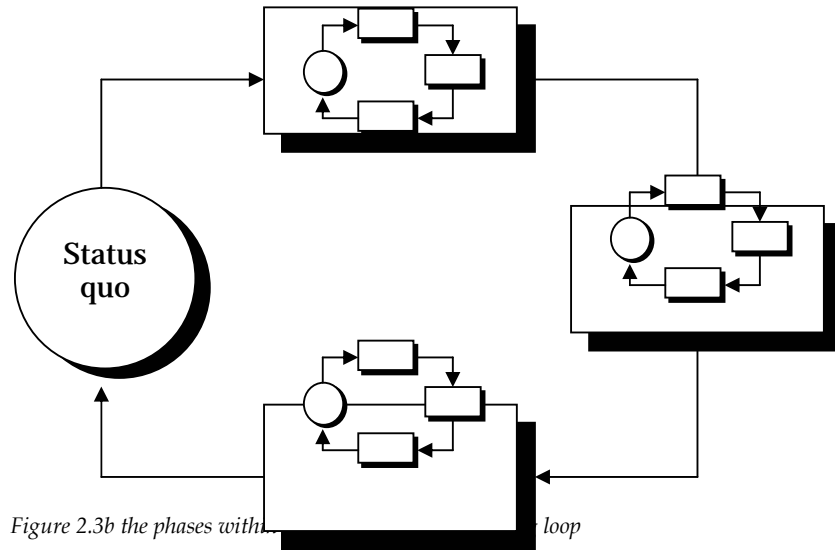


Figure 2.3a The phases of



Design Software design is actually a multistep process that focuses on four distinct attributes of a program; data structure, software architecture interface representations, and procedural detail. The design process translates requirements into a representation of the software that can be assessed for quality before code generation begins. Like requirements, the design is documented and becomes part of the software configuration.

Code generation The design must be translated into a machine readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, assuring that all statements have been tested, and on the functional externals that is conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required result.

Maintenance Software will undoubtedly undergo change after it is delivered to the customer. Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g. a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software maintenance reapplies each of the preceding phases to an existing program rather than a new one.

The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticisms of the paradigm has caused even active supporters to question its efficacy. Among the problems that are sometimes encountered when the linear sequential model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.
4. Developers are often delayed unnecessarily, in an interesting analysis of actual projects, Bradac found that the linear nature of the classic life cycle leads to “blocking states” in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking states tend to be more prevalent at the beginning and end of a linear sequential process.

Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and maintenance can be placed. The classic life cycle remains the most widely used process model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development.

2.8 Evolutionary Software Process Model

There is growing recognition that software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

The linear sequential model is designed for straight line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

2.9 The Incremental Model

The incremental model combines elements of the linear sequential model with the iterative philosophy of prototyping. As figure 2.5 shows, the incremental model applies linear sequences in a staggered fashion as calendar time progress. Each linear sequence produces a deliverable “increment” of the software. For example word processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and

grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a core product. That is basic requirements are addressed, but many supplementary features, (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced. Early increments are “stripped down” versions of the final product but they do provide capability that serves the user and also provide a platform for evaluation by the user.

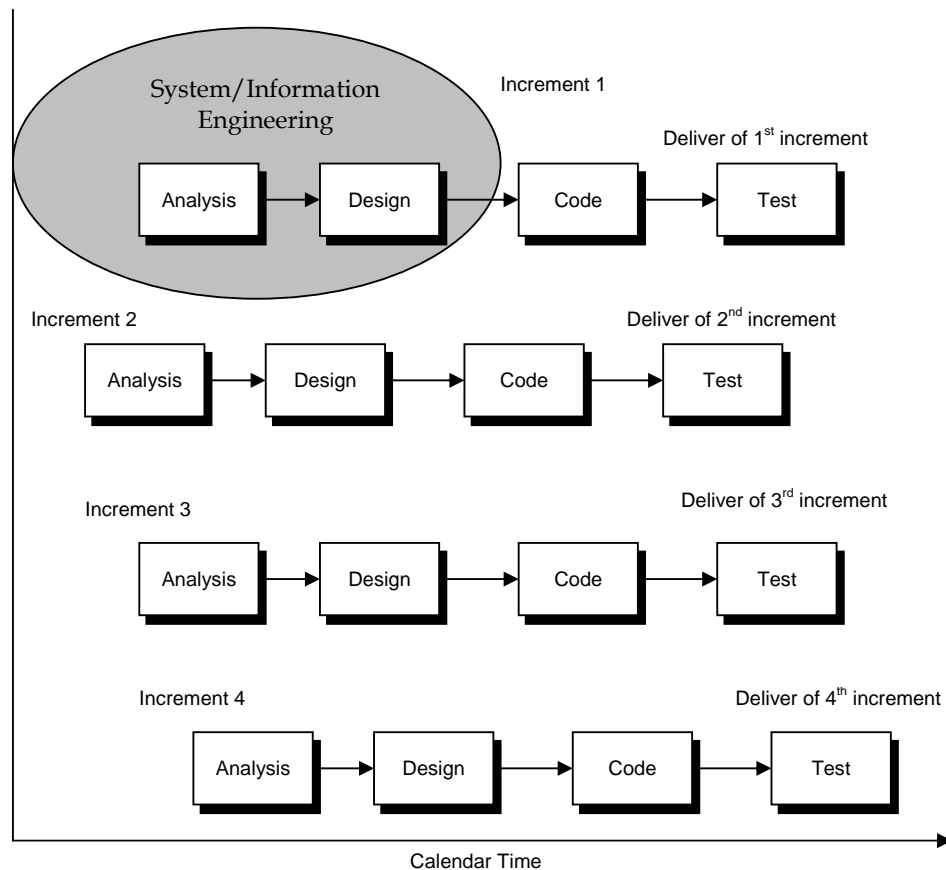


Figure 2.5 The incremental model

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

2.10 The Formal Methods Model

The formal methods model encompasses a set of activities that lead to mathematical specification of computer software. Formal methods enables a software engineer to specify, develop, and verify a computer based system by applying a rigorous, mathematical notation. A variation on this approach, called clean room software engineering, is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms, ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might otherwise go undetected.

Although not yet a mainstream approach, the formal methods model offers the promise of defect free software. Yet, concern about its applicability in a business environment has been voiced.

1. The development of formal models is currently quite time consuming and expensive .
2. Because few software developers have the necessary background to apply formal methods, extensive training is required.
3. It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, it is likely that the formal methods approach will gain adherents among software developers that must build safety critical software and among developers that would suffer severe economic hardship should software errors occur.

2.11 Short Summary

- Software engineering is a discipline that integrates process, methods, and tools for the development of computer software. A number of different process models for software engineering have been proposed, each exhibiting strengths and weaknesses, but all having a series of generic phases in common.
- The principles, concepts, and methods that enable us to perform the process that we call software engineering are considered throughout the remainder of this book.

2.12 Brain Storm

1. Explain Software Process
2. Write a note on Incremental Model
3. Write a note on Linear Sequential Model.



Lecture 3

Project Management Concept

Objectives

In this lecture you will learn the following

- ✧ About Management Spectrum
- ✧ About People
- ✧ About Problem
- ✧ About Process

Coverage Plan

Lecture 3
3.1 Snap Shot
3.2 The Management Spectrum
3.3 People
3.4 The Process
3.5 The Project
3.6 Short Summary
3.7 Brain Storm

3.1 Snap Shot

In this lecture we are going to learn about what is Management spectrum and people, the problem and the process in the Management spectrum.

3.2 The Management Spectrum

Effective software project management focuses on the three P's: people, problem, and process. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. Finally the manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum.

People

The cultivation of motivates, highly skilled software people has been discussed since the 1960s (e.g., [COU80, DeM87, WIT94]. In fact, the “people factor” is so important that the Software engineering Institute has developed a people management capability maturity model “ to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability”[CUR94].

The people management maturity model defines the following key practice areas for software people career development, selection performance management, training, compensation, career development, organization and work design and team culture development. Organization that achieve high levels of maturity in the people management area have a higher likelihood of implement effective software engineering practices.

The PM-CMM is a companion to the software capability maturity model which guides organizations in the creation of a mature software process. Issues associated with people management and structure for software projects are considered later in this lecture.

The problems

Before a project can be planned, its objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost; an effective assessment of risk; a realistic breakdown of project tasks; or a manageable project schedule that provides a meaningful indication of process.

The software developer and customer must meet to define project objectives and scope. In many cases, this activity begins as part of the system engineering process and continues as the first step in software requirements analysis. Objectives identify the overall goals of the project with out considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the problem, and more important attempts to bound these characteristics in a quantitative manner.

Once the project objectives and scope are understood, alternative solutions are considered. Although very little details is discussed, the alternatives enable managers and practitioners to select a “best” approach,

given the constraints imposed by delivery deadlines, budgetary restrictions personnel availability, technical interfaces, and myriad other factors.

The process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets tasks, milestones, deliverables, and quality assurance points enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities such as software quality assurance, software configuration management, and measurement overlay the process model. Umbrella activities are independent of any one-framework activity and occur throughout the process.

3.3 People

In a study published by the IEEE [CUR 88] the engineering vice presidents of three major technology companies were asked the most important contributor to a successful software project. They answered in the following way.

VP 1: I guess if you had to pick one thing out that is most important in our environment, I'd say it's not the tools that we use, it's the people.

VP 2: The most important ingredient that was successful on this project was having smart people. Very little else matters in my opinion. The most important thing you do for a project is selecting the staff. The success of the software development organization is very, very much associated with the ability to recruit good people.

VP3: The only rule I have in management is to ensure I have good people real good people and that I grow good people and that I provide an environment in which good people can produce.

Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue (as the group above had done) that people are primary, but their actions sometimes belie their words. In this section we examine the players who participate in the software process and the manner in which they are organized to perform effective engineering.

The Players

The software process (and every software project) is populated by players who can be categorized into one of five constituencies.

1. Senior managers, who define the business issues that often have significant influence on the project.
2. Project (technical) managers, who must plan, motivate, organize, and control the practitioners who do software work.
3. Practitioners, who deliver the technical skills that, are necessary to engineer a product or application.
4. Customers, who specify the requirements for the software to be engineered.
5. End users, who interact with the software once it is released for productions use.

Every software project is populated by the players noted above. To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. That's the job of the team leader.

What do we look for when we select someone to lead a software project? In an excellent book of technical leadership, Jerry Weinberg [WEI86] attempt to answer this question by suggesting the MOI Model leadership:

Motivation The ability to encourage (by "push or pull") technical people to produce to their best ability.

Organization The ability to mold existing processes(or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time letting everyone on the team know (by words, and far more important, by actions) that quality counts and that it will not be compromised.

Another view [EDG95] of the characteristics that define an effective project manager emphasizes four key traits.

Problem solving An effective software project manager can diagnose that technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement To optimize the productivity of a project team, a manager must reward initiative and accomplishment, and demonstrate through his own actions that controlled risk taking will not be punished.

Influence and Team Building An effective project manager must be able to read people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high stress situations.

The software team

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change is not within the software project manager's scope of responsibilities. However, the organization of the people directly involved in a new software project is within the project manager's purview.

The following options are available for applying human resources to a project that will require n people working for k years:

1. n individuals are assigned to m different functional tasks, relatively little combined work occurs coordination is the responsibility of a software manager who may have six other projects to be concerned with.
2. n individuals are assigned to m different functional tasks, ($m < n$) so that informal “teams” are established ;an ad hoc team leader may be appointed ;coordination among teams is the responsibility of a software manager;
3. n individuals are organized into t teams ;each team is assigned one or more functional tasks; each team has a specific structure that is defined for all teams working on a project coordination is controlled by both the team and a software project manager.

Although it is possible to voice pro and con arguments for each of the above approaches, there is a growing body of evidence that indicates that a formal team organization (option 3) is most productive.

The “best” team structure depends on the management style of an organization, the number of people who will populate the team and their skill levels and the overall problem difficulty. Mantei [MAN81] suggests three generic team organizations:

Democratic decentralized (DD). This software engineering team has no permanent leader. Rather, “ task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks.” Decisions on problems and approach are made by group consensus. Communication among team members is horizontal

Controlled decentralized (CD). This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtask. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical. Communication along the control hierarchy also occurs.

Controlled Centralized (CC). Top-level problem solving and internal team coordination are managed by a team leader. communication between the leader and team members is vertical.

Mantei also describes seven project factors that should be considered when planning the structure of software engineering teams:

- The difficulty of the problem to be solved
- The time the team will stay together (team lifetime)
- The degree to which the problem can be modularized
- The required quality and reliability of the system to be built
- The rigidity of the delivery date
- The degree of sociability (communication) required for the projects

Table 3.1 [MAN81] summarizes the impact of project characteristics on team organization. Because a centralized structure completes tasks faster, it is the most adept at handling simple problems. Decentralized teams generate more and better solutions than individuals. Therefore such teams have a greater probability of success when working on difficult problems. Since the CD team is centralized for

problem solving, either the CD or the CC team structure can be successfully applied to simple problems. A DD structure is best for difficult problems.

Because the performance of a team is inversely proportional to the amount of communication that must be conducted, very large projects are best addressed by teams with a CC or CD structure when sub grouping can be easily accommodated.

The impact of project characteristics on team structure [MAN81]

Team type:	DD	CD	CC
Difficulty			
High	x		
Low		x	x
Size			
Large		x	x
Small	x		
Team lifetime			
Short		x	x
Long	x		
Modularity			
High		x	x
Low	x		
Reliability			
High	x	x	
Low			x
Delivery date			
Strict			x
Lax	x	x	
Sociability			
High	x		
Low		x	x

Table 3.1 : The impact of project characteristics on team structure

The length of time the team will “live together” affects team morale. It has been found that DD team structures result in high morale and job satisfaction and are therefore good for long lifetime teams.

The DD team structure is best applied to problems with relatively low modularity because of the higher volume of communication that is needed. when high modularity is possible (and people can do their own thing) the CC or CD structure will work well.

CC and CD teams have been found to produce fewer defects than DD teams, but these data have much to do with the specific quality assurances activities that are applied by the team. Decentralized teams generally require more time to complete a project than a centralized structure and at the same time are best when high sociability is required.

Constantine [CON93] suggests four “organizational paradigms” for software engineering teams.

1. A closed paradigm structures a team along a traditional hierarchy of authority (similar to a CC team). Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. The random paradigm structures a team loosely and depends on individual initiative of the team members. When innovation or technological break through is required, teams following the random paradigm will excel. But such teams may\ struggle when “ orderly performance” is required.
3. The open paradigm attempts to structure a team in manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively with heavy communication and consensus based decision making. Open paradigm team structures are well suited to the solution of complex problems, but may not perform as efficiently as other teams.
4. The synchronous paradigm relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problems with little active communication among themselves.

As an historical footnote, the earliest software team organization was a controlled centralized (CD) structure originally called the chief programmer team. This structure was first proposed by Harlan Mills and described by Baker [BAK72]. The nucleus of the team is composed of a senior engineer (“the chief programmer”) who plans, co ordinates and reviews all technical activities of the team; technical staff (normally two to five people) who conduct analysis and development activities and a backup engineer who supports the senior engineer in project continuity.

The chief programmer may be served by one or more specialists (e.g. telecommunications expert, database designer),support staff(e.g., technical writers, clerical personal) and a *software librarian*. The librarian serves many teams and performs the following functions: maintains and controls all elements of the software configuration (i.e., documentation, source listings, data, magnetic media); helps collect and format software productivity data; catalogs and indexes reusable software modules; and assists the teams in research, evolution, and document preparation. The importance of a librarian cannot be overemphasized. The librarian acts as a controller , coordinator and potentially, an evaluator of the software configuration. Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book, *peopleware*, DeMarco and Lister [DeM87] discuss this issue:

We tend to use the word *team* fairly loosely in the business world, calling any group of people assigned to work together a “team”. But many of these groups just don’t seem like teams. They don’t have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell*.

A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts...

Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success... they don’t need to be managed in the traditional way, and they certainly don’t need to be motivated. They’ve got *momentum*.

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a “sense of eliteness” that makes them unique.

Coordination and Communication Issues

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. *Uncertainty* is common, resulting in a continuing stream of changes that ratchets the project team. *Interoperability* has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software-scale, uncertainty, and interoperability-are facts of life. To deal with them effectively, a software engineering team must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through “writing, structured meetings, and other relatively non-interactive and impersonal communication channels”. Informal communication is more personal. Members of a software engineering team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another daily.

Kraul and Streeter examine a collection of project coordination techniques that are categorized in the following manner.

Formal, impersonal approaches Include software engineering documents and deliverables technical memos, project milestones, schedules and project control tools changes requests and related documentation error tracking reports, and repository data.

Formal, interpersonal procedures Focus on quality assurance activities applied to software engineering work products. These include status review meetings and design and code inspections.

Informal, interpersonal procedures Include group meetings for information dissemination and problem solving and “collocation of requirements and developments staff”.

Electronic communication Encompasses electronic mail, electronic bulletin boards, Web sites, and by extension, video-based conferencing systems.

Interpersonal network Informal discussion with those outside the project who may have experience or insight that can assist team members.

3.4 The Problem

A software project manager is confronted with a dilemma at the very beginning of a software engineering project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or months to complete. Worse requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed “now”.

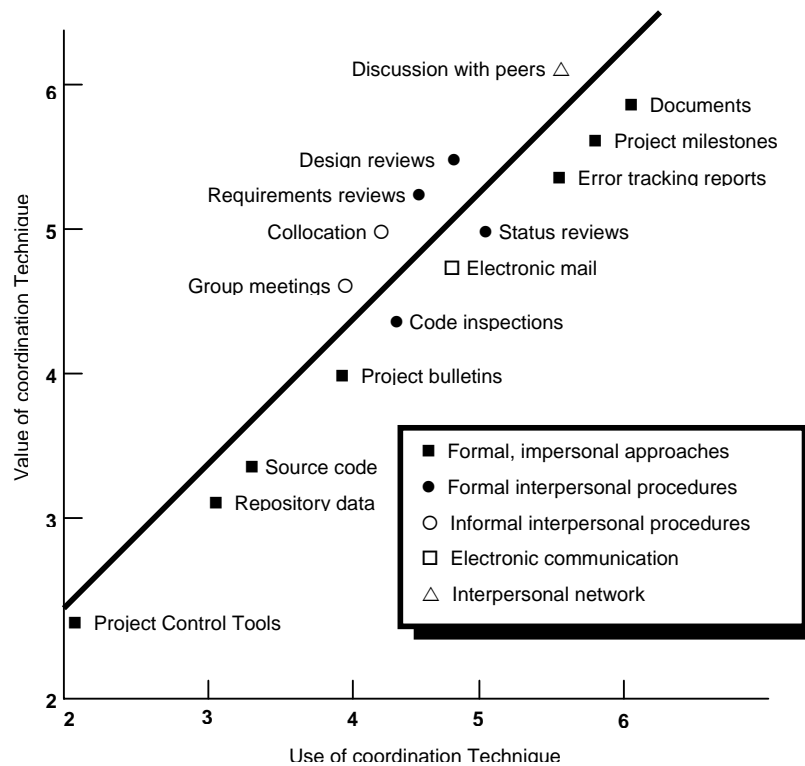
Therefore we must examine the problem at the very beginning of the project. At a minimum, the scope of the problem must be established and bounded.

Software Scope

The first software project management activity is the determination of software scope. Scope is defined by answering the following questions;

Context : how does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?

Information objectives: What customer visible data objects are produced as output from the software? what data objects are required for input?



Function and performance : What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Figure 3.1 Value and use of coordination and communication techniques

Software project scope must be unambiguous and understandable at management and technical levels. A statement of software scope must be bonded. That is, quantitative data (e.g., number of simultaneous users, size of mailing list, maximum allowable response time) are stated explicitly; constraints and/or limitations (e.g. product cost restricts memory size) are noted, and mitigating factors are described.

Problem Decomposition

Problem decomposition, sometimes called partitioning, is an activity that sits at core of software requirements analysis (later chapters). During the scoping activity there is no attempt to fully decompose the problem. Rather decomposition is applied in two major areas : 1. the functionality that must be delivered and (2) the process that will be used to deliver it.

Human beings tend to apply a divide and conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that is applied as project planning begins. Software functions, described

in the statement of scope, are evaluated and regained to provide more detail prior to the beginning of estimation. Because both cost and schedule estimated are functionally oriented, some degree of decomposition is often useful.

The Process

The generic phases that characterize the software process definition, development, and maintenance are applicable to all software. The problem is to select the process model that is appropriate for the software to be engineered by a project team.

- The linear sequential model
- The prototyping model
- The RAD model
- The incremental model
- The spiral model
- The component development model
- The formal methods model
- The fourth generation techniques model

The project manager must decide which process model is most appropriate for the project, then define a preliminary plan based on the set of common process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan reflecting the work tasks required to populate the framework activities must be created. We explore these activities briefly in the sections that follow and present a more detailed view in later chapters.

Melding the Problem and the Process

project planning begins with the melding of the problem and the process. Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization. Assume that the organization has adopted the following set of framework activities.

- Customer communication – tasks required to establish effective communication between developer and customer
- Planning – tasks required to define resources, timelines, and other project related information.
- Risk analysis – tasks required to assess both technical and management risk
- Engineering – tasks required to build one or more representations of the application
- Construction and release – tasks required to construct, test, install, and provide user support.
- Customer evaluation – tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

The team members who work on each function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in figure 3.2 is created. Each major problem function is listed in the left hand column. Framework activities are listed in the top row. Software engineering work tasks would be entered in the following row. The job of the project manager is to estimate resource requirements for each matrix, cell start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each cell, these issues are considered in later chapters.

COMMON PROCESS FRAMEWORK ACTIVITIES	Customer Communication				Planning				Risk Analysis				Engineering			
Software Engineering Tasks																
Product Functions																
Text Input																
Editing and Formatting																
Automatic copy edit																
Page Layout Capability																
Automatic Indexing and TOC																
File Management																
Document Production																

Figure 3.2 Process Decomposition

Process Decomposition

A software team should have a significant degree of flexibility in choosing the software engineering paradigm that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If very tight time constraints are imposed and the problem can be heavily compartmentalized, the RAD model is probably the right option. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics will lead to the selection of other process models.

Once the process model has been chosen, the common process framework is adapted to it. In every case, the CPF discussed earlier in this chapter customer communication, planning, risk analysis, engineering, construction and release, customer evaluation can be fitted to the paradigm. It will work for linear models, for iterative and incremental models, for evolution models, and even for concurrent or component assembly models. The CPF is invariant and serves as the basis for all software work performed by a software organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks: "how do we accomplish the CPF activity?". For example a small, relatively simple project might require the following work tasks for the customer communication activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope
4. Review the state of scope with all concerned
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

Now , we consider a more complete project that has a broader scope and more significant business impact. Such a project might require the following work tasks for the customer communication activity.

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer .
3. Conduct research to define proposed solutions and existing approaches.
4. Prepare a “working document” and an agenda for the formal meeting
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software.
7. Review each mini spec for correctness, consistency, and lack of ambiguity.
8. Assemble the mini specs into a scooping document.
9. Review the scooping document with all concerned
10. Modify the scooping document as required.

Both projects perform the frame work activity that we call customer communication, but the first project team performs half as many software engineering work tasks as the second.

3.5 The Project

Jaded industry professionals often refer to 90-90 rule when discussing particularly difficult software projects: the first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes the other 90 percent of the allotted effort and time. This statement tells us much about the state of a project that gets into trouble:

The manner in which progress is assessed is flawed. (Obviously, if the 90-90 rule is true, 90 percent complete is not an accurate indicator).

There is no way to calibrate progress because quantitative metrics are unavailable. The project plan has not been designed to accommodate resources required at the end of a project.

Risks have not been considered explicitly, and a plan for mitigating, monitoring, and managing them has not been created. The schedule is (1) unrealistic or (2) flawed.

To overcome these problems, time must be spent at the beginning of a project to establish a realistic plan, during the project to monitor the plan, and throughout the project to control quality and change.

3.6 Short Summary

- Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues through out the definition, development, and maintenance of computer software.
- There P's have a substantial influence on software project management people, problem and process.
- People must be organized into effective teams, motivated to do high quality software work, and coordinated to achieve effective communication.
- The problem must be communicated from customer to developer, partitioned into its constituent arts, and positioned for work by the software team.

- The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done.
- The pivotal element in all software projects is people, software engineers can be organized in a number of different team structures that range from traditional control hierarchies to “open paradigm” teams. A variety of coordination and communication techniques can be applied to support the work of the team.. In general, formal reviews and informal person to person communication have the most value for practitioners.

3.7 Brain Storm

1. What is Management Spectrum?
2. Explain briefly about Co-ordination and communication Issues?
3. Define Software Scope?
4. Discuss briefly about Problem Decomposition?
5. Write a Note on Project Decomposition

❧

Lecture 4

Software Project Planning

Objectives

In this lecture you will learn the following

- ✍ About Software Scope
- ✍ About Software Resources
- ✍ Software Project Estimations

Coverage Plan

Lecture 4

- 4.1 Snap Shot
- 4.2 Observations on Estimating
- 4.3 Project Planning Objectives
- 4.4 Software Scope
- 4.5 Obtaining Information Necessary for Scope
- 4.6 Resources
- 4.7 Human Resources
- 4.8 Reusable Software Resources
- 4.9 Environmental Resources
- 4.10 Short Summary
- 4.11 Brain Storm

4.1 Snap Shot

The software project management process begins with a set of activities that are collectively called project planning. The first of these activities is estimation. Whenever estimates are made, we look into the future and accept some degree of uncertainty as a matter of course.

Although estimating is as much art as it is science, this important activity need not be conducted in a haphazard manner. Useful techniques for time and effort estimation do exist. And because estimation lays a foundation for all other project planning activities, and project planning provides the road map for successful software engineering, we would be ill advised to embark without it.

4.2 Observations on Estimating

A leading executive was once asked what single characteristic was most important when selecting a project manager. His response “a person with the ability to know what will go wrong before it actually does”. We might add “and the courage to estimate when the future is cloudy.”

Estimation of resources, cost, and schedule for software development of course requires experience, access to good historical information, and the courage to commit to quantitative measures when qualitative data are all that exist. Estimation carries inherent risk and it is this risk that leads to uncertainty.

Project complexity has a strong effect on uncertainty that is inherent in planning. Complexity, however, is relative measure that is affected by familiarity with past effort. A real time application might be perceived as “exceedingly complex” to a software group that has previously developed only batch applications. The same real time application might be perceived as “run-of-the-mill” for a software group that has been heavily involved in high speed process control. A number of quantitative software complexity measures have been proposed. Such measures are applied at the design or code level and are therefore difficult to use during software planning (before a design and code exist). However other, more subjective assessments of complexity can be established early in the planning process.

Project size is another important factor that can affect the accuracy of estimates. As size increase, the interdependency among various elements of the software grows rapidly. Problem decomposition, an important approach to estimating, becomes more difficult because decomposed elements may still be formidable. To paraphrase Murphy’s law: What can go wrong will go wrong”-and if there are more things that can fail, more things will fail.

The degree of *structural uncertainty* also has an effect on estimation risk. In this context, structure refers to the degree to which requirements have been solidified, the ease with which function can be compartmentalized, and the hierarchical nature of information that must be processed.

The availability of historical information also determines estimation risk. Santayana once said, “Those who cannot remember that past are condemned to repeat it.” By looking back we can emulate things that worked and avoid areas where problem arose. When comprehensive software metrics are available for past projects, estimates can be made with greater assurance; schedules can be established to avoid past difficulties, and overall risk is reduced.

Risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and risk become dangerously high. The software planner should demand completeness of function, performance, and interface definitions (contained in a system specification). The planner, and more important the customer, should recognize that variability in software requirements means instability in cost and schedule.

A project manager should not become obsessive about estimation. Modern software engineering approaches (e.g., evolutionary process models) take an iterative view of development. In such approaches, it is possible to revisit the estimate (as more information is known) and revise it when the customer makes changes to requirements

4.3 Project Planning Objectives

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimate should attempt to define “best case” and “worst case” scenarios so that project outcomes can be bounded.

The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

4.4 Software Scope

The first activity in software project planning is the determination of software scope. Function and performance allocated to software during system engineering should be assessed to establish a project scope that is unambiguous and understandable at management and technical levels.

Software scope describes function, performance, constraints, interfaces ,and reliability. Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. *Performance* considerations encompass processing and response time requirements. *Constraints* identify limits placed on the software by external hardware, available memory or other existing systems.

4.5 Obtaining Information Necessary for Scope

Things are always somewhat hazy at the beginning of a software project. A need has been defined and basic goals and objectives have been enunciated, but the information necessary to define scope (a prerequisite for estimation) has not yet been defined.

The most commonly used technique to bridge the communication gap between the customer and developer and to get the communication process started is to conduct a preliminary meeting or interview. The first meeting between a software engineer(the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask both are worried that what they do say will be misinterpreted; both are worried that what they do say will be

misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here) both want to get the thing over with but at the same time, both want it to be a success.

Yet communication must be initiated. Gause and Weinberg suggest that the analyst start by asking *context free questions*. That, is set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself.

The first set of context free questions focus on the customer, the overall goals, and the benefits. For example, the analyst might ask:

- ◆ Who is behind the request for this work?
- ◆ Who will use the solution?
- ◆ What will be the economic benefit of a successful solution?
- ◆ Is there another source for the solution?

The next set of questions enable the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution.

- How would you [the customer] characterize “good” output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me(or describe) the environment in which the solution will be used?
- Are there special performance issues or constraints that will affect the way the solution is approached?

The final set of questions focus on the effectiveness of the meeting. Gause and Weinberg call there “meta-questions” and propose the following (abbreviated)list:

- Are you the right person to answer these questions? Are your answers “official”?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Is there anyone else who can provide additional information?
- Is there anything else that I should be asking you?

These questions (and others) will help to “break the ice” and initiate the communication that is essential to establish the scope of the project. But a question and answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then be replaced by a meeting format that combines elements of problem solving, negotiation and specification.

A number of independent investigators have developed a team oriented approach to requirements gathering that can be applied to help establish the scope of a project. Called *facilitated application*

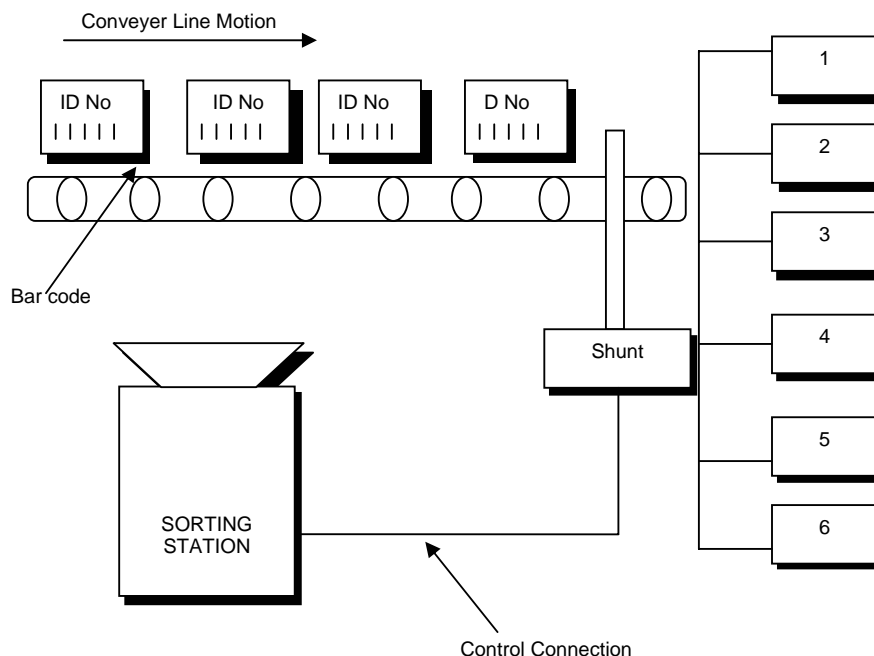
specification techniques (FAST), this approach encourages the creation of a joint team of customer and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of requirements.

A scooping Example

Communication with the customer leads to a definition of the data, functions, and behavior that must be implemented, the performance and constraints that bound the system and related information. As an example, consider software that must be developed to drive a *conveyor line sorting system*. The statement of scope for the CLSS follows.

The conveyor line sorting system (CLSS) sorts boxes moving along a conveyor line. Each box is identified by a bar code that contains a part number and is sorted into one of six bins at the end of the line. The boxes pass by a sorting station that contains a bar code reader and a PC. The sorting station PC is connected to a shunting mechanism that sorts the boxes into the bins. Boxes pass in random order and are evenly spaced. The line is moving at five feet per minute. A CLSS is depicted schematically in Figure 4.1.

CLSS software receives input information from a bar code reader at time intervals that conform to the conveyor line speed. Bar code data will be decoded into box identification format. The software will do a look-up in a part number data base containing a maximum of 1000 entries to determine proper bin location for the box currently at the reader (sorting station) . The proper bin location is passed to a sorting shunt that will position boxes in the appropriate bin. A record of the bin destination for each box will be maintained for later recovery and reporting. CLSS software will also receive input from a pulse tachometer that will be used to synchronize the control signal to the shunting mechanism. Based on the number of pulses that will be generated between the sorting station and the shunt, the software will produce a control signal to the shunt to properly position the box.



The project planner examines the statement of scope and extracts all important software functions. This process, called *decomposition*, is was discussed in the previous lecture and results in the following functions.

Figure 4.1 A Conveyer Line Sorting System

- Read bar code input
- Read pulse tachometer
- Decode part code data
- Do database look-up
- Determine bin location
- Produce control signal for shunt
- Maintain record of box destinations'

In this case, performance is dictated by conveyer line speed. Processing for each box must be completed before the next box arrives at the bar code reader. The CLSS software is constrained by the hardware it must access (the bar code reader, the shunt, the PC), the available memory, and the overall conveyer line configuration (evenly spaced boxes)

Function performance, and constraints must be evaluated together. The same function can precipitate and order of magnitude difference in development effort when considered in the context of different performance bounds. The effort and cost required to develop CLSS software would be dramatically it function remains the same but performance varies. For instance, if conveyor line average speed increase by a factor of 10 (performance) and boxes are no longer spaced evenly (a constraint) software would become considerably more complex and thereby require more effort. Function, performance, and constraint are intimately connected.

Software interacts with other elements of a computer based system. The planner considers the nature and complexity of each interface to determine any affect on development respire, cost, and schedule. The concept of an interface is interpreted to mean(1) hardware (e.g. machines, displays) that are indirectly controlled by the software(2) software that already exists (e.g. database access routines, reusable software components, operating system) and must be linked to the new software; (3) people who make use of the software via keyboard or other I/O devices; and (4) procedures that precede or succeed the software as a sequential series of operations. In each case the information transfer across the interface must be clearly understood.

The least precise aspect of software scope is a discussion of reliability. Software reliability measures do exist but they are rarely used at this stage of a project. Classic hardware reliability characteristics like meantime-between failure (MTBF) can be difficult to translate to the software domain. However, the general nature of the software may dictate special considerations to ensure “reliability”. For example, software for an are traffic control system or the Space Shuttle (both human-rated systems) must not fail or human life may be lost. An inventory control system or word-processing software should not fail, but the impact of failure is considerably less dramatic. Although it may not be possible to quantify software reliability as precisely as we would like in the statement of scope, we can use the nature of the project to aid in formulating estimates of effort and cost to assure reliability.

If a system specification has been properly developed, nearly all information required for a description of software scope is available and documented before software project planning begins. In cases where a specification has not been developed, the planner must take on the role of system analyst to determine attributes and bounds that will influence estimation tasks.

4.6 Resources

The second task of software planning is estimation of resources required to accomplish the software development effort. Figure 4.2 illustrates development resources as a pyramid. The *development environment*-hardware and software tools-sits at the foundation of the resources pyramid and provides the infrastructure to support the development effort. At a higher level we encounter *reusable software components* software building blocks that can dramatically reduce development costs and accelerate delivery. At the top of the pyramid is the primary resource-people. Each resources is specified with four characteristics; description of the resources, a statement of availability, chronological time that the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a *time window*. Availability of the resource of a specified window must be established at the earliest practical time.

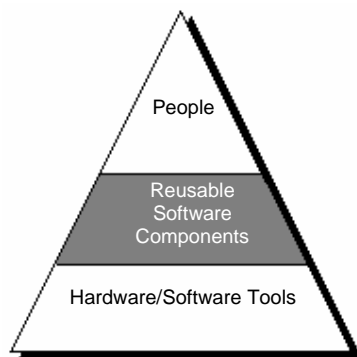


Figure 4.2 Resources

4.7 Human Resources

The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g manager, senior software engineer, etc) and specialty (e.g telecommunications, database, client/server) are specified. For relatively small projects (six person-months or less) a single individual may perform all software engineering steps, consulting with specialists as required.

The number of people required for a software project can be determined only after an estimate of development effort (e.g person-months or person years) is made. Techniques for estimating effort are discussed later in the chapter.

4.8 Reusable Software Resources

Any discussion of the software resource would be incomplete without recognition of *reusability* that is, the creation and reuse of software building blocks[HOO91]. Such building blocks must be catalogued for easy reference, standardized for easy application, and validated for easy integration.

Bennatan [BEN92] suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components Existing software that can be acquired from a third party or that has been developed internally for a past project. These components are ready for use on the current project and have been fully validated.

Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full experience components will be relatively low-risk.

Partial-experience components. Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project, but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications, required for partial experience components have a fair degree of risk.

New Components Software components that must be built by the software team specifically for the needs of the current project.

The following guidelines should be considered by the software planner when reusable components are specified as a resource.

1. If off-the-shelf components meet project requirements, acquire them. The cost for acquisition and integration of off-the-shelf components will almost always be less than the cost to develop equivalent software. In addition, risk is relatively low.
2. If full experience components are available the risks associated with modification and integration are generally acceptable. The project plan should reflect the use of these components.
3. If partial-experience components are available, their use for the current project must be analyzed in detail. If extensive modification is required before the components can be properly integrated with other elements of the software, proceed carefully. The cost to modify partial experience components can sometimes be greater than the cost to develop new components.

Ironically, the use of reusable software components is often neglected during planning, only to become a paramount concern during the development phase of the software process. It is far better to specify software resource requirements early. In this way technical evaluation of alternatives can be conducted and timely acquisition can occur.

4.9 Environmental Resources

The environment that supports the software project, often called a software engineering environment, incorporates hardware and software. Hardware provides a platform that supports the tools (software)

required to produce the work products that are an outcome of good software engineering practice. Because most software organizations have multiple constituencies that require access to the SEE, a project planner must prescribe the time window required for hardware and software and verify that these resources will be available.

When a computer based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements begin developed by other engineering teams. For example, software for a numerical control (NC) used on a class of machine tools may require a specific machine tool(e,g a NC lathe) as part of the validation test step; a software project for automated typesetting may need a photo-typesetter at some point during development. Each hardware element must be specified by the software project planner.

4.10 Short Summary

- The software project planner must estimate three things before a project begins: how long it will take, how much effort will be required and how many people will be involved.
- The planner must predict the resources (hardware and software) that will be required and the risk involved.

4.11 Brain storm

1. How do you obtain necessary informatin for scope ?
2. Short Note on Resources ?
3. Explain about Project Planning Objectives ?
4. Describe briefly about Reusable Software Resource ?
5. What is Environmental Resource ?

END

Lecture 5

Software Project Planning - II

Objectives

In this lecture you will learn the following

- ✧ About decomposition Techniques
- ✧ About empirical estimation models
- ✧ About automated estimation tools

Coverage Plan

Lecture 5
5.1 Software Project Estimation
5.2 Decomposition Techniques.
5.3 Problem Based Estimations
5.4 An example of LOC Based ESTIMATION
5.5 Short Summary
5.6 Brain Storm

5.1 Snap Shot

In the early days of computing, software costs comprised a small percentage of overall computer based system cost. An order of magnitude error in estimates of software cost had relatively little impact. Today, software is the most expensive elements in most computer-based systems. A large cost estimation error can make the difference between profit and loss. Cost overrun can be disastrous for the developer.

Software cost and effort estimation will never be an exact science. Too many variables human Technical, Environmental, Political can affect the ultimate cost of software and effort applied to develop it. However software project estimation can be transformed from a mysterious art to a series of systematic steps that provide estimates with acceptable risk.

To achieve reliable cost and effort estimates, a number of options arise.

1. Delay estimation until late in the project (obviously, we can achieve 100% accurate estimates after the project is completed)
2. Base estimates on similar projects that have already been completed.
3. use relatively simple “decomposition techniques” to generate project cost and effort estimates.
4. use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided “up-front”. However, we should recognize that the longer we wait, the more we know, and the more we know, the less likely we are to make serious errors in our estimates.

The second option can work reasonably well if the current project is quite similar to past efforts and other project influences (e.g the customer, business conditions, the SEE deadlines) are equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem, each used as a cross-check for the other. *Decomposition techniques* take a “divide and conquer” approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form.

$$d=f(V_i)$$

Where d is one of a number of estimated values (e.g. effort, cost, project duration) and V_i are selected independent parameters (e.g estimated LOC or FP)

Automated estimation tools implements one or more decomposition techniques or empirical models. When combined with an interactive human machine interface, automated tools provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data.

Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation.

5.2 Decomposition Techniques

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e, developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason we decompose the problem, recharacterizing it as a set of smaller (and hopefully more manageable) problems.

Software Sizing

The accuracy of a software project estimate is predicated on a number of things. (1) the degree to which the planner has properly estimated the size of the product to be built. (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects) (3) the degree to which the project plan reflects the abilities of the software team; and (4) the stability of product requirements and the environment that supports the software engineering effort.

In this section, we consider the software sizing problem. Because a project estimate is only as good as the estimate of the size of the work to be accomplished, sizing represents the project planner's first major challenge. In the context of project planning, size refers to a quantifiable outcome of the software text of project planning. Size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in LOC. If an indirect approach is chosen, size is represented as FP.

Putnam and Myers[PUT92] suggest four different approaches to the sizing problem:

"Fuzzy-Logic" Sizing

This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range. Although personal experience can be used, the planner should also have access to and historical database of projects so that estimates can be compared to actual experience.

Function Point Sizing

The planner develops estimates of the information domain characteristics discussed in the previous lecture

Standard Component Sizing

Software is composed of a number of different "standard component" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object level instruction. The project planner estimates the number of occurrence of each standard component and then uses historical project data to determine the delivered size per standard component. To illustrate, consider an information systems application, the planner estimates that 18 reports will be generated. Historical data indicates that 967 lines of Cobol are required per report. This enables the planner to estimate that 17,000 LOC will be required for the reports component. Similar estimates and calculations are made for other standard components, and a combined size value results.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished. Using the “effort ratio” for each type of change, the size of the change may be estimated.

Putnam and Myers suggest that the results of each of the sizing approaches noted above be combined statistically to create a three point or expected value estimate. This is accomplished by developing optimistic (low), most likely, and pessimistic (high) values for size and combining them using equations described in the next section.

5.3 Problem Based Estimations

Lines of code (LOC) and function points (FP) were described as basic measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as an estimation variable that is used to “size” each element of the software, and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques, yet both have a number of characteristics in common. The project planner begins with a bounded statement of software scope and from this statement attempts to decompose software into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, the planner may choose another component for sizing such as classes or objects, changes, or business processes impacted. Baseline productivity metrics (e.g., LOC/pm or FP/pm) are then applied to the appropriate estimation variable and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project.

It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely that reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics – inputs, outputs, data files, inquiries, and external interfaces – and the fourteen complexity adjustment values are estimated. The resultant estimates are used to derive a FP value that can be tied to past data and used to generate and estimate.

Regardless of the estimation variable that is used, the project planner begins by estimating a range of values for each function or information domain value. Using historical data or (when all else fail) intuition, the planner estimates an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided

when a range of values is specified.

A three point or expected value is then computer. The expected value for the estimation variable (size) EV, can be computed as a weighted average of the optimistic(s_{opt}), most likely(S_m) and pessimistic(p_{pess}) estimates. For example,

$$EV = (s_{opt} + 4s_m + s_{pess}) / 6 \text{ -----(5.1)}$$

Gives heaviest credence to the “most likely” estimate and follows a beta probability distribution.

We assume that there is a very small probability that the actual size result will fall outside the optimistic or pessimistic values. Using standard statistical techniques, we can compute the deviation of the estimates. However, it should be noted that a deviation based on uncertain (estimated) data must be used judiciously.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only reasonable answer to this question is : “we can’t be sure”. Any estimation technique no matter how sophisticated, must be cross checked with another approach. Even then, common sense and experience must prevail.

5.4 An Example of LOC Based Estimation

As an example of LOC and FP estimation techniques, let us consider a software package to be developed for a computer-aided design (CAD) application for mechanical components. A review of the system specification indicates that the software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high-resolution color display, and laser printer.

Using a system specification as a guide, a preliminary statement of software scope can be developed:

The CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human machine interface design. All geometric data and other supporting information will be maintained in a CAD database Design analysis modules will be developed to produce required output which will be displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, and laser printer.

The above statement of scope is preliminary—it is not bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin the planner must determine what “characteristics of good human-machine interface design” means or what the size and sophistication of the “CAD database” is to be.

For our purposes, we assume that further refinement has occurred and that the following major software functions are identified:

- User interface and control facilities (UICF)
- Two-dimensional geometric analysis (2DGA)
- Three-dimensional geometric analysis (3DGA)
- Database management (DBM)
- Computer graphics display facilities (CGDF)

- Peripheral control (PC)
- Design analysis modules(DAM)

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control (PC)	2,100
Design analysis modules (DAM)	8,400
<i>Estimated lines of code</i>	33,200

Figure 5.1 Estimation table of LOC method

Following the three-point estimation technique for LOC the table shown in Figure 5.1 is developed. For example, the range of LOC estimates for the 3D geometric analysis function is:

Optimistic : 4600
Most likely: 6900
Pessimistic: 8600

Applying equation (5.1), the expected value for the 3D geometric analysis function is 6800 LOC. This number is entered in the table. Other estimates are derived; in a similar fashion. By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system.

A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of \$8000 per month the cost per line of code is approximately \$13.00. Based on the LOC estimate and the historical productivity data, the total estimated project cost is \$431,000 and the estimated effort is 54 person-months.

5.5 Short Summary

- The statement of scope helps the planner to develop estimates using one or more techniques that fall into two broad categories: decomposition and empirical modeling.
- Decomposition Techniques require a delineation of major software functions, followed by estimates of either the size or the number of person months required to implement each function.
- Empirical techniques use empirically derived expressions for effort and time to predict these project quantities. Automated tools can be used to implement a specific empirical model. .
- Accurate project estimates generally make use of at least two of the three techniques noted above. By comparing and reconciling estimates derived using different techniques, the planner is more likely to derive an accurate estimate. Software project estimation can never be an exact science, but a combination of good historical data and systematic techniques can improve estimation accuracy.

5.6 Brain Storm

1. What is Software Project Estimation ?
2. Give a Short Note on Decomposition Technique ?
3. Explain briefly about Problem Based Estimation ?

❧❧❧

Lecture 6

Risk Management

Objectives

In this lecture you will learn the following

- ✧ About Reactive Vs Proactive Risk Strategies
- ✧ About Software Risk
- ✧ About Risk Identification
- ✧ About Risk Projections

Coverage Plan

Lecture 6

- 6.1 Snap Shot
- 6.2 Reactive Vs. Proactive Risk Strategies
- 6.3 Software Risks
- 6.4 Risk Identification
- 6.5 Risk Projection
- 6.6 Risk Mitigation, Monitoring and Management
- 6.7 Safety Risks and Hazards
- 6.8 Short Summary
- 6.9 Brain Strom

6.1 Snap Shot

In this lecture, we focus on Reactive and Proactive Risk Strategies, Software Risks, Risk Identification and Projection, Safety Risks and Hazards and also about RMMM.

6.2 Reactive Vs. Proactive Risk Strategies

Reactive risk strategies have been laughingly called the “Indiana Jones school of risk management”. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, “Don’t worry, I’ll think of something!” never worrying about problems until they happened, Indy would react in some heroic way.

Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called “fire fighting mode”. When this fails, “crisis management” takes over and the project is in real jeopardy.

A considerably more intelligent strategy for risk management is to be proactive. A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are prioritized by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner. Throughout the remainder of this chapter, discuss a proactive strategy for risk management.

6.3 Software Risks

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics

- **Uncertainty** The event that characterizes the risk may or may not happen; i.e. there are no 100% probable risks.
- **Loss** If the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel, resource, customer, and requirements problems and their impact on a software project. Project complexity, size, and the degree of structural uncertainty were also defined as project risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interfacing, verification, and maintenance problems. In addition, specification

ambiguity, technical uncertainty, technical obsolescence, and “leading-edge” technology are also risk factors. Technical risks occur because the problem is harder to solve than we thought it would be.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one really wants ; (2) building a product that no longer fits into the overall business strategy for the company ; (3) building a product that the sales force doesn’t understand how to sell; (4) losing the support of senior management due to a change in focus or a change in people and (5) losing budgetary or personnel commitment. It is extremely important to note the simple categorization won’t always work. Some risks are simply unpredictable in advance.

Another general categorization of risks has been proposed by Charette [CHA 89]. Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment). Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

6.4 Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan. By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories that are generic risks and product-specific risks. Generic risks are a potential threat to every software project. Product-specific risks can only be identified by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed: “ what special characteristics of this product may threaten our project plan?”

Both generic and product-specific risks should be identified systematically. Tom Glib drives this point home when he states: “if you don’t actively attack the risks they will actively attack you”.

One method for identifying risks is to create a risk item checklist. The Checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories.

- **Product size** – risks associated with the overall size of the software to be built or modified
- **Business impact** – risk associated with constraints imposed by management or the marketplace
- **Customer characteristics** – risks associated with the sophistication of the customer and the developer’s ability to communicate with the customer in a timely manner.
- **Process definition**- risks associated with the degree to which the software process has been defined and is followed by the development organization.

- **Development environment** – risks associated with the availability and quality of the tools to be used to build the product.
- **Technology to be built** – risks associated with the complexity of the system to be built and the “newness” of the technology that is packaged by the system.
- **Staff size and experience** – risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics noted above can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of “risk components and drivers” are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

Product size Risks

Few experienced managers would debate the following statement: Project risk is directly proportional to product size. The following risk item checklist identifies generic risks associated with product size:

- Estimated size of the product in LOC or FP?
- Degree of confidence in estimated size estimate?
- Estimated size of product in number of programs, files, and transactions?
- Percentage deviation in size of product from average for previous products?
- Size of database created or used by the product?
- Number of users of the product?
- Number of projected changes to the requirements for the product? Before delivery? After delivery?
- Amount of reused software?

In each case, the information for the product to be developed must be compared to past experience. If a large percentage deviation occurs or if numbers are similar, but past results were considerably less than satisfactory, risk is high.

Business Impact Risks

An engineering manager at a major software company placed the following framed plaque on his wall: “god grant me brains to be a good project manager and the common sense to run like hell whenever marketing sets project deadlines!”. The marketing department is driven by business considerations, and business considerations sometimes come into direct conflict with technical realities. The following risk item checklist identifies generic risks associated with business impact:

- Effect of this product on company revenue?
- Visibility of this product to senior management?
- Reasonableness of delivery deadline?
- Number of customers who will use this product and the consistency of their needs relative to the product?

- Number of other products/systems with which this product must be interoperable?
- Sophistication of end users?
- Amount and quality of product documentation that must be produced and delivered to the customer?
- Governmental constraints on the construction of the product?
- Costs associated with late delivery?
- Costs associated with a defective product?

Each response for the product to be developed must be compared to past experience. If a large percentage deviation occurs or if numbers are similar, but past results were considerably less than satisfactory, risk is high.

Customer Related Risks

All customers are not created equal. Pressman and Herron discuss this issue when they state.

Customers have different needs. Some know what they want ; others know what they don't want. Some know what they want others know what they don't want. Some customers are willing to sweat the details, while others are satisfied with a vague promises.

Customers have different personalities. Some enjoy being customers-the tension, the negotiation, the psychological rewards of a good product. Others would prefer not to be customers at all. Some will happily accept almost any thing that is delivered and make the very best of a poor product. Others will complain bitterly when quality is lacking, some will show their appreciation when quality is good a few will complain no matter what.

Customers also have varied associations with their suppliers. Some know the product and producer well. Others may be faceless, communicating with the producer only by written correspondence and a few hurried telephone calls.

Customers are often contradictory. They want everything yesterday for free. Often, the producer is caught among the customer's own contradictions.

A bad customer can have a profound impact on a software team's ability to complete a project on time and within budget. A bad customer represents a significant threat to the project plan and a substantial risk for the project manager. The following risk item checklist identifies generic risk associated with different customers.

- Have you worked with the customer in the past ?
- Does the customer have a solid idea of what is required? Has the customer spent the time to write it down?
- Will the customer agree to spend time in formal requirements gathering meetings to identify project scope?
- Is the customer willing to participate in reviews?
- Is the customer technically sophisticated in the product area?
- Is the customer willing to let your people do their job that is, will the customer resist looking over your shoulder during technically detailed work?
- Does the customer understand the software process?

If the answer to any of these questions is “no” further investigation should be undertaken to assess risk potential.

Process Risks.

If the software process is ill defined if analysis design, and testing are conducted in an ad hoc fashion if quality is a concept that everyone agrees is important, but no one acts to achieve in any tangible way, then the project is at risk. The following questions are extracted from a workshop on the assessment of software engineering practice developed by R.S. Pressman & Associates, Inc [PRE 95]. The questions themselves have been adapted from the Software engineering Institute software process assessment questionnaire.

- Does your senior management support a written policy statement that emphasizes the importance of standard process for software development.
- Has your organization developed a written description of the software process to be used on this project.
- Are staff members signed up to the software process as it is documented and willing to use it.
- Is the software process used for other projects?
- Has your organization developed or acquired a series of software engineering training courses for managers and technical staff?
- Are published software engineering standards provided for every software developer and software manager?
- Have document outlines and examples been developed for all deliverable defined as part of the software process?
- Are formal technical reviews of the requirements specification, design, and code conducted regularly?
- Are formal technical reviews of test procedures and test cases conducted regularly?
- Are the results of each formal technical review documented, including errors found and resources used?
- Is there some mechanism for ensuring that work conducted on a project conforms to software engineering standards?
- Is configuration management used to maintain consistency among system/software requirements, design, code and test cases?
- Is a mechanism used for controlling changes to customer requirements that impact the software?
- Is there a documented statement of work, a software requirements specification, and a software development plan for each subcontract?
- Is a procedure followed for tracking and reviewing the performance of subcontractors?

Technical Issues

- Are facilitated application specification techniques used to aid in communication between the customer and developer?
- Are specific methods used for software analysis?
- Do you use a specific method for data and architectural design?
- Is more than 90 percent of your code written in a high-order language?
- Are specific conventions for code documentation defined and used?

- Do you use specific methods for test case design?
- Are software tools used to support planning and tracking activities?
- Are configuration management software tools used to control and track change activity throughout the software process?
- Are software tools used to support the software analysis and design process?
- Are tools used to create software prototypes?
- Are software tools used to support the testing process?
- Are software tools used to support the production and management of documentation?
- Are quality metrics collected for all software projects?
- Are productivity metrics collected for all software projects?

If a majority of the above questions are answered “no”, software process is weak and risk is high.

Technology Risk

Pushing the limits of the technology is challenging and exciting. It’s the dream of almost every technical person, because it forces a practitioner to use his or her skills to the fullest. But it’s also very risky. Murphy’s law seems to hold sway in this part of the development universe, making it extremely difficult to foresee risks, much less plan for them. The following risk item checklist identifies generic risks associated with the technology to be built:

- Do the customer’s requirements demand the creation of new algorithms or input or output technology?
- Does the software interface with new or proven hardware?
- Does the software to be built interface with vendor supplied software products that are unproven?
- Does the software to be built interface with a database system whose function and performance have not been proven in this application area?
- Is a specialized user interface demanded by product requirements?
- Do requirements for the product demand the creation of program components that are unlike any previously developed by your organization?
- Do requirements demand the use of new analysis, design, or testing methods?
- Do requirements demand the use of unconventional software development methods such as formal methods, AI-based approaches, and artificial neural networks?
- Do requirements put excessive performance constraints on the product?
- Is the customer uncertain that the functionality requested is “doable”?

If the answer to any of these questions is “yes” further investigation should be undertaken to assess risk potential.

Development Environment Risks

If a carpenter were asked to create a fine piece of furniture with a bent, dull handsaw, the quality of the end product would be suspect. Inappropriate or in-effective tools can blunt the efforts of even a skilled practitioner. The software engineering environment supports the project team, the process, and the product. But if the environment is flawed, it can be the source of significant risk. The following risk item checklist identifies generic risks associated with the development environment.

- Is a software project management tool available?
- Is a software process management tool available?
- Are tools for analysis and design available?
- Do analysis and design tools deliver methods that are appropriate for the product to be built?
- Are compilers or code generators available and appropriate for the product to be built?
- Are testing tools available and appropriate for the product to be built?
- Are software configuration management tools available?
- Does the environment make use of a database or repository?
- Are all software tools integrated with one another?
- Have members of the project team received training in each of the tools?
- Are local experts available to answer questions about the tools?
- Is on-line help and documentation for the tools adequate?

If a majority of the above questions are answered “no”, the software development environment is weak and risk is high

Risks associated with staff size and experience

Boehm [BOE 89] suggests the following questions to assess risks associated with staff size and experience:

- Are the best people available?
- Do the people have the right combination of skills?
- Are enough people available?
- Are staff committed for entire duration of the project?
- Will some project staff be working only part time on this project?
- Does staff have the right expectations about the job at hand?
- Has staff received necessary training?
- Will turnover among staff be low enough to allow continuity?

If the answer to any of these questions is “ no” further investigation should be undertaken to assess risk potential.

Risk Components and Drivers.

The U.S. Air Force has written a pamphlet that contains excellent guidelines for software risk identification and abatement. The Air force approach requires that the project manager identify the risk drivers that affect software risk components performance cost, support, and schedule. In the context of this discussion, the risk components are defined in the following manner.

- Performance risk the degree of uncertainty that the product will meet its requirements and be fit for its intended use
- Cost risk the degree of uncertainty that the project budget will be maintained
- Support risk the degree of uncertainty that the software will be easy to correct, adapt, and enhance

- Schedule risk the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time

The impact of each risk driver on the risk component is divided into one of four impact categories negligible marginal, critical, and catastrophic. Indicates the potential consequences of errors or a failure to achieve a desired outcome. The impact category is chosen based on the characterization that best fits the description in the table.

6.5 Risk Projection

Risk projection, also called risk estimation, attempts to rate each risk in two ways the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. The project planner, along with other managers and technical staff, performs four risk projection activities: (1) establish a scale that reflects the perceived likelihood if a risk ; (2) delineate the consequences of the risk (3) estimate the impact of the risk on the project and the product and (4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

COMPONENTS		PERFORMANCE SUPPORT	SUPPORT	COST	SCHEDULE
CATEGORY					
CATASTROPHIC	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of \$500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable delivery date
CRITICAL	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of \$100K to \$500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in delivery date
MARGINAL	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of \$1 to \$100k	
	2	Minimal to small reduction in Technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
NEGLECTIBLE	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than \$1K	
	2	No reduction in Technical performance	Easily supportable software	Possible budget underrun	Early achievable delivery date

Figure 6.1 Impact assessment (BOE89)

A project team begins by listing all risks (no matter how remote) in the first column of the table. Each risk is categorized in the second column (e.g. implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. Individual values are averaged to develop a single consensus probability. Next the impact of each risk is assessed. Each risk component is assessed using the characterization presented in and an impact category is determined. The categories for each of the four risk components performance, support, cost, and schedule are averaged to determine an overall impact value.

Risk	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

Impact values.

1. - Catastrophic 2. - Critical 3. - Marginal 4. - Negligible

Figure 6.2 Sample risk table prior to sorting

Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high impact risks percolate to the top of the table, and low probability risks drop to the bottom. This accomplished first order risk prioritization.

The project manager studies the resultant sorted table and defines a cut off line. The cut off line (drawn horizontally at some point in the table) implies that only risks which lie above the line will be given further attention. Risks that fall below the line are re-evaluated to accomplish second order prioritization. Risk impact and probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high impact risk with moderate to high probability and low impact risk with high probability should be carried forward into the management steps that follow.

All risks that lie above the cut off line must be managed. The column labeled RMMM contains a pointer into a Risk Mitigation, Monitoring and Management Plan developed for all risk that lie above cut off.

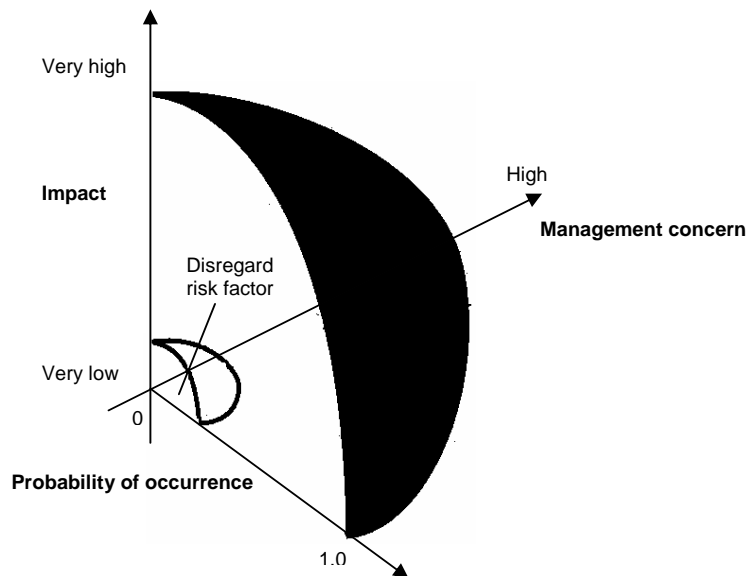


Figure 6.3 risk and management concern

Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed. Risk drivers can be assessed on a qualitative probability scale that has the following values, impossible, improbable, probable, and frequent. Mathematical probability can then be associated with each qualitative value (e.g. probability of 0.7 to 1.0 implies a highly probable risk)

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur its nature, its scope and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution how much of the project will be affected or how many customers are harmed. Finally the timing of a risk considers when and for how long the impact will be felt. In most cases, a project manager might want the “bad news” to occur as soon as possible but in some cases, the longer the delay, the better.

Returning once more to the risk analysis approach proposed by the U.S Air Force the following steps are recommended to determine the overall consequences of a risk.

1. Determine the average probability of occurrence value for each risk component.
2. Using determine the impact for each component based on the criteria shown.
3. Complete the risk table and analyze the results as described in the preceding sections.

The risk projection and analysis techniques applied iteratively as the software project proceeds. The project team should revisit the risk table at regular intervals, re-evaluating each risk to determine when new circumstances, cause its probability and impact to change. As a consequence of this activity, it may

be necessary to add new risks to the table, remove some risks that are no longer relevant, and change the relative positioning of still others.

Risk Assessment

At this point in the risk management process, we have established a set of triplets of the form [CHA 89]

$$[r_i, l_i, x_i]$$

where r_i is risk, l_i is the likelihood (probability) of the risk, and x_i is the impact of the risk. During risk assessment, we further examine the accuracy of the estimates that were made during risk projection, attempt to prioritize the risks that have been uncovered, and begin thinking about ways to control and/ or avert risks that are likely to occur.

For assessment to be useful, a risk referent level [CHA 89] must be defined. For most software projects, the risk components discussed earlier performance, cost, support and schedule also represent risk referent levels. That is, there is a level for performance degradation cost overrun, support difficulty, or schedule slippage (or any combination of the four) that will cause the project to be terminated. If a combination of risks create problems that cause one or more of these referent levels to be exceeded, work will stop. In the context of software risk analysis, a risk referent level has a single point, called the referent point or break point, at which the decision to proceed with the project or terminate it (problems are just too great) are equally acceptable.

Represents this situation graphically. If a combination of risks leads to problems that cause cost and schedule overruns, there will be a level, represented by the curve in the figure, that (when exceeded) will cause project termination (the shaded region). At a referent point, the decisions to proceed or to terminate are equally weighted.

In reality the referent level can rarely be represented as a smooth line on a graph. In most cases it is a region in which there are areas of uncertainty (i.e, attempting to predict a management decision based on the combination of referent values is often impossible)

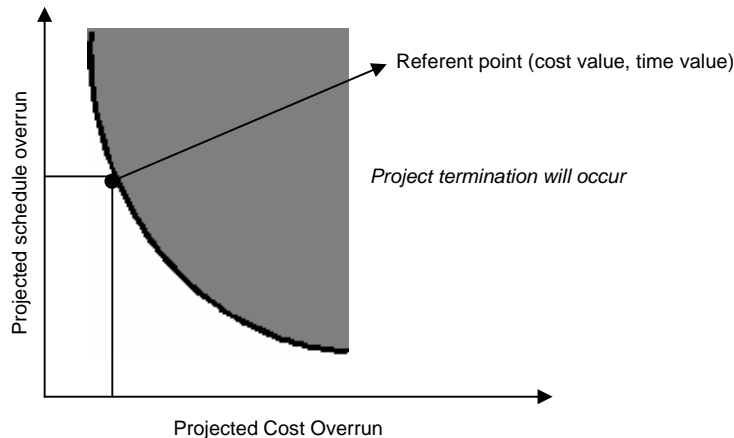


Figure 6.4 Risk referent level

Therefore, during risk assessment, we perform the following steps.

1. Define the risk referent levels for the project
2. Attempt to develop a relationship between each $[r_i, l_i, x_i]$ and each of the referent levels.

3. Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.
4. Try to predict how compound combinations of risks will affect a referent level

A detailed discussion is best left to books that are dedicated to risk analysis (e.g. [CHA89,], [ROW 88]).

6.6 Risk Mitigation, Monitoring and Management

All of the risk analysis activities presented to this point have a single goal to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:

- Risk avoidance.
- Risk monitoring, and
- Risk management and contingency planning

If a software team adopts a proactive approach to risk avoidance is always to best strategy. This is achieved by developing a plan for risk mitigation. For example, assume that high staff turnover is noted as a project risk, r_1 . Based on past history and management intuition, the likelihood, l_1 , of high turnover is estimated to be .70 and impact, x_1 is projected to have a critical impact on project cost and schedule.

To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are these:

- Meet with current staff to determine caused for turnover (E.g. poor working conditions, low pay, competitive job market)
- Act to mitigate those causes that are under management control before the project starts.
- Once project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
- Organize project teams so that information about each development activity is widely dispersed .
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner
- Conduct peer reviews of all work so that more than one person is familiar with the work
- Define a backup staff member for every critical technologist.

As the project proceeds, risk monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the following factors can be monitored:

In addition to monitoring the factors noted above, the project manager should also monitor the effectiveness of risk mitigation step. For example, a risk mitigation step noted above called for the definition of “documentation standards and mechanisms to be sure that documents are developed in a standards and mechanisms to be sure that documents are developed in a timely manner”. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were force to join the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, suppose that the project is well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, the project manager may temporarily refocus resources to those functions that are fully staffed, enabling newcomers who must be added to the team to “get up to speed”. Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode”. This includes video based knowledge capture, the development of “commentary documents” and/or meeting with other team members who will remain on the project.

It is important to note that RMMM steps incur additional project cost. For example, spending the time to “back up” every critical technologist cost accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, the project planner performs a classic cost benefit analysis. If risk aversion steps for high turnover will increase project cost and duration by an estimated 15 percent, but the predominant cost factor is “backup”, management may decide not to implement this step. On the other hand if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent, management will likely put all into place.

For a large project, 30 or 40 risks may be identified. If between three and seven risk management steps are identified for each, risk management may become a project in itself. For this reason, we adapt the Pareto 80-20 rule to software risk. Experience indicates that 80 percent of the overall project risk can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help the planner to determine which of the risks reside in that 20 percent. For this reason some of the risks identified, assessed and projected may not make it into the RMMM plan—they don’t comprise the critical 20 percent (the risks with highest project priority).

6.7 Safety Risks and Hazards

Risk is not limited to the software project itself. Risks can occur after the software has been successfully developed and delivered to the customer. These risks are typically associated with the consequences of software failure in the field.

Although the probability of failure of a well-engineered system is small an undetected fault in a computer-based control or monitoring system could result in enormous economic damage or worse, significant human injury or loss of life. But the cost and monitoring often outweigh the risk. Today computer hardware and software are used regularly to control safety-critical systems.

When software is used as part of the control system complexity can increase by an order of magnitude or more. Subtle design flaws induced by human error— Something that can be uncovered and eliminated in hardware based conventional controls become much more difficult to uncover when software is used

Software safety and hazard analysis are software quality assurance activities that focus on the identification and assessment of potential hazards that may impact software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

6.8 Short Summary

- when ever a lot is riding on a software project, common sense dictates risk analysis. And yet, most software project managers do it informally and superficially, if they do it at all. The time spent identifying, analyzing, and managing risk pays itself back in many ways: less upheaval during the project; a greater ability to track and control a project; and the confidence that comes with planning for problems before they occur.
- Risk analysis can absorb a significant amount of project planning effort. Identification, projection, assessment, management, and monitoring all take time. But the effort is worth it. To quote Sun Tzu, a Chinese general who lived 2500 years ago, "if you know the enemy and know yourself, you need not fear the result of a hundred battles". For the software project manager, the enemy is risk.

6.9 Brain Storm

1. Explain briefly about Software Risks ?
2. What is Risk Identification ? Give a brief note on Risk Identification ?
3. How do you develop a Risk Table ?
4. Explain briefly about RMMM ?



Lecture 7

Project Scheduling and Tracking - I

Objectives

In this lecture you will learn the following

- ✧ Basic Concepts
- ✧ About The Relationship between People and Effort
- ✧ Defining a task set for the Software Project

Coverage Plan

Lecture 7	
7.1	Snap Shot- Basic Concepts 7.1.1 Comments on “ Lateness” 7.1.2 Basic principles
7.2	The Relationship Between People And Effort 7.2.1 An empirical relationship 7.2.2 Effort Distribution
7.3	Defining a Task Set for the Software Project 7.3.1 Degree of Rigor 7.3.2 Defining Adaptation Criteria ⁹ 7.3.3 Computing a Task Set Selector Value 7.3.4 Interpreting the TSS Value and Selecting the Task Set
7.4	Short Summary
7.5	Brain Storm

7.1 Snap Shot

Although there are many reasons why software is delivered late, most can be traced to one or more of the following root causes.

- An unrealistic deadline established by someone outside the software engineering group and forced on managers and practitioners within the group.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Aggressive deadlines are a fact of life in the software business. Sometimes such deadlines are demanded for reasons that are legitimate from the point of view of the person who sets the deadline, but common sense says that legitimacy must also be perceived by the people doing the work.

7. 1.1. Comments on “ Lateness”

The scheduling techniques described in this lecture, are often implemented under the constraint of a defined deadline. If best estimates indicate that the deadline is unrealistic, a competent project manager should “ protect his or her team from undue pressure reflect the pressure back to its originators”.

To illustrate, assume that a software development group has been asked to build a real time controller for a medical diagnostic instrument that is to be introduced to the market in 9 months. After careful estimation and risk analysis the software project manager comes to the conclusion that the software as requested, will require 14 calendar months to create with available staff. How does the project manager proceed?

It is unrealistic to march into the customer’s office and demand that the delivery date be changed. External market pressures have dictated the date, and the product must be released. It is equally foolhardy to refuse to undertake the work . So, what to do?

The following steps are recommended in this situation:

1. Perform a detailed estimate using data from past projects. Determine the estimated effort and duration for the project.
2. Using an incremental process model develop a development strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
3. Meet with the customer and explain why the imposed deadline is unrealistic. Be certain to note that all estimates are based on performance on past projects. Also be certain to indicate the percentage

improvement that would be required to achieve the deadline as it currently exists. The following comment is appropriate:

“ I think we may have a problem with the delivery date for the XYZ controller software. I’ve given each of you an abbreviated breakdown of production rates for past projects and an estimate that we’ve done a number of different ways. You’ll note that I’ve assumed a 20 percent improvement in past production rates, but we still get a delivery date that’s 14 calendar months rather than 9 months away”.

4. Offer the incremental development strategy as an alternative.

“we have a few options, and I’d like you to make a decision based on them. First, we can increase the budget and bring on additional resources so that we’ll have a shot at getting this job done in nine months. But understand that this will increase risk of poor quality due to the tight time line. Second, we can remove a number of the software functions and capabilities that you’re requesting. This will make the preliminary version of the product somewhat less functionality and then deliver over the 14 month period. Third, we can dispense with reality and wish the project complete in 9 months. We’ll wind up with nothing that can be delivered to a customer. The third option, I hope you will agree, is unacceptable. Past history and our best estimates say that it is unrealistic and a recipe for disaster”.

There will be some grumbling but if solid estimates based on good historical data are presented, it’s likely that negotiated versions of either option 1 or option 2 will be chosen. The unrealistic deadline evaporates.

7.1.2 Basic principles

The reality of a technical project is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other tasks lie on the “Critical path.” If these “critical” tasks fall behind schedule, the completion date of the entire project is put into jeopardy.

The project manager’s objective is to define all project tasks, identify the ones that are critical, and then track their progress to ensure that delay is recognized “one day at a time”. To accomplish this, the manager must have a schedule that has been defined at a degree of resolution that enables the manager to monitor progress and control the project.

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major software engineering activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software tasks are identified and scheduled.

Scheduling for software development projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer based system has already been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately the first situation is encountered far more frequently than the second.

Like all other areas of software engineering, a number of basic principles guide software project scheduling:

Compartmentalization The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

Interdependency The interdependencies of each compartmentalized activity or task must be determined. Some tasks must occur in sequence; others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time Allocation Each task to be scheduled must be allocated some number of work units. In addition, each task must be assigned a start date and a completion date that are functions of the interdependencies and whether work will be conducted on a full time or part time basis.

Effort validation Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been allocated at any given time. For example, consider a project that has three assigned staff members. On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products have been reviewed for quality and have been approved.

Each of the above principles is applied as the project schedule evolves.

7.2 The Relationship Between People and Effort

In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.

There is a common myth that is still believed by many managers who are responsible for software development effort: “If we fall the project”, Unfortunately, adding people late in a project often has a disruptive effect, causing schedules to slip even further. People late in a project often has disruptive effect, causing schedules to slip even further. People who are added must learn the system, and the people who teach them are the same people who were doing the work. While they are teaching, no work is done and the project falls further behind.

In addition to the time it takes to learn the system, involving more people increases the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

An example

Consider four software engineers, each capable of producing 5000 LOC/year when working on an individual project. When these four engineers are placed on a team project, six potential communication paths are possible. Each communication path requires time that could otherwise be spent developing software. We shall assume that team productivity will be associated with communication. Therefore, team productivity will be reduced by 250 LOC/year for each communication path, due to the overhead associated with communication. Therefore, team productivity is $20,000 - (250 \times 6) = 18,500$ LOC/year - 7.5 percent less than what we might expect.

The one year project on which the above team is working falls behind schedule and with two months remaining, two additional people are added to the team. The number of communication paths escalates to 14. The productivity input of the new staff is the equivalent of $840 \times 2 = 1680$ LOC for the two months remaining before delivery. Team productivity now is $20,000 + 1680 - (250 \times 14) = 18,180$ LOC / year.

Although the above example is a gross oversimplification of real world circumstances, it does serve to illustrate another key point; The relationship between the number of people working on a software project and overall productivity is not linear.

Based on the people - work relationship, are teams counterproductive? The answer is an emphatic “no,” if communication serves to improve software quality. In fact, formal technical reviews conducted by software engineering teams can lead to better analysis and design, and more important, can reduce the number of errors that go undetected until testing. Hence, productivity and quality, when measured by time to project completion and customer satisfaction, can actually improve.

7.2.1 An empirical relationship

Recalling the “software equation” that was introduced in the previous lecture. We can demonstrate the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code L are related to effort and development time by the equation :

$$L = P \times (E/B)^{1/3} T^{4/3}$$

Where E is development effort in person months ; P is a productivity parameter that reflects a variety of factors that lead to high quality software engineering work B is a special skill factor that ranges between 0.16 and 0.39 and is a function of the size of the software to be produced; and t is the project duration in calendar months.

Rearranging the software equation (above), we arrive at an expression for development effort E .

$$E = L^3 / (P^3 T^4)$$

Where E is the effort expended over the entire life cycle for software development and maintenance, and T is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor (\$/person-year).

This leads to some interesting results. Consider a complex, real time software project estimated at 33,000 LOC, 12 person years of efforts. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years the highly nonlinear nature of the model described in equation 7.1 yields.

$$E = L^3 / (P^3 T^4) \sim 3.8 \text{ person years.}$$

This implies that by extending the end date six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear; Benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

7.2.2 Effort Distribution

Each of the software project estimation techniques discussed in previous lecture leads to estimates of person months required to complete software development. A recommended distribution of effort across the definition and development phases is often referred to as the 40-20-40 rule. Forty percent or more of all effort is allocated to front end analysis and design tasks. A similar percentage is applied to back end testing. You can correctly infer that coding is de-emphasized.

This effort distribution should be used as a guideline only. The characteristics of each project dictate the distribution of effort. Effort expended on project planning rarely accounts for more than 2 or 3 percent of effort, unless the plan commits and organization to large expenditures with high risk. Requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages may be considered.

7.3 Defining a Task Set for the Software Project

A number of different process models were described in lecture 2. These models offer different paradigms for software development. Regardless of whether a software team chooses a linear sequential paradigm an iterative paradigm an evolutionary paradigm a concurrent paradigm or some permutation the process model is populated by a set of tasks that enable the software team to define, develop, and ultimately maintain computer software.

There is no single set of tasks that is appropriate for all projects. The set of tasks that would be appropriate for a large complex system would likely be perceived as overkill for a small relatively simple project. Therefore an effective software process should define a collection of task sets each designed to meet the needs of different types of projects.

A task set is a collection of software engineering work tasks, milestone and deliverables that must be accomplished to complete a particular project. The task set to be chosen must provide enough discipline to achieve high software quality. But at the same time, it must not burden the project team with unnecessary work.

Task sets are designed to accommodate different type of projects and different degrees of rigor. Although it is difficult to develop a comprehensive taxonomy, most software organizations encounter projects of the following types :

- I. **Concept Development Projects.** That are initiated to explore some new business concept or application of some new technology.
- II. **New Application Development Projects** That are undertaken as a consequence of a specific customer request.
- III. **Application Enhancement Projects** That occur when existing software undergoes major modification to function performance or interfaces that are observable by the end user.
- IV. **Application Maintenance Projects** That correct, adapt or extend, existing software in ways that may not be immediately obvious to the end user.
- V. **Reengineering Projects** That are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

Even within a single project type there are many factors that influence the task set to be chosen. When taken in combination these factors provide an indication of the degree of rigor with which the software process should be applied.

7.3.1 Degree of Rigor

Even for a project of a particular type the degree of rigor with which the software process is applied may vary significantly. The degree of rigor is function of many project characteristics. As an example, small, non-business, critical projects can generally be addressed with somewhat less rigor than large complex baseline critical applications. It should be noted however that all projects must be conducted in manner that results in timely high quality deliverables. Four different degrees of rigor can be defined:

Casual. All process framework activities are applied but only a minimum task set is required. In general umbrella tasks will be minimized and documentation requirements will be reduced. All basic principles of software engineering are still applicable.

Structured. The process framework will be applied for this project. Framework activities and related tasks appropriate to the project type will be applied and umbrella activities necessary to ensure high quality will be applied. SQA, SCM documentation and measurement tasks will be conducted in a streamlined manner.

Strict. The full process will be applied for this project with a degree of discipline that will ensure high quality. All umbrella activities will be applied and robust documentation will be produced.

Quick Reaction. The process framework will be applied for this project but because of an emergency situation only those tasks essential to maintaining good quality will be applied. "Back-filling " (i.e., developing a complete set of documentation conducting additional reviews) will be accomplished after the application/product is delivered to the customer.

The project manager must develop a systematic approach for selecting the degree of rigor that is appropriate for a particular project. To accomplish this project adaptation criteria are defined and a task set selector value is computed.

7.3.2 Defining Adaptation Criteria

Adaptation criteria are used to determine the recommended degree of rigor with which the software process should be applied on a project. Eleven adaptation criteria are defined for software projects:

- Size of the project
- Number of potential users
- Mission critically
- Application longevity
- Stability of requirements
- Ease of customer/developer communication
- Maturity of applicable technology
- Performance constraints
- Embedded /non embedded characteristics
- Project staffing
- Reengineering factors

Each of the adaptation criteria is assigned a grade that ranges between 1 and 5 where 1 represents a project in which a small subset of process tasks are required and over all methodological and documentation of requirements are minima 1 and 5 represents a project in which a complete set of process tasks should be applied and overall methodological and documentation requirements are substantial

7.3.3 Computing a Task Set Selector Value

To select the appropriate task set for a project the following steps should be conducted:

1. Review each of the adaptation criteria sin Section 7.3.2 and assign the appropriate grades (1 to 5) based on the characteristic of the project. These grades should be entered into Table 7.1
2. Review the weighting factors assigned to each of the criteria. The value of a weighting factor ranges from 0.8 to 1.2 and provides an indication of the relative importance of a particular adaptation criterion to the types of software developed within the local environments. If modification are required to better reflect local circumstances they should be made.
3. Multiply the grade entered in Table 7.1 by the weighting factor and by the entry point multiplier for the type of project to be undertaken. The entry point multiple user takes on a value of 0 or 1 and indicates that relevance of the adaptation criterion to the project type. The result of the product: *Grade*weighting factor* entry point multiplier*
4. Compute the average of all entries in the “Product” column and place the result in the space marked task set selector. This value will be used to help you select the task set that is most appropriate for the project.

Adaptation criteria	Grade	Weight	Entry Point Multiplier				Product	
			Conc.	N.Dev.	Enhan.	Main.		Reeng.
Size of project	----	1.20	0	1	1	1	1	----
No. of users	----	1.10	0	1	1	1	1	----
Business criticality	----	1.10	0	1	1	1	1	----
Longevity	----	0.90	0	1	1	0	0	----
Stability of work	----	1.20	0	1	1	1	1	----

Ease of communication	----	0.90	1	1	1	1	1	----
Modularity of Tech.	----	0.90	1	1	0	0	1	----
Performance constraint	----	0.80	0	1	1	0	1	----
Embedded/nonembedded	----	1.20	1	1	1	0	1	----
Project staffing	----	1.00	1	1	1	1	1	----
Interoperability	----	1.10	0	1	1	1	1	----
Reengineering factors	----	1.20	0	0	0	0	1	----

Table 7.1 Computing the task set selector

7.3.4 Interpreting the TSS Value and Selecting the Task Set

Once the task set selector is computed, the following guidelines can be used to select the appropriate task set for a project.

Task Set Selector Value	Degree of Rigor
TSS<1.2	casual
1.0 <TSS <3.0	structured
TSS>2.4	strict

The overlap in TSS values from one recommended task set to another is purposeful and is intended to illustrate that sharp boundaries are impossible to define when making ask set selections. In the final analysis the task set selector value past experience and common sense must all be factored into the choice of the task set for a project.

Adaptation criteria	Grade	Weight	Entry Point Multiplier				Product	
			Conc.	N.Dev.	Enhan.	Main.		Reeng.
Size of project	2	1.2	----	1	----	----	----	2.4
No. of users	3	1.1	----	1	----	----	----	3.3
Business criticality	4	1.1	----	1	----	----	----	4.4
Longevity	3	0.9	----	1	----	----	----	2.7
Stability of work	2	1.2	----	1	----	----	----	2.4
Ease of communication	2	0.9	----	1	----	----	----	1.8
Modularity of Tech.	2	0.9	----	1	----	----	----	1.8
Performance constraint	3	0.8	----	1	----	----	----	2.4
Embeddeed/nonembedded	3	1.2	----	1	----	----	----	3.6
Project staffing	2	1.0	----	1	----	----	----	2.0
Interoperability	4	1.1	----	1	----	----	----	4.4
Reengineering factors	0	1.2	----	0	----	----	----	2.8

Table 7.2 Computing the task set selector – An example

Table 7.2 illustrates how TSS might be computed for a hypothetical project. The project manager selects the grades shown in the “grade” column. The project type is new application development therefore entry point multipliers are selected from the “Ndev” column. The entry in the “Product” column is computed using

Grade*Weight* New Dev Entry Point Multiplier

The value of TSS (computed as the average of all entries in the product column) is 2.8 Using the criteria discussed above, the manager has the option of using either the structured or the strict task set. The final decision is made once all project factors have been considered.

7.4 Short Summary

- Scheduling is the culmination of a planning activity that is a primary component of software project management.
- When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

7.5 Brain Storm

1. What is Effort Distribution ?
2. Define Empirical Relationship ?
3. What is Degree of Rigor ?
4. How do you compute a TSS value ?



Lecture 8

Project Scheduling and Tracking - II

Objectives

In this lecture you will learn the following

- ✧ About selecting Software Engineering tasks
- ✧ Defining a Task Network
- ✧ About Scheduling

Coverage Plan

Lecture 8

- 8.1 Snap shot
- 8.2 Selecting Software Engineering Tasks
- 8.3 Refinement of Major Tasks
- 8.4 Defining a Tasks Network
- 8.5 Scheduling
- 8.6 Timeline Charts
- 8.7 Tracking the Schedule
- 8.8 The Project Plan
- 8.9 Short Summary
- 8.10 Brain Storm

8.1 Snap Shot

In this lecture we are going to learn about Software Engineering tasks, scheduling, Timeline charts and project plan.

8.2 Selecting Software Engineering Tasks

In order to develop a project schedule a task set must be distributed on the project time line. As we noted in the previous software project task section the task set will vary depending upon the project type and the degree of rigor. Each of the project types described in the previous software project task Section may be approached using a process model that is linear sequential iterative (e.g., the prototyping model) or evolutionary (e.g., the spiral model). In Some cases one project type flows smoothly into the next. For example concept development projects that succeed often evolve into new application development projects. As a new application development project ends an application enhancement project sometimes begins. This progression is both natural and predictable and will occur regardless of the process model that is adopted by an organization. As an example we consider the major software engineering tasks for concept development projects.

Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable but a customer believes that potential benefit exists. Concept development projects are approached by applying the following major tasks:

Concept Scoping determines the overall scope of the project

Preliminary Concept Planning establishes the organization's ability to undertake the work implied by the project scope.

Technology risk assessment evaluates the risk associated with the technology to be implemented as part of project scope.

Proof of concept demonstrates the viability of a new technology in the software context.

Concept implementation implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

Customer reaction to concept solicits feedback on a new technology concept and targets specific customer applications.

A quick scan of these major tasks should yield few surprises. In fact, the flow of software engineering tasks for concept development projects is little more than common sense.

The software team must understand what must be done; the team must determine whether there's anyone available to do it; consider the risks associated with the work; prove the technology in some way; and implement it in a prototypical manner so that the customer can evaluate it. Finally, if the concept is viable, a production version must be produced.

It is important to note that concept development tasks are iterative in nature. That is, an actual concept development project might approach the above tasks in a number of planned increments, each designed to produce a deliverable that can be evaluated by the customer.

If a linear process model flow is chosen. Each of these increments is defined in a repeating sequence. During each sequence, umbrella activities are applied; quality is monitored, and at the end of each sequence, a deliverable is produced. With each iteration, the deliverable should converge toward the defined end product for the concept development stage. If an evolutionary model is chosen, the layout of tasks I.1 through I.6 would appear in the previous lecture. Major software engineering tasks for other project types can be defined and applied in a similar manner.

8.3 Refinement of Major Tasks

The major tasks described in previous section may be used to define a macroscopic schedule would be used to define a task network for the project.

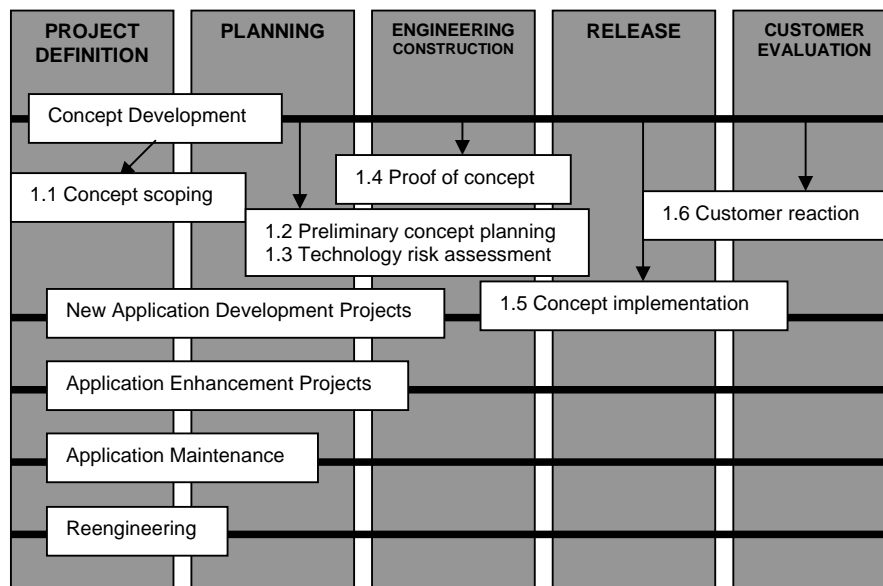


Fig 8.1 Concept development tasks in a linear, sequential model

The macroscopic schedule must be refined to create a detailed project schedule. Refinement begins by taking each major task and decomposing it into a set of subtasks.

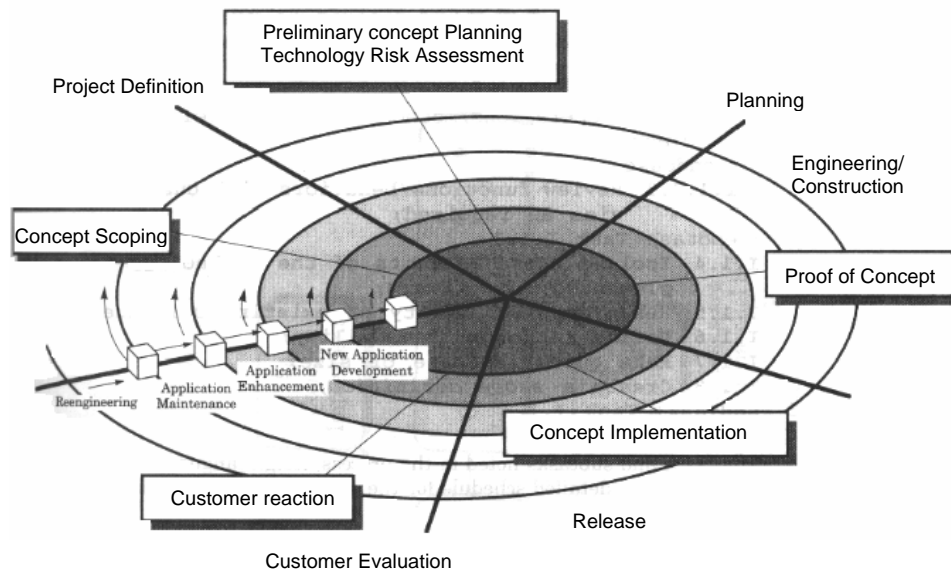


Figure 8.2 Concept development tasks using an evolutionary model

8.4 Defining a Tasks Network

Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project. It is likely that development activities and tasks will be performed in parallel. When this occurs, concurrent tasks must be coordinated so that they will be complete when later tasks require their work product.

A task network is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form, the task network depicts major software engineering tasks. The following figure 8.1 shows a schematic task network for a concept development project.

The concurrent nature of software engineering activities leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, the planner must determine intertask dependencies to ensure continuous progress toward completion. In addition, the project manager should be aware of these tasks that lie on the critical path. That is, tasks that must be completed on schedule if the project as a whole is to be completed on schedule.

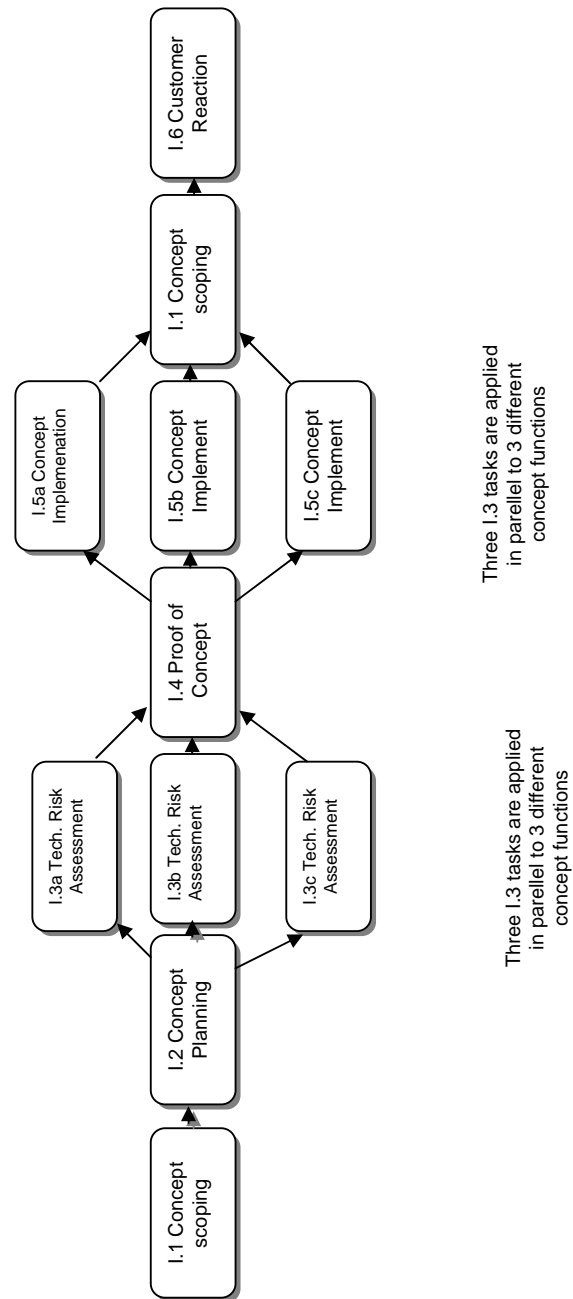


Figure :8.3 A task network for a concept development project

It is important to note that the task network shown in figure 8.1 macroscopic. In a detailed task network each activity shown in figure 8.1 would be expanded. For example task I.1 would be expanded to show all tasks detailed in the refinement.

8.5 Scheduling

Scheduling of a software projects does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied to software with little modification.

Program evaluation and review technique (PERT) and critical path method (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities:

- Estimates of effort
- A decomposition of product function
- The selection of the appropriate process model
- The selection of project type and task set

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project work breakdown structure, are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow the software planner to (1) determine the critical path- the chain of tasks that determines the duration of the project; (2) establish most likely time estimates for individual tasks by applying statistical models; and (3) calculate boundary times that define a time “window” for a particular task.

Boundary time calculations can be very useful in software project scheduling. Slippage in the design of one function, for example, can retard further development of other functions. Riggs describes important boundary times that may be discerned from a PERT or CPM network (1) the earliest time that a task can begin when all preceding tasks are completed in the shortest possible time;(2) the latest time for task initiation before the minimum project completion time is delayed;(3) the earliest finish-the sum of the earliest start and the task duration (4) the latest finish-the latest start time added to task duration and (5) the total float the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule. Boundary time calculations lead to a determination of critical path and provide the manager with a quantitative method for evaluating progress as tasks are completed.

Both PERT and CPM have been implemented in a wide variety of automated tools that are available for virtually every personal computer. Such tools are easy to use and make the scheduling methods described above available to every software project manager.

8.6 Timeline Charts

When creating a software project schedule, the planner begins with a set of tasks(the work breakdown structure). If automated tools are used, the work breakdown is input as a task network or task outline. Effort, duration and start date are then input for each tasks. In addition, tasks may be assigned to specific individuals.

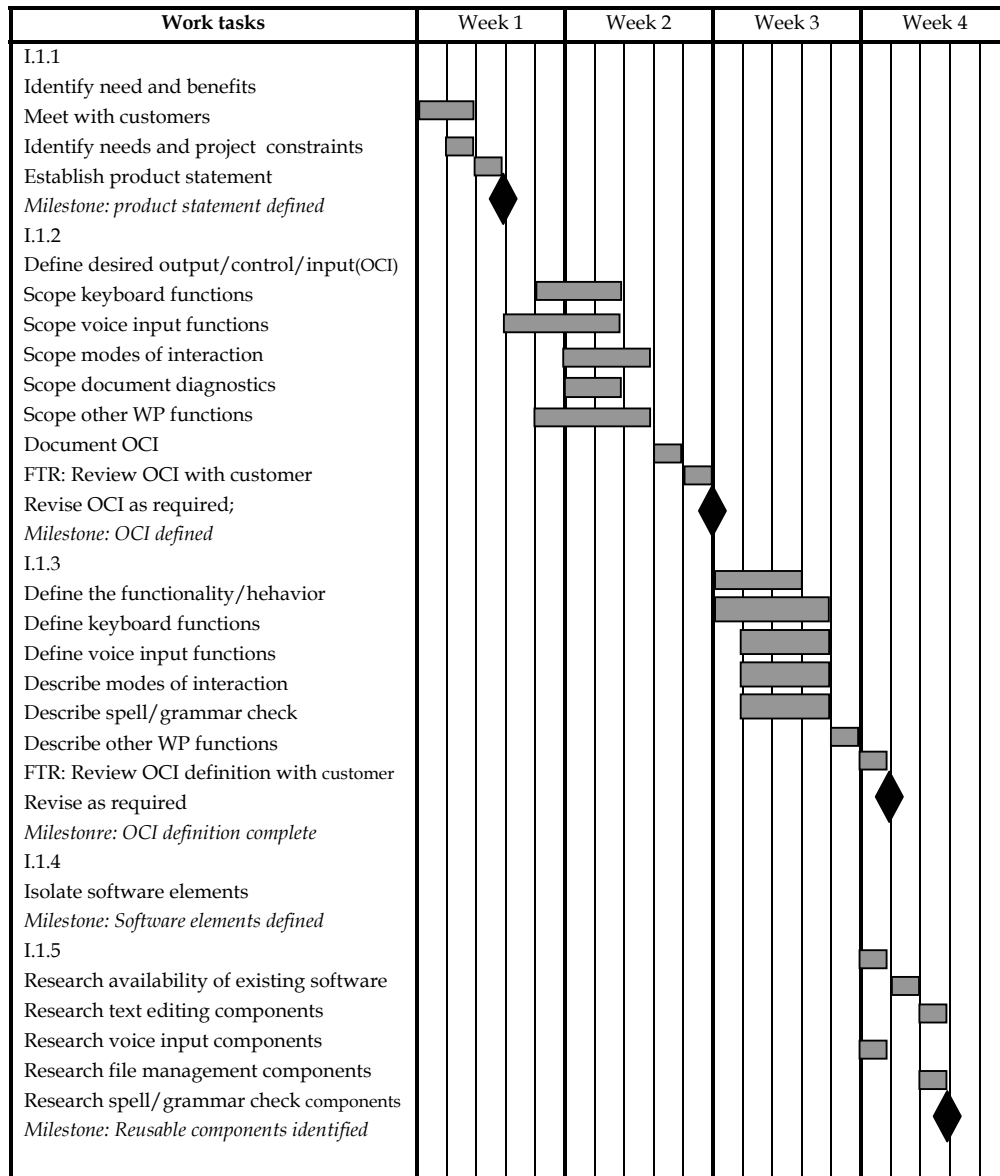


Figure 8.4 An example timeline chart

As a consequence of this input, a timeline chart, also called a Gantt chart, is generated. A timeline chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Figure 8.4 illustrates the format of a timeline chart. It depicts a part of a software project schedule that emphasizes the concept scoping task for a new work-processing software product. All project tasks (for concept scoping) are listed in the left hand column. The horizontal bars indicate the during of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.

Once the information necessary for the generation of a timeline chart has been input, the majority of software project scheduling tools produce project tables a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information. Used in conjunction with the timeline chart, project tables enable the project manager to track progress.

8.7 Tracking the Schedule

The project schedule provides a road map for a software project manager. If it has been properly developed, the project schedule defines the tasks and milestones that must be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways.

- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process
- Determining whether formal project milestones (the diamonds shown in figure 8.4) have been accomplish by the scheduled date.
- Comparing actual start date to planned start date for each project task listed in the project table.
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.

In reality, all of these tracking techniques are used by experienced project managers.

Control is employed by a software project manager to administer project resource, cope with problems and direct project manager to administered project (i.e., the project is on schedule and within budget reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems occur the project manager must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed additional resources may be focused on the problem area: Staff may be redeployed or the project schedule can be redefined.

Work Tasks	Planned start	Actual start	Planned Complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1							
Identify needs and benefits							
Meet with customers	Wk1,d1	Wk1,d1	Wk1,d2	Wk1,d2	Wk1,d1	BLS	Scoping will require more effort/time
Identify needs and project constraints	Wk1,d2	Wk1,d2	Wk1,d2	Wk1,d2	Wk1,d2	JPP	
Establish product statement	Wk1,d3	Wk1,d3	Wk1,d3	Wk1,d3	Wk1,d3	BLS/JPP	
Milestone: product statement defined	Wk1,d3	Wk1,d3	Wk1,d3	Wk1,d3	Wk1,d3		
I.1.2							
Defined desired output/control /input (OCI)							
Scope keyboard functions	Wk1,d4	Wk1,d4	Wk2,d2		Wk1,d4	BLS	
Scope voice input functions	Wk1,d3	Wk1,d3	Wk2,d2		Wk1,d3	JPP	
Scope modes of interaction	Wk2,d1		Wk2,d3		Wk2,d1	MLL	
Scope document diagnostics	Wk2,d1		Wk2,d2		Wk2,d1	BLS	
Scope other WP functions	Wk1,d4	Wk2,d1	Wk2,d3		Wk1,d4	JPP	
Document OCI	Wk2,d1		Wk2,d3		Wk2,d1	MILL	
FTR: Review OCI with customer	Wk2,d3		Wk2,d3		Wk2,d3	All	
Revise OCI as required;	Wk2,d4		Wk2,d4		Wk2,d4	All	
Milestone: OCI defined	Wk2,d5		Wk2,d5		Wk2,d5		
I.1.3							
Define the functionality/behavior							

Figure 8.5 An example project table

When faced with severe deadline pressure experienced project managers sometimes use a project scheduling and control technique called time-boxing. The time boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore an incremental software paradigm is chosen and a schedule is derived for each incremental delivery.

The tasks associated with each increment are then time -boxed . This means that the schedule for each task is adjusted by working backward from the delivery date for the increment . A “box” is put around each task. When a task hits the boundary of its time-box (plus or minus 10 percent) work stops and the next task begins.

The initial reaction to the time-boxing approach is often negative. “ if the work isn’t finished how can we proceed?” The answer lies in the way work is accomplished. By the time that time-box boundary is encountered it is likely that 90 percent of the task has been completed. The remaining 10 percent although important can (1) be delayed until the next increment for (2) be completed late if required. Rather than becoming “stuck” on a task the project proceeds toward the delivery date.

8.8 The Project Plan

Each step in the software engineering process should produce a work product that can be reviewed and that can act as a foundation for the steps that follow the Software project plan is produced at the culmination of the planning tasks . It provides baseline cost and scheduling information that will be used through out the software engineering process.

The software project plan is a relatively brief document that is addressed to a diverse audience. It must (1) communicate scope and resources to software management technical staff and the customer; (2) define risks and suggest risk management techniques ; (3) define cost and schedule for management review ; (4) provide an overall approach to software development for all people associated with the project; and (5)outline how quality will be ensured and change will be managed. An outline of the software project plan is presented in Figure 8.6

A presentation of cost and schedule will vary with the audience to whom it is addressed. If the plan is used only as an internal document, the result of each costing technique can be presented. When the plan is disseminated outside the organization a reconciled cost breakdown (combining the results of all costing techniques) is provided. Similarly the degree of detail contained within the schedule section may vary with the audience and formality of the plan.

The software project plan need not be a lengthy, complex document. Its purpose is to help establish the viability of the software development effort. The plan concentrates on a general statement of what and a specific statement of

I. Introduction

- A. Purpose of Plan
- B. Project Scope and Objectives
 - 1. Statement of Scope
 - 2. Major Functions
 - 3. Performance Issues
 - 4. Management and Technical Constraints

II. Project Estimates

- A. Historical Data used for Estimates
- B. Estimation Techniques
- C. Estimates of Effort, Cost , Duration

III. Risks Management Strategy

- A. Risk Table
- B. Discussion of Risks to be Managed
- C. RMMM Plan for each risk:
 - 1. Risk Mitigation
 - 2. Risk Monitoring
 - 3. Risk Management (contingency plans)

IV. Schedule

- A. Project Work Breakdown Structure
- B. Task Network
- C. Timeline Chart (Gantt Chart)
- D. Resource Table

V. Project Resources

- A. People
- B. Hardware and Software
- C. Special Resources

VI. Staff Organization

- A. Team Structure (if applicable)
- B. Management Reporting

VII. Tracking and Control Mechanisms

- A. Quality Assurance and Control
- B. Change Management and Control

VIII. Appendices

Fig 8.6. Software Project Plan

How much and how long. Subsequent steps in the software engineering process will concentrate on definition, development and maintenance.

8.9 Short Summary

- Scheduling begins with process decomposition. The characteristics of the project are used to adapt an appropriate task set for the work to be done.
- A task network depicts each engineering task, its dependency on other tasks, and its projected duration. The task network is used to compute the critical project path, a timeline chart, and a variety of project information.

- Using the schedule as a guide, the project manager can track and control each step in the software engineering process.

8.10 Brain Storm

1. How do you define a Task Network?
2. Give a short note on Timeline Charts?
3. Explain briefly about Tracking the Schedule?
4. What is Project Plan?

END

Lecture 9

Software Quality Assurance - I

Objectives

In this lecture you will learn the following

- ✧ About Quality Concepts
- ✧ About Quality Movement
- ✧ About Software Reviews

Coverage Plan

Lecture 9
9.1 Snap Shot
9.2 Quality Concept
9.3 The quality Movement
9.4 Software Quality Assurance
9.5 Software Reviews
9.6 Short Summary
9.7 Brain Storm

9.1 Snap Shot

In this lecture, we focus on the Management issues and the process specific activities that enable a software organization ensure that it does the right thing at the right time in the right way. A quantitative discussion of quality is presented in the lecture Technical Metric for software.

9.2 Quality Concept

QUALITY

The American Heritage Dictionary defines quality as “a characteristic or attribute of something. As an attribute of an item, quality refers to measurable characteristics-thing we are able to compare to known standards such as length, color, electrical properties, malleability, and so on. However, software, largely an intellectual entity, is more challenging to characterize than physical objects.

Nevertheless, measures of a program’s characteristics do exist. These properties include cyclomatic complexity, cohesion, number of function points, lines of code and many others discussed in the lecture coming. When we examine an item based on its measurable characteristics, two kinds of quality may be encountered quality of design and quality of conformance.

Quality of design refers to the characteristics that designers specify for an item. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-graded materials are used and tighter tolerances and greater levels of performance are specified, the design quality of a product increases, If the product is manufactured according to specifications.

Quality of conformance is the degree to which the design specifications are followed during manufacturing. Again the greater the degree of conformance, the higher the level of quality of conformance.

In software development quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high

Quality control

Variation control may be equated to quality control. But how do we achieve quality control? Quality control is the series of inspections reviews, and tests used throughout the development cycle to ensure that each work product meets the requirements placed upon it, Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications. This approach views quality control as part of the manufacturing process.

Quality control activities may be fully automated, entirely manual or a combination of automated tools and human interaction. A key concept of quality controls is that all work products have defined and measurable specifications. To which we may compare the outputs of each process. The feedback loop is essential to minimize the defects produced.

Quality Assurance

Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues .

Cost of Quality

Cost of quality includes all costs incurred in the pursuit of quality or in performing quality related activities. Cost of quality studies are conducted to provide a baseline for the current. Cost of quality, to identify opportunities for reducing the cost of quality and ;to provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the affect of changes in dollar-based terms.

Quality costs may be divided into costs associated with prevention, appraisal, and failure. Prevention costs include:

- Quality planning
- Formal technical reviews
- Test equipment
- Training

Appraisal costs include activities to gain insight into product condition "first time through" each process. Examples of appraisal costs include:

- In-process and inter process inspection
- Equipment calibration and maintenance
- Testing

Failure costs are costs that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. Internal failure costs are the costs incurred when we detect an error in our product prior to shipment . Internal failure costs include:

- Rework
- Repair
- Failure mode analysis

External failure costs are the costs associated with defects found after the product has been shipped to the customer. Examples of external failure s costs are:

- Complaint resolution
- Product return and replacement
- Help line support
- Warranty work

As expected, the relative costs to find and repair a defect increase dramatically as we go from prevention to detection and from internal failure to external failure. Figure 9.1 based on data collected by Boehm [BOE81], illustrates this phenomenon.

More recent anecdotal data is reported by Kaplan and his colleagues [KAP95] and is based on work at IBM's Rochester development facility:

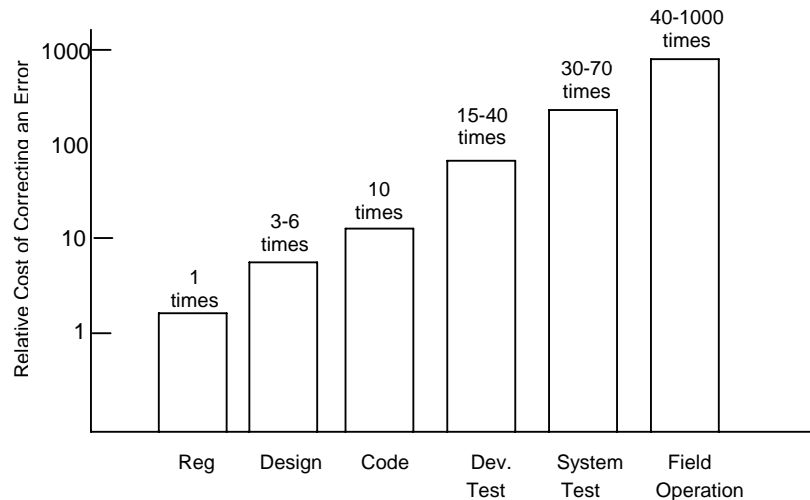


Fig 9.1 Relative cost of correcting on error

A total of 7053 hours was spent inspecting 2,00,000 lines of code with the result that 3112 potential defects were prevented. Assuming a programmer cost of \$40.00 per hour the total cost of preventing 3112 defects was \$282,120 or roughly \$91.00 per defect.

Compare these numbers to the cost of defect removal once the product has been shipped to the customer. Suppose that there had been no inspections, but that programmers had been extra careful and only one defect per 1000 lines of code [significantly better than industry average] escaped into the shipped product. That would mean that 200 defects would still have to be fixed in the field. At an estimated cost of \$25,000 per field fix, the cost would be \$5 million or approximately 18 times more expensive than the total cost of the defect prevention effect.

It is true that IBM produces software that is used by tens of thousands of customers and that their costs for field fixes may be higher than average. This in no way negates the results noted above. Even if the average software organization has field fix costs that are 25 percent of IBM's (most have no idea what their costs are!), the cost savings associated with quality control and assurance activities are compelling.

9.3 The Quality Movement

Today senior managers at companies throughout the industrialized world recognize that high product quality translates to cost savings and an improved bottom line. However, this was not always the case. The quality movement began in the 1940's with the seminal work of W. Edwards Deming [DEM86] and had its first true test in Japan. Using Deming's ideas as a cornerstone, the Japanese have developed a systematic approach to the elimination of the root causes of product defects. Throughout the 1970s and 1980s their work migrated to the Western world and is sometimes called "total quality management (TQM)," Although terminology differs across different companies and authors, a basic four-step progression is normally encountered and forms the foundation of any good TQM program.

The first step is called kaizen and refers to a system of continuous process improvement. The goal of kaizen is to develop a process (in this case, the software process) that is visible, repeatable, and measurable.

The second step, invoked only after kaizen has been achieved, is called atarimae hinshitsu. This step examines intangibles that affect the process and works to optimize their impact on the process. For example the software process may be affected by high staff turnover, which itself is caused by constant reorganizations within a company. It may be that a stable organizational structure could much to improve the quality of software. Atarimae hinshitsu would lead management to suggest changes in the way reorganization occurs.

While the first two steps focus on the process, the next step called kansei (translated as “the five senses”) concentrates on the user of the product (in this case, software) . In essence by examining the way the user applies the product kansei leads to improvement in the product itself, and potentially to the process that created it.

Finally a step called miryokuteki hinshitsu broadens management concern beyond the immediate product. This is a business-oriented step that looks for opportunity in related areas that can be identified by observing the use of the product in the marketplace. In the software world, miryokuteki hinshitsu might be viewed as an attempt to uncover new and profitable products or applications that are an outgrowth from an existing computer-based system .

For most companies kaizen should be of immediate concern. Until a mature software process has been achieved, there is little point in moving to the next steps.

9.4 Software Quality Assurance

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define quality? A wag once said , “Every program does something right, it just may not be the thing that we want it to do”

There have been many definitions of software quality proposed in the literature. For our purposes, software quality is defined as:

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

There is little question that the above definition could be modified or extended . In fact a definitive definition of software quality could be debated endlessly. For the purposes of this book, the above definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for good maintainability) . If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Background issues

Quality assurance is an essential activity for any business that produces products to be used by others. Prior to the twentieth century, quality assurance was the sole responsibility of the craftsman who built a product. The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world.

During the early days of computing (the 1950s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world [IEE94] Extending the definition presented earlier, software quality assurance is a “planned and systematic pattern of actions” [SCH87] that are required to ensure quality in software. Today the implication is that many different constituencies in an organization have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer’s in-house representative. That is the people who perform SQA must look at the software from the customer’s point of view. Does the software adequately meet the quality factors. Has software development been conducted according to pre-established standards? Have technical disciplines properly performed their roles as part of the SQA group attempts to answer these and other questions to ensure that software quality is maintained.

SQA Activities

Software quality assurance is comprised of a variety of tasks associated with two different constituencies—the software engineers who do technical work and a SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis and reporting.

Software engineers address quality (and perform quality assurance) by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing.

The charter of the SQA group is to assist the software engineering team in achieving a high quality end product. The software Engineering Institute [PAU93] recommends a set of SQA activities that address quality assuring planning, oversight, record keeping, analysis, and reporting. It is these activities that are performed (or facilitated) by an independent SQA group.

Prepare a SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies:

- Evaluations to be performed
- Audits and reviews to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking
- Documents to be produced by the SQA group
- Amount of feedback provided to software project team

Participates in the development of the project's software process description. The software engineering team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO 9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products, identifies, documents and tracks deviations, verifies that corrections have been made and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or technical work products

Record any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

In addition to these activities the SQA group coordinates the control and management of change and helps to collect and analyze software metrics.

9.5 Software Reviews

Software reviews are a "filter" for the software engineering process. That is reviews are applied at various points during Software development and serve to uncover errors that can then be removed. Software reviews serve to "purify" the software work products that occur as a result of analysis design, and coding. Freedman and Weinberg [FRE90] discuss the need for reviews this way:

Technical work needs reviewing for the same reason that pencils need erasers: To err is human. The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else. The review process is, therefore, the answer to the prayer of Robert Burns:

O wad some power the giftie give us
To see ourselves as other set us

A review-any review-is a way of using the diversity of a group of people to:

1. Point out needed improvement in the product of single person or team
2. Confirm those parts of a product in which improvement is either not desired or not needed; and
3. Achieve technical work of more uniform, or at least more predictable. Quality than can be achieved without reviews, in order to make technical work more manageable.

There are many different types of reviews the can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and

technical staff is a form of review. In this book however we focus on the formal technical review-sometimes called a walkthrough.

A formal technical review is the most effective filter from a quality assurance standpoint Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

Cost Impact of Software Defects

The IEEE Standard Dictionary of Electrical and Electronics Terms (IEEE Standard 100-1992) defines a defect as “a product anomaly.” The definition for “fault” in the hardware context can be found in IEEE Standard 610.12-1990 ;

(a) A defect in a hardware device or component; for example, a short circuit or broken wire. (b) An incorrect step process, or data definition in a computer program. Note This definition is used primarily by the fault tolerance discipline. In common usage, the terms “error” and “bug” are used to express this meaning See also data-sensitive fault; program-sensitive fault; equivalent faults; fault masking; intermittent fault.

Within the context of the software process, the terms “defect” and “fault” are synonymous. Both imply a quality problem that is discovered after the software has been released to end users. In earlier chapters we used the term “error” to depict a quality problem that is discovered by software engineers (or others) before the software is released to the end user.

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of formal technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

A number of industry studies (TRW, Nippon Electric, and Mitre Corp., among others) indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, formal review techniques have been shown to be up to 75 percent effective [JON86] in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and maintenance phases.

To illustrate the cost impact of a early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects [IBM81] Assume that an error uncovered during design will cost 1.0 monetary unit to correct. Relative to this cost, the same error uncovered just before testing commences will cost 6.5 units during testing 15 units and after release between 60 and 100 units.

Defect Amplification and Removal

A defect amplification model [IBM81] can be used to illustrate the generation and detection of errors during preliminary design, detail design, and coding steps of the software engineering process. The model is illustrated schematically in Figure 9.2 A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors, passed through from previous steps are amplified (amplification factor, x) by current work. The box subdivisions represent each of these characteristics and the percent efficiency for detecting errors a function of the thoroughness of review.

Figure 9.3 illustrates a hypothetical example of defect amplification for a software development process in which no reviews are conducted. As shown in the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design errors are amplified to 94 errors before testing commences. Twelve latent defects are released to the field. Figure 9.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, 10 initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent defects exist. By recalling the relative costs associated with the discovery and correction of errors overall cost (with and without review for Your hypothetical example) can be established. In Table 10.1 it can be seen that total cost for development and maintenance when reviews are conducted is 783 cost units. When no reviews are conducted total cost is 2177 units-nearly three times more costly.

To conduct reviews, a developer must expend time and effort and the development organization must spend money. However, the results of the preceding example leave little doubt that we have encountered a “pay now or pay much more later” syndrome. Formal technical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

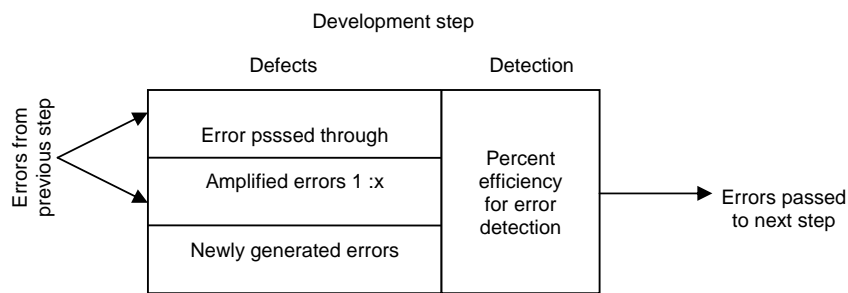


Fig 9.2 Defect amplification model

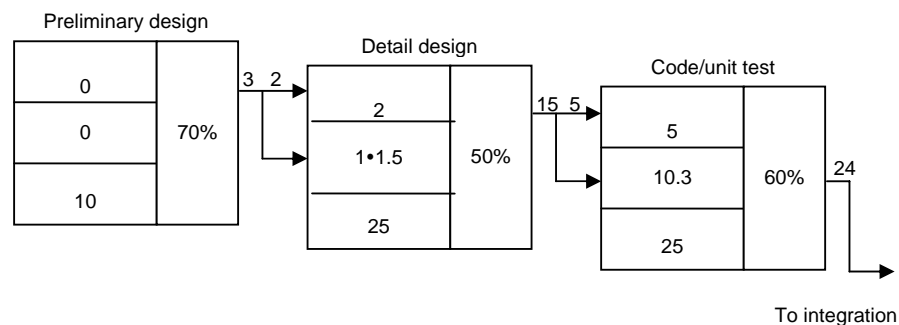


Fig 9.3 Defect amplification no reviews

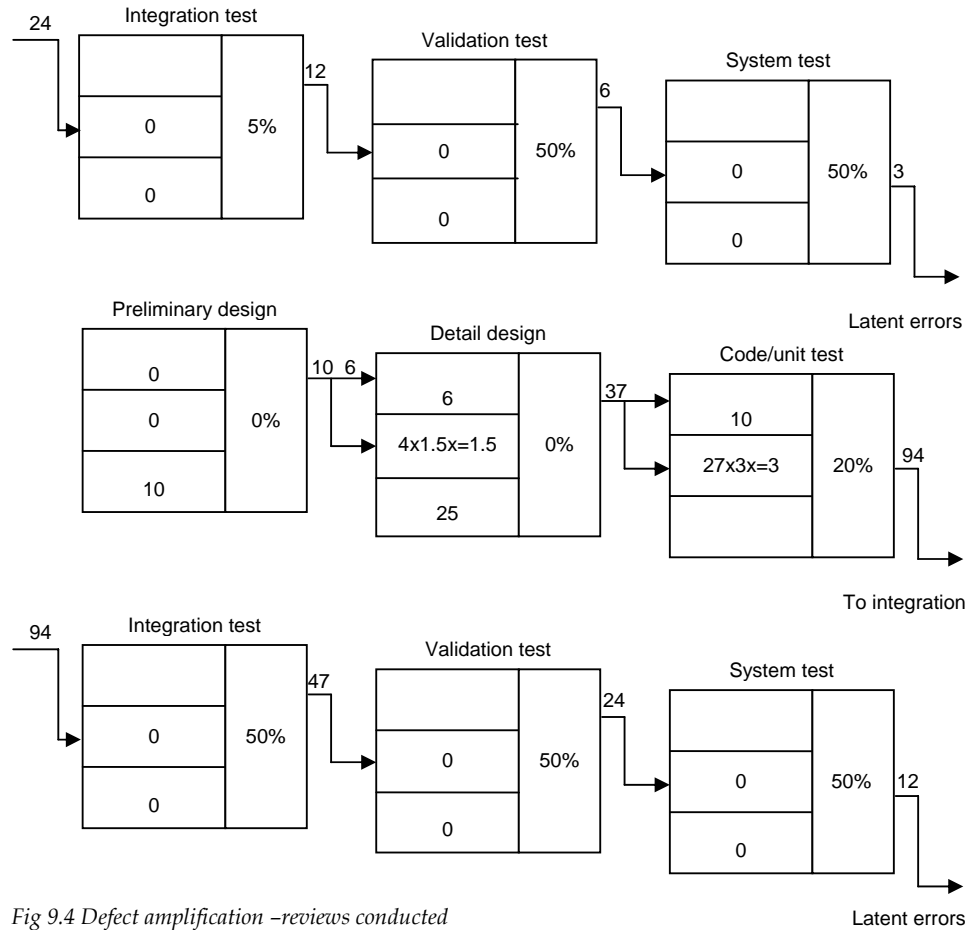


Fig 9.4 Defect amplification –reviews conducted

9.6 Short Summary

- Software quality assurance is an “umbrella activity” that is applied at each step in the software process, SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.
- SQA is complicated by the complex nature of software quality –an attribute of computer programs that is defined as “conformance to explicitly and implicitly defined requirements.” But when considered more generally, software quality encompasses many different product and process factors and related metrics.

9.7 Brain Strom

1. Explain briefly about Quality Concept ?
2. What is SQA? Explain it in a detailed manner ?
3. Give short note on Software Reviews ?

Lecture 10

Software Quality Assurance - II

Objectives

In this lecture you will learn the following

- ✎ About Formal Technical Reviews
- ✎ About Formal Approaches to SQA
- ✎ About SQA Plan

Coverage Plan

Lecture 10

- 10.1 Snap Shot
- 10.2 Formal Technical Reviews
- 10.3 Formal Approaches to SQA
- 10.4 Statistical Quality Assurance
- 10.5 Software Reliability
- 10.6 The SQA Plan
- 10.7 The ISO 9000 Quality Standards
- 10.8 Short Summary
- 10.9 Brain Storm

10.1 Snap Shot

In this lecture we are going to learn about the Formal Technical Reviews. Software Reliability, SQA plan and also about ISO9000 Quality standards.

10.2 Formal Technical Reviews

A formal technical review (FTR) is a software quality assurance activity is performed by software engineers. The objectives of the FTR are (1) to

Development Cost Comparison

Errors Found	Number	Cost Unit	Total
Reviews conducted			
During design	22	1.5	33
Before test	36	6.5	234
During test	15	15	315
After release	3	67	<u>201</u>
			783
No Reviews Conducted			
Before test	22	6.5	143
During test	82	15	1230
After release	12	67	<u>804</u>
			2177

Table 10.1 Development cost comparison

Cover errors in function logic, or implementation for any representation of the software; (2) to verify that the software under review meets its requirements (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

The FTR is actually a class of reviews that include walkthroughs, inspections, round-robin reviews, and other small group technical assessments of software. Each ETR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the paragraphs that follow, guidelines similar to those for a walkthrough [ERE90], [GIL 93] are presented as a representative formal technical review.

The Review Meeting

Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review;

- Advance preparation should occur but should require no more than two hours of work for each person; and
- The duration of the review meeting should be less than two hours.

Given the above constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example rather than attempting to review an entire design, walkthroughs are conducted for each module or small group of modules. With a narrower focus, the FTR has a higher likelihood of uncovering errors.

The focus of the FTR is on a work product—a component of the software (e.g., a portion of a requirements specification a detailed module design, a source code listing for a module) . The individual who has developed the work product—at the producer –inform the project leader that the work product is complete and that a review is required. The project leader contacts a review leader, who evaluates the work product for readiness. Generates copies, and distributes them to two or three reviewers for advance preparation. Each reviewer is expected to spend between one and two hours reviewing the work product, making notes and otherwise becoming familiar with the work. Concurrently, the review leader also reviews the work product and establishes an end a for the review meeting which is typically scheduled for the next day.

The review meeting is attended by the review leader, all reviewers and the producer. One of the reviewers takes on the role of the recorder, that is, the individual who records (in writing) all important issues raised during the review. The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to “walk through” the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered the recorder notes each.

At the end of the review, all attendees of the FTR must decide whether to (1) accept the work product without further modification. (2) reject the work product due to severe errors (once corrected, another review must be performed) or (3) accept the work product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required). The decision made, all FTR attendees complete a sign-off indicating their participation in the review and their concurrence with the review team’s findings.

Review Reporting and Record Keeping

During the FTR a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a simple review summary report is completed. A review summary report answers three questions:

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The review summary report is typically a single page form (with possible attachments) . It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

The review issues list serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can “fall between the cracks.” One approach is to assign the responsibility for follow-up to the review leader. A more formal approach assigns responsibility to an independent SQA group.

Review Guidelines

Guidelines for the conduct of formal technical reviews must be established in advance, distributed to all reviewers, agreed upon and then followed. A review that is uncontrolled can often be worse than no review at all.

The following represents a minimum set of guidelines for formal technical reviews:

1. Review the product, not the producer. An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; and the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.
2. Set an agenda and maintain it. One of the key maladies of meetings of all types is drift. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.
3. Limit debate and rebuttal. When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.
4. Enunciate problem areas, but don't attempt to solve every problem noted. A review is not a problem solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. Take written notes. It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and prioritization can be assessed by other reviewers as information is recorded.
6. Limit the number of participants and insist upon advance preparation. Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material).
7. Develop a checklist for each work product that is likely to be reviewed. A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, coding, and even test documents.

8. Allocate resources and time schedule for FTRs. For reviews to be effective, they should be scheduled as a task during the software engineering process. In addition time should be scheduled for the inevitable modification that will occur as the result of an FTR.
9. Conduct meaningful training for all reviewers. To be effective all review participants should receive some formal training. The training should stress both process related issues and the human psychological side of reviews. Freedman and Weinberg [FRE90] estimate a one month learning curve for every 20 people who are to participate effectively in reviews.
10. Review your early reviews. Debriefing can be beneficial in uncovering problems with the review process itself. The very first work product to be reviewed might be the review guidelines themselves.

Because there are many variables (e.g., number of participants, type of work products, timing and length, specific review approach) that have an impact on a successful review a software organization should experiment to determine what approach works best in a local context. Porter [POR95] and his colleagues provide excellent guidance for this type of experimentation

10.3 Formal Approaches To SQA

In preceding sections, we have argued that software quality is everyone's job and that it can be achieved through competent analysis, design, coding, and testing as well as through the application of formal technical reviews, a multi tiered testing strategy, better control of software documentation and the changes made to it and the application of accepted software development standards. In addition, quality can be defined in terms of a broad array of quality factors and measured (indirectly) using a variety of indices and metrics.

Over the past two decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required . It can be argued that a computer program is a mathematical object. A rigorous syntax and semantics can be defined for every programming language and a similarly rigorous approach to the specification of software requirements is also available. Once the requirements model (specification) has been represented in a rigorous manner, mathematic proofs of correctness can be applied to demonstrate that a program conforms exactly to its specification.

Extensive information on review checklists can be found at the FTR Archive on the wall wide web. See "Further Readings and Other Information Sources" for additional information.

Attempts to prove programs correct are not new. Dijkstra [DIJ76] and Linger et al. [LIN79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts. Today , a number of different approaches to formal proof of correctness have been proposed.

10.4 Statistical Quality Assurance

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software defects is collected and categorized.

2. An attempt is made to trace each defect to its underlying cause (e.g., nonconformance to specification, design error, violation of standards, poor communication with customer)
- 3 Using the pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the “vital few”).
- 4 Once the vital few causes have been identified, move to correct the problems that have caused the defects.

This relatively simple concept represents an important step toward the creation of an adaptive software engineering process in which changes are made to improve those elements of the process that introduce error.

To illustrate the process, assume that a software development organization collects information on defects for a period of one year. Some errors are uncovered as software is being developed. Other defects are encountered after the software has been released to its end user. Although hundreds of different errors are uncovered, all can be tracked to one (or more) of the following causes:

- Incomplete or erroneous specification (IES)
- Misinterpretation of customer communication (MCC)
- Intentional deviation from specification (IDS)
- Violation of programming standards (VPS)
- Error in data representation (EDR)
- Inconsistent module interface (IMI)
- Error in design logic (EDL)
- Incomplete or erroneous testing (IET)
- Inaccurate or incomplete documentation (IID)
- Error in programming language translation of design (PLT)
- Ambiguous or inconsistent human-computer interface (HCI)
- Miscellaneous (MIS).

To apply statistical SQA, Table 10.2 is built. The table indicates that IES, MCC, and EDR are the vital few cause that account for 53 percent of all errors. It should be noted however that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few cases are determined, the software development organization can begin corrective action. For example, to correct MCC the software developer might implement facilitated application specification techniques to improve the quality of customer communication and specification. To improve EDR the developer might acquire CASE tools for data modeling and perform more stringent data design reviews. As the vital few causes are corrected, new candidates pop to the top of the stack.

In conjunction with the collection of defect information, software developers can calculate an error index (EI) for each major step in the software engineering process [IEE94]. After analysis, design, coding, testing, and release, the following data are gathered:

E_i = the total number of errors uncovered during the i th step in the software Engineering process

S_i =the number of serious errors

M_i =the number of moderate errors

T_i =the number of minor errors

PS =size of the product (LOC design statements pages of documentation) at the i th step

Data Collection Statistical SQA

Error	Total		Serious		Moderate		Minor	
	No	%	No	%	No	%	No	%
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
IMI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
<u>MIS</u>	<u>56</u>	<u>6%</u>	<u>0</u>	<u>0%</u>	<u>15</u>	<u>4%</u>	<u>41</u>	<u>9%</u>
Totals	942	100%	128	100%	379	100%	435	100%

Table 10.2 Data Collection for statistical SQA

W_s , W_m , W_t =weighting factors for serious moderate and trivial errors where recommended values are $W_s = 10$, $W_m = 3$, $W_t = 1$. The weighting factors for each phase should become larger as development progresses. This rewards an organization that finds errors early.

At each step in the software engineering process, a phase index PI_i is computed.:

$$PI_i = W_s (S/E_i) + W_t (T/E_i)$$

The error index (EI) is computed by calculating the cumulative effect or earn P_{ii} weighting errors encountered later in the software engineering process more heavily than those encountered earlier.

$$EI = \frac{\sum (I \cdot PI)}{PS} \\ = (PI_1 + 2PI_2 + 3PI_3 + \dots + iPI_i) / PS$$

The error index can be used in conjunction with information collected in Table 10.2 to develop an overall indication of improvement in software quality.

The application of statistical SQA and the pareto principle can be summarized in a single sentence. Spend your time focusing on things that, really matter but first be sure that you understand what really matters ! Experienced industry practitioners agree that most really difficult defects can be traced to a relatively limited number of root causes. In fact, most practitioners have an intuitive feeling for the “real” causes of software defects , but few have spent, time collecting data to support their feelings. By performing the

basic steps of statistical SQA the vital few causes for defects can be isolated and appropriate corrections can be made.

A comprehensive discussion of statistical SQA is beyond the scope of this book. Interested readers should see [SCH87], [KAP95], or [KAN95].

10.5 Software Reliability

There is no doubt that the reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable.

Software reliability, unlike many other quality factors, can be measured, directed, and estimated using historical and developmental data. Software reliability is defined in statistical terms as “the probability of failure free operation of a computer program in a specified environment for a specified time” [MUS87]. To illustrate, program X is estimated to have a reliability of 0.96 over eight elapsed processing hours. In other words, if program X were to be executed 100 times and require eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 96 times out of 100.

Whenever software reliability is discussed, a pivotal question arises: What is meant by the term “failure”? In the context of any discussion of software quality and reliability, a failure is nonconformance to software requirements. Yet, even within this definition there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds, while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

Measures of Reliability and Availability

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory (e.g., [ALV64] to the prediction of software reliability). Most hardware related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature corrosion, shock) are more likely than a design related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture.

There is still debate over the relationship between key concepts in hardware reliability and their applicability to software (e.g., [LIT89], [ROO90]), although an irrefutable link has yet to be established, it is worthwhile to consider a few simple concepts that apply to both system elements.

If we consider a computer-based system, a simple measure of reliability is mean time between failure (MTBF), where

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

(The acronyms MTTF and MTTR are mean time to failure and mean time to repair, respectively)

Many researchers argue that MTBF is a far more useful measure than defects/KLOC. Stated simply an end user is concerned with failures, not with the total defect count. Because each defect contained within a program does not have the same failure rate, the total defect count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 14 months. Many

defects in this program may remain undetected for decades before they are discovered. The MTBF of such obscure defect might be 50 or even 100 years. Other defects in as yet undiscovered, might have a failure rate of 18 or 24 months. Even if every one of the first category of defects (those with long MTBF) is removed, the impact on software reliability is negligible.

In addition to a reliability measure, we must develop a measure of availability. Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as:

$$\text{Availability} = \text{MTTE} / (\text{MTTE} + \text{MTTR}) * 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

Software Safety and Hazard Analysis

Leveson [LEV86] discusses the impact of software in safety critical system when she writes:

Before software was used in safety critical systems, they were often controlled by conventional (nonprogrammable) mechanical and electronic devices. System safety Techniques are designed to cope with Random failure in these [nonprogrammable] systems. Human design errors are not considered since it is assumed that all faults caused by human errors can be avoided completely or removed prior to delivery and operation.

When software is used as part of the control system, complexity can increase by an order of magnitude or more. Subtle design faults induced by human error something that can be uncovered and eliminated in hardware-based conventional control- become much more difficult to uncover when software is used.

Software safety and hazard analysis are software quality assurance activities that focus on the identification and assessment of potential hazards that may impact software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk. For example, some of the hazards associated with a computer-based cruise control for an automobile might be:

- Cause uncontrolled acceleration that cannot be stopped
- Does not disengage when the brake pedal is depressed
- Does not engage when switch is activated
- Slowly loses or gains speed

Once these system-level hazards are identified analysis techniques are used to assign severity and probability of occurrence. To be effective, software must be analyzed in the context of the entire system. For example, a subtle user input error (people are system components) may be magnified by a software fault to produce control data that improperly positions a mechanical device. If a set of external environmental conditions are met (and only if they are met), the improper position of the mechanical

device will cause a disastrous failure. Analysis techniques such as fault tree analysis [VES81] can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

Fault tree analysis builds a graphical model of the sequential and concurrent combinations of events that can lead to a hazardous event or system state. Using a well-developed fault tree, it is possible to observe the consequences of a sequence of interrelated failures that occur in different system components. Real-time logic (RTL) builds a system model by specifying events and corresponding actions. The event-action model can be analyzed using logic operations to test safety assertions about system components and their timing. Petrinet models can be used to determine the faults that are most hazardous.

This approach is analogous to the risk analysis approach described for software project management. The primary difference is the emphasis on technology issues as opposed to project related topics.

Once hazards are identified and analyzed, safety related requirements can be specified for the software. That is the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is failures are not considered in a vacuum but are evaluated in the context of an entire computer-based system.

A comprehensive discussion of software safety and hazard analysis is beyond the scope of this book. Those readers with further interest should refer to Leveson's [LEV95] book on the subject.

10.6 The SQA Plan

The SQA plan provides a road map for instituting software quality assurance. Developed by the SQA group and the project team, the plan serves as a template for SQA activities that are instituted for each software project.

Figure 10.3 presents an outline for SQA plans recommended by the IEEE [IEEE94]. Initial sections describe the purpose and scope of the document and indicate those software process activities that are covered by quality assurance. All documents noted in the SQA plan are listed and all applicable standards are noted. The Management section of the plan describes SQA's place in the organizational structure; SQA tasks and activities and their placement throughout the software process; and the organizational roles and responsibilities relative to product quality.

The Documentation section describes (by reference) each of the work products produced as part of the software process. These include:

- Project documents (e.g., project plan)
- Models (e.g., ERDs, class hierarchies)
- Technical documents (e.g., specifications, test plans)
- User documents (e.g., help files)

In addition this section defines the minimum set of work products that are acceptable to achieve high quality.

Standards, Practices, and Conventions lists all applicable standards/practices that are applied during the software process (e.g., document standards, coding standards, and review guidelines. In addition all project, process, and (in some instances) product metrics that are to be collected as part of software engineering work are listed.

The Reviews and Audits section of the plan identifies the reviews and audits to be conducted by the software engineering team, the SQA group and the customer. It provides an overview of the approach for each review and audit.

- I. Purpose of plan
- II. References
- III. Management
 1. Organization
 2. Tasks
 3. Responsibilities
- IV. Documentation
 1. Purpose
 2. Required software engineering documents
 3. Other documents
- V. Standards, Practices, and Conventions
 1. Purpose
 2. Conventions
- VI. Reviews and Audits
 1. Purpose
 2. Review Requirements
 - a. software requirements review
 - b. design reviews
 - c. software verification and validation reviews
 - d. functional audit
 - e. physical audit
 - f. in-process audits
 - g. management reviews
- VII. Test
- VIII. Problem Reporting and Corrective Action
- IX. Tools, Techniques, and Methodologies
- X. Code Control
- XI. Media Control
- XII. Supplier Control
- XIII. Records Collection, Maintenance, and Retention
- XIV. Training
- XV. Risk Management

Fig 10.3. ANSI/IEEE Std 730 1984 and 983-1986 Software quality Assurance plans.

The Test section references the software test plan and procedure. It also defines test record-keeping requirements, Problem Reporting and Corrective Action defines procedures for reporting, tracking, and resolving errors and defects, and identifies the organizational responsibilities for these activities.

The remainder of the SQA plan identifies the tools and methods that support SQA activities and tasks; references software configuration management procedures for controlling change; defines a contract management approach; establishes methods for assembling, safeguarding, and maintaining all records; identifies training required to meet the needs of the plan, and defines methods for identifying, assessing, monitoring, and controlling risks.

10.7 The ISO 9000 Quality Standards⁶

A Quality assurance system may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management [ANS87] ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

To become registered to one of the quality assurance system models contained in ISO 9000 a company's quality system and operations are scrutinized by third party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semiannual surveillance audits ensure continued compliance to the standard.

The ISO Approach to Quality Assurance Systems

The ISO 9000 quality assurance models treat an enterprise as a network of interconnected processes. For a quality system to be ISO-compliant, these processes must address the areas identified in the standard and must be documented and practiced as described. Documenting a process helps an organization understand, control, and improve it. It is the opportunity to understand control and improve the process network that offers, perhaps, the greatest benefit to organizations that design and implement ISO-compliant quality systems.

ISO 9000 describes the elements of a quality assurance system in general terms. These elements include the organizational structure, procedures, processes, and resources needed to implement quality planning, quality control, quality assurance, and quality improvement. However, ISO 9000 does not describe how an organization should implement these quality system elements. Consequently, the challenge lies in designing and implementing a quality assurance system that meets the standard and fits the company's products, services, and culture.

The ISO 9001 Standard

ISO 9001 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001 standard is applicable to all engineering disciplines, a special set of ISO guidelines (ISO 9000-3) have been developed to help interest the standard for use in the software process.

This section written by Michael Stovsky has been adapted from "Fundamentals of ISO 9000 and "ISO 9001 Standard," workbooks developed for Essential Software Engineering, a video curriculum developed by R.S. Pressman & Associates, Inc Reprinted with permission.

The 20 requirements delineated by ISO 9001 address the following topics:

1. Management responsibility
2. Quality system

3. Contract review
4. Design control
5. Document and data control
6. Purchasing
7. Control of customer supplied product
8. Product identification and tractability
9. Process control
10. Inspection and testing
11. Control of inspection, measuring, and test equipment
12. Inspection and test status
13. Control of nonconforming product
14. Corrective and preventive action
15. Handling, storage, packaging, preservation, and delivery
16. Control of quality records
17. Internal quality audits
18. Training
19. Servicing
20. Statistical techniques

In order for a software organization to become registered to ISO 9001 it, must establish policies and procedures to address each of the requirements noted above and then be able to demonstrate that these policies and procedures are being followed. For further information on ISO 9001 the interested reader should see [SCH94] and [ESE95].

10.8 Short Summary

- Software reviews are one of the most important SQA activities. Reviews serve as a filter for the software process, removing errors while they are relatively inexpensive to find and correct. The formal technical review or walk through is a stylized review meeting that has been shown to be extremely effective in uncovering errors.
- To properly conduct software quality assurance data about the software engineering process should be collected, evaluated, and disseminated.
- Statistical SQA helps to improve the quality of the product and the software process itself.
- Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.
- In summary, we recall the words of Dunn and Ullman [DUN82] “Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering” The ability to ensure quality is the measure of a mature engineering discipline. When the mapping alluded to above is successfully accomplished, mature software engineering is the result.

10.9 Brain Storm

1. Discuss about Formal Technical Reviews ?
2. What is Statistical Quality Assurance ?
3. Explain briefly about the SQA plan ?
4. Discuss about ISO 9000 standards ?

❧

Lecture 11

Software Configuration Management

Objectives

In this lecture you will learn the following

- ✧ About SCM Process
- ✧ About Version Control
- ✧ About Change Control

Coverage Plan

Lecture 11

- | |
|---|
| 11.1 Snap Shot |
| 11.2 Software Configuration Management. |
| 11.3 The SCM Process |
| 11.4 Identification of Objects in the Software Configuration. |
| 11.5 Version Control |
| 11.6 Change Control |
| 11.7 Configuration Audit |
| 11.8 Status Reporting |
| 11.9 SCM Standards |
| 11.10 Short Summary |
| 11.11 Brain Storm |

11.1 Snap Shot

Software configuration management is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change (2) control change, (3) ensure that change is being properly implemented and (4) report change to others who may have an interest.

It is important to make a clear distinction between software maintenance and software configuration management. Maintenance is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that begin when a software project begins and terminate only when the software is taken out of operation

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. In this chapter we discuss the specific activities that enable us to manage change.

11.2 Software Configuration Management.

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source-level and executable forms) (2) documents that describe the computer programs (targeted at both technical practitioners and users), and (3) data (contained within the program or external to it) The items that comprise all information produced as part of the software process are collectively called a software configuration.

As the software process progresses, the number of software configuration items (SCIs) grows rapidly. A system specification spawns a software project plan and software requirements specification (as well as hardware related documents). These in turn spawn other documents to create a hierarchy of information. If each SCI simply spawned other SCIs little confusion would result. Unfortunately, another variable enters the process—change. Change may occur at any time for any reason. In fact the First Law of System Engineering [BER80] states: No matter where you are in the system life cycle the system will change and the desire to change it will persist throughout the life cycle.

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However there are four fundamental sources of changes:

- New business or market conditions that dictate changes in product requirements or business rules.
- New customer needs that demand modification of data produced by information systems, functionality delivered by products or services delivered by a computer-based system.
- Reorganization and /or business downsizing that causes changes in project priorities of software engineering team structure.
- Budgetary or scheduling constraints management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we examine major SCM tasks and important concepts that help us to manage change.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance

activity that is applied throughout the software process. In the sections that follow , we examine major SCM tasks and important concepts that help us to manage change.

Baselines

Change is a fact of life in software development. Customers want to modify requirements. Developers want to ,modify technical approach. Managers want to modify project approach. Why all this modification? The answer is really quite simple. As time passes all constituencies know more (about what they need, which approach would be best and how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: Most changes are justified!

A baseline is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE defines a baseline as:

A specification or product that has been formally reviewed and agreed upon that thereafter serves as the basis for further development and that can be changed only through formal change control procedures.

One way to describe a baseline is through analogy: Consider the doors to the kitchen of a large restaurant. To eliminate collisions one door is marked OUT and the other is marked IN. The doors have stops that allow them to be opened only in the appropriate direction.

If a waiter picks up an order in the kitchen, places it on a tray and then realizes he has selected the wrong dish, he may change to the correct dish quickly and informally before he leaves the kitchen.

If, however he leaves the kitchen gives the customer the dish and then is informed of his error he must follow a set procedure: (1) look at the check to determine if an error has occurred; (2) apologize profusely; (3) return to the kitchen through the IN door. (4) explain the problem and so forth.

A baseline is analogous to a dish as it passes through the kitchen door in the restaurant. Before a software configuration item becomes a baseline, change may be made quickly and informally. However once a baseline change may be made quickly and informally. However once a baseline is established we figuratively pass through a swinging one-way door. Changes can be made but a specific formal procedure must be applied to evaluate and verify each change.

In the context of software engineering a baseline is a milestone in the development of software of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review . For example the elements of a design specification have been documented and reviewed. Errors are found and corrected. Once all parts of the specification have been reviewed corrected and then approved the design specification becomes a baseline. Further changes to the after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure11.1.

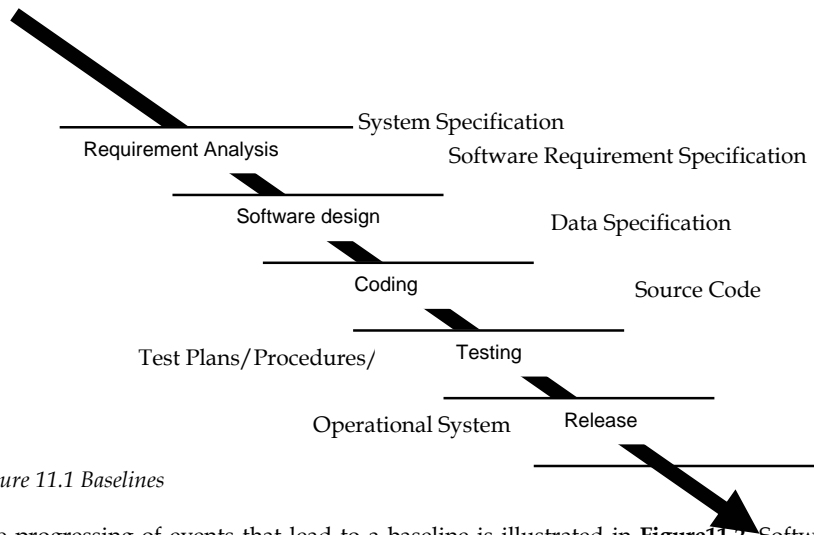


Figure 11.1 Baselines

The progressing of events that lead to a baseline is illustrated in **Figure 11.2**. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a project database (also called a project library or software repository). When a member of a software engineering team wants to make a modification to a baselined SCI it is copied from the project database into the engineer's private work space. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The dashed arrows noted in **Figure 11.2** illustrate the modification path for a baselined SCI.

Software Configuration Items

We have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme an SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically an SCI is a document an entire suite of test cases or a named program component (e.g., a C++ function or an Ada95 package).

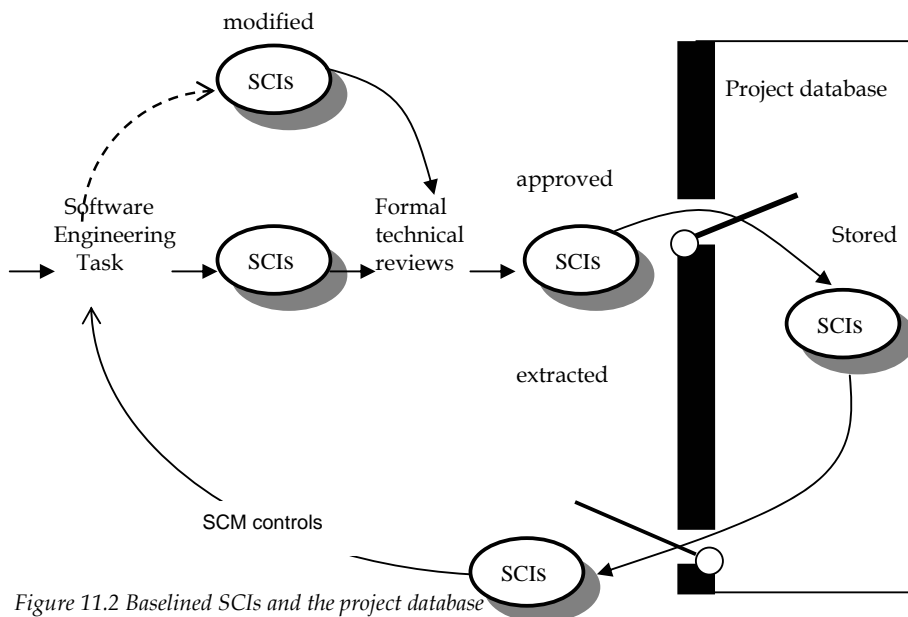


Figure 11.2 Baselined SCIs and the project database

The following SCIs become the target for configuration management techniques and form a set of baselines:

1. System Specification
2. Software Project Plan
3. Software Requirements Specification
 - a. Graphical analysis models
 - b. Process specifications
 - c. Prototype
 - d. Mathematical specification
4. Preliminary User Manual
5. Design Specification
 - a. Data design description
 - b. Architectural design description
 - c. Module design descriptions
 - d. Interface design descriptions
 - e. Object descriptions(if object-oriented techniques are used)
6. Source Code Listing
7. Test Specification
 - a. Test plan and procedure
 - b. Test cases and recorded results
8. Operation and Installation Manuals
9. Executable Program
 - a. Module executable code
 - b. Linked modules
10. Database Description
 - a. Schema and file structure
 - b. Initial Content
11. As-built User Manual
12. Maintenance Documents
 - a. Software problem reports
 - b. Maintenance requests
 - c. Engineering change orders
13. Standards and Procedures for Software Engineering

In addition to the SCIs noted above , many software engineering organizations also place software tools under configuration control. That is Specific versions of editors, compilers and other CASE tools are “frozen” as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare it is possible that a new version of a tool(e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form configuration objects that may be catalogued in the project database with a single name. A configuration object has a name, attributes and is “connected” to other objects by relationships. In **Figure 11.3** the configuration objects **design specification, data model, module N, source code, and test specification** are each defined separately. However each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is **data model** and

module N are part of the object **design specification**. A double headed straight arrow indicates an interrelationship. If a change were made to the **source code** object interrelationships enable a software engineer to determine what other objects (and SCIs) might be affected.

11.3 The SCM Process

Software configuration management is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is also responsible for the identification of individual SCIs and various versions of the software the auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration.

Any discussion of SCM introduces a set of complex questions:

- How does an organization identify and manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?.
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and prioritizing changes?
- How can we assure that changes have been made properly?
- What mechanism is used to apprise others of changes that are made?

These questions lead us to the definition of five SCM tasks: identification, version control, change control, configuration auditing and reporting.

11.4 Identification of Objects in the Software Configuration

To control and manage software configuration items each must be separately named and then organized using an object-oriented approach. Two types of objects can be identified [CHO89]: basic objects and aggregate objects. A basic objects air a “unit of text” that has been created by a software engineer during analysis, design, coding of testing. For example, a basic object might be a section of a requirements specification, a source listing for a module or a suite of test cases that are used to exercise the code. An aggregate object is a collection of basic objects and other aggregate objects. In **Figure 11.3 design specification** is an aggregate object. Conceptually it can be viewed as a named list of pointers that specify basic objects such as **data model** and **module N**.

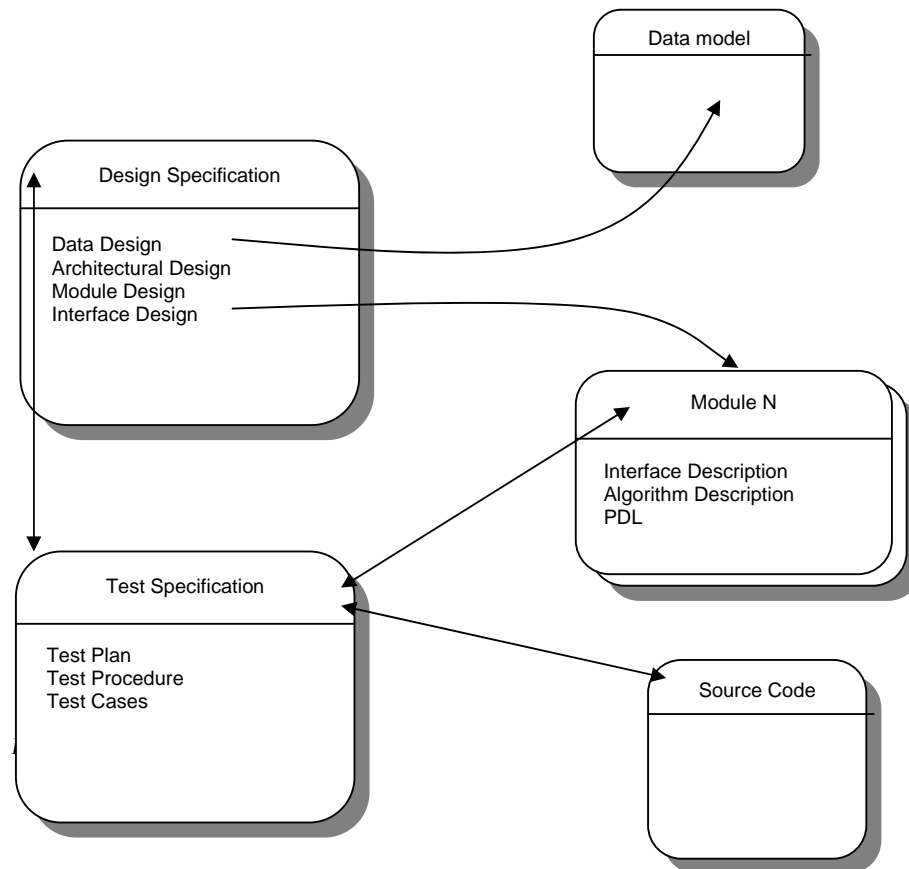
Each object has a set of distinct features that identify it uniquely: a name. A description, a list of resources and a “realization”; the object name is a character string that identifies the object unambiguously. The object description is a list of data items that identify:

- The SCI type (e.g., document, program, data) that is represented by the object;
- A project identifier; and change and /or version information.

Resources are “entities that are provided, processed, referenced or otherwise required by the object” [CHO 89]. For example data types, specific functions or even variable names may be considered to be object and null for an aggregate object.

Configuration object identification must also consider the relationships that exist between named objects. An object can be identified as **<part-of>** defines a hierarchy of objects. For example using the simple notation.

E-R diagram 1-4 <part-of> data model;
Data model <part-of> Design Specification;
We create a hierarchy of SCIs.



It is unrealistic to assume that the only relationship among objects in an object hierarchy are along direct paths of the hierarchical tree. In many cases, objects are interrelated across branches of the objects hierarchy. For example, **data model** is interrelated to data flow diagrams (assuming the use of structured analysis) and also interrelated to a set of test cases for a specific equivalence class. These cross-structural relationships can be represented in the following manner:

Data model <interrelated> data flow model;
Data model <interrelated> test case class m;

In the first case, the interrelationship is between a composite object while the second relationship is between an aggregate object (**data model**), and a basic object(**test case class m**).

The interrelationship between configuration objects can be represented with a module interconnection language (MIL) [NAR87]. A MIL description interdependencies among configuration objects and enables any version system to be constructed automatically.

The identification scheme for software objects must recognize that objects evolve throughout the software process. Before an object is baselined, it is change many times and even after a baseline has been established, change may be quite frequent. It is possible to create an evolution graph [GUS89] any object. The evolution graph describes the change history of the objects and comes object 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2 which is followed by a major update that is object 1.2. The evolution or object 1.0 continues through 1.3 and 1.4 but at the same time a major modification to the object results in a new evolutionary path version 2.0 Both versions are currently supported.

It is possible that changes may be made to any version , but not necessarily of all versions. How does the developer reference all modules, documents and test cases for version 1.4? How does the marketing department know what customers currently have version 2.1? How can we be sure that changes to version 2.1 source code are properly reflected in corresponding design documentation? A key element in the answer to all of the above questions is identification.

A variety of automated SCM tools(e.g., CCC, RCS, SCCS, Aide-de-Camp) have been developed to aid in identification (and other SCM) tasks. In some cases a tool is designed to maintain full copies of only the most recent version. To achieve earlier versions (of documents of programs) changes (catalogued by the tool) are “subtracted” from the most recent version[TIC82]. This scheme makes the current configuration immediately available and other versions easily available.

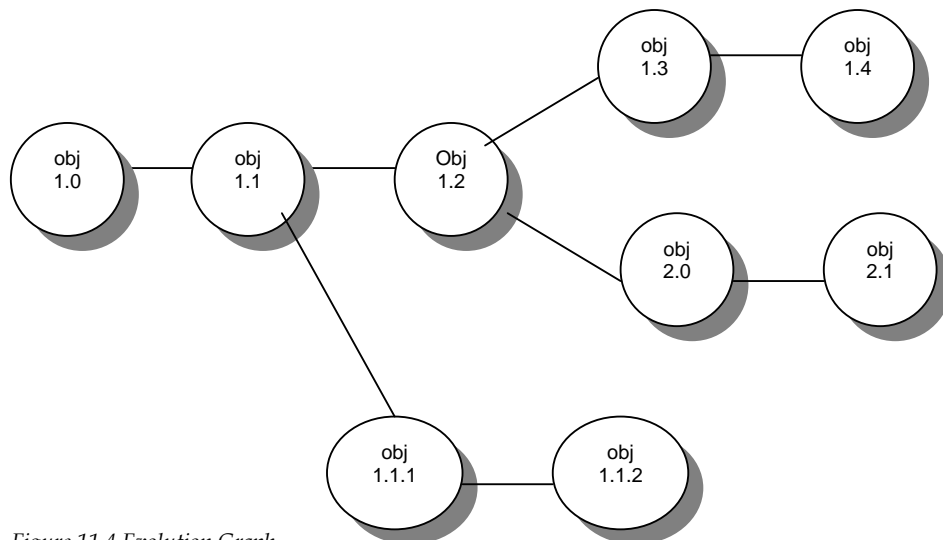


Figure 11.4 Evolution Graph

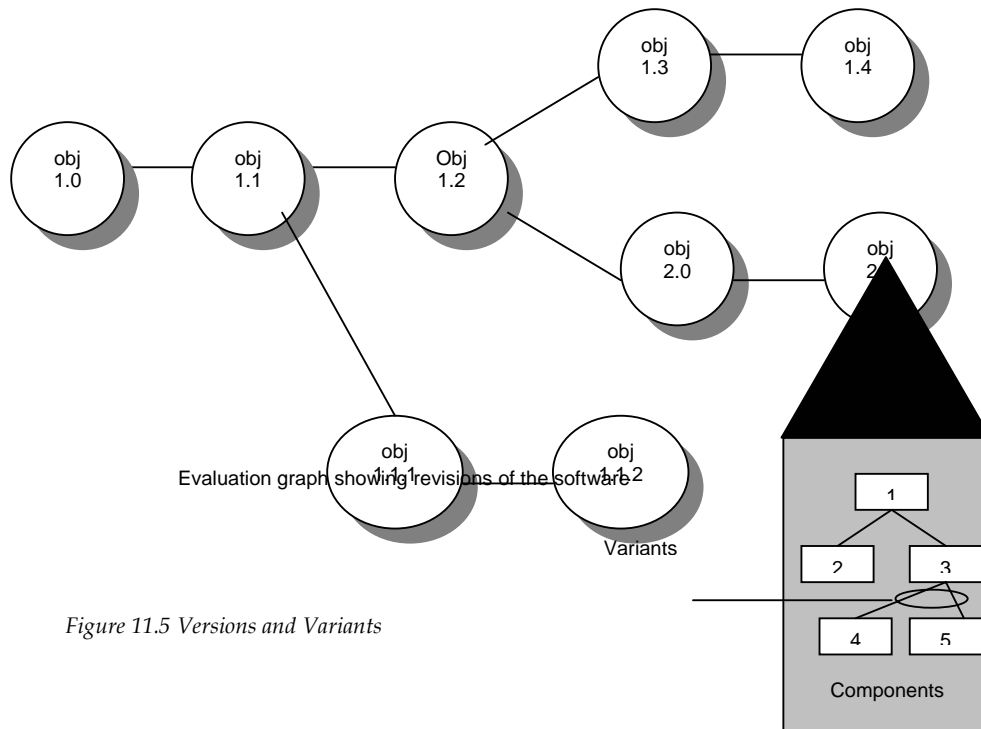


Figure 11.5 Versions and Variants

11.5 Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software engineering process Clemm [CLE89] describes version control in the context of SCM:

Configuration management's allows a user o specify alternative configuration of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version , and then allowing a configuration to be specified [and constructed] by describing the set of desired attributes.

The “attributes” mentioned above can be as simple as a specific version number that is attached to each object or as complex as a string of Boolean variables (switches) that indicate specific types of functional changes that have been applied to the system. [LIE89].

One representation of the different versions of a system is the evolution graph presented in **Figure 11.4** Each node on the graph is an aggregate object that is a complete version of the software. Each versions of the software is a collection of SCIs (source code, documents, data) and each version may be composed of different variants. To illustrate this concept consider a versions f a simple program that is composed of components 1,2,3,4 and 5 (**Figure 11.5**) Component 4 is used only when the software is implemented using color displays. Component 5 is implemented when monochrome displays are available. Therefore two variants of the version can be defined : (1) Components 1,2,3,4; and (2) Components1,2,3 and 5.

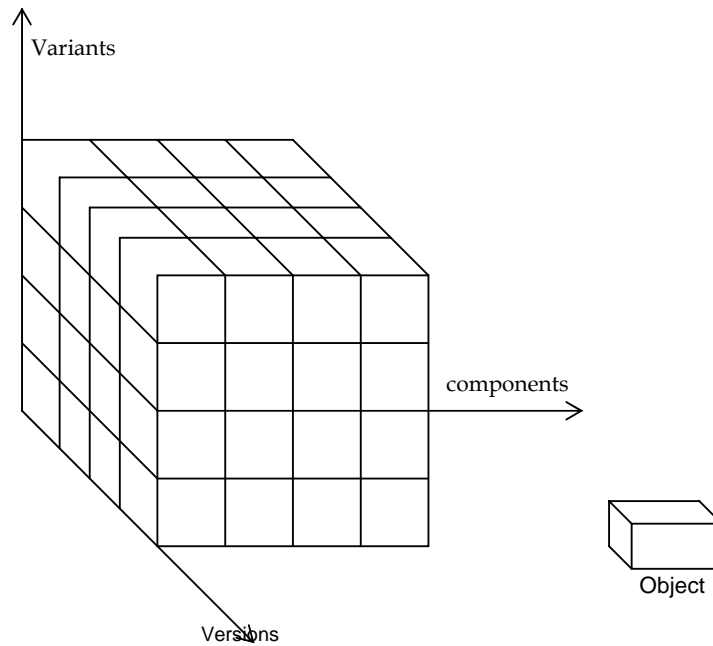


Figure 11.6 Object pool representation of components, variants, and versions

To construct the appropriate variant of a given version of a program, each component can be assigned an “attribute-tuple” a list of features that will define whether the component should be used when a particular variant of a software version is to be constructed. One or more attributes is assigned of each variant. For example a color displays are to be supported.

Another way to conceptualize the relationship between components, variants and version is to represent them as an object pool [REI 89]. As **Figure 11.6** shows the relationship between configuration objects and components, variants and versions can be represented as a three-dimensional space. A component is composed of a collection of objects at the soame revision level. A variant is a different collection of objects at the same revision level and therefore coexists in parallel with other variants. A new version is defined when major changes are made to one or more objects.

A number of different automated approaches to version control have been proposed over the past decade. The primary difference in approcaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process of construction. In early systems such as SCCS [ROC75], attributes took on numeric values. In later systems such as RCS [TIC82], symbolic revision keys were used, Modern systems such as NSE or DSEE [ADA89] create version specifications that can be used to construct variants or new versions. These systems also support the baselining concept, thereby precluding uncontrolled modification (or deletion) of a particular version.

11.6 Change Control

For a large software development project, uncontrolled change rapidly leads to chaos. Change Control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in **figure 11.7** A change request is submitted and evaluated to assess technical merit potential side effects, overall impact on other configuration objects and system functions and the projected cost of the change. The results of the evaluation are presented as a

change report that is used by a change control authority (CCA) a person or group who makes a final decision on the status and priority of the change. An engineering change order (ECO) is generated for each approved change. The ECO describes the change to be made; the constraints that must be respected, and the criteria for review and audit. The object to be changed is “checked out” of the project database the change is made and appropriate SQA activities are applied. The object is then “checked in” to the database and appropriate version control mechanism are used to create the next version of the software.

The “Check in” and “Check out” processes implement two important elements of change control access control and synchronization control. Access control governs which software engineers have the authority to access and modify a particular configuration object. Synchronization control helps to ensure that parallel changes performed by two different people don’t overwrite one another[HAR89].

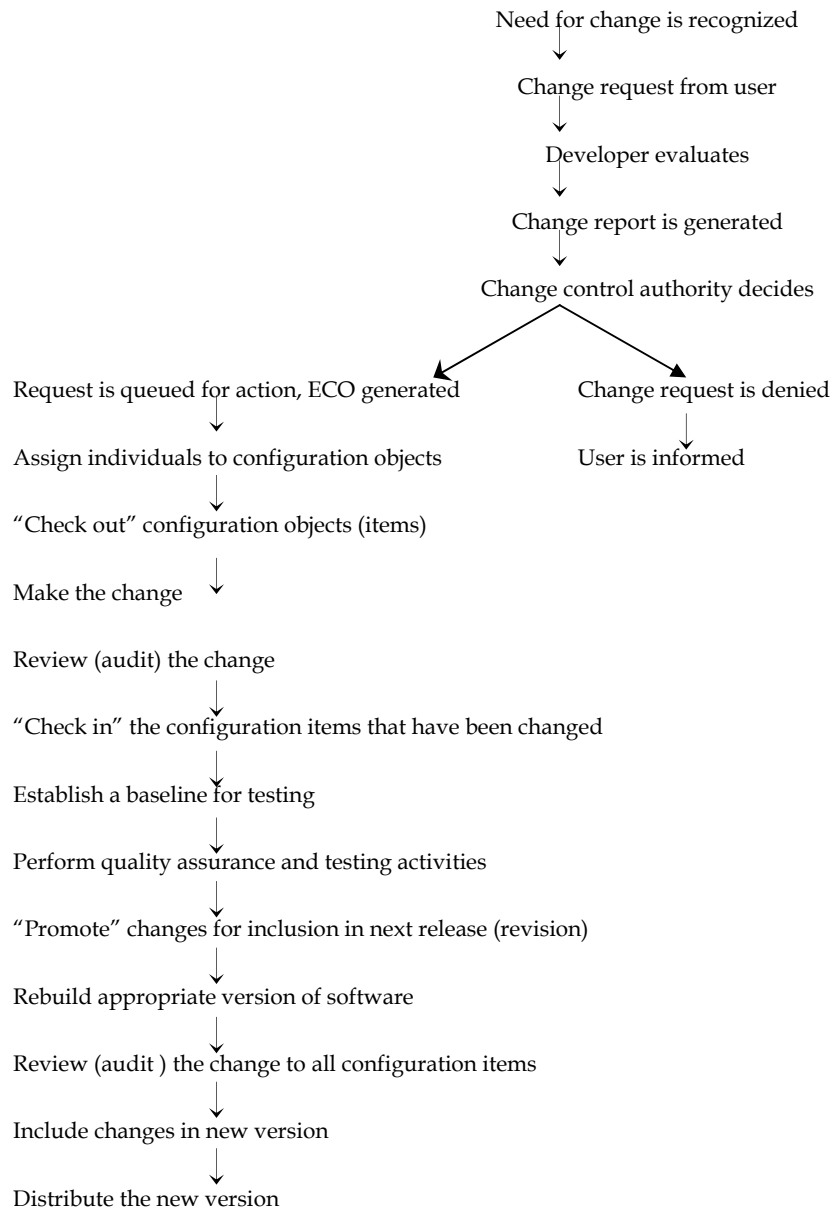


Figure 11.7 The change control process

Access and synchronization control flow is illustrated schematically in **Figure 11.8** based on approved change request and ECO, a software engineer checks out a configuration object. An access control function ensures that the software engineer has authority to check out the object, and synchronization control locks the object in the project database so that no updates can be made to it until the version currently checked out has been replaced. Note that other copies can be checked out, but other updates cannot be made. A copy of the baselined object called the “extracted version” is modified by the software engineer. After appropriate SQA and testing the modified version of the object is checked in and the new baseline object is unlocked.

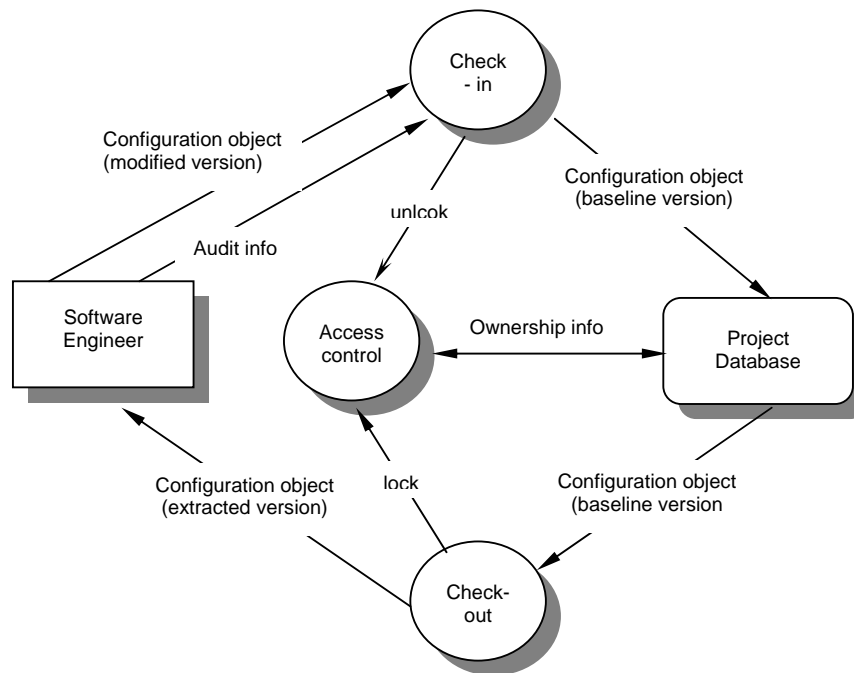


Figure 11.8 Access and synchronization control

Some readers may begin to feel uncomfortable with the level of bureaucracy implied by the change control process description. This feeling is not uncommon. Without proper safeguards changed control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms have created a number of layers of control to help avoid the problem alluded to above.

Prior to an SCI becoming a baseline, only informal change control need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not impact broader system requirements that lie outside the developer's cope of work). Once the object has undergone formal technical review and had been approved, a baseline is created. Once an SCI becomes a baseline project level change control is implemented. Now, to make a change the developer must gain approval form the project manager (if the change is "local") or form the CCA if the change impacts other SCIs. In some cases normal generation of change requests change reports, and ECOs is dispensed with. However, assessment of each change is conducted and all changes are tracked and reviewed.

When the software product is release to customers formal change control is instituted. The formal change control procedure has been outlined in **figure 11.7**.

The change control authority (CCA) plays an active role in the second and third layer of control. Depending on the size and character of a software project, the CCA may be comprised of one person – the project manager – or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing, etc.). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change impact hardware? How will the change impact performance? How will the change modify the customer's perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

11.7 Configuration Audit

Identification, version control, and change control help the software developer to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can we ensure that the change has been properly implemented? The answer is twofold: (1) formal technical reviews and (2) the software configuration audit.

The formal technical review focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, and potential side effects. A formal technical review should be conducted for all but the most trivial changes.

A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a formal technical review been conducted to assess technical correctness?
3. Have software engineering standards been properly followed?
4. Has the change been “highlighted” in the SCI? Have the change date and change author been specified? Do the attributes of the configuration Object reflect the change?
5. Have SCM procedures for noting the change, recording it, and Reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a formal technical review. However, when SCM is a formal activity, the SCM audit is conducted separately by the quality assurance group.

11.8 Status Reporting

Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions; (1) What happened? (2) Who did it?(3) When did it happened ? (4) What else will be affected?.

The flow of information of configuration status reporting is illustrated in **Figure 11.7**. Each time a new or updated identification of a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued) a CSR entry is made. Each time a change is made. Each time a configuration audit is conducted the results are reported as part of the CSR task. Output from CSR may be placed in an on-line database [TAY85] so that software developers or maintainers can access change information by keyword category. In addition a SCR report is generated on a regular basis and is intended to keep management and practitioners apprised of important changes.

Configuration status reporting plays a vital role in the success of a large software development project. When many people are involved it is likely that “the left hand not knowing what the right hand is doing” syndrome will occur. Two developers may attempt to modify the same SCI with different and conflicting intent. A software engineering team may spend months of effort building software to a specification for a proposed change is not aware that the change is being made. CSR helps to eliminate these problems by improving communication among all people involved.

11.9 SCM Standards

Over the past two decades a number of software configuration management standards have been proposed. Many early SCM standards such as MIL-STD-483, DOD-STD-480A and MIL-STD-1512A, focused on software developed for military applications. However more recent ANSI/IEEE standards such as ANSI/IEEE Std. No. 828-1983, Std. No. 1042-1987 and Std. No.1028-1988 [IEEE94] are applicable for commercial software and are recommended for both large and small software engineering organizations.

11.10 Short Summary

- Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies, controls, audits and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All information produced as part of the software process becomes part of a software configuration. The configuration is organized in a manner that enables orderly control of change.
- The software configuration is composed of a set of interrelated objects also called software configuration items that are produced as a result of some software engineering activity. In addition to documents, programs and data, the development environment that is used to create software can also be placed under configuration control.
- Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baselined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Basic and aggregate objects form an object pool from which variants and versions are created. Version control is a set of procedures and tools for managing the use of these objects.
- Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change and culminates with a controlled update of the SCI that is to be changed.

- The configuration audit is an SQA activity that help to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

11.11 Brain storm

1. Explain about Software Configuration Management ?
2. Give a Short Note on SCM Process ?
3. Explain briefly version and change control ?
4. What is Status Reporting ?
5. Describe about SCM standards in a brief manner ?

❧❧

Lecture 12

System Engineering - I

Objectives

In this lecture you will learn the following

- ✧ About System Engineering Hierarchy
- ✧ About Information Engineering

Coverage Plan

Lecture 12

- 12.1 Snap Shot
- 12.2 The System Engineering Hierarchy
- 12.3 System Modeling
- 12.4 Information Engineering : An Overview
- 12.5 Product Engineering : An Overview
- 12.6 Information Engineering
- 12.7 Short Summary
- 12.8 Brain Storm

12.1 Snap Shot

The word “system” is possibly the most overused and abused term in the technical lexicon. We speak of political systems and educational systems of avionics systems and manufacturing systems of banking systems and subway systems. The word tells us little. We use the adjective describing “system” to understand the context in which the word is used. Webster’s Dictionary defines “system” as “ a set or arrangement of things so related as to form a unity or organic whole... a set of facts, principles ,rules etc ., classified and arranged in an orderly form so as to show a logical plan linking the various parts ... a method or plan of classification or arrangement ...an established way of doing something ;method; procedure...” Five additional definitions are provided in the dictionary yet no precise synonym is suggested “system “ is a special word.

Borrowing from Webster’s definition we define a computer-based system as:

A set of arrangement of elements that are organized to accomplish some predefined goal by processing information.

The goal may be to support some business function or to develop a product that can be sold to generate business revenue. To accomplish the goal a computer based system makes use of a variety of system elements:

Software: Computer programs, data structures and related documentation that serve to effect the logical method, procedure or control that is required.

Hardware: Electronic devices that provide computing capability and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function.

People: Users and operators of hardware and software

Database: A large, organized collection of information that is accessed via software.

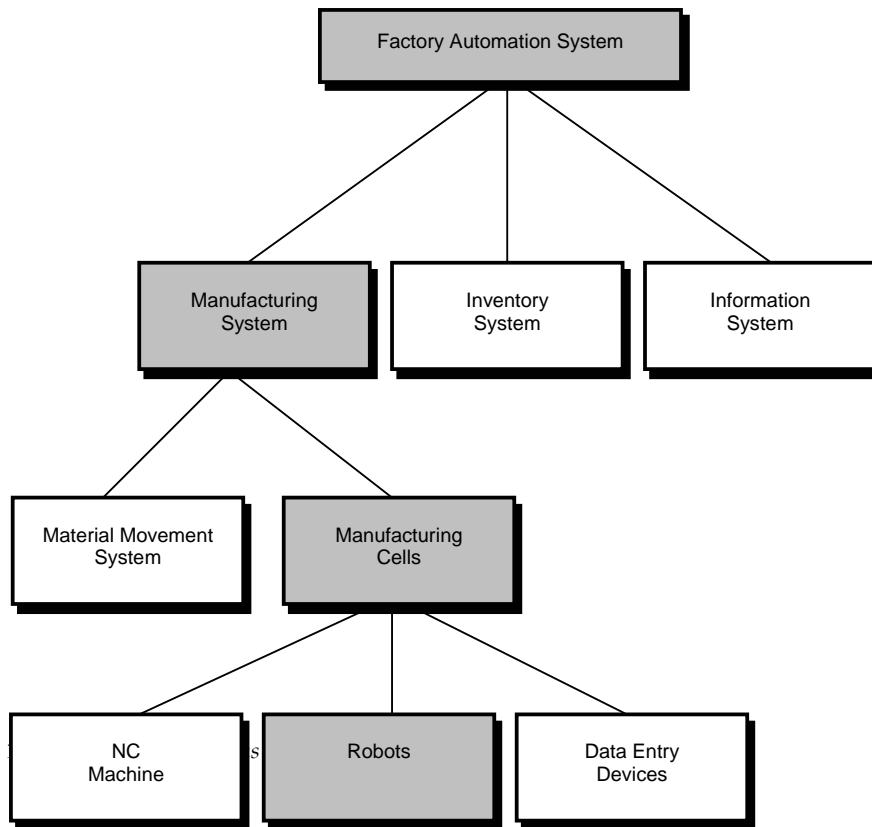
Documentation: Manuals, forms and other descriptive information that portrays the use and /or operation of the system.

Procedures : The steps that define the specific use of each system element or the procedural context in which the system resides.

The elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product and a robot transforms a command file containing specific instruction into a set of control signals that cause some specific physical action. Creating an information system to assist the marketing department and control software to support the robot both require system engineering

One complication characteristic of computer-based systems is that the elements comprising one system may also represent one macro element of a still larger system. The macro element is a computer-based system that is one part of a larger computer-based system. As an example we consider a “factor automation system “ that is essentially a hierarchy of systems shown in the figure 12.11. At the lowest level of the hierarchy we have a numerical control machine robots and data entry devices. Each is a computer-based system in its own right. The elements of the numerical control machine include electronic and electromechanical hardware (e.g., Processor and memory motors, sensors); software (for communications, machine control, and interpolation); people (the machine operator); a database (the stored NC program); and documentation and procedures. A similar decomposition could be applied to the robot and data entry device. Each is a computer-based system.

At the next level in the hierarchy a manufacturing cell is defined. The manufacturing cell is a computer-based system that may have element of its won (e.g., computers, mechanical fixtures) and also integrates the macro elements that we have called numerical control machine, robot, and data entry device.



To summarize, the manufacturing cell and its macro elements each are comprised of system elements with the generic labels software, hardware, people, database, procedures, and documentation. In some cases, macro elements may share a generic element. For example, the robot and the NC machine might both be managed by a single operator (the people element). In other cases, generic elements are exclusive to one system.

The role of the system engineer is to define the elements for a specific computer-based system in the context of the overall hierarchy of systems (macro elements). In the sections that follow, we examine the tasks that constitute computer system engineering.

12.2 The System Engineering Hierarchy

Regardless of its domain of focus, system engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy illustrated in Figure 12.2. The system engineering process usually begins with a “world view” That is the entire business of product domain is examined to ensure that the proper business or technology context established. The world view is refined to focus more fully on specific domain of interest. Within a specific domain, the need for targeted system elements (e.g., data, software, hardware, people) is analyzed. Finally the analysis design and construction of a targeted system element is initiated. At the top of the hierarchy a very broad context is established and at the bottom, detailed technical activities performed by the relevant engineering discipline (e.g., hardware or software engineering) are conducted.

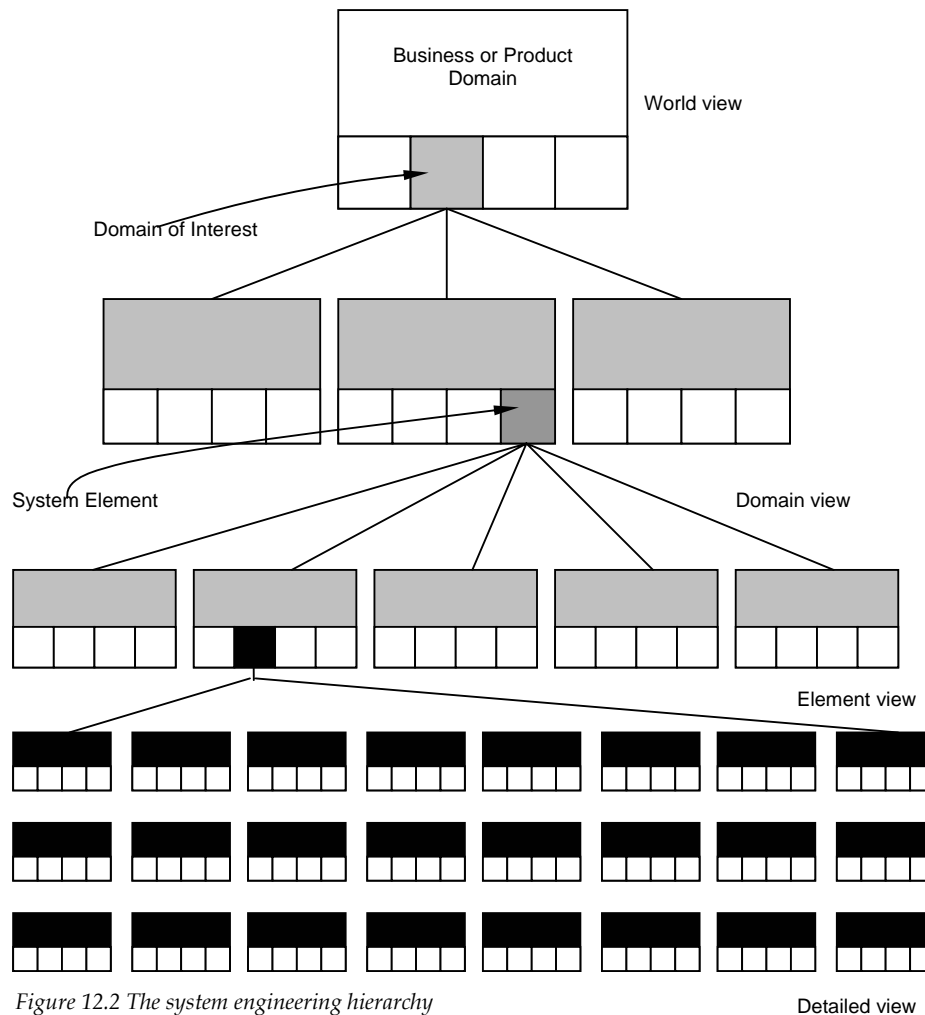


Figure 12.2 The system engineering hierarchy

Stated in a slightly more formal manner the world view (WV) is composed of a set of systems in its own right.

$$WV=\{D1,D2,D3....D_n\}$$

Each domain is composed of specific elements (E_i) each of which serves some role in accomplishing the objectives and goals of the domain:

$$D_i=\{E1,E2,E3....E_m\}$$

Finally, each element is implemented by specifying the technical components(C_k) that achieve the necessary function for an element.

$$E_j=\{C1,C2,C3....C_k\}$$

In the software context, a component could be a computer program a reusable program component, a module, an class or object or even a programming language statement.

It is important to note that the system engineer narrows the focus of work as he or she moves downward in the hierarchy described above. However the world view portrays a clear definition of overall functionality that will enable the engineer to understand the domain and ultimately the system or product in the proper context.

12.3 System Modeling

System engineering is a modeling process. Whether the focus is on the world view or the detailed view the engineer creates models that [MOT92]:

- ❑ Define the processes that serve the needs of the view under consideration
- ❑ Represent the behavior of the processes and the assumptions on which the behavior is based
- ❑ Explicitly define both exogenous and endogenous input to the model
- ❑ Represent all linkages (including output) that will enable the engineer to better understand the view.

To construct a system model, the engineer should consider a number of restraining factors:

1. Assumption that reduce the number of possible permutations and variations thus enabling a model to reflect the problem in a reasonable manner. As an example consider a three-dimensional rendering product used by the entertainment industry to create realistic animation. One domain of the product enables the representation of 3D human forms. Input to this domain encompasses the ability of input movement from a live human "actor" from video or by the creation of graphical models. The system engineering makes certain assumptions about the range of allowable human movement so that the range of inputs and processing can be limited.
2. Simplifications that enable the model to be created in timely manner. To illustrate, consider an office products company that sells and services a broad range of copiers fax machines and related equipment. The system engineer is modeling the needs of the service organization and is working to understand the flow of information that spawns a service order. Although a service order can be derived from many origin, the engineer categorizes only two source: internal demand or external request. This enables a simplified partitioning of input that is required to generate the work order.
3. Limitation that help to bound the system. For example an aircraft avionics system is being modeled for a next generation aircraft. Since the aircraft will be a two-engine design, all monitoring domains for propulsion will be modeled to accommodate a maximum of two engines and associated redundant systems.

4. Constraints that will guide the manner in which the model is created and the approach taken when the model is implemented. For example, the technology infrastructure for the three-dimensional rendering system described above is a single Power PC-based processor. The computational complexity of problems must be constrained to fit within the processing bounds imposed by the processor.
5. Preferences that indicate the preferred architecture for all data, functions, and technology. The preferred solution sometimes comes into conflict with other restraining factors. Yet, customer satisfaction is often predicated on the degree to which the preferred approach is realized.

The resultant system model may call for a completed automated solution, a semi-automated solution or a manual approach. In fact it is often possible to characterize models of each type that serve as alternative solutions to the problem at hand. In essence the system engineer simply modifies the relative influence of different system elements to derive models of each type.

12.4 Information Engineering : An Overview

The goal of information engineering (IE) is to define architecture that will enable a business to use information effectively. In addition information engineering works to create an overall plan for implementing those architecture {SPE93}. Three different architecture must be analyzed and designed within the context of business objectives and goals:

- Data architecture
- Application architecture
- Technology infrastructure

The data architecture provides a framework for the information needs of a business or business function. The individual building blocks of the architecture are the data objects that are used by the business. The data objects flow between business functions are organized within a database and are transformed to provide information that serves the needs of the business.

The application architecture encompasses those elements of a system that transform objects within the data architecture for some business purpose. In the context of this book, we normally consider the application architecture to be the system of programs that performs this transformation. However in a broader context, the application architecture might incorporate the role of people and business procedures that have not been automated.

The technology infrastructure provides the foundation for the data and application architectures. The infrastructure encompasses the hardware and software that are used to support application and data. The includes computers and computer networks, telecommunication links, storage technologies and the architecture that has been designed to implement these technologies.

The model the system architectures described earlier a hierarchy of information engineering activities is defined. As shown in the figure 12.3 the world view is achieved through information strategy planning (ISP) ISP views the entire business as an entity and isolates the domains of the business that are important

to the overall enterprise. ISP defines the data objects that are visible at the enterprise level, their relationships and how they flow between the business domains.

The domain view is addressed with an IE activity called business area analysis (BAA). Hares [HAR93] describes BAA in the following manner:

BAA is concerned with identifying in detail data and function requirements of selected business area identified during ISP and ascertaining their interactions. It is only concerned with specifying what is required in a business area.

As the information engineer begins BAA the focus narrows to a specific business domain. BAA views the business area as an entity and isolates the business functions and procedures that enable the business area to meet its objectives and goals. BAA like ISP defines data objects their relationship and how data flow. But at this level these characteristic are all bounded by the business area being analyzed. The outcome of NAA is to isolate areas of opportunity in which information systems may support the business area.

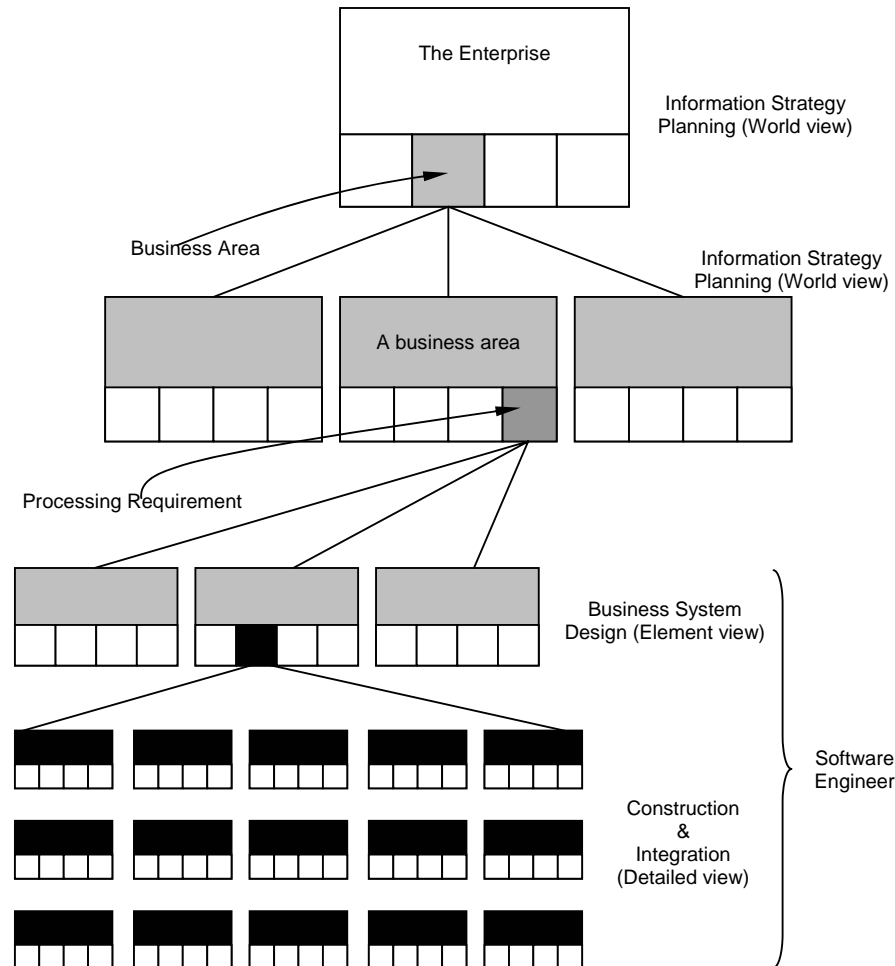


Fig 12.3 The Information Engineering Hierarchy

Once an information system has been isolated for further development, IE makes a transition into software engineering. By invoking a business system design step the basic requirements of a specific information system are modeled and these requirements are translated into data architecture, applications architecture and technology infrastructure.

The final IE step—construction and integration (C&I) focuses on implementation detail. The architecture and infrastructure are implemented by constructing an appropriate database and internal data structures by building application using program components and by selecting appropriate elements of a technology infrastructure to support the design created during BSD. Each of these system components must then be integrated to form a complete information system into the business area context, performing all user training and logistics support to achieve a smooth transition.

12.5 Product Engineering : An Overview

The goal of product engineering is to translate the customer's desire for a set of defined capabilities into a working product. To achieve this goal, product engineering-like information engineering – must derive architecture and infrastructure. The architecture encompasses four distinct system components: software, hardware, data (and databases) and people. A support infrastructure is established and includes the technology required to tie the components together and the information(e.g, documents, CD_ROM, video) that is used to support the components.

As shown in figure 12.4, the world view is achieved through system analysis. The overall requirements of the product are elicited from the customer. These requirements encompass information and control needs, product function and behavior, overall product performance, design, and interfacing constraints and other special needs. Once these requirements are known, the job of system analysis is to allocate function and behaviour to each of the four components noted above.

Once allocation has occurred, component engineering commences. Component engineering is actually a set of concurrent activities that address each of the system components separately; software engineering, hardware engineering, human engineering and database engineering. Each of these engineering disciplines takes a domain-specific view, but it is important to note that the engineering disciplines must establish and maintain active communication with one another. Part of the role of system analysis is to establish the interfacing mechanisms that will enable this to happen.

The element view for product engineering is the engineering discipline itself applied to the allocated component. For software engineering, this means analysis and design modeling activities and construction and integration activities that encompass code generation, testing and support steps. Analysis modeling allocates requirements into representations of data, function and behavior. Design maps the analysis model into data, architectural, interface and procedural designs for the software.

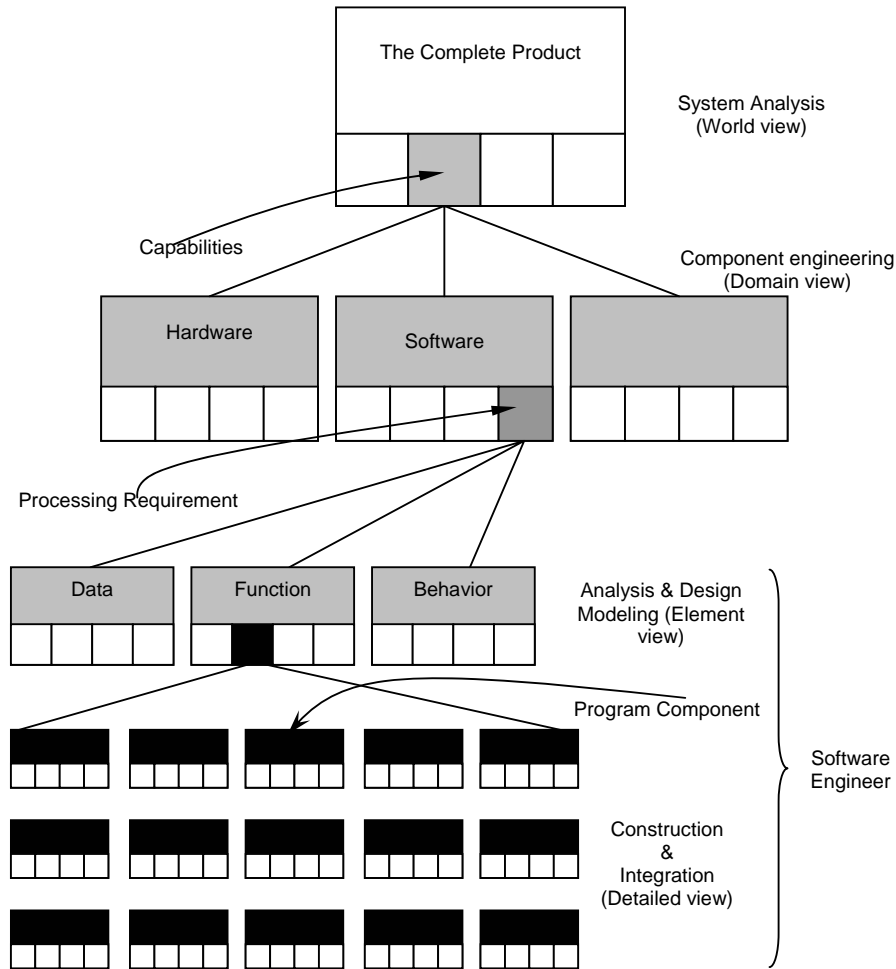


Fig 12.4 The Product Engineering Hierarchy

12.6 Information Engineering

When business automation was first introduced in the early 1960s, companies looked for areas of opportunity and simply automated business functions that were previously performed in a manual fashion. As time passed, individual computer programs were combined to encompass business applications. The applications were grouped into major information systems that served specific business areas. Although this approach was workable, it resulted in problems. Systems were difficult to 'connect' to one another; redundant data was every where; the impact of changes to applications that served one area of the business was difficult to project and even more difficult to implement; and old programs outlived their usefulness, but lack of resources caused them to be used long past their prime.

In their book on 'reengineering the corporation', Hammer and Champy state: Information technology plays a crucial role in business reengineering, but one that is easily miscast. Modern, state-of-the-art

information technology is part of any reengineering effort, an essential enabler permits companies to reengineer business process. But to paraphrase what is often said about money and governments, merely throwing computers at an existing business problem does not cause it to be reengineered.

The global objective of information engineering is to apply "information technology" in a way that best serves the overall needs of the business. To accomplish this, IE must begin by analyzing business objectives and goals, understanding the many business areas that must define the information needs of each business area and the business as a whole. Only after this is done does IE make a transition into the more technical domain of software engineering – the process where information systems, applications and programs are analyzed, designed and built.

12.7 Short Summary

- A high technology system encompasses a number of components: software, hardware, people, database, documentation, and procedures. System engineering helps to translate a customer's needs into a model of a system that makes use of one or more of these components.
- System engineering begins by taking a "world view". A business domain or product is analyzed to establish all basic requirements focus is then narrowed to a "domain view", where each of the system elements is analyzed individually.
- Each element is allocated to one or more engineering components which are then addressed by the relevant engineering discipline.

12.8 Brain Storm

1. Explain System Engineering Hierarchy briefly ?
2. Give a short note on System Modeling ?
3. Discuss about Information Engineering ?
4. Give an overview of Product Engineering ?



Lecture 13

System Engineering - II

Objectives

In this lecture you will learn the following

- ✧ About Information Strategy Planning
- ✧ About Business Area Analysis

Coverage Plan

Lecture 13
13.1 Snap Shot
13.2 Information Strategy Planning
13.3 Enterprise Modeling
13.4 Business-Level Data Modeling
13.5 Business Area Analysis
13.6 Process Modeling
13.7 Information Flow Modeling
13.8 Short Summary
13.9 Brain Storm

13.1 Snap Shot

In this lecture we focus on the Information Strategy Planning, Enterprise Modeling, Business – level Data Modeling, Business Area Analysis and Process Modeling.

13.2 Information Strategy Planning

The first information engineering steps is information strategy planning (ISP). The overall objectives of ISP are (1) to define strategic business objectives and goals, (2) to isolate the critical success factors that will enable the business to achieve these goals and objectives, (3) to analyze the impact of technology and automation on the goals and objectives and (4) to analyze existing information to determine its role in achieving goals and objectives. ISP also creates a business-level data model that defines key data objects and their relationship to one another and to various business areas.

The terms ‘objectives’ and ‘goals’ take on a specific meaning in ISP. An objective is a general statement of direction. For example, a business objective for a maker of cellular telephone might be to reduce the manufactured cost of the product. Goals define a quantitative course of action. To achieve the objective noted above, the manufacture might state the following goals:

- Decrease reject rate by 20 percent within 9 months
- Gain 10 percent price concessions from suppliers
- Reengineer keypads to reduce assembly cost by 30 percent
- Automate manual assembly of components
- Implement a real-time production control system

Objectives tend to be strategic. Goals are tactical

Critical success factors can be tied to an objective or to individual goals. A CSF must be present if the objective or goal is to be achieved. Therefore management planning must accommodate it. For example, CSFs for the manufacturing objective noted above might be:

- Total quality management strategy for the manufacturing organization
- Worker training and motivation
- Higher-reliability machines
- Higher-quality parts
- A ‘sales plan’ to convince suppliers to reduce prices
- Availability of engineering staff

Technology impact analysis examines objectives and goals and provides an indication of those technologies that will have a direct or indirect impact on achieving them successfully. The information engineer addresses the following questions: How critical is the technology to the achievement of a business objective? Is the technology available today? How will the technology change the way business is conducted? What are the direct and indirect costs? How should the business adapt or extend objectives and goals to accommodate the technology?

Because every business area makes some use of information technologies, ISP must also identify what

currently exists and how it is currently used to achieve objectives and goals. Business process reengineering is an activity that examines existing systems with the intent of reengineering them to better meet business needs.

13.3 Enterprise Modeling

Enterprise modeling creates a three-dimensional view of a business. The first dimension addresses the organizational structure and the functions that are performed within the business area defined by the organizational structure. The second dimension decomposes business function to isolate the processes that make function happen. Finally, the third dimension relates objectives, goals and CSFs to the organization and its functions. In addition, enterprise modeling creates a business-levels data model that defines data objects and their relationships to other elements of the enterprise model.

The business organization is defined in a classical business unit hierarchy. Each box in the org chart represents a business area of the company. Like all hierarchies, it is generally possible to refine the boxes within the org chart until small working groups or even individuals are noted. However, for ISP purpose, business areas are all that is required.

Business functions are identified and the processes that are required to implement the business functions are defined. Each of the business functions is then related to the business area that has responsibility for it. In general a business function is some ongoing activity that must be accomplished to support the overall business. It can usually be described as a noun phrase. A business process is a transform that accepts specific inputs and produces specific outputs. It can generally be described as a verb phrases.

Business Function

- o Product development and engineering
- o Marketing
- o Demographic research
- o Market analysis
- o Forecasting
- o Product specification
- o Product engineering
- o Technology research
- o New product development
- o System analysis
- o Component engineering
- o Hardware engineering
- o Software engineering
- o Human engineering
- o Product V & V
- o Quality assurance

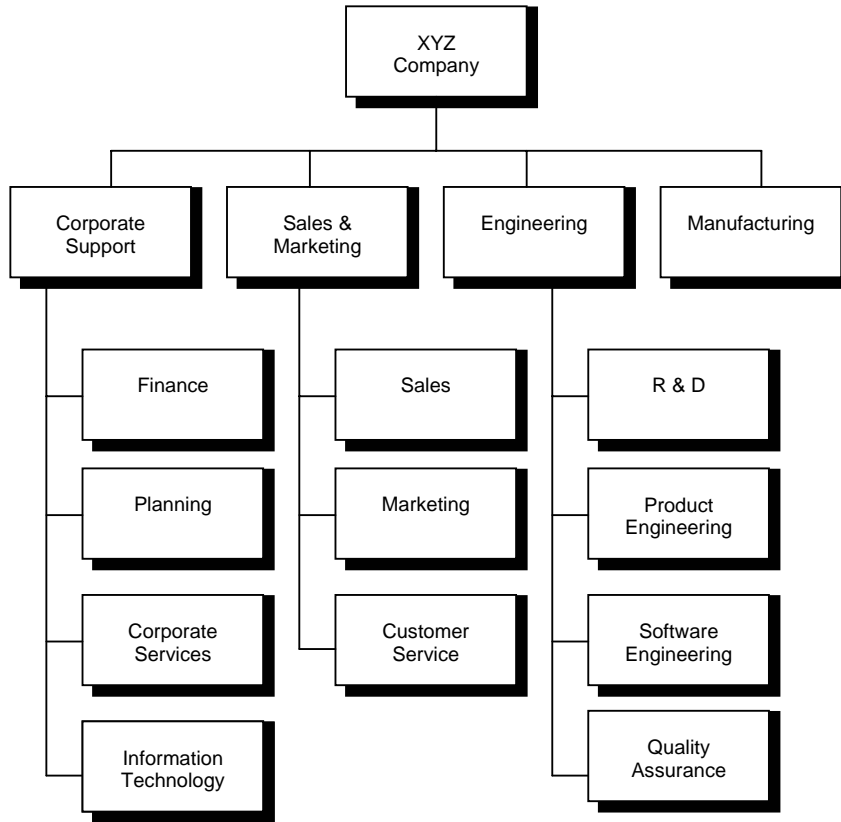


Figure 13.1 Deriving an organizational chart and coupling business areas to function.

To illustrate how a business function is refined into a set of supporting processes, consider the market analysis function shown figure 13.1. A process refinement follows:

Market analysis:

- Collect data on all sales inquiries
- Collect data on all sales
- Analyze data on inquiries and sales
- Develop buyer profile
- Compare profile to demographic research
- Study industry buying trends
- Establish focus groups to determine best sales message
- Design rough sales materials
- Test sales materials and refine
- Finalize sales approach

Each of the bulleted process steps could be further refined to provide a detailed road map for accomplishing the business function. During ISP, the information engineer does not become concerned with areas of automation opportunity. The intent is simply to understand and model the business.

13.4 Business-Level Data Modeling

Business-level data modeling is an enterprise modeling activity that focuses on the data objects (also called entities) that are required to achieve the business functions. At the business level, typical data objects include producers and consumers of information (e.g., a customer), things (e.g., a report), occurrences of events (e.g., a sales conference), organization units (e.g., sales and marketing), places (e.g., manufacturing cell), or information structures (e.g., an employee file). A data object contains a set of attributes that define some aspect, quality, characteristic or descriptor of the data that is being described.

For example during enterprise modeling an information engineer might define the data object customer. To more fully describe customer, the following attributes are defined:

Object : **Customer**

Attributes:

Name
 Company name
 Job classification and purchase authority
 Business address and contact information
 Product interest(s)
 Past purchase(s)
 Date of last contact
 Status of contact

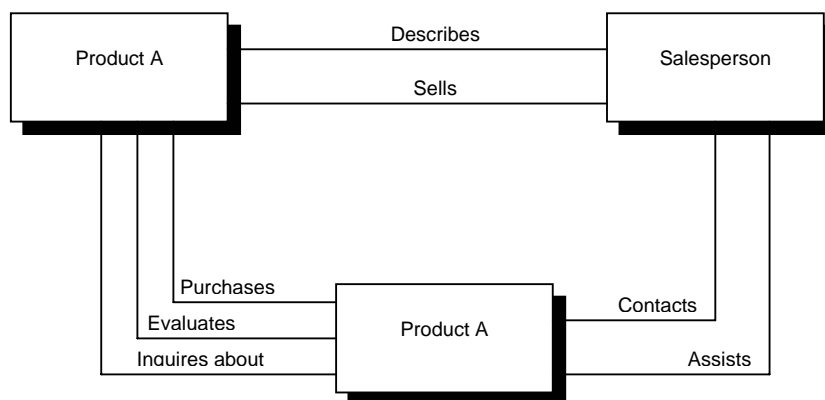


Figure 13.2 Depicting relation ship among business level data objects.

Once a set of data objects is defined, their relationships are identified. A relationship indicates how objects are connected to one another. As an example, consider the objects customer, product A and salesperson. An information engineer creates a diagram (figure 13.2) to depict these relationships. Referring to the figure relationships imply a connection between data objects. In general, relationships can be read in either direction; hence a customer purchase products A and product A is purchased by a customer. In reality additional information is provided as part of the data model.

The culmination of the ISP activity is the creation of a series of cross-reference matrices that establish the overall relationships between the organization (and its business areas), business objectives and goals, business functions, and data objects. Examples of such matrices are shown in figure 13.3

13.5 Business Area Analysis

In his book on information engineering, Martin describes business area analysis in the following manner:

Business areas analysis(BAA) establishes a detailed framework for building an information-based enterprise. It takes one business area at a time and analyzes it in detail. It uses diagrams and matrices to model and record the data and activities in the enterprise and to give a clear understanding of the elaborate and subtle ways in which the information aspects of the enterprise interrelate.

During BAA, our focus shifts from the world view to the domain view. To model 'the elaborate and subtle ways in which the information aspects of the enterprise interrelate' the information engineer must depict how data objects (described during ISP and refined during BAA) are used and transformed within each business area and how the business functions and processes within each business area transform these data objects. In essence, both exogenous and endogenous data are analyzed and modeled for each business area.

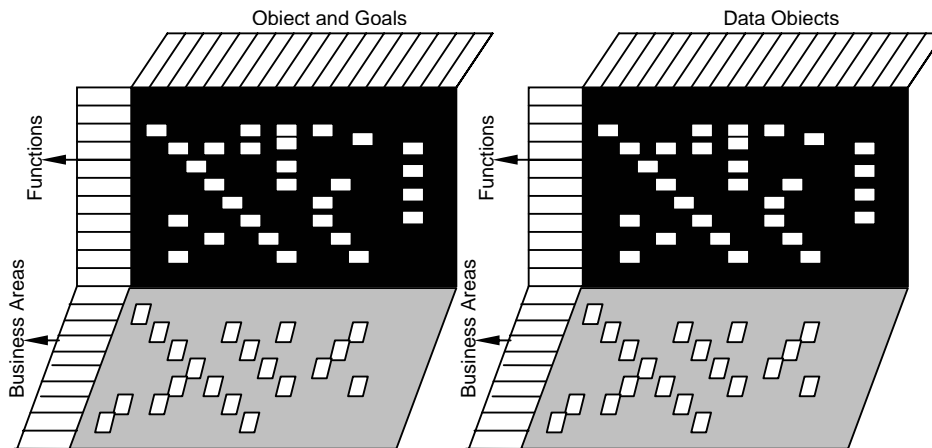


Figure 13.3 Typical cross-reference matrices used during ISP

To accomplish this work, BAA makes use a number of different models;

- Data models (now refined to the business area level)
- Process flow models
- Process decomposition diagrams
- A variety of cross-reference matrices

The data objects defined during ISP are refined for use within each business area. For example the data object customer described in the preceding section is used by the sales department. After evaluation of the needs of the sales department (an analysis of the sales domain), the original definition of customer is further refined to meet the needs of sales:

Object :**Customer**

Attributes:

Name

Company name → Object **Company**

Job classification and purchase authority

Business address and contact information

Product interest(s)
 Past purchase(s)
 Date of last contact – record of contacts
 Status of contact – status of last contact
 → next contact date
 → recommended nature of contact

The attribute company name has been modified to point to another object called **Company**. This object contains not only the company name but additional information about the size of the company, its purchasing requirements, the name of other contacts and so on. This information will be useful in the sales domain. Other attributes have been modified and added as noted above.

13.6 Process Modeling

The work performed within a business area encompasses a set of business functions that are further refined into business processes. To illustrate consider a simplified version of the sales function discussed in section 13.4. The processes that occur to accomplish sales are:

Sales Function

- Establish customer contact
- Provide product literature and related information
- Address questions and concerns
- Provide evaluation product
- Accept sales order
- Check availability of configuration ordered
- Prepare delivery order
- Confirm configuration, pricing , ship data with customer
- Transmit delivery order to fulfillment department
- Follow up with customer.

A process flow diagram (Figure 13.4) can be developed for this sequence of processing. It should be noted that each business function relevant to the business area can be refined in a similar manner.

13.7 Information Flow Modeling

The process flow model is integrated with the data model to provide an indication of how information flows through a business area. Input and output data objects are shown for each process, providing an indication of how the process transforms information to accomplish a business function.

Once a complete set of process flow models has been created the information engineer(along with others) examines how the existing process can be reengineered (e.g., [HAM93],[JAY94]) and where existing information systems or applications might be modified or replaced by more efficient information technologies. The revised process model is used as a basis for the specification of new or revised software to support the business function.

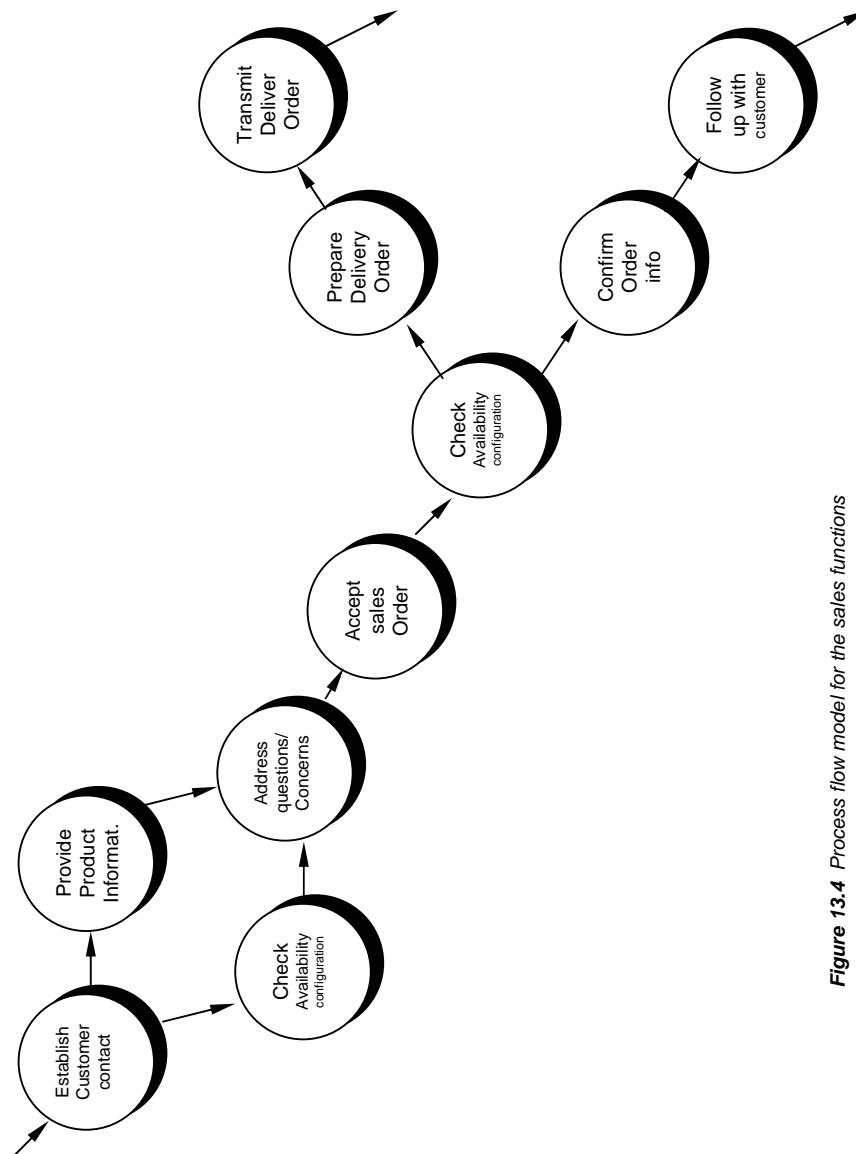
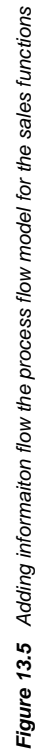


Figure 13.4 Process flow model for the sales functions



Centre for Information Technology and Engineering, Manonmaniam Sundaranar University 159

13.8 Short Summary

- Information engineering is a system engineering approach that is used to define architectures that enable a business to use information effectively.
- The intent of information engineering is to derive comprehensive data architectures, an application architecture, and a technology infrastructure that will meet the needs of the business strategy and the objectives and goals of each business area.
- Information engineering encompasses information strategy planning business area analysis and application specific analysis that is actually part of software engineering.

13.9 Brain Sorm

1. Discuss about Information Strategy Planning ?
2. Explain briefly about Enterprise Modeling ?
3. Give a Short note on Business Area Analysis ?
4. Discuss abot process Modelilng ?

❧❧

Lecture 14

System Engineering - III

Objectives

In this lecture you will learn the following

- ✧ About Product Engineering
- ✧ About Modeling the System Architecture
- ✧ About System Modeling and System Simulation
- ✧ About System Specification

Coverage Plan

Lecture 14
14.1 Snap Shot
14.2 Product Engineering
14.3 Modeling the System Architecture
14.4 System Modeling and Simulation
14.5 System Specification
14.6 Short Summary
14.7 Brain Storm

14.1 Snap Shot

Software Engineering occurs as a consequence of a process called system engineering. Instead of concentrating solely on software, system engineering. Instead of concentrating solely on software, system engineering focuses on a variety of elements, analyzing and designing and organizing those element into a system that can be a product, a service or a technology for the transformation of information or control.

14.2 Product Engineering

Product engineering (also called system engineering) is a problem solving activity. Desired product data, function, and behavior are uncovered analyzed and allocated to individual engineering components. The system engineer begins with customer-defined objectives and goals for the product and proceeds to model these requirement in a manner that allocates them to as set of engineering components— software, hardware, data and people. The components are tied together with a support infrastructure the technology required to integrate the components and the information that is used to support the components.

The genesis of most new products and systems begins with a rather nebulous concept of desired function. Therefore the system engineer must bound the product requirements by identifying the scope of function and performance desired. For example, it is not enough to say that the control software for the robot in a manufacturing automation system will "respond rapidly if a parts tray is empty." The system engineer must define (1) what indicates an empty tray to the robot, (2) the precise time bounds within which software response is expected and (3) what form the response mistake. That is the system engineer must describes the events that drive the behavior of the robot the nature of the behavior and the quantitative bounds placed on the behavior.

Once function, performance, constraints and interfaces are bounded the system engineer moves onto a task that is called allocation. During allocation function is assigned to one or more engineering components. Often alternative allocations are proposed and evaluated. To illustrate the process of allocation we consider a macro element of the factory automation system the conveyor line sorting system (CLSS). The system engineer is presented with the following statement of objectives for CLSS.

CLSS must be developed such that boxes moving along a conveyor line will be identified and sorted into one of six bins at the end of the line. The boxes will pass by a sorting station where they will be identified. Based on an identification number printed on the side of the box (an equivalent bar code is provided), the boxes will be shunted into the appropriate bins. Boxes pass in random order and are evenly spaced. The line is moving slowly.

CLSS is depicted schematically before continuing make a list of questions that you would ask if you were the system engineer.

Among the many questions that should be asked and answered are the following:

1. How many different identification number must be processed and what is their form?
2. What is the speed of the conveyor line in feet per second and what is the distance between boxes in feet?
3. How far is the sorting station from the bins?
4. How far apart are the bins?

5. What should happen if a box doesn't have an identification number or an incorrect number is present?
6. What happens when a bin fills to capacity?
7. Is information about box destination and bin contents to be passed else where in the factory automation system? Is real-time data acquisition required?
8. What error/failure rate is acceptable?
9. What pieces to the conveyor line system currently exist and are operational?
10. What schedule and budgetary constraints are imposed?

Note that the above questions focus on function, performance, can information flow and content. The system engineer does not ask the customer how the task is to be done; rather the engineer asks what is required.

Assuming reasonable answers, the system engineer develops a number of alternative allocations. Note that function and performance are assigned to different generic system elements in each allocation.

Allocation 1. A sorting operator is training and place at the sorting station location. He/She reads the box and places it into an appropriate bin.

Allocation 1 represents a purely manual (but nevertheless, effective) solutions to the CLSS problem. The primary engineering component is people (the sorting operator). The person performs all sorting functions. Some documentation (in the form of a table relating identification number to bin location and procedural description for operator training) may be required. Therefore this allocation uses only the people and documentation elements.

Allocation 2. A bar code reader and controller are placed at the sorting station. Bar code output is passed to a programmable controller that controls a technical shunting mechanism the shunt slides the box to the appropriate bin.

For allocation 2, hardware (bar code reader, programmable control, shunt hardware, etc.,) software (for the bar code reader and programmable controller) and database (a look-up table that relates box ID with bin location) components are used to provide a fully automated solution. It is likely that each of these components may have corresponding manuals and other documentation adding another component.

Allocation 3. A bar code reader and controller are placed at the sorting station. Bar code output is passed to a robot arm that grasps a box and moves it to the appropriate bin location.

Allocation 3 makes use of one macro element— the robot. Like allocation 2, this allocation uses hardware, software, a database and documentation as engineering components. The robot is a macro element of CLSS and itself contains a set of engineering components.

By examining the three alternative allocations for CLSS it should be obvious that the same function can be allocated to different components. In order to choose the most effective allocation a set of trade-off criteria should be applied to each alternative.

The following trade-off criteria govern the selection of a product configuration based on a specific allocation of function and performance to generic system elements:

Project considerations. Can the configuration be built within pre-established cost and schedule bound? What is the risk associated with cost and schedule estimates?

Business considerations. Does the configuration represent the most profitable solution? Can it be marketed successfully? Will ultimate payoff justify development risk?

Technical analysis. Does the technology exist develop all elements of the system? Are function and performance assured? Can the configuration be adequately maintained? Do technical resource exist? What is the risk associated with the technology?

Manufacturing evaluation. Are manufacturing facilities and equipment available? Is there a shortage of necessary component? Can quality assurance be adequately performed?

Human Issues. Are trained personnel available for development and manufacture? Do political problems exist? Does the customer understand what the system is to accomplish?

Environmental Interfaces. Does the proposed configuration properly interface with the system's external environment? Are machine-to-machine and human-to-machine communication handled in an intelligent manner?

Legal Considerations. Does this configuration introduce undue liability risk? Can proprietary aspects be adequately protected? Is there potential infringement?

We examine some of these issues in more detail late in this lecture.

It is important to note that the system engineer should also consider off-the-shelf solutions to the customer's problem. Does an equivalent system already exist? Can major parts of a solution be purchased from a third party?

The application of trade-off criteria results in the selection of a specific system configuration and the specification of function and performance allocated to hardware software, people, databases, documentation and procedures. Essentially the scope of function and performance is allocated to each engineering component of the product. The role of hardware engineering, software engineering, human engineering and database engineering is to refine scope and procedure an operational product component that can be properly integrated with other components.

System Analysis

System analysis is conducted with the following objectives in mind: (1) identify the customer's need; (2) evaluate the system concept for feasibility; (3) perform economic and technical analysis; (4) allocate function to hardware, software, people, database and other system elements; (5) establish cost and schedule constraints; and (6) create a system definition that forms the foundation for all subsequent engineering work. Both hardware and software expertise (as well as human and database engineering) are required to successfully attain the objectives listed above.

Identification of Need

The first step of the system analysis process involves the identification of need. The analyst (system engineer) meets with the customer and the end user (if different from the customer). The customer may be

a representative of an outside company, the marketing department of the analyst's company (when a product is being defined), or another technical department (when an internal system is to be developed). Like information engineering the intent is to understand the product's objective(s) and to define the goals required to meet the objective(s).

Once overall goals are identified, the analyst moves on to an evaluation of supplementary information: Does the technology exist to build the system? What special development and manufacturing resources will be required? What bounds have been placed on costs and schedule? If the new system is actually a product to be developed for sale to many customers the following questions are also asked: what is the potential market for the product? How does this product compare with competitive products? What position does this product take in the overall product line of the company?

Information gathered during the needs identification step is specified in a system concept document. The original concept document is sometimes prepared by the customer before meetings with the analyst. Invariably customer-analyst communication results in modification to the document.

Feasibility Study

All projects are feasible -- given unlimited resources and infinite time! Unfortunately, the development of a computer-based system or product is more likely plagued by a scarcity of resources and difficult (if not downright unrealistic) delivery dates. It is both necessary and prudent to evaluate the feasibility of a project at the earliest possible time. Months or years of effort thousand for millions of dollars and untold professional embarrassment can be averted if an ill-conceived system is recognized early in the definition phase.

Feasibility and risk analysis are related in many ways. If project risk is great the feasibility of producing quality software is reduced. During product engineering however, we concentrate our attention on four primary areas of interest:

Economic feasibility. An evaluation of development cost weighed against the ultimate income or benefit derived from the developed system or product.

Technical Feasibility. A study of function, performance and constraints that may affect the ability to achieve an acceptable system.

Legal feasibility. A determination of any infringement violation or liability that could result from development of the system.

Alternative An evaluation of alternative approaches to the development of the system or product.

A feasibility study is not warranted for systems in which economic justification is obvious technical risk is low, few legal problems are expected and no reasonable alternative exists. However if any of the preceding conditions fail a study of that area should be conducted.

Economic justification is generally the "bottom-line" consideration for most systems (notable exceptions sometimes include national defense systems, systems mandated by law, and high-technology applications such as the space program). Economic justification includes a broad range of concerns that include cost-

benefits analysis long-term corporate income strategies impact on other profit centers or products cost of resources needed for development and potential market growth.

Technical feasibility is frequently the most difficult area to assess at this stage of the product engineering process. Because objectives functions and performance are somewhat hazy anything seems possible if the “right” assumptions are made. It is essential that the process of analysis and definition be conducted in parallel with an assessment of technical feasibility. In this way concrete specification may be judged, as they are determined.

The consideration that are normally associated with technical feasibility include:

Development risk. Can the system element be designed so that necessary function and performance are achieved within the constraints uncovered during analysis?

Resource availability. Is skilled staff available to develop the system element in question? Are other necessary resources (hardware and software) available to build the system?

Technology. Has the relevant technology progressed to a state that will support the system?

Developers of computer-based systems are optimists by nature. However, during an evaluation of technical feasibility a cynical if not pessimistic attitude should prevail. Misjudgment at this stage can be disastrous.

Legal feasibility encompasses a broad range of concerns that include contracts, liability, infringement and myriad other traps frequently unknown to technical staff. A discussion of legal issues and software is beyond the scope of this book. The integrated reader should see [SCO89].

The degree to which alternatives are considered is often limited by cost and time constraints; however a legitimate but “unsponsored” variation should not be buried.

The feasibility study may be documented as a separate report to upper management and included as an appendix to the system specification. Although the format of a feasibility study may vary the outline provided in Figure 14.1 covers most important topics.

The feasibility study is reviewed first by project management (to assess content reliability) and by upper management (to assess project status). The study should result in a “go/no-go” decision. It should be noted that other go/no-go decisions will be made during the planning, specification, and development steps of both hardware and software engineering.

Economic Analysis

Among the most important information contained in feasibility study is cost-benefit analysis an assessment of the economic justification for a computer-based system project. Cost-benefit analysis delineates costs for project development and weighs them against tangible. (i.e., measurable directly in dollars) and intangible benefits of a system.

- I. Introduction
 - A. Statement of the problem
 - B. Implementation environment
 - C. Constraints
- II. Management Summary and Recommendations
 - A. Important Findings
 - B. Comments
 - C. Recommendations
 - D. Impact
- III. Alternatives
 - A. Alternative system configurations
 - B. Criteria used in selecting the final approach
- IV. System Description
 - A. Abbreviated statement of scope
 - B. Feasibility of allocated elements
- V. Cost-Benefit Analysis
- VI. Evaluation of Technical Risk
- VII. Legal Ramifications
- VIII. Other Project Specific Topics.

Figure 14.1 Feasibility Study Outline

Cost-benefit analysis is complicated by criteria that vary with the characteristics of the system to be developed, the relative size of the project and the expected return on investment desired as part of a company's strategic plan. In addition many benefits derived from computer-based systems are intangible (e.g., better design quality through iterative optimization, increased customer satisfaction through programmable control, and better business decisions through reformatted and preanalyzed sales data). Direct quantitative comparisons may be difficult to achieve.

As we noted above, analysis of benefits will differ depending on system characteristics. To illustrate, consider the benefits for management information systems most data-processing systems are developed with 'better information quantity, quality, timeliness or organization' as a primary objectives. Therefore the concentrate on information access and its impact on the user environment. The benefits that might be associated with an engineering-scientific analysis program or a computer-based product could differ substantially.

Costs associated with the development of a computer-based system the analyst can estimate each cost and then use development and ongoing costs to determine a return on investment, a break-even point and a payback period.

The following excerpt (FRI77) may best characterize cost-benefit analysis:

Like political rhetoric after the election, the cost-benefit analysis may be forgotten after the project implementation begins. However it is extremely important because it has been the vehicle by which management approval has been obtained.

Only by sending the time to evaluate feasibility do we reduce the chances for extreme embarrassment (or worse) at later stages of a system project. Effort spent on a feasibility analysis that results in cancellation of a proposed project is not wasted effort.

Technical Analysis

During technical analysis, the analyst evaluates the technical merits of the system concept, at the same time collecting additional information about performance, reliability, maintainability and producibility. In some cases, this system analysis step also includes a limited amount of research and design.

Technical analysis begins with an assessment of the technical viability of the proposed system. What technologies are required to accomplish system function and performance? What new material, methods, algorithms or processes are required and what is their development risk? How will these technology issues affect cost?

The tools available for technical analysis are derived from mathematical modeling and optimization techniques, probability and statistics queuing theory and control theory – to name a few. It is important to note, however that analytical evaluation is not always possible. Modeling (either mathematical or physical) is an effective mechanism for technical analysis of computer-based systems.

Blanchard and Fabrycky [BLA81] define a set of criteria for the use of models during technical analysis of systems:

1. The model should represent the dynamics of the system configuration being evaluated in a way that is simple enough to understand and manipulate and yet close enough to the operating reality to yield results.
2. The models should highlight those factors that are most relevant to the problem at hand and suppress (with discretion) those that are not as important.
3. The model should be made comprehensive by including all relevant factors and should be reliable in terms of repeatability of results.
4. Model design should be simple enough to allow for timely implementation in problem solving. Unless the model can be utilized in a timely and efficient manner by the analyst or the manager, it is of little value. If the model is large and highly complex, it may be appropriate to develop a series of smaller models in which the output of one can be tied to the input of another. Also it may be desirable to evaluate a specific element of the system independently from other elements.
5. Models design should incorporate provisions for ease of modification and/or expansion to permit the evaluation of additional factors as required. Successful model development often includes a series of trials before the overall objective is met. Initial attempts may suggest information gaps which are not immediately apparent and consequently may suggest beneficial changes.

The results obtained from technical analysis form the basis for another ‘go/no-go’ decision on the system. If technical risk is severe, if models indicate that desired function or performance cannot be achieved, if the pieces just won’t fit together smoothly – it’s back to the drawing board!

14.3 Modeling The System Architecture

Every computer-based system can be modeling as information transforms using an input-processing-output architecture. Hatley and Pirbhai (HAT87) have extended this view to include two additional system features-user interface processing and maintenance and self-test processing. Although these additional features are not present for every computer-based system, they are very common, and their specification makes any system model more robust. Using a representation of input, processing, output, user interface processing, and self-test processing, a system engineer can create a model of system components that sets a foundation for later requirements analysis and design steps in each of the engineering disciplines.

To develop the system model, an architecture template [HAT87] is used. The system engineer allocates system elements to each of five processing regions within the template (1) user interface,(2) input, (3) system function and control, (4) output and (5) maintainenacne and self-test. The format of the architecture template is shown in fig 14.2.

Like nearly all modeling techniques used in system and software engineering, the architecture template enables the analyste to create a hierarchy of detail. An architecture context diagram (ACD) resides at the top level of the hierarchy.

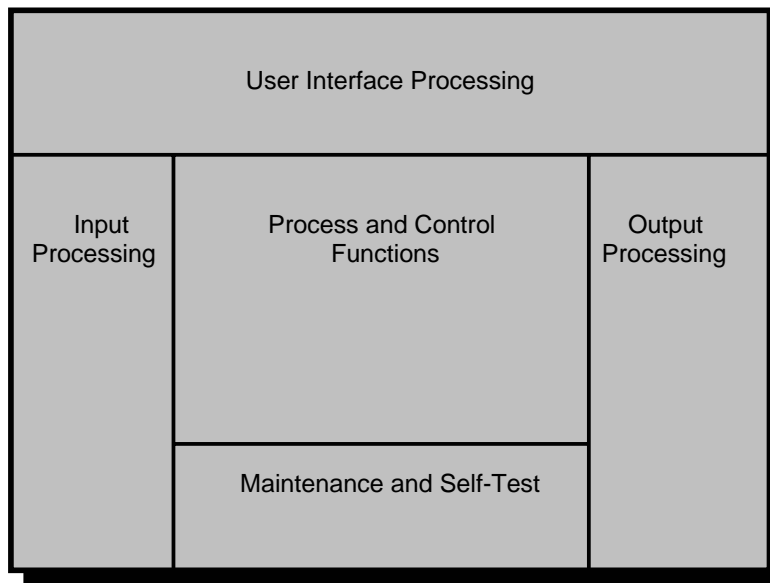


Figure 14.2 Architecture Template

The context diagram ‘establishes the information boundary between the system being implemented and the environment in which the system is to operate’ [HAT87]. That is the ACD defines all external producers of information used by the system, all external consumers of information created by the system and all entities that communicate through the interface or perform maintenance and self-test.

To illustrate the use of the ACD, consider an extended version of the conveyor line sorting system (CLSS) discussed earlier in this. The extended version makes use of personal computer at the sorting station site. The PC executes all CLSS software; interfaces with the bar code reader to read part number on each box; interfaces with the conveyor line monitoring equipment to acquire conveyor line speed; stores all part number sorted; interacts with a sorting station operator to produce a variety of reports and diagnostics; sends control signals to the shutting hardware to sort the boxes; and communicates with a central factory automation mainframe. The ACD for CLSS is shown in figure 14.3.

Each box shown in figure 14.3 represents an external entity- that is a producer or consumer of information from the system. For example, the barcode reader produces information that is input to the CLSS system. The symbol for the entire system (or at lower levels, major subsystems) is a rectangle with rounded corners. Hence, CLSS is represented in the ACD represent information(data and control) as it moves from the external environment into the CLSS system. The external entity bar code reader produces input information that is labeled bar code. In essence, the ACD places any system into the context of its external environment.

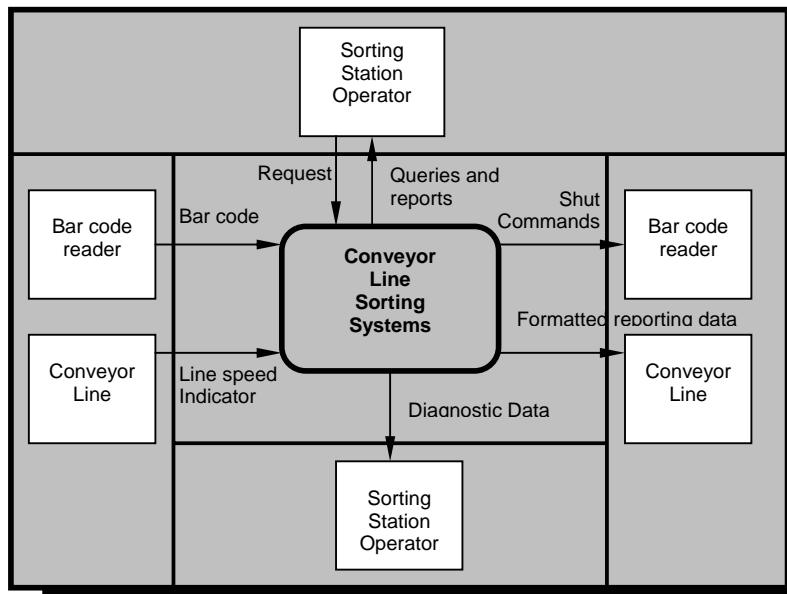


Figure 14.3 Architecture context diagram for CLSS

The system engineer refines the architecture context diagram by considering the shaded rectangle in figure 14.3 in more detail. The major subsystems that enable the conveyor line sorting system to function within the context defined by the ACD are identified. In figure 14.4 the major subsystems are defined in an architecture flow diagram (AFD) that is derived from the ACD. Information flow across the regions of the ACD is used to guide the system engineer in developing the AFD – a more detailed 'schematic' for CLSS. The architecture flow diagram shows major subsystems and important lines of information (data and control) flow. In addition, the architecture template partitions the subsystem processing into each of the five processing regions discussed earlier. At this stage, each of the subsystems can contain one or more system elements (e.g., hardware, software, people) as allocated by the system engineer.

The initial architecture flow diagram (AFD) becomes the top node of a hierarchy of AFDs. Each rounded rectangle in the original AFD can be expanded into another architecture template dedicated solely to it.

This process is illustrated systematically in figure 14.5. Each of the AFDs for the system can be used as a starting point for subsequent engineering steps for the subsystem that has been described.

Subsystems and the information that flows between them can be specified (bounded) for subsequent engineering work. A narrative description of each subsystem and a definition of all data that flow between subsystems become important elements of the system specification.

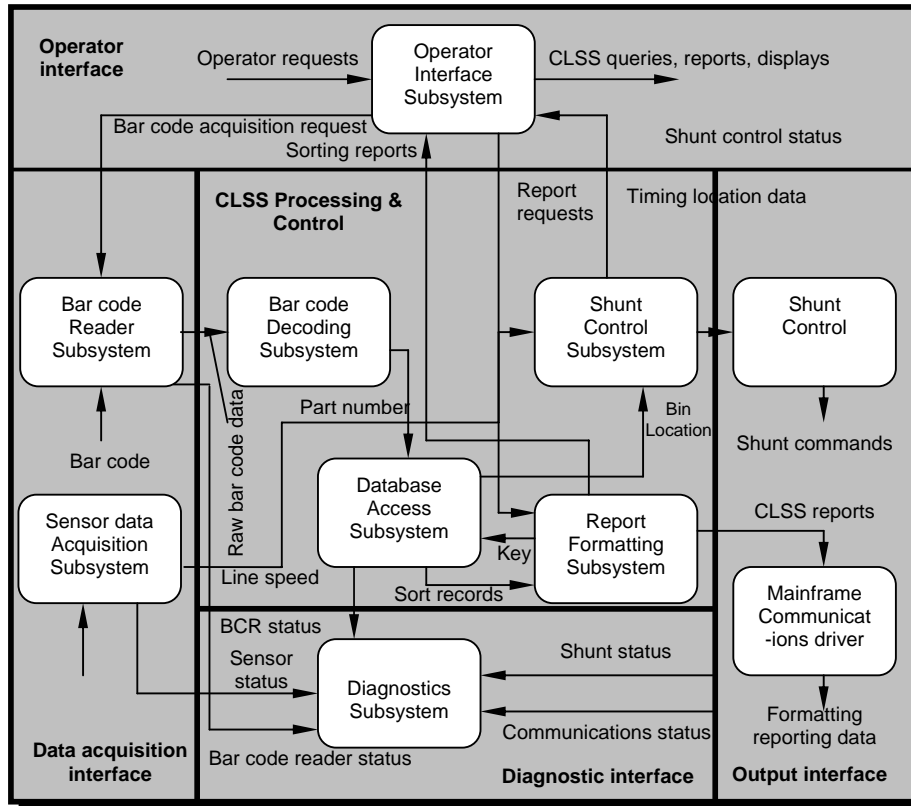


Figure 14.4 Architecture flow diagram for CLSS

14.4 System Modeling and Simulation

Almost three decades ago, R.M. Graham [GRA69] made a distressing comment about the way we built computer-based systems: “We build systems like the Wright brothers built airplanes – build the whole thing, push it off a cliff, let it crash and start over again”. In fact, for at least one class of system – the reactive system – we continue to do this today.

Many computer-based systems interact with the real world in a reactive fashion. That is real world events are monitored by the hardware and software that comprise the computer-based system, and based on these events, the system imposes control on the machines, processes and even people who cause the event to occur. Real-time and embedded systems often fall into the reactive systems category.

Unfortunately, the developers of reactive systems sometimes struggle to make them perform properly. Until recently, it has been difficult to predict the performance, efficiency and behavior of such systems prior to building them. In a very real sense, the construction of many real-time systems was an adventure in ‘flying’. Surprise (most of them unpleasant) were not discovered until the system was built and ‘pushed

off a cliff'. If the system crashed due to incorrect function, inappropriate behavior, or poor performance, we picked up the pieces and started over again.

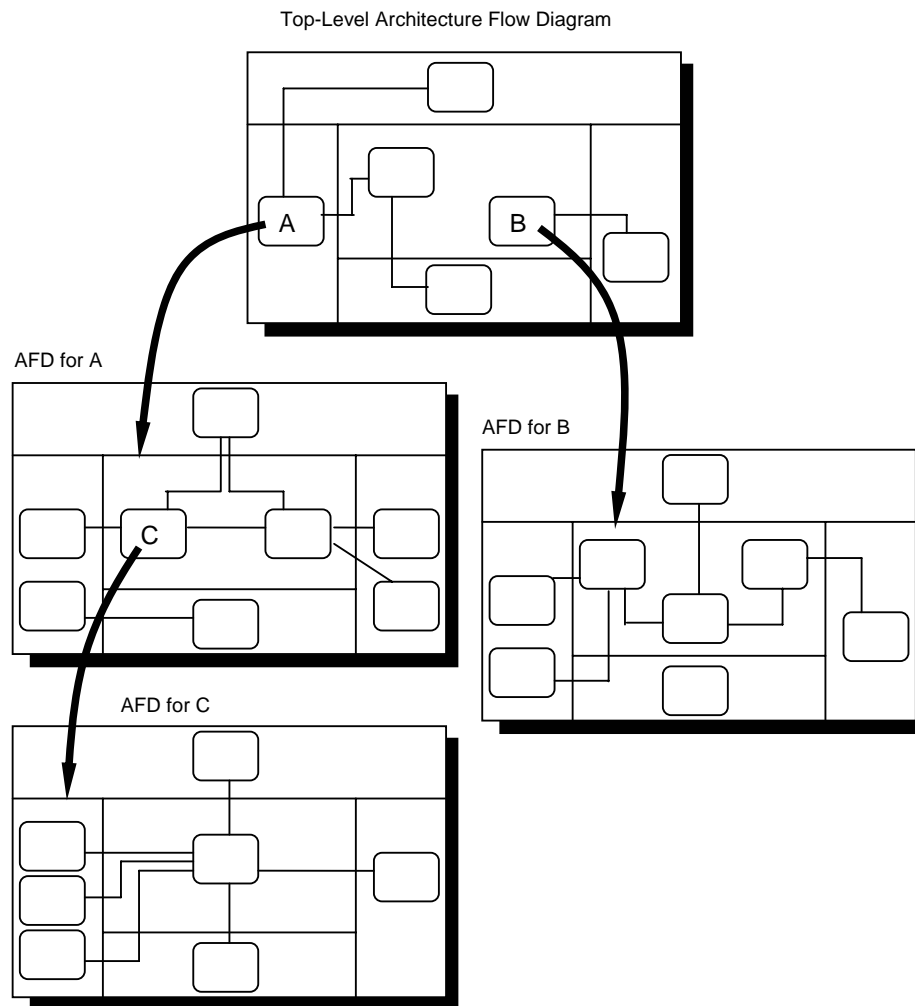


Figure 14.5 Building an AFD hierarchy

Many systems in the reactive category control machines and/or processes (e.g., commercial airliners or petroleum refineries) that must operate with an extremely high degree of reliability. If the system fails, significant economic or human loss could occur. For this reason, the approach described by Graham is both painful and dangerous.

14.5 System Specification

The system specification is a document that serves as the foundation for hardware engineering, software engineering, data base engineering and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element. For example, it provides the software engineer with an indication of the role of software within the context of the system as a whole and the various subsystems described in the

architecture flow diagrams. The system specification also describes the information (data and control) that is input to and output from the system.

It should be noted, however that this is but one of many outlines that can be used to define a system description document. The actual format and content may be dictated by software or system engineering standards or local custom and preferences.

14.6 Short Summary

- Product engineering is a system engineering approach that begins with system analysis.
- The system engineer identifies the customer's needs, determines economic and technical feasibility, and allocates function and performance to software, hardware, people and databases – the key engineering components.
- An architectural model of the system or product is produced and representations of each major subsystem can be developed. Finally the system engineer can create a reactive system model that can be used as the basis for a simulation of performance and behavior.
- The system engineering task culminates with the creation of a system specification a document that forms the foundation for all engineering work that follows.

14.7 Brain Storm

1. Give a brief explanation on system engineering ?
2. Explain briefly about Modeling the System Architecture ?
3. What is System Specification ?
4. Write a Note on System Modeling and Simulation ?



Lecture 15

Analysis Modeling - I

Objectives

In this lecture you will learn the following

- ✎ About Elements of Analysis Modeling
- ✎ About Data Modeling
- ✎ About Functional Modeling

Coverage Plan

Lecture 15
15.1 Snap Shot
15.2 The Elements of the Analysis Model
15.3 Data Modeling
15.4 Functional Modeling and Information Flow
15.5 Hatly and Pirbhai Extension
15.6 Short summary
15.7 Brain storm

15.1 Snap Shot

Like many important contributions to software engineering, structured analysis was not introduced with a single landmark paper or book that was a definitive treatment of the subject. Early work in analysis modeling was begun in the late 1960s and early 1970s but the first appearance of the structured analysis approach was as an adjunct to another important topic-structured design. Research hers needed a graphical notation for representing data and the processes that transformed it. These processes would ultimately be mapped into design architecture.

The term “structured analysis” originally coined by Douglas Ross, was popularized by DeMarco [DEM79]. In his book on the subject, DeMarco introduced and named the key graphical symbols that enabled an analyst to create information flow models, suggested heuristics for the use of these symbols suggested that a data dictionary and processing narratives could be used as a supplement to the information flow models, and presented numerous examples that illustrated the use of this new method. In the years that followed, variations of the structured analysis approach were suggested by Page-Jones [PAG80] Gane and Sarson [GAN82], and many others. In every instance the method focused on information systems applications and did not provide an adequate notation to address the control and behavioral aspects of real-time engineering problems.

By the mid-1980 the deficiencies of structured analysis (when attempts were made to apply the method to control-oriented application) became painfully apparent. Real-time “extension” was introduced by Ward and Mellor [WAR 85] and later by Hatley and Pirbhai [HAT87] these extension resulted in amore robust analysis method that could be applied effectively to engineering problems. Attempts to develop one consistent notation have been suggested [BRU88] and modernized treatments have been published to accommodate the use of CASE tools [YOU89].

15.2 The Elements of the Analysis Model

The analysis model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design and (3) to define a set of requirements that can be validated once the software is built. To accomplish the analysis model derived during structured analysis takes the form illustrated in figure 15.1.

At the core of the model lies the data dictionary – a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the core. The entity-relationship diagram (ERD) depicts relationships between data objects. The ERAD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.

The data flow diagram serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and sub function) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a process specification (PSPEC).

The state-transition diagram (STD) indicates how the system behaves as a consequence of external events. To accomplish this the STD represents the various are made from state to state. The STD serves as the basis for behavior (called states) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about control aspects

of the software is contained in the control specification

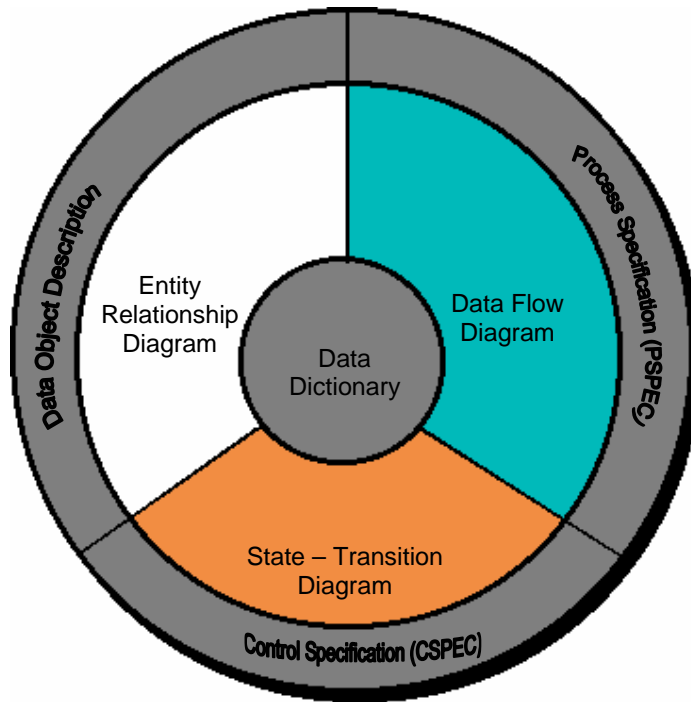


Figure 15.1 The structure of the analysis model

The analysis model encompasses each of the diagrams specification, and descriptions and the dictionary noted give more detailed discussion of these elements of the analysis model is presented in the sections that follow.

15.3 Data Modeling

Data modeling answers of specific questions that are relevant to any data processing application. What are the primary data objects to be processed by the system? What is the composition of each data object and what attributes describe the object? Where do the objects currently reside? What are the relationships between each object and other objects? What is the relationship between the objects and the processes that transform them?

To answer these questions, data modeling methods make use of the entity relationship diagram (ERD). The ERD described in detail later in this section enables a software engineer to identify data objects and their relationships using a graphical notation. In the context of structured analysis, the ERD defines all data that are input, stored, transformed and produced within an application.

The entity-relationship diagram focuses solely on data (and therefore satisfies the first operational analysis principle), representing a “data network” that exists for a given system. The ERD is especially useful for applications in which data and the relationships that govern data are complex. Unlike the data flow diagram data modeling considers data independently of the processing that transforms the data.

15.3.1 Data Objects, Attributes and Relationships

The data model consists of three interrelated pieces of information: the data objects the attributes that describe the data object and the relationship that connect data objects to one another.

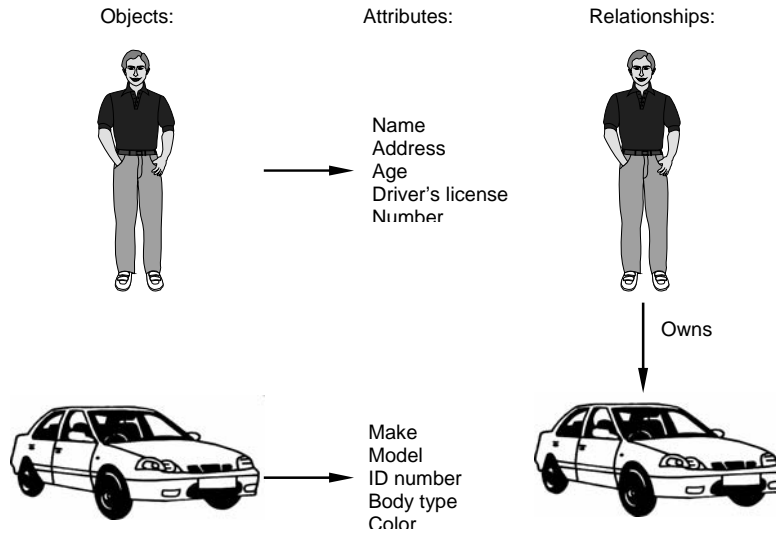


Figure 15.2 Data objects, attributes, and relationships

Data objects A data object is a representation of almost any composite information that must be understood by software. By composite information we mean something that has a number of different properties or attributes. Therefore "width " would not be a valid data object, but dimensions (incorporating height, width and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attribute. The data object description incorporates the data object and all of its attributes.

Data objects are related to one another. For example, person can own car where the relationship own connotes a specific connection between person and car. The relationships are always defined by the context of the problem that is being analyzed.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented. The headings in the table reflect attribute of the object. In this case, a car is defined in terms of make, model, ID# body type, color and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.

Attributes: Attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier that is the identifier attribute becomes a "key" when we want to find an instance of

the data object. In some case, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID#

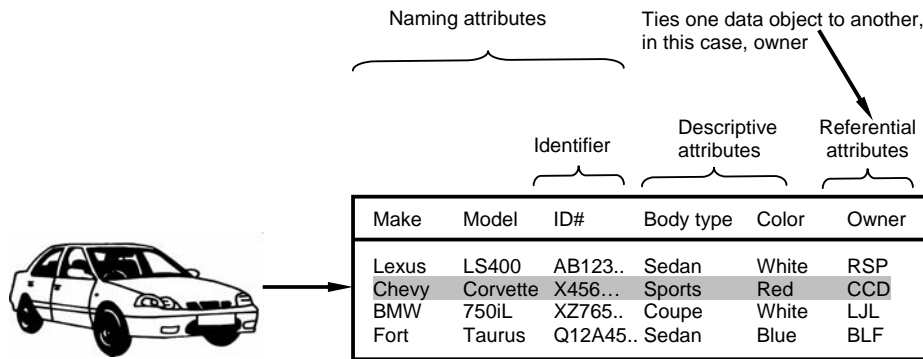


Figure 15.3 Tabulator representation of data objects

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for car described above might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case the attributes for car might also include ID#, body type, and color, but many additional attributes (e.g., interior code, driver train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

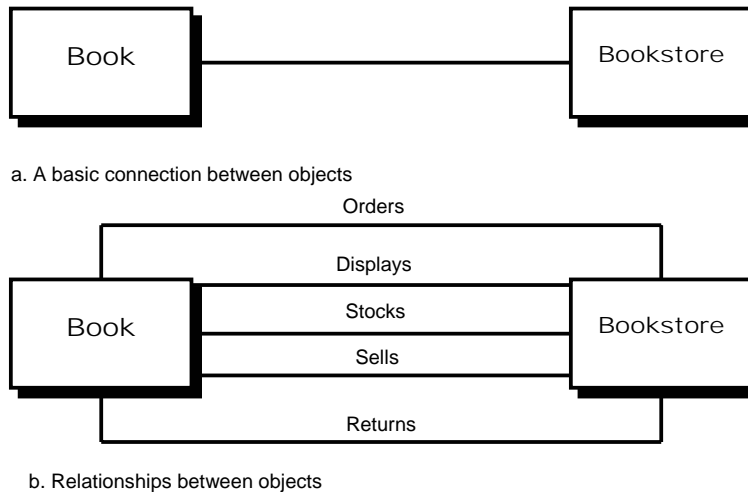


Figure 15.4 Relationships

Relationship Data object are connected to one another in a variety of different ways. Consider two data objects, book and bookstore. Those objects can be represented using the simple notation illustrated. A connection is established between book and bookstore because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set object relationship pairs that define the relevant relationships.

- a bookstore orders books
- a bookstore displays books
- a bookstore stocks books
- a bookstore sells books
- a bookstore returns books

The relationships orders, displays, stocks, sells, and returns define the relevant connections between book and bookstore. Figure 15.4b illustrate three object-relationship pairs graphically.

It is important to note that object relationship pairs are bi-directional; that is they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

15.3.2 Cardinality and Modality

the basic elements of data modeling – data objects, attributes and relationships—provide the basis for understanding the information domain of a problem. However additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object relationship pairs that bind them. But a simple pair that states: **object X** relates to **object Y** does not provide enough information for software engineering purposes. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y**. This leads to a data-modeling concept called cardinality.

Cardinality The data model must be capable of representing the number of occurrences of objects in a given relationship. Tillmann [TIL93] defines the cardinality of an object relationship pair in the following manner:

Cardinality is the specification of the number occurrence of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply ‘one’ or ‘many’ For example a husband can have only one wife while a parent can have many children. Taking into consideration all combination of ‘one’ and ‘many’ two [objects] can be related as

- One-to-one (1:1) – An occurrence of [object] ‘A’ can relate to one and only one occurrence of [object] ‘B’ can relate to only one occurrence of ‘A’. For example, a husband can have only one wife and a wife only one husband.
- One-to-many (1:N) – One occurrence of [object] ‘A’ can relate to more than one occurrences of [object] of ‘B’; but an occurrence of ‘B’ can relate to only one occurrence of ‘A’. For example, a mother can have many children, but a child can have only one mother.
- Many-to-many – An occurrence of [object] ‘A’ can relate to one or more occurrences of ‘B’; while an occurrence of ‘B’ can relate to one or more occurrences of ‘A’. For example an uncle can have many nephews, while a nephew can have many uncles

Cardinality defines “ the maximum number of object relationships that can participate in a relationship”[TIL93]. It does not; however provide an indication of whether or not a particular data object must participate in the relationship. To specify this information the data model adds modality to the object relationship pair.

Modality The modality of a relationship is zero if there is no explicit need for the relationship to occur to the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However if the problem is complex, multiple repair actions may be required. Figure 15.5 illustrates the relationship cardinality and modality between the data objects customer and repair action.

In the figure, a 1 to many cardinality relationship is established. That is a single customer can be provided with zero or many repair actions.

On the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar indicates 1, and the three-pronged fork indicates many. Modality is indicated by the symbols that are further away from the data object rectangles. The second vertical bar on the left indicates that there must be a customer of a repair action to occur. The circle on the right indicates that there may be no repair action required for the type of problem reported by the customer.

15.3.3 Entity-Relationship Diagrams

The object-relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the entity relationship diagram (ERD) ⁵ The ERD was originally proposed by Peter Chen [CHE77] for the design of relational database systems and has been extended by others. A set of primary components is identified for the ER: data objects, attributes, relationships and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

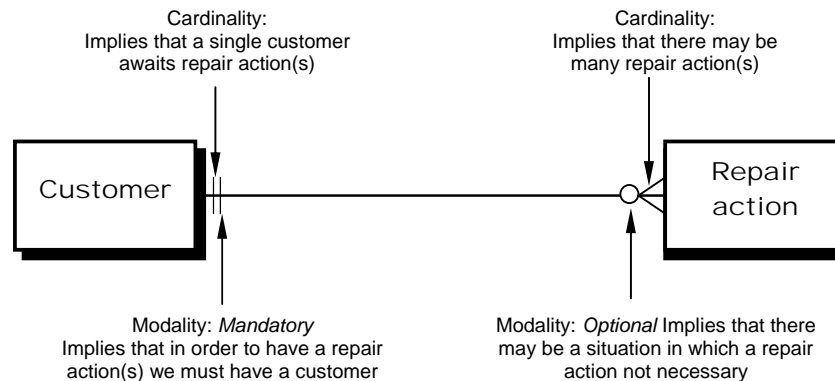


Figure 15.5 Cardinality and modality

Rudimentary ERD notation has already been introduced in Section 15.3 Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD the connecting line contains a diamond that is labeled with the relationship connection between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality. (Section 15.3.2)

The relationship between data objects card and manufacturer would be represented as shown in Figure 15.6 One manufacturer build one or many cars. given the context implied by the ERD, the specification of the data object car (see the data object table) By examining the symbols at the end of the connection line between objects it can be seen that the modality of both occurrences is mandatory (the vertical lines).

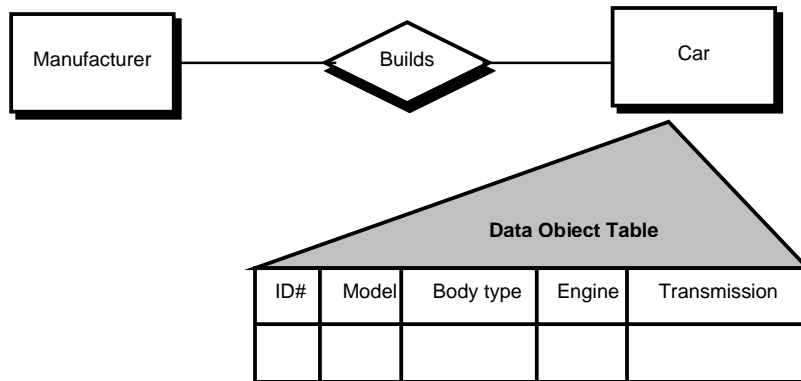


Figure 15.6 A simple ERD and data object table

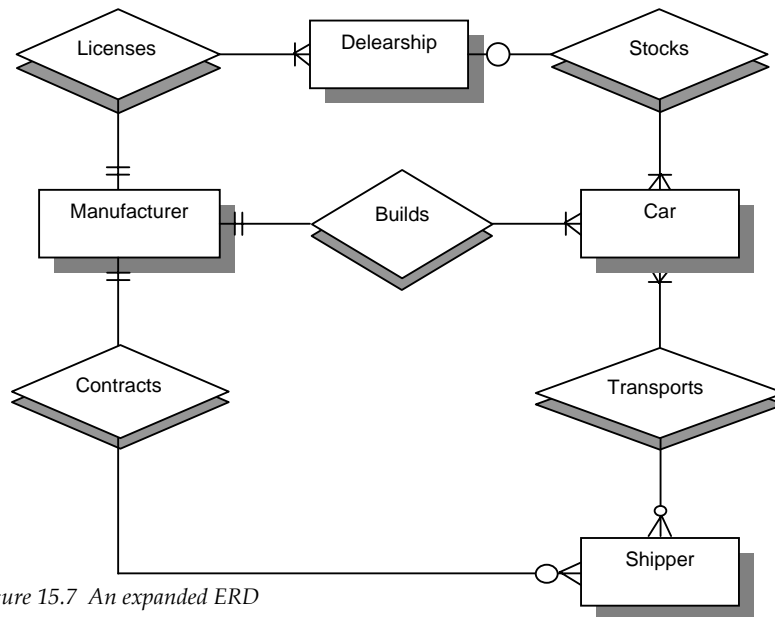


Figure 15.7 An expanded ERD

Expanding the model, we represent a grossly oversimplified ERD (Figure 15.6) of the distribution element of the automobile business. New data objects shipper and dealership are introduced. In addition new relationship transports, contracts, licenses and stocks—indicate how the data objects contained in the ERD would have to be developed according to the rules introduced earlier in this chapter.

In addition to the basic ERD notation introduced in Figures 15.6 and 15.7 the analyst can represent data object type hierarchies. In many instances a data object hierarchies. In many instances, a data object may actually represent a class or category of information For example; the data object car can be categorized as domestic, European, or Asian. The ERD notation shown in Figure 15.8 represents this categorization in the form of a hierarchy.

ERD notation also provides a mechanism that represents the associativity between objects. An associative data object is represented as shown in Figure 15.9. In the figure the data objects that model individual subsystem are each associated with the data object car.

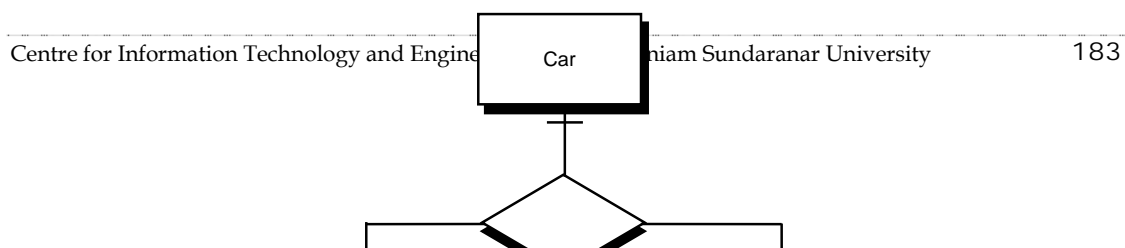


Figure 15.8 Data object type hierarchies

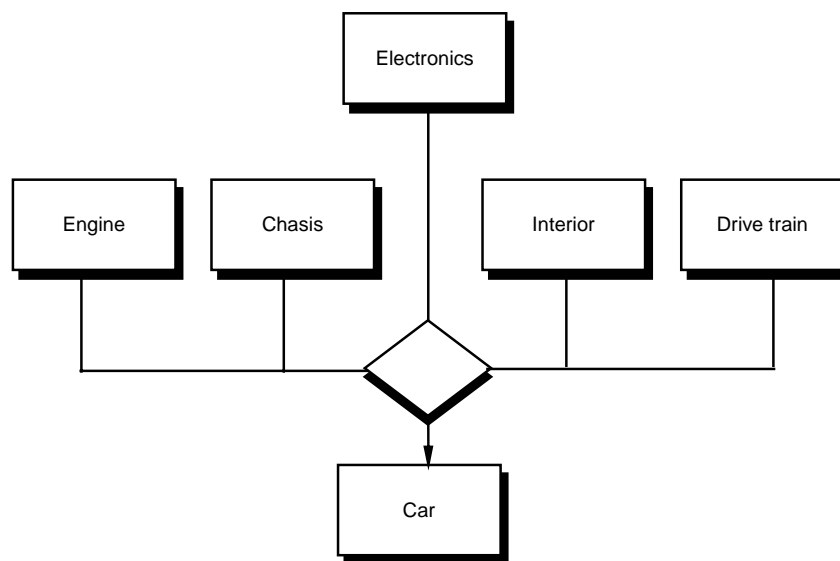


Figure 15.9 Associating data objects

Data modeling and the entity relationship diagram provide that analysis with a concise notation for examining data within the context of a data processing application. In most cases, the data modeling approach is used to create one piece of the analysis model, but it can also be used for database design and to support any other requirements analysis method.

15.4 Functional Modeling and Information Flow

Information is transformed as it flows through a computer based system. The system accepts input in a variety of forms; applies hardware, software and human elements to transform input into output and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from a CD-ROM. The transforms may comprise approach of an expert system. Output may light a single LED or produce a 200-page report. In effect we can create a flow model for any computer-based system, regardless of size and complexity.

Structured analysis began as an information flow modeling technique. A computer-based system is represented as an information transform as shown in Figure 15.10. Overall function of the system is represented as a single information transform noted as a bubble in the figure. One or more inputs shown as labeled arrows originate from external entities represented as a box. The input drives the transform to produce output information (also represented as labeled arrows) that is passed to the external entity. It should be noted that the model might be applied to the entire system or to the software element only. The key is to represent the information fed into and produced by the transform.

15.4.1 Data Flow Diagrams

As information moves through software it is modified by a series of transformations. A data flow diagram is a graphical technique that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram is illustrated in figure 15.10. The DFD is also known as a data flow graph or a bubble chart.

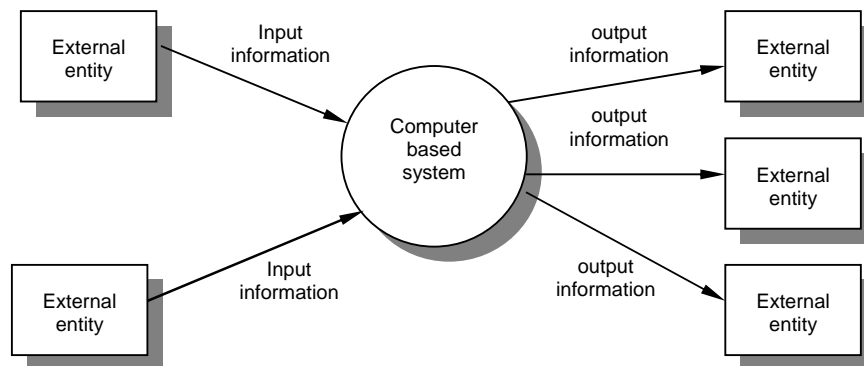


Figure 15.10 Information flow

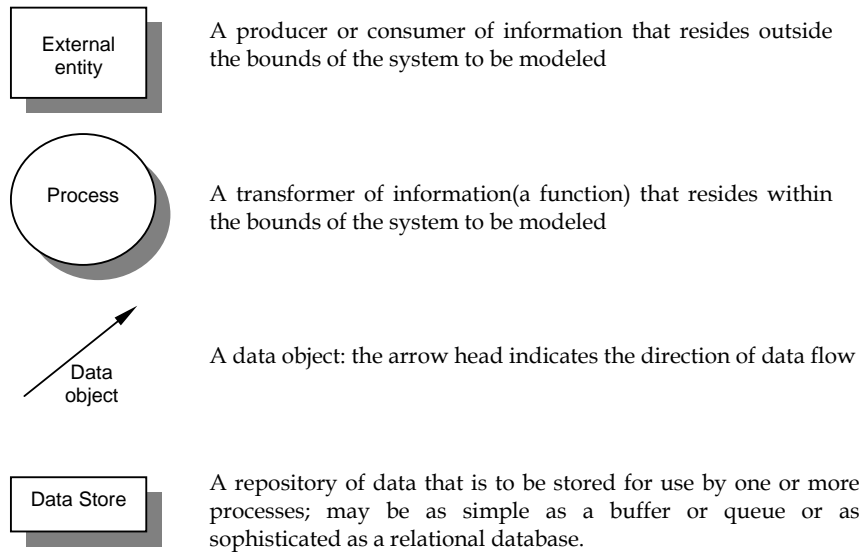


Figure 15.11 Basic D notation

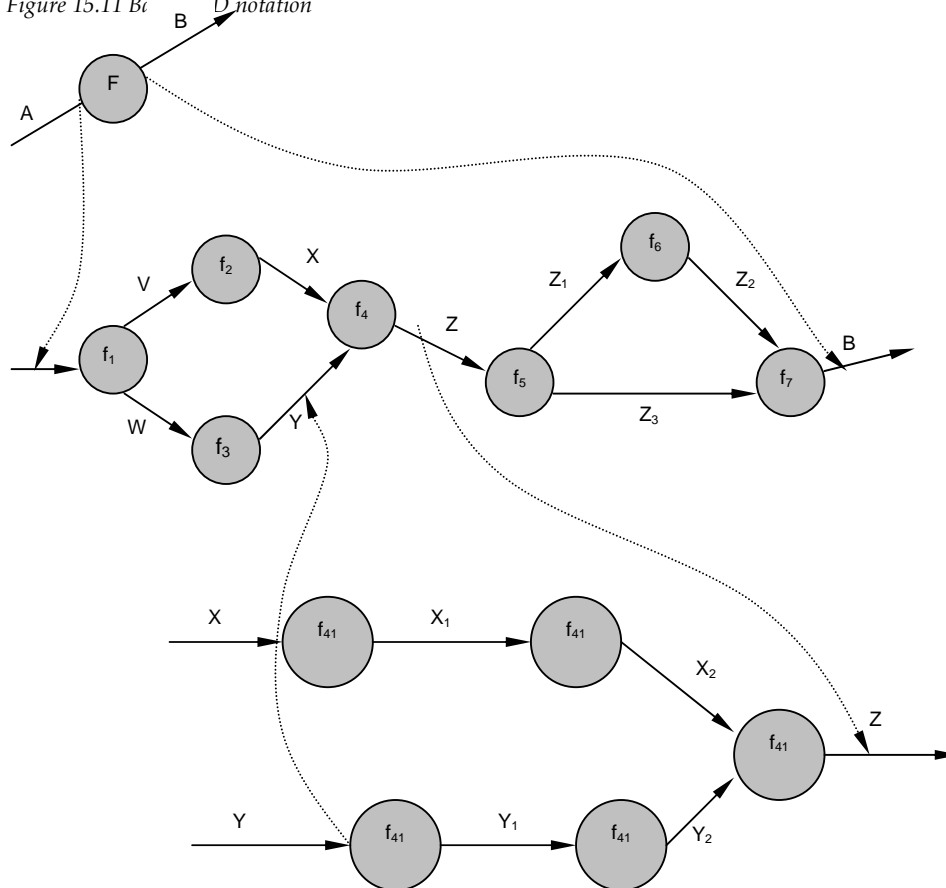


Figure 15.12 Information flow refinement

The data flow diagram may be used to represent a system or software at any level of abstraction. In fact,

DFDs may be partitioned into levels that represent increasing information flow and functional detail. Therefore the DFD provides a mechanism for functional modeling as well as information flow modeling. In so doing it satisfies the second operational analysis principle (i.e., creating a functional model) .

A level 0 DFD also called a fundamental system model or a context model represents the entire software element as a single bubble input and output data indicated by incoming outgoing arrows respectively. Additional processes and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example a level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 are sub functions of the overall system depicted in each context model.

The basic notation used to create a DFD is illustrated in Figure 15.11. A rectangle is used to represent an external entity that is a system element (e.g., hardware, a person, another program) or another system that produces information for transformation by the software or receives information produced by the software. A circle represents a process or transform that is applied to data and changes it in some way. An arrow represents one or more data items or data objects. All arrows on a data flow diagram should be labeled. The double line represents data store— stored information that is used by the software. The simplicity of DFD notation is one reason why structured analysis techniques are the most widely used.

It is important to note that no explicit indication of the sequence of processing is supplied by the diagram. Procedure or sequence may be implicit in the diagram, but explicit procedural representation is generally delayed until software design.

As we noted earlier each of the bubbles may be refined or layered to depict more detail. Figure 15.12 illustrates this concept. A fundamental model for system F indicates the primary input is A and ultimate output is B. We refine the F model into transforms f_1 to f_7 . Note that information flow continuity must be maintained that is input and output to each refinement must remain the same. The concept sometimes called balancing is essential for the development of consistent models. Further refinement of f_4 depicts detail in the form of transforms f_{41} to f_{45} . Again the input (X, Y) and output (Z) remain unchanged.

The data flow diagram is a graphical tool that can be very valuable during software requirements analysis. However the diagram can be misinterpreted if its function is confused with the flowchart. A data flow diagram depicts information flow without explicit representation of procedural logic (e.g., conditions or loops). It is not a flowchart with rounded edges!

The basic notation used to develop a DFD is not in itself sufficient to describe requirements for software. For example an arrow shown in a DFD represents a data object that is input to or output from a process. A data store represents some organized collection of data. But what is the content of the data implied by the arrow to depict by the store? If the arrow (or the store) represents a collection of objects, what are they? These questions are answered by applying another component of the basic notation for structured analysis the data dictionary. The format and use of the data dictionary are presented later in this chapter.

Finally the graphical notation represented in Figure 15.11 must be augmented with descriptive text. A processing specification can be used to specify the processing details implied by a bubble within a DFD. The processing specification describes the input to a function the algorithm that is applied to the input and the output that is produced. In addition the PSPEC indicates restrictions and limitations imposed on the process and design constraints that may influence the way in which the process will be implemented.

15.4.2 Extensions for Real-Time Systems

Many software applications are time dependent and process as much or more control oriented information as data. A real time system must interact with the real world in a time frame dictated by the real world. Aircraft avionics manufacturing process control consumer products and industrial instrumentation are but a few of hundreds of real-time software applications.

To accommodate the analysis of real-time software, a number of extensions to the basic notation for structured analysis have been proposed. These extensions, developed by Ward and Mellor and Hatley and Pirbhai and shown in figure 15.13 enable the analyst to represent control flow and control processing as well as data flow and processing

15.4.3 Ward and Mellor Extensions

Ward and Mellor extend basic structured analysis notation to accommodate the following demands imposed by a real-time system:

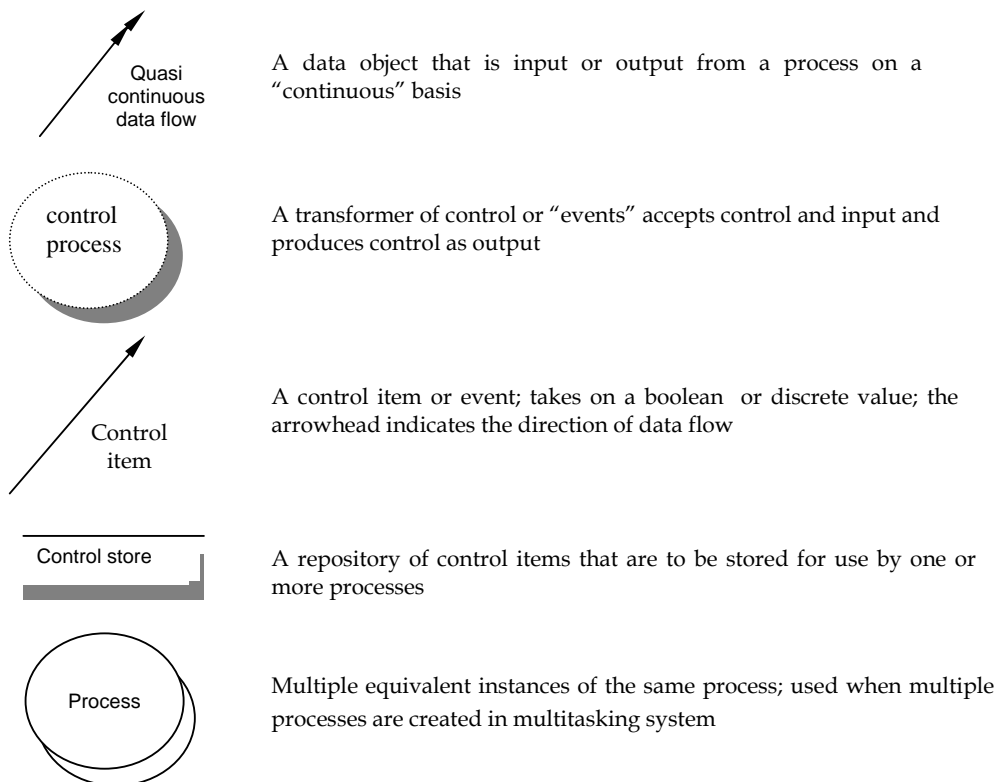


Figure 15.13 Extended structured analysis notation for real-time systems developed by Ward and Mellor

In a significant percentage of real-time application, the system must monitor time-continuous information generated by some real world process. For example a real time test monitoring system for gas turbine engines might be required to monitor turbine speed, combustor temperature and a variety of pressure probes on a continuous basis.

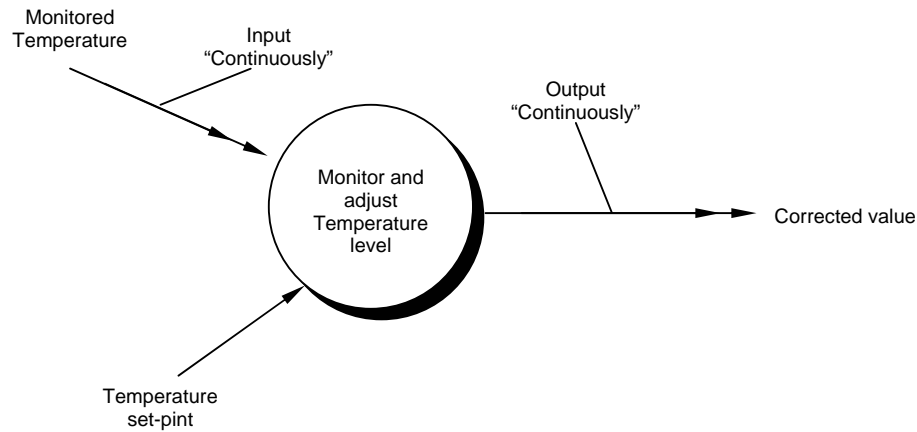


Figure 15.14 Time continuous Data Flow

Conventional data flow notation does not make a distinction between discrete data and time – continuous data. An extension to basic structured analysis notation, shown in figure 15.14 provides a mechanism for representing time-continuous flow, and a single headed arrow is used to indicate discrete data flow. In the figure, monitored temperature is measured continuously while a single value for temperature set-point is also provided. The process shown in figure produces a time-continuous output, corrected value.

The distinction between discrete and time-continuous data flow has important implication for both the system engineer and the software designer. During the creation of the system model, a system engineer will be better able to isolate those processes that may be performance critical (it is likely that the input and output of time-continuous data will be performance sensitive). As the physical or implementation model is created, the designer must establish a mechanism for collection of time-continuous data. Obviously, the digital system collects data in a quasi-continuous fashion using techniques such as high-speed polling. The notation indicates where analog to digital hardware will be required and which transforms are likely to demand high-performance software.

In conventional data flow diagrams, control or event flows are not represented explicitly. In fact, the analyst is cautioned to specifically exclude the representation of control flow from the data flow diagram. This exclusion is overly restrictive when real-time applications are considered and for this reason, a specialized notation for representing event flows and control processing has been developed. Continuing the convention established for data flow diagrams, data flow is represented using a solid arrow. Control flow however is represented using a dashed or shaded arrow. A process that handles only control flows, called a control process, is similarly represented using a dashed bubble.

Control flow can be input directly to a conventional process or into a control process. Figure 15.15 illustrates control flow and processing as it would be represented using Ward and Mellor notation. The figure illustrates a top-level view of data and control flow for a manufacturing cell. As components to be assembled by a robot are placed on fixtures, a status bit is set within a parts status buffer (a control store) that indicates the presence or absence of each component. Even information contained within the parts status buffer is passed as a bit string to a process, monitor fixture and operator interface. The process will

read operator commands only when the control information, bit string, indicates that all fixtures contain components. An event flag, start/stop flag, is sent to robot initiation control, a control process that enables further command processing. Other data flows occur as a consequence of the process activate event that is sent to process robot commands.

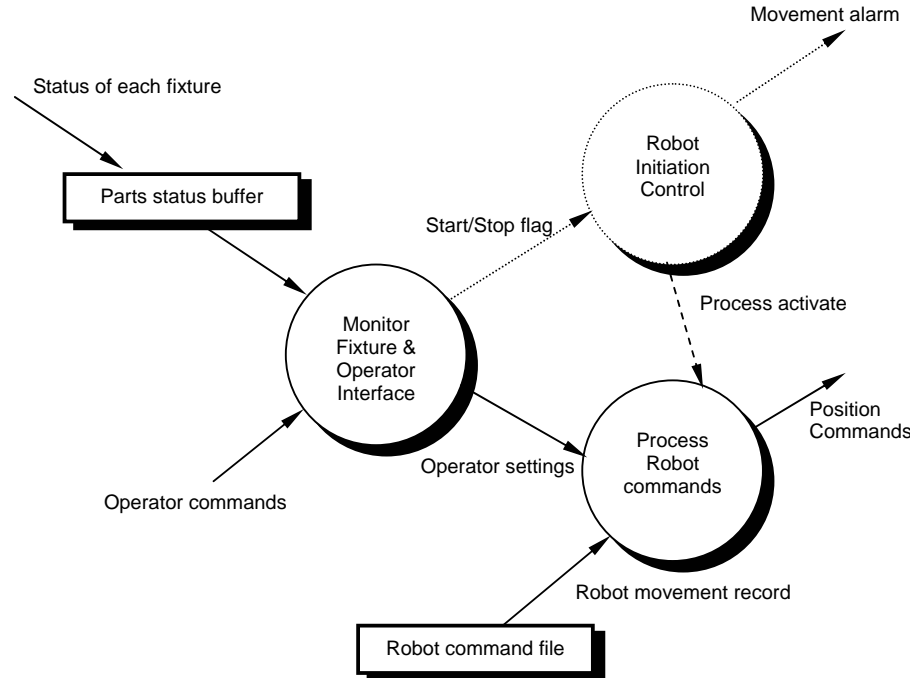


Figure 15.15 Data and control flows using ward and mellor notation

In some situations multiple instance of the same control or data transformation process may occur in a real-time system. This can occur in a multitasking environment when tasks are spawned as a result of internal processing or external events. For example a number of part status buffers may be monitored so that different robots can be signaled at the appropriate time. In addition, each robot may have its own robot control system. The Ward and Mellor notation used to represent multiple equivalent instances of the same process is shown in figure 15.13.

15.5 Hatley and Pirbhai Extensions

The Hatley and Pirbhai extensions to basic structured analysis notation focus less on the creation of additional graphical symbols and more on the representation and specification of the control oriented aspects of the software. Unlike ward and mellor hatley and pirbhai suggest that dashed and solid notation be represented separately. Therefore a control flow diagram (CFD) is defined. The CFD contains the same process as the DFD, but shows control flow rather than data flow. Instead of representing control process directly within the flow model, a notational reference to a control specification is used. In essence the solid bar can be viewed as a “window” into an “executive” that controls the process represented in the DFD based on the event that is passed through the window. The CSPEC, described in detail in section 12.6.4 is used to indicate (1) how the software behaves when an event or control signal is sensed and (2) which process are invoked as a consequence of the occurrence of the event. A process specification is used to describe the inner workings of a process represented in a flow diagram.

Using the notation described in figure 15.11 along with additional information contained in PSPECs and

CSPECs hatley and pirbhai create a model of a real time system. Data flow diagrams are used to represent data and the processes that manipulate it. Control flow diagrams show how events flow among processes and illustrate those external events that cause various process to be activated. The interrelationship between the process various processes and control models. The process model is connected to the control model through data conditions. The control model is connected to the process model through process activation information contained in the CSPEC.

A data condition occurs whenever data input to a process results in a control output. The process check & convert pressure implements the algorithm described in the PSPEC pseudocode shown. When the absolute tank pressure is greater than an allowable maximum, an above pressure event is generated. Note that when hatley and pirbhai notation is used, the data flow is shown as part of a DFD. While the control flow is noted separately as part of a control flow diagram. To determine what happens when this event occurs, we must check the CSPEC.

The control specification contains a number of important modeling tools. A process activation table is used to indicate which processes are activated by a given event that flows through the vertical bar. For example, a process activation table(PAT) might indicate that the above pressure event would cause a process reduce tank pressure to be invoked. In addition to the PAT the CDPEC may contain a state transition diagram(STD). In addition to the PAT, the CSPEC may contain a state transition diagram. The STD is a behavioral model that relies on the definition of a set of system states and is described in the following section.

15.6 Short Summary

- Structured analysis the most widely used of requirements modeling methods relies on data modeling and flow modeling to create the basis for a comprehensive analysis model.
- Data and control flow diagrams are used as a basis for representing the transformation of data and control.

15.7 Brain Storm

1. Explain briefly about Data Modeling ?
2. Discuss about cardinality and Modality ?
3. Write a note on Functional Modeling and Information flow?
4. Explain briefly about Data flow Diagram ?
5. Short note on Hatley and Pirbhai Extension ?



Lecture 16

Analysis Modeling - II

Objectives

In this lecture you will learn the following

- ✧ About Behavioral Modeling
- ✧ About Mechanics of Structured Analysis

Coverage Plan

Lecture 16

- 16.1 Snap Shot
- 16.2 Behavioral modeling
- 16.3 The Mechanics Of Structured Analysis
- 16.4 Creating an entity relationship diagram
- 16.5 Creating a Data Flow Model
- 16.6 Creating a Control Flow Model
- 16.7 The Control specification
- 16.8 The process Specification
- 16.9 Short Summary
- 16.10 Brain Stom

16.1 Snap Shot

In this lecture, we are going to learn about Behavioral Modeling, Mechanics of Structured Analysis, Entity Relationship Diagram, Data Flow Model and Control Flow Model and also about Control and Process Specification.

16.2 Behavioral Modeling

Behavioral modeling is an operational principle for all requirements analysis methods. Yet only extended versions of structured analysis provide a notation for this type of modeling. The state transition diagram represents the behavior of a system by depicting its states and the events that cause the system to change state. In addition, the STD indicated what actions are taken as a consequence of a particular event.

A state is any observable mode of behavior. For example states for a monitoring and control system for pressure vessel might be monitoring state, alarm state, pressure release state, and so on. Each on these states represents a mode of behavior of the system. A state transition diagram indicated how the system moves from state to state.

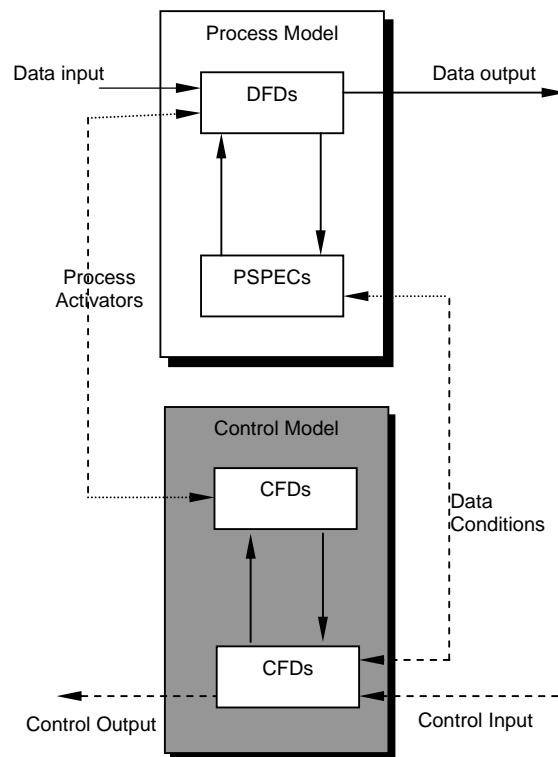


Figure 16.1 The relationship between data and control models

To illustrate the use of the hatley and pirbhai control an behavioral extension, consider software embedded within an office photocopying machine. The photocopier performs a number of functions that are implied by the level 1 DFD shown in figure 16.3. It should be noted that additional refinement of the data flow and definition of each data item would be required.

The control flow for the photocopier software is shown entering and exiting individual process and the CSPEC “window”. For example, the paper feed status and start/stop events flow into the CSPEC bar. This implies that each of these events will cause some process represented in the CFD to be activated. If

we were to examine the CSPEC internals, the start/stop event would be shown to activate/deactivate the manage copying process. Similarly, the jammed event (part of paper feed status) would activate perform problem diagnosis. It should be noted that all vertical bars within the CFD refer to the same CSPEC.

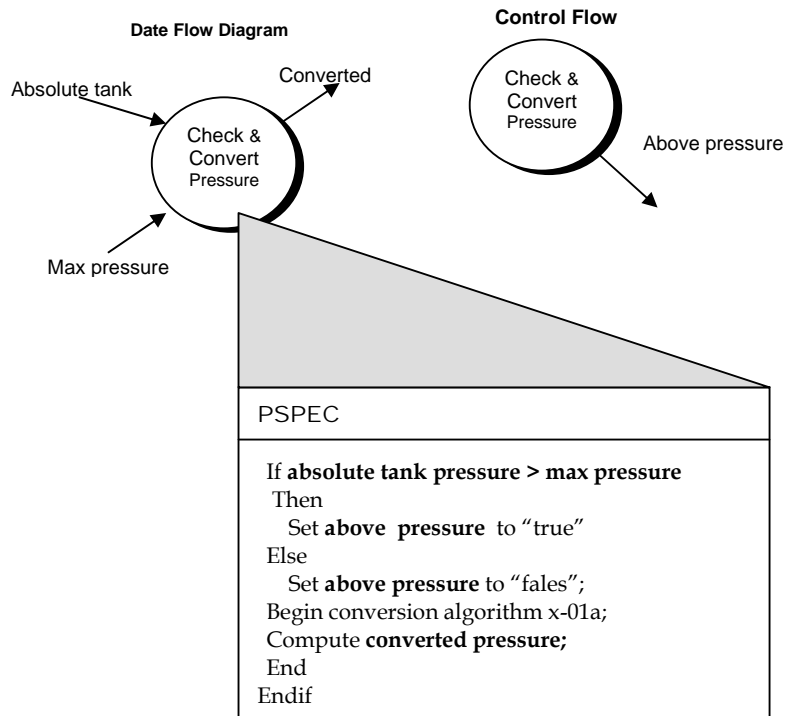


Figure 16.2 Data Conditions

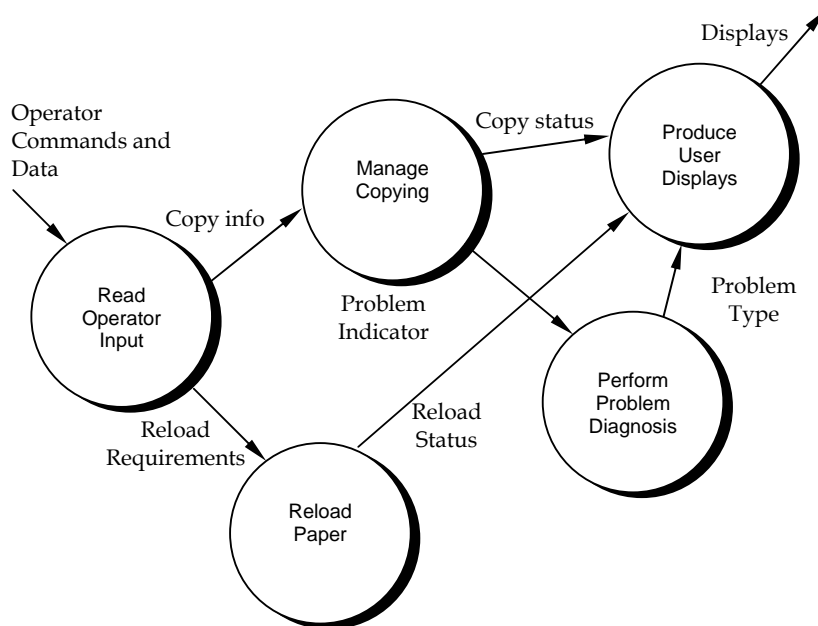


Figure 16.3 Level 1 CFD for photocopier Software

An event flow can be input directly into a process as shown with reprofault. However, this flow does not activate the process, but rather provides control information for the process algorithm. Data flow arrows have been lightly shaded for illustrative purposes, but in reality they are not shown as part of a control flow diagram.

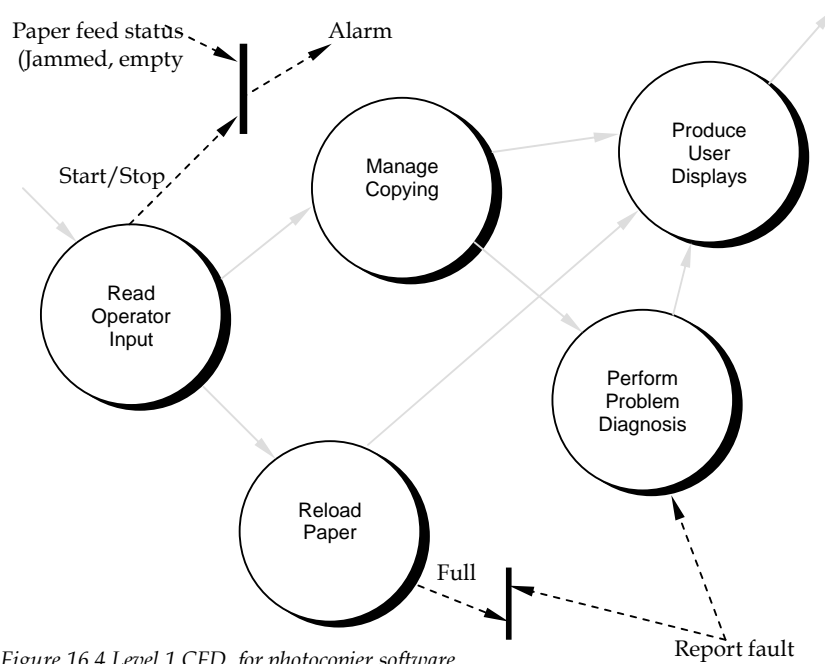


Figure 16.4 Level 1 CFD for photocopier software

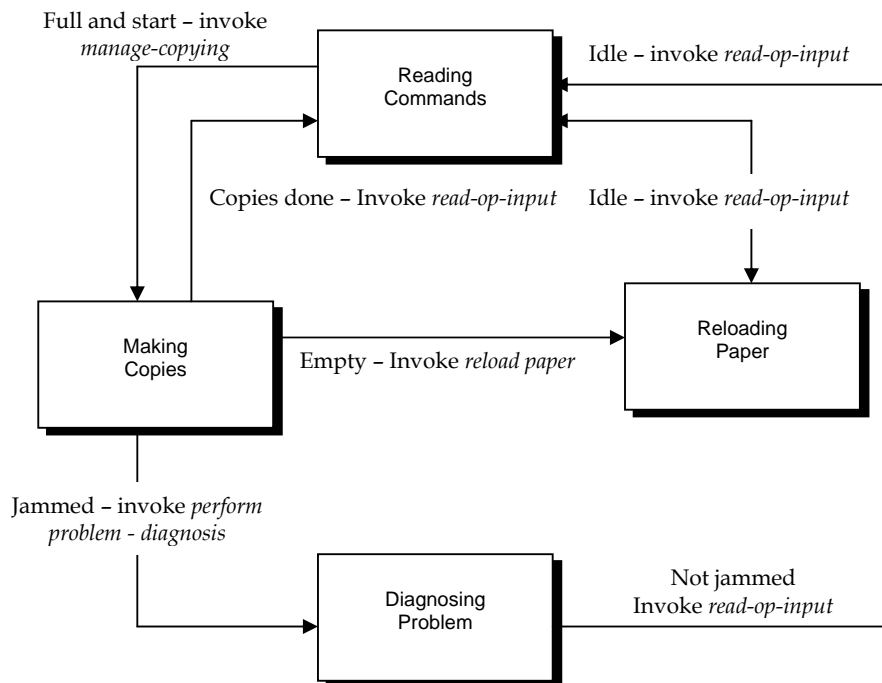


Figure 16.5 Simplified state transition diagram for photocopier software

A simplified state transition diagram for the photocopier software described above is shown in figure 16.5. The rectangles represent system states and the arrows represent transitions between states. Each arrow is labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. Therefore, when the paper tray is full and the start button is pressed, the system moves from the reading commands state to the making copies state. Note that states do not necessarily correspond to processes on a one to one basis.

For example, the state making copies would encompass both the manage copying and produce user displays processes shown in figure 16.4.

16.3 The Mechanics of Structured Analysis

In the previous section, we discussed basic and extended notation for structured analysis. To be used effectively in software requirements analysis, this notation must be combined with a set of heuristics that enable a software engineer to derive a good analysis model, to illustrate the use of these heuristics an adapted version of the hatley and pirbhai extensions to the basic structured analysis notation will be used throughout the remainder of this lecture.

In the sections that follow we examine each of the steps that should be applied to develop complete and accurate models using structured analysis. Through this discussion, the notation introduced will be used, and other notational forms, alluded to earlier, will be presented in some detail.

16.4 Creating an Entity Relationship Diagram

The entity relationship diagram enables a software engineer to fully specify the data objects that are input and output from a system, the attributes and define the properties of these objects, and the relationships between objects. Like most elements of the analysis model, the ERD is constructed in an iterative manner. The following approach is taken;

1. During requirements gathering, customers are asked to list the “things” that the application or business process addresses. These “things” evolve into a list of input and output data objects as well as external entities that produce or consume information.
2. Taking the objects one at a time, the analyst and customer define whether or not a connection exists between the data object and other objects.
3. Wherever a connection exists, the analyst and customer create one or more object relationship pairs.
4. For each object relationship pair, cardinality and modality are explored.
5. Step 2 through 4 are continued iteratively until all object relationship pairs have been defined. It is common to discover omissions as this process continues. New objects and relationships will invariably be added as the number of iterations grows.
6. The attributes of each entity are defined.
7. An entity relationship diagram is formalized and reviewed.
8. Steps 1 through 7 are repeated until data modeling is complete

To illustrate the use of these basic guidelines, the safehome security system example is discussed in previous lecture. A processing narrative for safehome is reproduced below:

Safe Home software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the Safe Home control panel.

During installation the Safe Home control panel is used to 'program' and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.

When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting and the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until telephone connection is obtained.

All interaction with Safe Home is managed by a user-interaction subsystem that receives input provided through the keypad and function keys, displays prompting messages and system status on the LCD display. Keyboard interaction takes the following form...

Discussion between the analyst and the customer indicate the following list of things that are relevant to the problem.

- Homeowner
- Control panel
- Sensors
- Security system
- Monitoring service

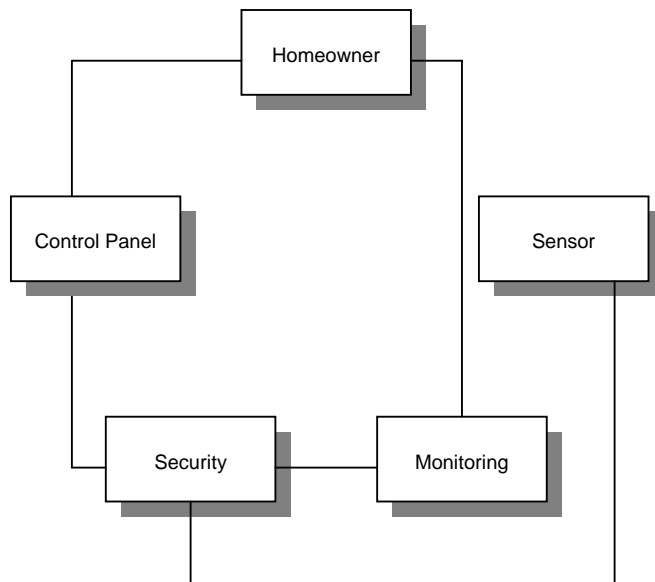


Figure 16.6 Establishing connections

Taking these things one at a time, connections are explored. To accomplish this, each object is drawn and lines connecting the objects are noted. For example, figure 16.6 shows that a direct connection exists between the homeowner and control panel, security system, and monitoring service. A single connection exists between sensor and security system, and so forth.

Once all connections have been defined, one or more object relationship pairs are identified for each connection. For example, the connection between sensor and security system is determined to have the following object relationship pairs.

Security system *monitors* **sensors**
Security system *enables/disables* **sensor**
Security system *tests* **sensor**
Security system *programs* **sensor**

Each of the above object relationship pairs is analyzed to determine cardinality and modality. For example in the object relationship pair security system monitors sensor, the cardinality between security system and sensor is one to many. The modality is one occurrence of security system and at least one occurrence of sensor. Using the ERD notation introduced the connecting line between security system and sensor would be modified as shown in figure 16.7 similar analysis would be applied to all other data objects.

Each object is studied to determine its attributes. Since we are considering the software that must support SafeHome the attributes should focus on data that must be stored to enable the system to operate. For example, the sensor object might have the following attributes: sensor type, internal identification number, Zone location and alarm level.

16.5 Creating a Data Flow Model

The data flow diagram (DFD) enables the software engineer to develop models of the information domain and functional domain at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system thereby accomplishing the fourth operational analysis principle. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during derivation of a data flow diagram; (1) The level data flow diagram should depict the software/system as a single bubble; (2) primary input and output should be carefully noted; (3) refinement should begin by isolating candidate processes data objects and stores to be represented at the next level; (4) all arrows and bubbles should be labeled with meaningful names; (5) information flow continuity must be maintained from level to level; and (6) one bubble data time should be refined. There is a natural tendency to overcomplicate the data flow diagram. This occurs when the analyst attempts to overcomplicate the data flow diagram. This occurs when the analyst attempts to show too much detail too early or represents procedural aspects of the software in lieu of information flow.

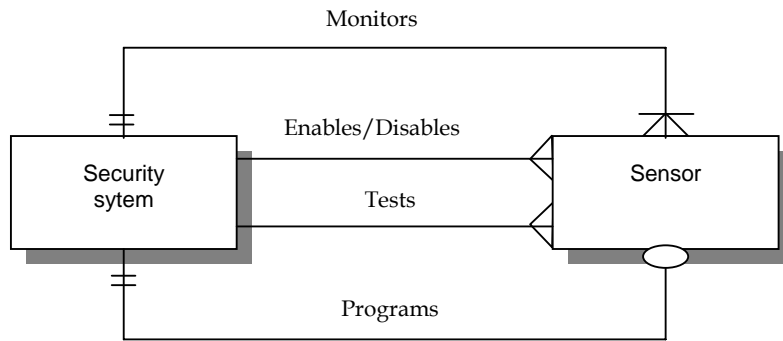


Figure 16.7 Developing relationships and cardinality / modality

Again considering the Safe Home product a level 0 DFD for the system is shown in Figure 16.8. The primary external entities (boxes) produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object type hierarchies. For example user commands and data encompasses all configuration commands all activations/deactivation commands all miscellaneous interactions and all data that are input to qualify or expand a command.

The level 0 DFD is now expanded into a level 1 model. But how do we proceed? A simple, yet effective approach is to perform a “Grammatical parse” on the processing narrative that describes the context level bubble. That is we isolate all nouns and verbs in the narrative. To illustrate we again reproduce the processing narrative underlining the first occurrence of all nouns and italicizing the first occurrence of all verbs. (It should be noted that nouns and verbs that are synonyms or have no direct bearing on the modeling process are omitted).

SafeHome software enables the homeowner to configure the security system when it is installed monitors all sensors connected to the security system and interacts with the homeowner through a keypad and function keys contained in the SafeHome control panel.

During installation, the SafeHome control panel is used to “program” and configure the system. Each sensor is assigned a number and type a master password in programmed for arming and disarming the system and telephone numbers are input for dialing when a sensor event occurs.

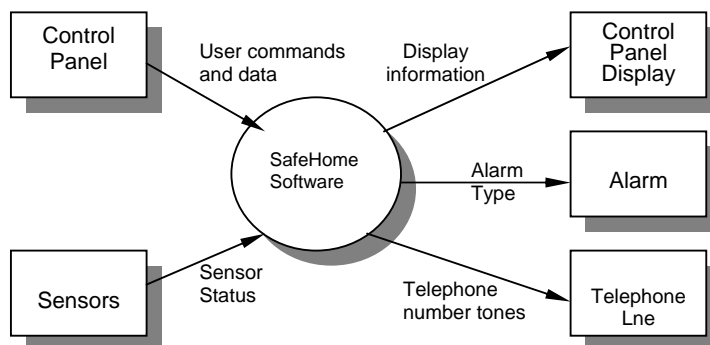


Figure 16.8 Context level DFD for safehome

When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities the software dials a telephone number of a monitoring service, provide information about the location, reporting and the nature of the event 20 seconds until telephone connection is obtained.

All interaction with SafeHome is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, display prompting messages and system status on the LCD display. Keyboard interaction takes the following form...

In examining the grammatical parse, we see a pattern begin to emerge. All verbs are SafeHome processes ; that is they may ultimately be represented as bubbles in a subsequent DFD. All nouns are either external entities data or control objects (arrows) or data stores (double lines) . Note further that nouns and verbs can be attached to one another (e.g., sensor assigned number and type) Therefore by performing a grammatical parse on the processing narrative for a bubble at any DFD level we can generate much useful information about how to proceed with the refinement to the next level. Using this information a level 1 DFD is shown in Figure 16.9 the context level process shown in Figure 16.8 has been expanded into seven processes derived from an examination of the grammatical parse. Similarly the information flow between processes at level 1 has been derived from the parse.

It should be noted that information flow continuity is maintained between levels 0 and 1. Elaboration of the content of inputs and output a DFD levels 0 and 1 is postponed.

The processes represented at DFD continues until each bubble performs a simple function. That is until the process represented by the bubble performs a function that would be easily implemented as a program component. In the previous lecture we discuss a concept called cohesion that can be used to assess the simplicity of a given function. For now we strive to refine DFDs until each bubble is "single minded".

16.6 Creating a Control Flow Model

For many types of data processing applications the data model and the data flow diagram, are all that is necessary to obtain meaningful insight into software requirement. As we have already noted however there exists a large class of applications that are driven by events rather than data that produce control information rather than reports or displays and that process information with heavy concern for time performance,. Such application require the use of control flow modeling in addition to data flow modeling.

The graphical notation required to create a control flow diagram (CFD) was presented. To review the approach for creating a CFED a data flow model is "stripped" of all data flow arrows¹⁰ Events and control items (dashed arrows) are then added to the diagram and a "window" (a vertical bar) into the control specification is shown. But how are events selected?

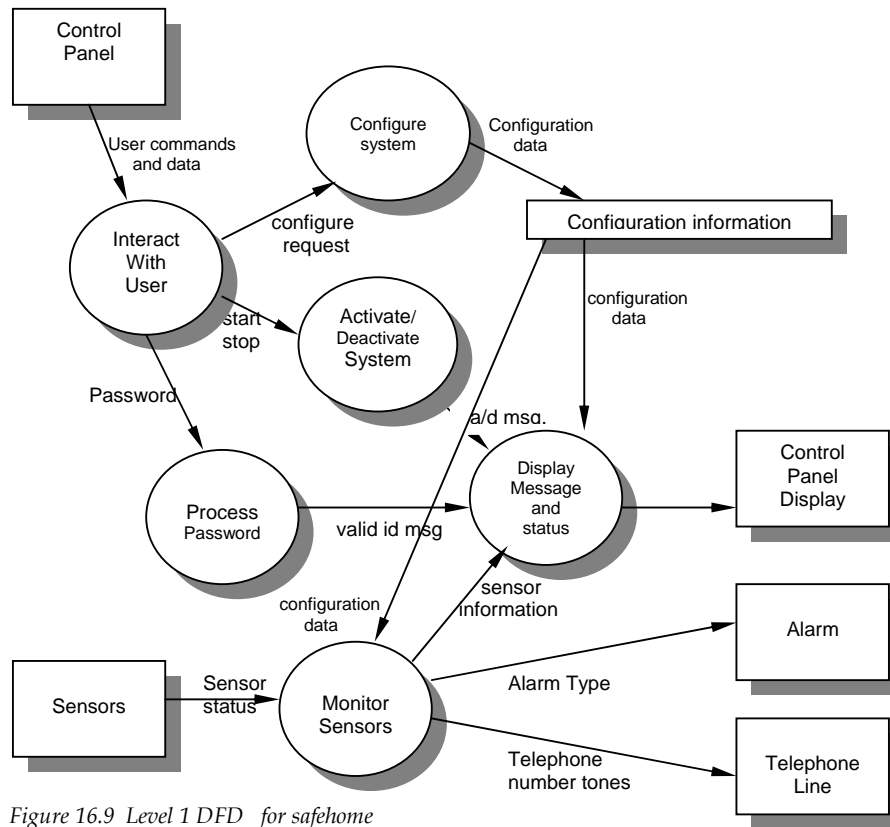


Figure 16.9 Level 1 DFD for safehome

We have already noted that an event or control item is implemented as a Boolean value (e.g., true or false, on or off, 1 or 0) or a discrete list of conditions (empty, jammed, full). To select potential candidate events the following guidelines are suggested.

- List all sensors that are “read” by the software
- List all interrupt conditions
- List all “switches” that are actuated by an operator
- List all data conditions.
- Recalling the noun-verb parse that was applied to the processing narrative review all “control items” as possible CSPEC inputs/outputs
- Describe the behavior of a system by identifying its states; identify how each state is reached and define the transitions between states.
- Focus on possible omissions – a very common error in specifying control (e.g., ask: “Is there any other way I can get to this state or exit from it?”)

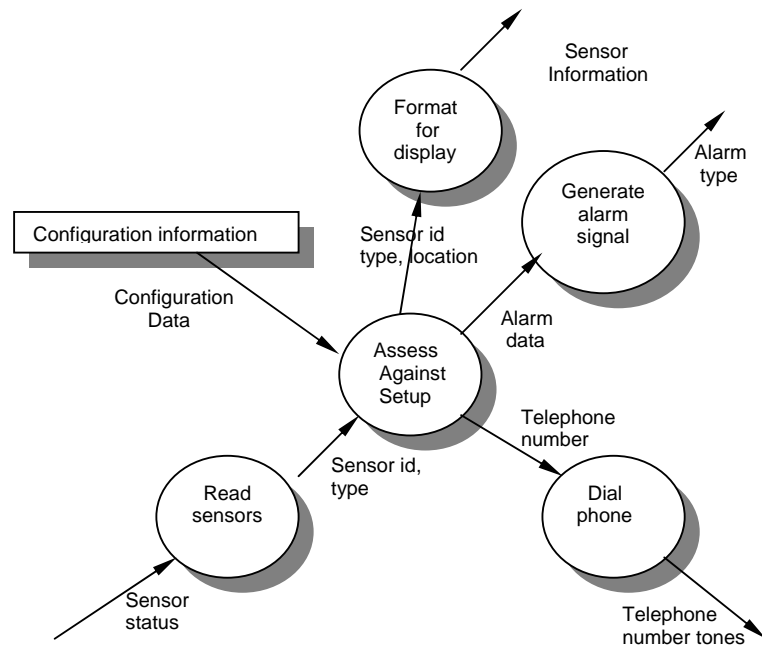


Figure : 16.10 Level 2 DFD that refines the monitor sensors process

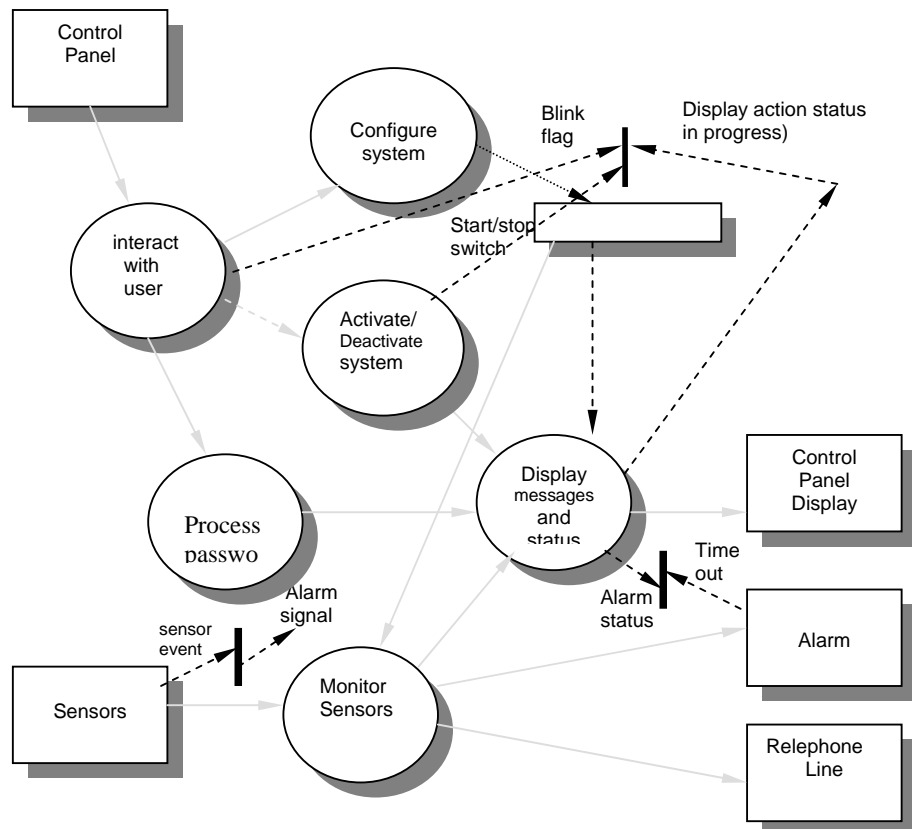


Figure 16.11 Level 1 CFD for safehome

A level 1 CFD for SafeHome software is illustrated in Figure 16.11. Among the events and control items noted are **sensor event** (i.e., a sensor has been tripped), **blink flag** (a signal to blink the LCD display) and **start/stop switch** (a signal to turn the system on or off). An event flowing into the CSPEC window from the outside world implies that the CSPEC will activate one or more of the processes shown in the CFD. When a control item emanates from a process and flows into the CSPEC window, control and activation of some other process or an outside entity is implied.

16.7 The Control Specification

The control specification (CSPEC) represents the behavior of the system (at the level from which it has been referenced) in two different ways. The CSPEC contains a state transition diagram (STD) that is a sequential specification of behavior. It can also contain a process activation table (PAT) a combinatorial specification of behavior. The underlying attributes of the CSPEC were introduced. It is now time to consider an example of this important modeling notation for structured analysis.

Figure 16.12 depicts a state-transition diagram for the level 1 flow model for SafeHome. The labeled transition arrows indicate how the system responds to events as it traverses the four states defined at this level. By studying the STD a software engineer can determine the behavior of the system and more important can ascertain whether there are “holes” in the specified behavior. For example the STD indicates that the only transition from the reading user input state occurs when the **start/stop switch** is encountered and a transition to the monitoring system status state occurs. Yet there appears to be no way other than the occurrence of **sensor event** that will allow the system to return to reading user input. This is an error in specifications and to determine whether there are any other anomalies.

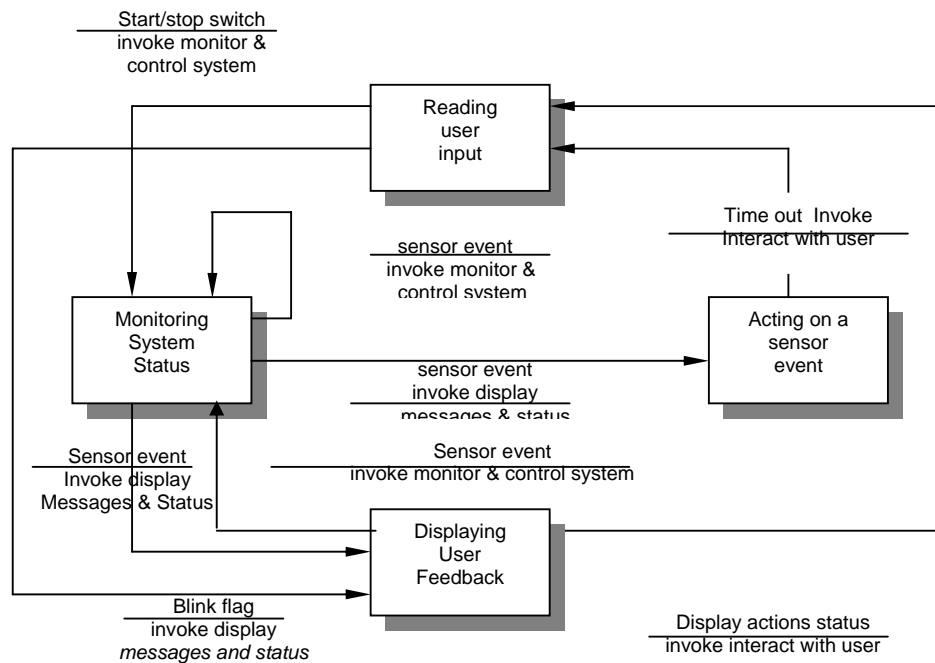


Fig 16.12 State transition diagram for safehome

A somewhat different move of behavioral representation is the process activation table(PAT) The PAT represents information contained in the STD in the context of processes not states. That is the table indicates which processes in the flow model will be invoked when an event occurs. The PAT can be used as a guide for a designer who must build an executive that controls the processes represented at this level. A PAT for the level 1 flow model of SafeHome software is shown in Figure 16.13.

The CSPEC describes the behavior of the system but it does not give us any information about the inner working of the processes that are activated as a result of this behavior. The modeling notation that provides this information is discussed in the next section.

Input events						
Sensor event	0	0	0	0	1	0
Blink flag	0	0	1	1	0	0
Start stop switch	0	1	0	0	0	0
Display action status Complete	0	0	0	1	0	0
In-progress	0	0	1	0	0	0
Time out	0	0	0	0	0	1
Output						
Alarm signal	0	0	0	0	1	0
Process activation						
Monitor and control system	0	1	0	0	1	1
Activate/deactivate system	0	1	0	0	0	0
Display messages and status	1	0	1	1	1	1
Interact with user	1	0	0	1	0	1

Figure 16.13 process activation table for safehome

16.8 The process Specification

The process specification (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text a program design language(PDL) description¹¹ of the process algorithm mathematical equations tables diagrams or charts By providing a PSPEC to accompany each bubble in the flow model, the software engineer creates a “mini-spec” that can serve as a first step in the creation of the software requirements specification and as a guide for design of the program component that will implement the process.

To illustrate the use of the PSPEC, consider a software application in which the dimension of various geometric objects are analyzed to identify the shape of the object. Refinement of a context level data flow diagram continues until level 2 processes are derived. One of these named analyzed triangle. The PSPEC for analyze triangle is first written as an English language narrative as shown in the figure. In additional algorithmic detail is desired at this stage a program design language representation may also be included as part of the PSPEC. However many believe that the PDL version should be postponed until design commences.

16.9 Short Summary

- Behavioral analysis is an operational method for all requirement analysis method.
- The entity relationship diagram enables a software to fully specify data objects that are input and output from a system, the attributes and the define the properties of the object and the relationship between the objects.
- When a sensor event is recognized the software invokes an alarm attached to the system.

16.10 Brain Storm

1. Explain briefly about the Behavioral Modeling ?
2. Explain the mechanism of Structured Analysis ?
3. How do you create an Entity Relation Diagram ?
4. How do you create Data flow and Control Flow Model ?
5. Explain briefly about control and process specification ?



Lecture 17

Analysis Modeling - III

Objectives

In this lecture you will learn the following

- ✎ About Data Dictionary
- ✎ About Overview of other Classical Analysis Methods

Coverage Plan

Lecture 17
17.1 Snap Shot
17.2 Data Dictionary
17.3 An Overview of other Classical Analysis Methods
17.4 Data Structured System Development
17.5 Jackson System Development
17.6 SADT
17.7 Short Summary
17.8 Brain Storm

17.1 Snap Shot

In this lecture we are going to learn about Data Dictionary, Data Structured System Development Jacks on system Development and also about Structured analysis and Design Techniques.

17.2 Data Dictionary

The analysis model encompasses representations of data objects function and control. In each representation data objects and /or control items play a role. Therefore it is necessary to provide an organized approach fo rrepresenting the characteristics of each data objects and control item. This is accomplished with the data dictionary.

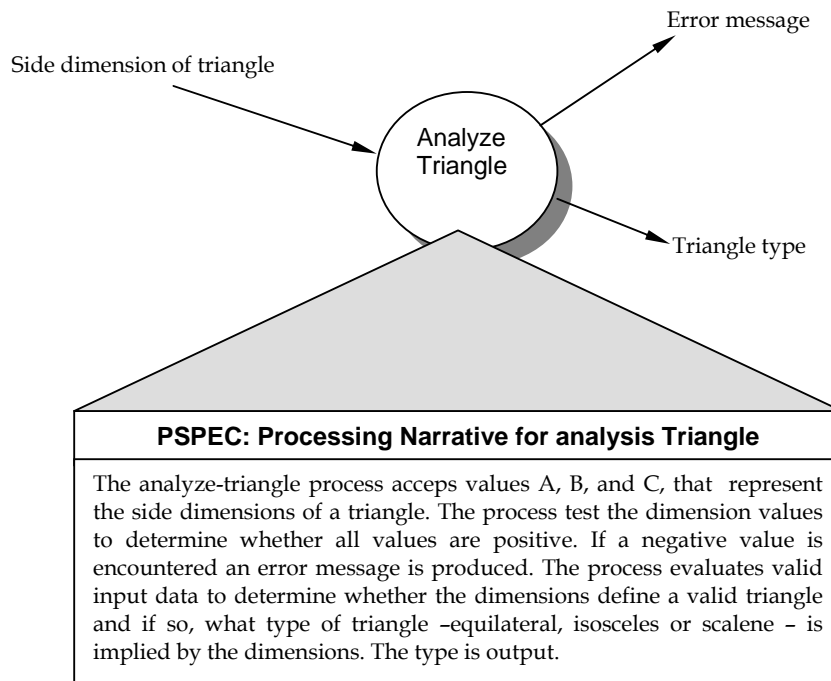


Figure 17.1 Process specification for a DFD process

The data dictionary has been proposed as a quasi –formal grammar for describing the content of objects defined during structured analysis. This important modeling notation has been defined in the following manner[YOU89]

The data dictionary is an organized listing of all data elements that are pertinent to the system with precise rigorous definitions so that both user and system analyst will have a common understanding of inputs outputs components of stores and (even) intermediate calculations.

Today the data dictionary is almost always implemented as part of a CASE “structured analysis and design too;” Although the format of dictionaries varies from tool to tool most contain the following information:

- Name—the primary name of the data or control item, the data store or an external entity.
- Alias—other names used for the first entry
- Where-used /how-used—a listing of the processes that use the data or control item and how it is

used(e.g., input to the process output from the process ,as a store as an external entity)

- Content description—a notation for representing content
- Supplementary information—other information about data types preset values(if known) restrictions or limitations etc.,

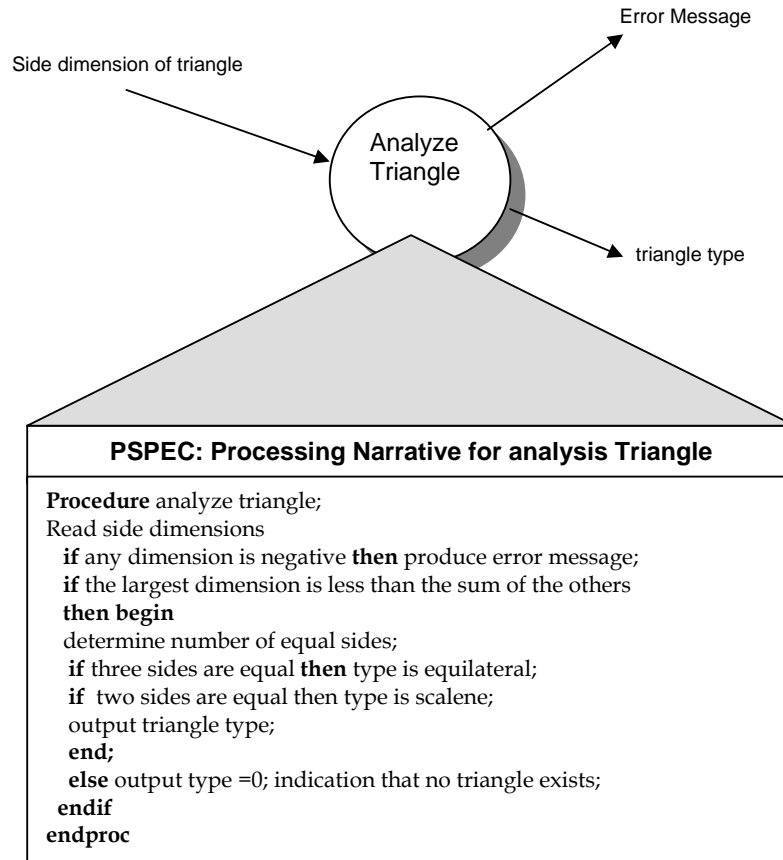


Figure 17.2 Process specification using PDL for a DFD process

Once a data object or control item name and its aliases are entered into the data dictionary consistency in naming can be enforced. That is if an analysis team member decides to name a newly derived data item xyz but xyz is already in the dictionary the CASE tool supporting the dictionary posts a warning to indicate duplicate names. This improves the consistency of the analysis model and helps to reduce errors.

Where-used/how-used” information is recorded automatically from the flow models. when a dictionary entry is created, the CASE tool scans DFDs and CFDs to determine which processes use the data or control information and most important benefits for the dictionary. During analysis there is an almost continuous stream of changes. For large projects it is often quite difficult to determine the impact of a change. For large projects it is often quite difficult to determine the impact of a change. Many a software engineer has asked” Where is this data object used? What else will have to change if we modify it? What will the overall impact of the change be?” Because the data dictionary can be treated as a database the analyst can ask “where-used/how-used” questions and get answers to queries noted above.

The notation used to develop a content description illustrated in? Figure 17.3 enables the analyst to represent composite data in one of the three fundamental ways that it can be constructed

a 3 digit number that never starts with 0 or 1) It is also important to note that a specification of elementary data often restricts a system. For example the definition of prefix indicates that only four branch exchanges can be accessed locally.

The data dictionary defines information items unambiguously. Although we might assume that the telephone number represented by the DFD in Figure 16.10 could accommodate a 25 digit long distance carrier access number the data dictionary content description tells us that such numbers are not part of the data that may be used.

For large computer-based systems the data dictionary grows rapidly in size and complexity. In fact it is extremely difficult to maintain a dictionary manually. For this reason, CASE tools should be used.

17.3 An Overview of other Classical Analysis Methods

Over the years many other worth while software requirements analysis methods have been used throughout the industry while all follow the operational analysis principles each introduces a different notation and heuristics for constructing the analysis model. In this section we present a very brief overview of three of the more common methods. For further information the interested reader should refer to the references noted.

17.4 Data Structured System Development

Data Structured Systems Development (DSSD) also called the Warnier –Orr methodology evolved from pioneering work on information domain analysis conducted by J.D. Warnier[WAR74,WAR81]. Warnier developed a notation for representing information hierarchy using the three constructs for sequence selection and repetition and demonstrated that the software structure could be derived directly from the data structure.

Ken Orr [ORR 77,ORR81]extended Warnier's work to encompass a some what broader view of the information domain that has evolved into Data Structured Systems Development. DSSD considers information flow and functional characteristic as well as data hierarchy.

17.5 Jackson System Development

Jackson System Development (JSD) evolved out of work conducted by M.A. Jackson [JAC75,JAC83] on information domain analysis and its relationship to program and system design. Similar in some ways to Warnier's approach and DSSD, JSD focuses on models of the "real world" information domain. In Jackson's words[JAC83], "[t]he developer begins by creating a model of the reality with which the system is concerned the reality which furnishes its the system's subject matter..."

To conduct JSD the analyst applies the following steps:

Entity Action Step. Using an approach that is quite similar to the object-oriented analysis techniques entities (people objects or organizations that a system needs to produce or use information) and actions (the events that occur in the real world that affect entities) are identified.

Entity Structure Step Actions that affect each entity are ordered by time and represented with Jackson Diagrams(a tree-like notation)

Initial Modeling Step. Entities and action are represents as a process model connections between the

model and the real world are defined.

Function Step. Functions that correspond to defined actions are specified.

Implementation Step. Hardware and software are specified as a design

The last three steps in JSD are closely aligned with system, or software design.

17.6 SADT

Structured analysis and design technique(SADT) is a technique that has been widely used as annotation for system definition, process representations, software requirements analysis and system/software design[ROS77.ROS85] SADT consists of procedures that allow the analyst to decompose software(or system) functions a graphical notation the SADT actigram and datagram that communicates the relationships of information (data and control) and function within software and project control guidelines for applying the methodology.

The SADT methodology encompasses automated tools to support analysis procedures and a well-defined organizational harness through which the tools are applied. Review and milestone are specified allowing validating of developer-customer communication.

17.7 Short Summary

- Structured analysis, the most widely used of requirements modeling methods, relies on data modeling and flow modeling to create the basis for a comprehensive analysis model.
- Using entity relationship diagrams, the software engineer creates a representation of all data objects that are important for the system.
- Data and control flow diagrams are used as a basis for representing the transformation of data and control. At the same time, these models are used to create a functional model of the software and to provide a mechanism for partitioning function.
- A behavioral model is created using the state transition diagram, and data content is developed with a data dictionary. Process and control specifications provide additional elaboration of detail.
- The original notation for structured analysis was developed for conventional data processing applications, but extensions now make the method applicable to real time systems. Structured analysis is supported by an array of CASE tools that assist in the creation of each element of the model and also help to ensure consistency and correctness.

17.8 Brain Storm

1. Write a short note on Data Dictionary ?
2. Explain briefly about DSSD and JSD ?
3. What is SADT ?

❧❧❧

Lecture 18

Design Concepts and Principles - I

Objectives

In this lecture you will learn the following

- ✧ About Software Design and Software Engineering
- ✧ About Design Process
- ✧ About Design Principles
- ✧ About Design Concepts

Coverage Plan

Lecture 18
18.1 Snap Shot
18.2 Software Design and Software Engineering
18.3 The Design Process
18.4 Design Principles
18.5 Design Concepts
18.6 Short Summary
18.7 Brain Storm

18.1 Snap Shot

Design is the first step in the development phase for any engineering product or system. The designer's goal is to produce a model or representation of an entity that will later be built. This lecture give you an idea about Software Design and Software Engineering, Design principles, process and concepts and also about Software Architecture.

18.2 Software Design and Software Engineering

Software design sits at the technical kernel of the software engineering process and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities - design, code generation, and testing - that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

Each of the elements of the analysis model provides information that is required to create a design model. The flow of information during software design is illustrated in following figure 18.1. Software requirements, manifested by the data, functional, and behavioral models, feed the design step. Using one of a number of design methods (discussed in later chapters), the design step produces a data design, an architectural design, an interface design, and a procedural design.

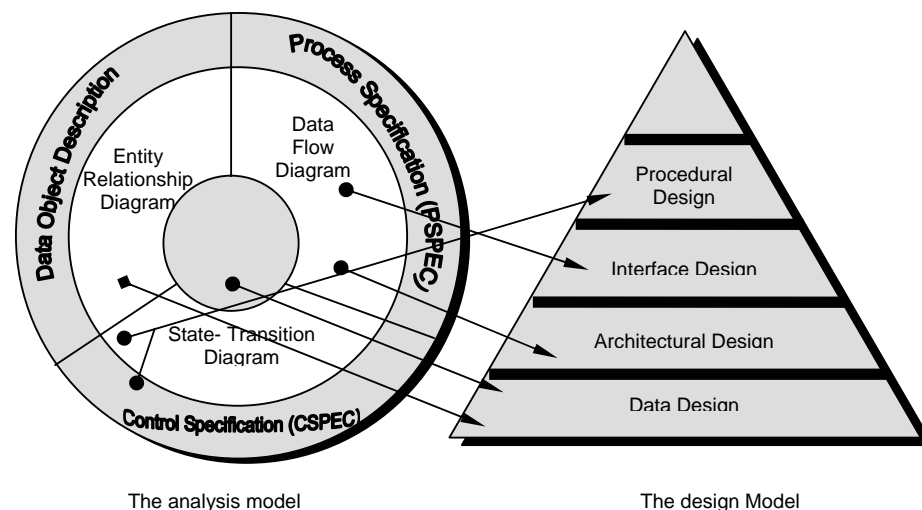


Figure 18.1 Translating the analysis model into a software desing

The data design transforms the information domain created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity-relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity.

The architectural design defines the relationship among major structural elements of the program. This design representation - the modular framework of a computer program - can be derived from the analysis model(s) and the interaction of subsystems defined within the analysis model.

The interface design describes how the software communicates within itself, to systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control). Therefore, the data and control flow diagrams provide the information required for interface design.

The procedural design transforms structural elements of the program architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, CSPEC, and STD serve as the basis for procedural design.

During design we make decisions that will ultimately affect the success of software construction, and as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word-quality. Design is the place where quality is fostered in software development. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all software engineering and software maintenance steps that follow. Without design, we risk building an unstable system-one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software engineering process, when time is short and many dollars have already been spent.

18.3 The Design Process

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction - a level that can be directly traced to specific data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Design and Software Quality

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs discussed in following chapters. McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently maintain the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

In order to evaluate the quality of a design representation, we must establish technical criteria for good design. Later in this lecture we discuss design quality criteria in some detail. For the time being, we present the following guidelines:

1. A design should exhibit a hierarchical organization that makes intelligent use of control among elements of software.
2. A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.
3. A design should contain both data and procedural abstractions.
4. A design should lead to modules (e.g., subroutines or procedures) that exhibit independent functional characteristics.
5. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
6. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

The Evolution of Software Design

The evolution of software design is a continuing process that has spanned the past three decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software architecture in a top-down manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches propose an object-oriented approach to design derivation.

Many design methods, growing out of the work noted above, are being applied throughout the industry. Like the analysis methods each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes lead to design quality. Yet, each of these methods has a number of common characteristics: (1) a mechanism for the translation of an analysis model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, a software engineer should apply a set of fundamental principles and basic concepts to data, architectural, interface, and procedural design. These principles and concepts are considered in the sections that follow.

18.4 Design Principles

Software design is both a process and a model. The design process is a set of iterative steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design.

The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three dimensional rendering of the house) and slowly refines the thing to

provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer program.

Basic design principles enable the software engineer to navigate the design process. Davis suggests a set of principles for software design, which have been adapted and extended in the following list:

- The design process should not suffer from "tunnel vision." A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts.
- The design should be traceable to the analysis model. Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- The design should not reinvent the wheel. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns, often called reusable design components, should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
- The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world. That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- The design should exhibit uniformity and integration. A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
- The design should be structured to accommodate change. Many of the design concepts discussed in the next section enable a design to achieve this principles.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. A well-designed computer program should never "bomb". It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
- Design is not coding, coding is not design. Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
- The design should be assessed for quality as it is being created, not after the fact. A variety of design concepts and design measures are available to assist the designer in assessing quality.
- The design should be reviewed to minimize conceptual errors. There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A designer should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

When the design principles described above are properly applied, the software engineer creates a design that exhibits both external and internal quality factors. External quality factors are those properties of the

software that can be readily observed by users (e.g., speed, reliability, correctness, usability). Internal quality factors are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.

18.5 Design Concepts

A set of fundamental software design concepts has evolved over the past three decades. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps the software engineer to answer the following questions:

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- Are there uniform criteria that define the technical quality of a software design?

M. A. Jackson once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right". Fundamental software design concepts provide the necessary framework for "getting it right".

Abstraction

When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. Wasserman provides a useful definition:

[T]he psychological notion of "abstraction" permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details; use of abstraction also permits one to work with concepts and terms that are familiar that can be directly implemented. Wasserman provides environment without having to transform them to an unfamiliar structure....

Each step in the software engineering process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A procedural abstraction is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word "open" on a door. "Open" implies a long sequence of procedural steps (e.g., walk to the door; reach out and grasp knob; turn knob and pull door; step away from moving door, etc.). A data abstraction is a named collection of data that describes a data object.

In the context of the procedural abstraction open noted above, we can define a data abstraction called **door**. Like any data object, the data abstraction for door would encompass a set of attributes that describe

the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction **door**.

A number of programming languages (e.g., Ada, Modula, CLU) provide mechanisms for creating abstract data types. For example, the Ada package is a programming language mechanism that provides support for both data and procedural abstraction. The original abstract data type is used as a template or generic data structure from which other data structures can be instantiated.

Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the synchronization semaphore (KA183) used to coordinate activities in an operating system.

Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. The architecture of a program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a step wise fashion until programming language statements are reached. An overview of the concept is provided by Wirth:

In each step (of the refinement), one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of any underlying computer or programming language... As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine the program and the data specifications in parallel.

Every refinement step implies some design decisions. It is important that... the programmer be aware of the underlying criteria (for design decisions) and of the existence of alternative solutions.

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach.

Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement function or information conceptually, but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

Modularity

The concept of modularity in computer software has been espoused for almost four decades. Software architecture embodies modularity; that is, software is divided into separately named and addressable components, called modules, that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program comprised of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. To illustrate this point, consider the following argument based on observations of human problem solving.

Let $C(x)$ be a function that defines the perceived complexity of a problem x , and $E(x)$ be a function that defines the effort (in time) required to solve a problem x . For two problems, p_1 and p_2 , if

$$C(p_1) > C(p_2) \quad (13.1a)$$

it follows that

$$E(p_1) > E(p_2) \quad (13.1b)$$

As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

$$C(p_1+p_2) > C(p_1) + C(p_2) \quad (13.2)$$

Equation (13.2) implies that the perceived complexity of a problem that combines p_1 and p_2 is greater than the perceived complexity when each problem is considered separately. Considering equation (13.2) and the condition implied by equations (13.1), it follows that

$$E(p_1+p_2) > E(p_1)+E(p_2) \quad (13.3)$$

This leads to a "divide and conquer" conclusion - it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in inequality (13.3) has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

It is possible to conclude from inequality (13.3) that if we subdivide software indefinitely, the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. As **figure 18.2** shows, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the **figure 18.2**. There is a number, M , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

The curves shown in **Figure 18.2** do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of M . Undermodularity or overmodularity should be avoided. But how do we know "the vicinity of M "? How modular should we make software? The answers to these questions require an understanding of other design concepts considered later in this lecture.

Another important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system. Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

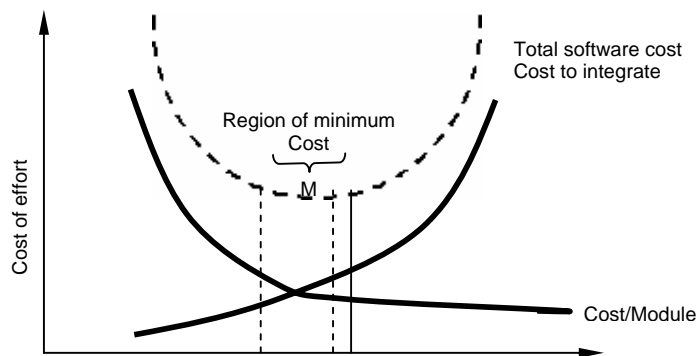


Figure 18.2 Modularity and software cost

Modular decomposability. If a design method provides a systematic mechanism for decomposing the problem into subproblems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

Modular composability. If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understandability. If a module can be understood as a stand alone unit (without reference to other modules) it will be easier to build and easier to change.

Modular continuity. If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of change-induced side effects will be minimized.

Modular protection. If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Finally, it is important to note that a system may be designed modularly, even if its implementation must be "monolithic". There are situations (e.g., real time software, embedded software) in which relatively minimal speed and memory overhead introduced by subprograms (i.e., subroutines, procedures) is unacceptable. In such situations software can and should be designed with modularity as an overriding philosophy. Code may be developed "in-line." Although the program source code may not look modular at first glance the philosophy has been maintained, and the program will provide the benefits a modular system.

Software Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact, and the structure of the data that are used by the components. In a broader sense, however, "components" can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data, and to interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models. Structural models represent architecture as an organized collection of program components. Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. Process models focus on the design of the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system.

A number of different architectural description languages (ADLs) have been developed to represent the models noted above. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

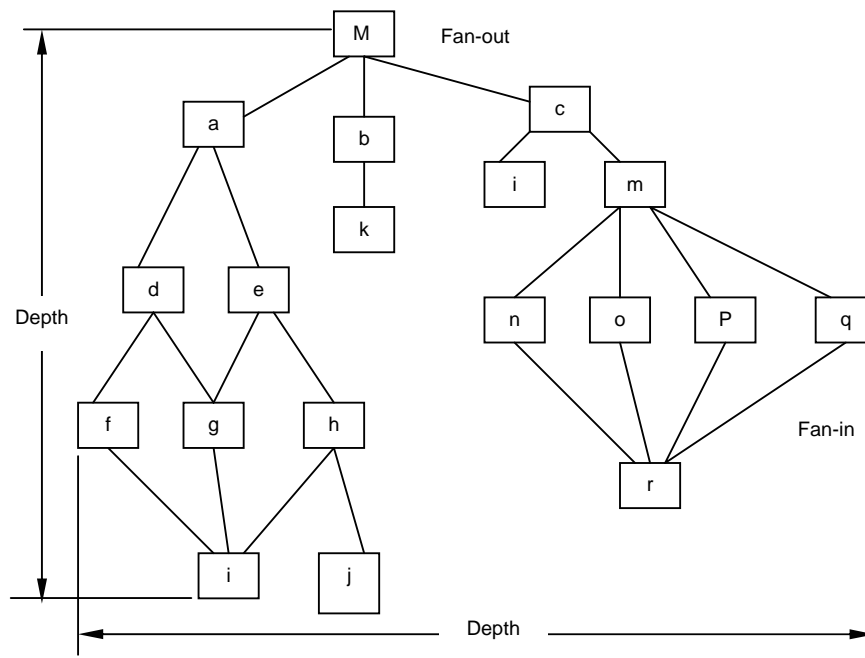


Figure 18.3 Structure Terminology

Control Hierarchy

Control hierarchy, also called program structure, represents the organization of program components and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence/order of decisions, or repetition of operations.

Many different notations are used to represent control hierarchy. The most common is the tree-like diagram that represents the hierarchy. However, other notations, such as Warnier-Orr and Jackson diagrams may also be used with equal effectiveness. In order to facilitate later discussions of structure, we define a few simple measures and terms. In **Figure 18.3**, depth and width provide an indication of the number of levels of control and overall span of control, respectively. Fan-out is a measure of the number of modules that are directly controlled by another module. Fan-in indicates how many modules directly control a given module.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be superordinate to it; conversely, a module controlled by another is said to be subordinate to the controller. For example, as shown in **Figure 13.3**, module M is superordinate to modules a, b and c. Module h is subordinate to module e and is ultimately subordinate to module M. Width-oriented relationships (e.g., between modules d and e), although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity. Visibility indicates the set of programs components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented, but only make use of a small number of these data attributes. All of the attributes are

visible to the module. Connectivity indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.

Structural Partitioning

The program structure should be partitioned both horizontally and vertically. As shown in **Figure 18.4 a**, horizontal partitioning defines separate branches of the modular hierarchy to reach major program function. Control modules, represented in a darker shade, are used to coordinate communication between and execution of program functions. The simplest approach to horizontal partitioning defines three partitions - input, data transformation (often called processing), and output. Partitioning the architecture horizontally provides a number of distinct benefits:

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

Because major functions are decoupled from one another, change tends to be less complex and extensions

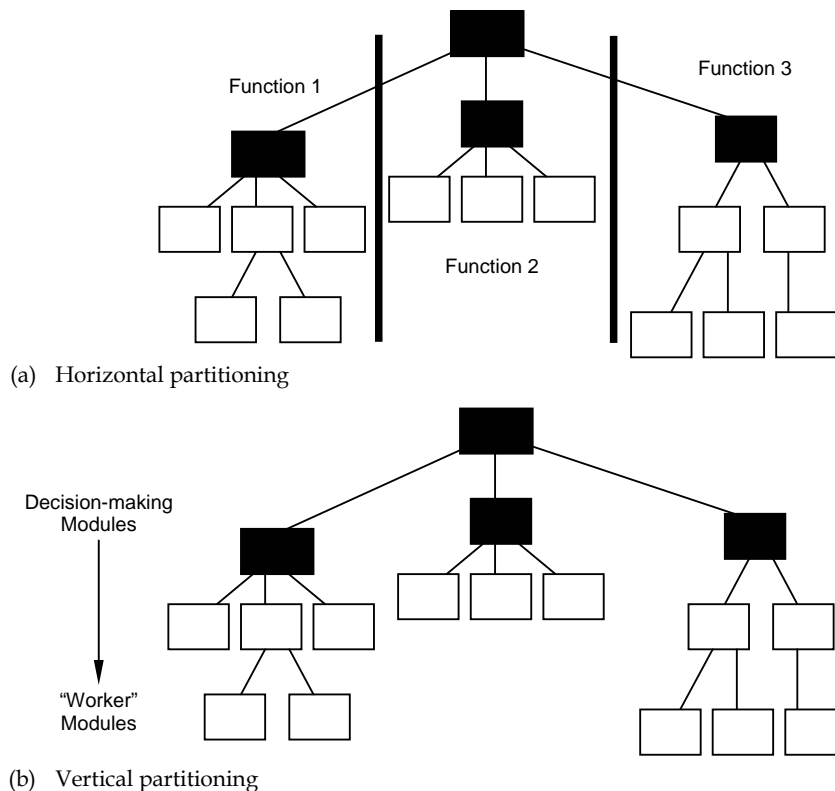


Figure 18.4 Architectural partitioning

to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

Vertical partitioning, often called factoring, suggests that control and work should be distributed top-down in the program architecture. Top-level modules should perform control functions and do little actual processing work. Modules that reside low in the architecture should be the workers, performing all input, computational, and output tasks.

The nature of change in program architectures justifies the need for vertical partitioning. A change in a control module (high in the architecture) will have a higher probability of propagating side effects to modules that are subordinate to it. A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects. In general, changes to computer programs revolve around changes to input, computation or transformation, and output. The overall control structure of the program (i.e., its basic behavior) is far less likely to change. For this reason vertically partitioned architectures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable – a key quality factor.

Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure the representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. Entire texts have been dedicated to these topics, and a complete discussion is beyond the scope of this book. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.

The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A scalar item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in storage.

When scalar items are organized as a list or contiguous group, a sequential vector is formed. Vectors are the most common of all data structures and open the door to variable indexing of information.

When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an n-dimensional space is created. The most common n-dimensional space is the two-dimensional matrix. In most programming languages, an n-dimensional space is called an array.

Items, vectors, and spaces may be organized in a variety of formats. A linked list is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner that enables them to be processed as a list. Each node contains the appropriate data organization and one or more pointers that indicate the

address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

Other data structures incorporate or are constructed using the fundamental data structures described above. For example, a hierarchical data structure is implemented using multilinked lists that contain scalar items, vectors, and possible n-dimensional spaces. A hierarchical structure is commonly encountered in applications that require information categorization and associativity. Categorization implies a grouping of information by some generic category (e.g., all subcompact automobiles or all 64-bit microprocessors). Associativity implies the ability to associate information from different categories; for example, find all entries in the microprocessor category that cost less than \$100.00 (cost subcategory), run at 100 MHz (cycle time subcategory), and are made by U.S. vendors (vendor subcategory).

It is important to note that data structures, like program structures, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structures that can be implemented as a vector or a linked list. Depending on the level of design detail, the internal workings of stack may or may not be specified.

Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization / structure.

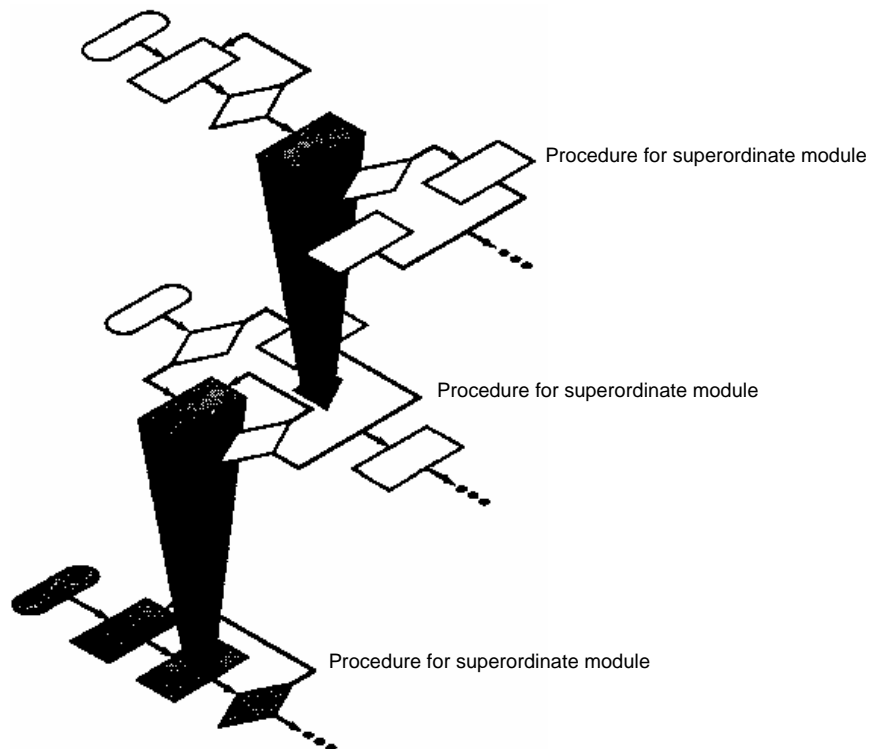


Figure 18.5 Procedure is layered

There is, of course, a relationship between structure and procedure. Processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered.

Information Hiding

the concept of modularity leads every software designer to a fundamental question: “How do we decompose a software solution to obtain the best set of modules?” The principle of information hiding suggests that modules be “characterized by design decisions that (each) hides from all others.” In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information that is necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that comprise the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides its greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

18.6 Short Summary

- Modularity (in both program and data) and the concept of abstraction enable the designer to simplify and reuse software components.
- Refinement provides a mechanism for representing successive layers of functional detail.
- Program and data structure contribute to an overall view of software architecture, while procedure provides the detail necessary for algorithm implementation.
- Information hiding and functional independence provide heuristics for achieving effective modularity.
- We conclude our discussion of design fundamentals with the words of Glenford Myers:

[W]e try to solve the problem by rushing through the design process so that enough time will be left at the end of the project to uncover errors that were made because we rushed through the design process. The moral is don't rush through it! Design is worth the effort.

18.7 Brain Storm

1. Write a short note on Software Design and Software Engineering ?
2. Explain briefly about Design Process ?
3. Explain briefly about Design principles and Design Concepts ?
4. What is Software Architecture ? Explain.

Lecture 19

Design Concepts and Principles - II

Objectives

In this lecture you will learn the following

- ✧ About Effective Modular Design
- ✧ About Design Heuristics for Effective Modularity
- ✧ About Design Model
- ✧ About Design Documentation

Coverage Plan

Lecture 19
19.1 Snap Shot
19.2 Design Heuristics for Effective Modularity
19.3 The Design Model
19.4 Design Documentatioan
19.5 Short Summary
19.6 Brain Storm

19.1 Snap Shot

The fundamental design concepts described in the preceding section all serve to precipitate modular designs. In fact, modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity facilitates change and results in easier implementation by encouraging parallel development of different parts of a system.

Functional Independence

The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding. In landmark papers on software design Parnas and Wirth allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine solidified the concept.

Functional independence is achieved by developing modules with “single-minded” function and an “aversion” to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity (i.e., independent modules), is easier to develop because function may be compartmentalized and interfaces are simplified (consider ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design/code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is measured using two qualitative criteria: cohesion and coupling. Cohesion is a measure of the relative functional strength of a module. Coupling is a measure of the relative interdependence among modules.

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed on other parts of a program. Stated simply, an cohesive module should (ideally) do just one thing.

Cohesion may be represented as a “spectrum”. We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear. That is, low-end cohesiveness is much “worse” than the middle range, which is nearly as “good” as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed.

To illustrate (somewhat facetiously) the low end of the spectrum, we relate the following story:

In the late 1960s most DP managers began to recognize the worth of modularity. Unfortunately many existing programs were monolithic – e.g., 20,000 lines of undocumented Fortran with one 2500 line subroutine! To bring a large computer program to the state of the art, a manager asked her staff to modularize the program. This was to be done “in your spare time”.

Under the gun, one staff member asked (innocently) the proper length for a module. “Seventy-five lines of code,” came the reply. He then obtained a red pen and a ruler, measured the linear distance taken by 75

lines of source code, and drew a red line on the source listing, then another and another. Each red line indicated a module boundary. This technique is akin to developing software with coincidental cohesion!

A module that performs a set of tasks that relate to each other loosely, if at all, is termed coincidentally cohesive. A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is logically cohesive. When a module contains tasks that are related by the fact that all must be executed within the same span of time, the module exhibits temporal cohesion.

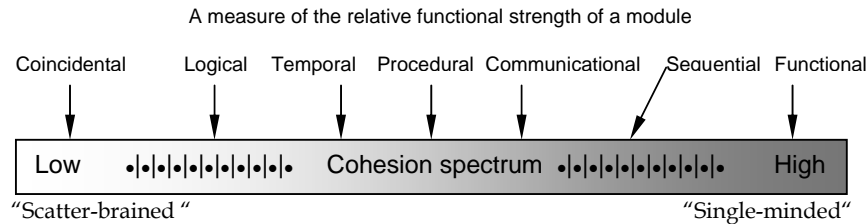


Figure 19.1 Cohesion

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed prespecified bound. It performs the following tasks: (1) computes supplementary data based on original computed data; (2) produces an error report (with graphical content) on the user's workstation; (3) performs follow-up calculations requested by the user; (4) updates a data base; and (5) enables menu selection for subsequent processing. Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can only serve to increase the likelihood of error propagation when a modification is made to any one of the processing tasks noted above.

Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, procedural cohesion exists. When all processing elements concentrate on one area of a data structure, communicational cohesion is present. High cohesion is characterized by a module that performs one distinct procedural task.

As we have already noted, it is unnecessary to determine the precise level of cohesion. Rather it is important to strive for high cohesion and recognizes low cohesion so that software design can be modified to achieve greater functional independence.

Coupling

Coupling is a measure of interconnection among modules in a program structure. Like cohesion, coupling may be represented on a spectrum as shown in figure 19.2. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" caused when errors occur at one location and propagate through a system.

Figure 19.3 provides examples of different types of module coupling. Modules a and d are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module c is subordinate to module a and is accessed via a conventional argument list through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (data coupling on the spectrum) is exhibited in this portion of structure. A variation of data coupling, called stamp coupling, is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules b and a.

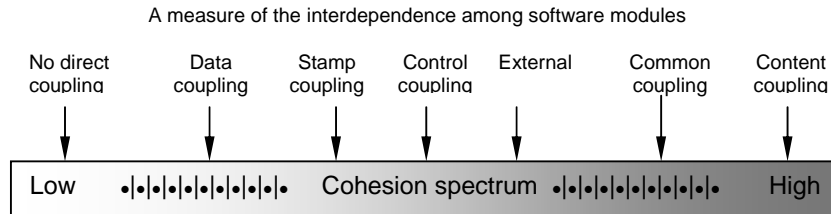


Figure 19.2 Coupling

At moderate levels coupling is characterized by passage of control between modules. Control coupling is very common in most software designs and is shown in figure 19.3, where a “control flag” is passed between modules d and e.

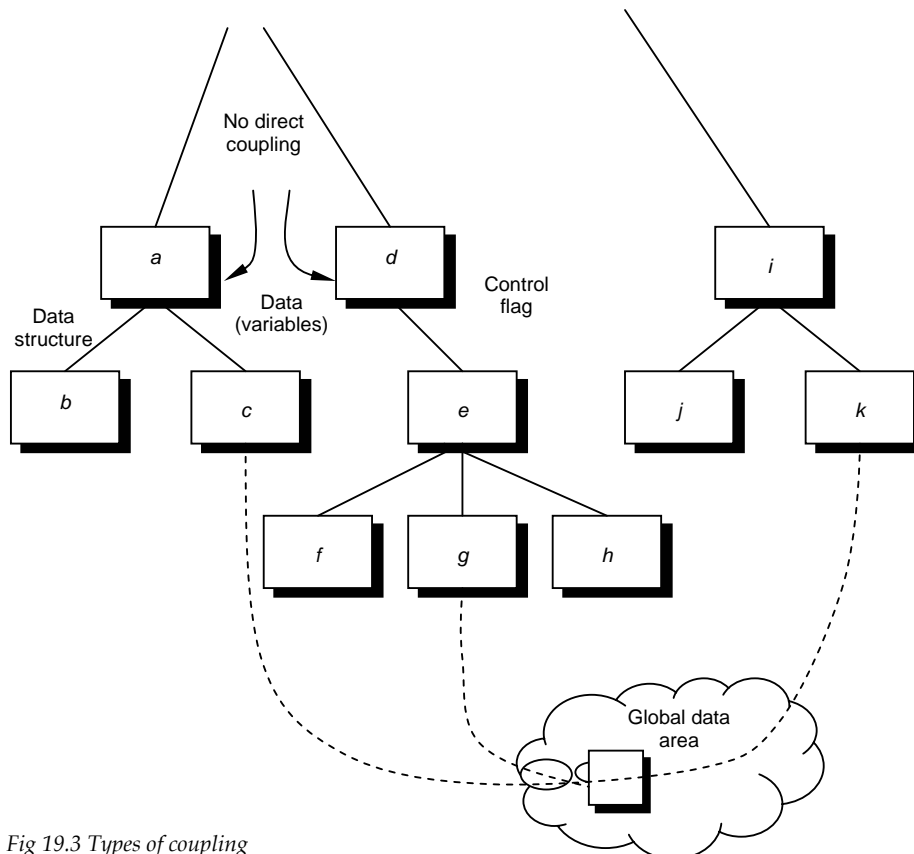


Fig 19.3 Types of coupling

Types of Coupling

Relatively high levels of coupling occur when modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols. External coupling is essential, but also occurs when a number of modules reference a global data area. Common coupling, as this mode is called, is shown in figure 19.3. Modules c, g, and h each access a data item in a global data area (e.g., a disk file, Fortran COMMON, external data types in the C programming language). Module c initializes the item. Later module g recomputes and updates the item. Let's assume that an error occurs and g updates the item incorrectly. Much later in processing, module k reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module k; the actual cause, module g. Diagnosing problems in structures with considerable common coupling is time-consuming and difficult. However, this does not mean that the use of global data is necessarily "bad". It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them.

The highest degree of coupling, content coupling, occurs when one module makes use of data or control information maintained within the boundary of another module. Secondly, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

The coupling modes discussed above occur because of design decision made when the program structure was developed. Variants of external coupling, however, may be introduced during coding. For example, compiler coupling ties source code to specific (and often nonstandard) attributes of a compiler; operating system (OS) coupling ties design and resultant code to operation system "hooks" that can create havoc when OS changes occur.

19.2 Design Heuristics for Effective Modularity

Once a program structure has been developed, effective modularity can be achieved by applying the design concepts introduced earlier in this chapter. The program architecture is manipulated according to a set of heuristics (guidelines) presented in this section.

I. Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.

Once program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence. An exploded module becomes two or more modules in the final program structure. An imploded module is the result of combining the processing implied by two or more modules.

An exploded module often results when a common process component exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data and interface complexity.

II. Attempt to minimize structures with high fan-out; strive for fan-in as depth increases. The structure shown inside the cloud in Figure 19.4 does not make effective use of factoring. All modules are "pancaked" below a single control module. In general, a more reasonable distribution of control is shown in the upper structure. The Structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

III. Keep scope of effect of a module within the scope of control of that module.

The scope of effect of a module is defined as all other modules that are affected by a decision made in module e. The scope of control of module e is all modules that are subordinate and ultimately

subordinate to module e. As shown in Figure 19.4, if module e makes a decision that affects module r, we have a violation of heuristic III, because module r lies outside the scope of control of module e.

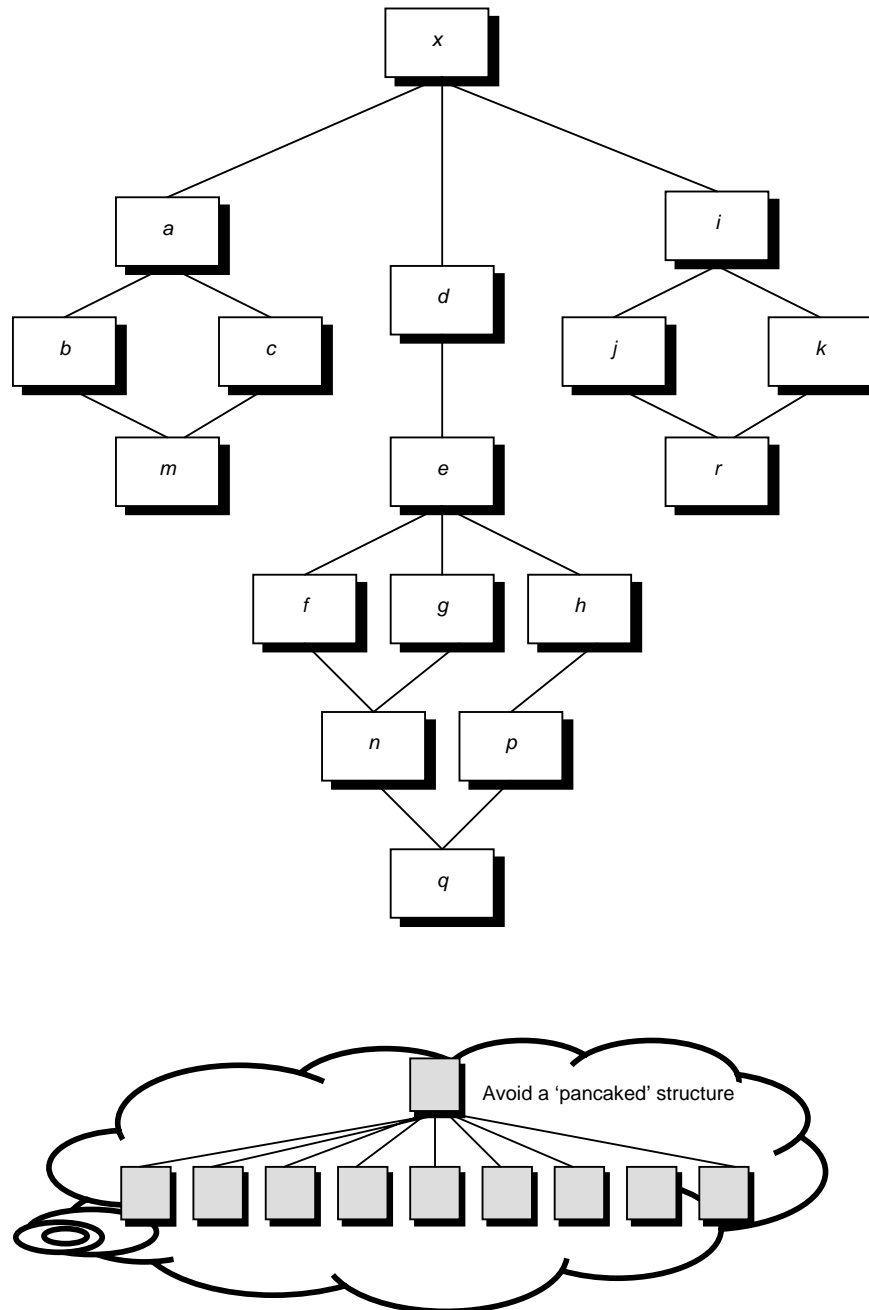


Figure 19.4 Program structure

IV. Evaluate module interfaces to reduce complexity and redundancy and improve consistency. Module interface complexity is a prime cause of software errors. Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e

seemingly unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be re-evaluated.

V. Define modules whose function is predictable, but avoid modules that are overly restrictive. A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal memory can be unpredictable unless care is taken in their use.

A module that restricts processing to a single subfunction exhibits high cohesion and is viewed with favor by a designer. However, a module that arbitrarily restricts size of a local data structure, option within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

VI. Strive for “controlled entry” modules, avoiding “pathological connections”. This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when modules interfaced are constrained and controlled. Pathological connection refers to branches or references into the middle of a module.

VII. Package software based on design constraints and portability requirements. Packaging alludes to the techniques used to assemble software for a specific processing environment. Design constraints sometimes dictate that a program “overlay” itself in memory. When this must occur, the design structure may have to be reorganized to group modules by degree of repetition, frequency of access and interval between calls. In addition, optional or “one-shot” modules may be separated in the structure so that they may be effectively overlaid.

19.3 The Design Model

The design principles and concepts discussed in this chapter establish a foundation for the creation of the design model that encompasses representation of data, architecture, interfaces, and procedures. Like the analysis model before it, in the design model each of these design representation is tied to the others, and all can be traced back to software requirements.

In figure 19.1, the design model was represented as pyramid. The symbolism of this shape is important. A pyramid is an extremely stable object with a wide base and a low center of gravity. Like the pyramid, we want to create a software design that is stable. By establishing a broad foundation using data design, a stable mid-region with architectural and interface design, and a sharp point by applying procedural design, we create a design model that is not easily “tipped over” by the winds of change.

It is interesting to note that some programmers continue to design implicitly, conducting procedural design as they code. This is akin to taking the design pyramid and standing it on its point an extremely unstable design results. The smallest change may cause the pyramid (and the program) to topple.

The methods that lead to the creation of the design model are presented in lectures. Each method enables the designer to create a stable design that conforms to the fundamental concepts that lead to high quality software.

19.4 Design Documentation

The document outlined in figure 19.5 can be used as template for a design specification. Each numbered paragraph addresses different aspects of the design model. The numbered sections of the design specification are completed as the designer refines his or her representation of the software.

The overall scope of the design effort is described in section I (section numbers refer to design specification outline) Much of the information contained in this section is derived from the system specification and the analysis model (software requirements specification).

I. Scope

- A. System objectives
- B. Major software requirements
- C. Design constraints, limitations

II. Data Design

- A. Data objects and resultant data structures
- B. File and database structures
 - 1. External file structure
 - a. Logical structure
 - b. Logical record description
 - c. Access method
 - 2. Global data
 - 3. File and data cross reference

III. Architectural Design

- A. Review of data and control flow
- B. Derived program structure

IV. Interface Design

- A. Human-machine interface specification
- B. Human-machine interface design rules
- C. External interface design
 - 1. Interfaces to external data
 - 2. Interfaces to external systems or devices
- D. Internal interface design rules

V. Procedural Design

For each module:

- A. Processing narrative
- B. Interface description
- C. Design language (or other) description
- D. Modules used
- E. Internal data structures
- F. Comments / restrictions / limitations

VI. Requirements Cross-Reference

VII. Test Provisions

- 1. Test guidelines

2. Integration strategy
3. Special considerations

VIII. Special Notes

IX. Appendices

Figure 19.5 Design specification outline

Section II presents the data design, describing external file structures, internal data structures and a cross reference that connects data objects to specific files. Section III, the architectural design, indicates how the program architecture has been derived from the analysis model. Structure charts (a representation of program structure) are used to represent the module hierarchy.

Sections IV and V evolve as interface and procedural design commence. External and internal program interfaces are represented and a detailed design of the human-machine interface is described. Modules – separately addressable elements of software such as subroutines, functions, or procedures – are initially described with an English-language processing narrative. The processing narrative explains the procedural function of a module. Later, a procedural design tool is used to translate the narrative into a structured description.

Section VI of the design specification contains a requirements cross-reference. The purpose of this cross-reference matrix is (1) to establish that all requirements are satisfied by the software design, and (2) to indicate which modules are critical to the implementation of specific requirements.

The first stage in the development of test documentation is contained in section VII of the design document. Once software structure and interfaces have been established, we can develop guidelines for testing of individual modules and integration of the entire package. In some cases, a detailed specification of test procedure occurs in parallel with design. In such cases, this section may be deleted from the design specification.

Design constraints, such as physical memory limitations or the necessity for a specialized external interface, may dictate special requirements for assembling or packaging of software. Special considerations caused by the necessity for program overlay, virtual memory management, high-speed processing, or other factors may cause modification in design derived from information flow or structure. Requirements and considerations for software packaging are presented in section VII. Secondly, this section describes the approach that will be used to transfer software to a customer site.

Section IX of the design specification contains supplementary data. Algorithm descriptions, alternative procedures, tabular data, excerpts from other documents and other relevant information are presented as a special not or as a separate appendix. It may be advisable to develop a preliminary operations / installation manual and include it as appendix to the design document.

19.5 Short Summary

- Design is the technical kernel of software engineering. During design, progressive refinements of data structure, program architecture, interfaces, and procedural detail are developed, reviewed, and documented.
- Design results in representations of software that can be assessed for quality.

- A number of fundamental software design principles and concepts have been proposed over the past three decades.
- Design principles guide the software engineer as the design process proceeds.
- Design concepts provide basic criteria for design quality.

19. 6 Brain Storm

1. Give a short note on Functional Independence ?
2. What is coupling ? Explain.
3. Explain briefly about Design Model and Design Documentation ?

END

Lecture 20

Design Methods

Objectives

In this lecture you will learn the following

- ✎ About Data Design
- ✎ About Architectural Design Process
- ✎ About Transform Mapping
- ✎ About Transaction Mapping

Coverage Plan

Lecture 20

- | |
|---|
| <ul style="list-style-type: none">20.1 Snap Shot20.2 Architectural Design20.3 The Architectural Design Process20.4 Transform Mapping20.5 Transaction Mapping20.6 Design Postprocessing20.7 Short Summary20.8 Brain Storm |
|---|

20.1 Snap Shot

Data design is the first (and some would say the most important) of four design activities that are conducted during software engineering. The impact of data structure on program structure and procedural complexity causes data design to have a profound influence on software quality. The concepts of information hiding and data abstraction provide the foundation for an approach to data design.

The process of data design summarized by Wasserman [WAS80]:

The primary activity during data design is to select logical representations of data objects (data structures) identified during the requirements definition and specification phase. The selection process may involve algorithmic analysis of alternative structures in order to determine the most efficient design or may simply involve the use of a set of modules (a “package”) that provide the desired operations upon some representation of an object.

An important related activity during design is to identify those program modules that must operate directly upon the logical data structures. In this way the scope of effect of individual data design decisions can be constrained.

Regardless of the design techniques to be used, well designed data can lead to better program structure and modularity, and reduced procedural complexity.

Wasserman has proposed a set of principles that may be used to specify and design data. In actuality, the design of data begins during the creation of the analysis model. Recalling that requirements analysis and design often overlap, we consider the following set of principles [WAS80] for data specification:

1. Systematic analysis principles applied to function and behavior should also be applied to data. We spend much time and effort deriving, reviewing, and specifying functional requirements and preliminary design. Representations of data objects, relationships, dataflow, and content should also be developed and reviewed, alternative data organizations should be considered, and the impact of data modeling on software design should be evaluated. For example specification of multiringed linked list may nicely satisfy data requirements but may lead to an unwieldy software design. An alternative data organization may lead to better results.
2. All data structures and the operations to be performed on each should be identified. The design of an efficient data structure must take the operations to be performed on the data structure must take the operations to be performed on the data structure into account. For example, consider a data structure made up of a set of diverse data elements. The data structure is to be manipulated in a number of major software functions. Upon evaluation of the operation performed on the data structure, an abstract data type is defined for use in subsequent software design. Specification of the abstract data type may simplify software design considerable.
3. A data dictionary should be established and used to define both data and program design. The concept of a data dictionary was introduced later chapters. A data dictionary explicitly represents the relationships among data objects and the constraints on the elements of a data structure. Algorithms that must take advantage of specific relationships can be more easily defined if a dictionary like data specification exists.

4. Low level data design decisions should be deferred until late in the design process. A process of stepwise refinement may be used for the design of data. That is, overall data organization may be defined during requirements analysis, refined during preliminary design, and specified in detail during later design iterations. The top-down approach to data design provides benefits that are analogous to a top-down approach to software design major structural attributes are designed and evaluated first so that the architecture of the data may be established.
5. The representation of data structures should be known only to those modules that must make direct use of the data contained within the structure. The concept of information hiding and the related concept of coupling provide important insight into the quality of a software design. Principle 5 alludes to the importance of these concepts as well as “the importance of separating the logical view of a data object from its physical view”[WAS80].
6. A library of use full data structures and the operations that may be applied to them should be developed. Data structures and operations should be viewed as resources for software design. Data structures can be designed can reduce both specification and design effort for data.
7. A software design and programming language should support the specification and realization of abstract data types. The implementation (and corresponding design) of s sophisticated data structure can be made exceedingly difficult if no means for direct specification of the structure exists. For example, implementation (or design) of a linked list structure of a multi level heterogeneous array would be difficult if the target programming language was Fortran because the language does not support direct specification of the data structures.

The principles described above form a basis for a data design approach that can be integrated into both the definition and development phases of the software engineering process. As we have noted elsewhere in this book, a clear definition of information is essential to successful software development.

20.2 Architectural Design

The primary objective of architectural design is to develop a modular program structure and represent the control relationships between modules. In addition, architectural design melds program structure and data structure, defining interfaces that enable data to flow throughout the program.

To understand the importance of architecture design, we present a brief story from everyday life:

You have saved your money, you’ve purchased a beautiful piece of land, and you’ve decided to build the house of your dreams. Having no experience in such matters, you visit a builder and explain your desires – number and size of rooms, contemporary styling, spa(of course!). cathedral ceilings, lots of glass, etc. The builder listens carefully, asks a few questions, and then tells you that he’ll have a design in a few weeks.

As you wait anxiously for his call, you conjure up many different (and outrageously expensive) images of your new house. What will he come up with? Finally, the phone rings and you rush to his office.

Pulling out a large manila folder, the builder spreads a diagram of the plumbing for the second floor bathroom in front of you and proceeds to explain it in great detail.

“But what about the overall design?” you say.

“Don’t worry, “says the builder, “we’ll get to that later”.

Does the builder’s final response? Of course not! You want to see a sketch of the house, a floor plan, and other information that will provide an architectural view. Yet many software developers act like the builder in our story. They concentrate on the “plumbing” (procedural details and code) to the exclusion of the software architecture. The design method presented in this section encourages the software engineer to concentrate on architectural design before worrying about the plumbing.

Contributors

Architectural design (and software design generally) has its origins in earlier design concepts that stressed modularity [DEN 73], top-down design [WIR 71], and structured programming [DAH72, LIN79]. Stevens, Myers, and Constantine [STE74] were early proponents of software design based on the flow of data through a system. Early work was refined and presented in books by Myers [MYE78] and yourdon and Constantine[YOU79]

Areas of Application

Each software design method has strengths and weakness. An important selection factor for a design method is the breadth of applications to which it can be applied. Data flow oriented design is amenable to a broad range of application areas. In fact, because all software can be represented by a data flow diagram, a design method that makes use of the diagram could theoretically be applied in every software development effort. A data flow- oriented approach to design is particularly useful when information is processed sequentially and no formal hierarchical data structure exists. For example, microprocessor control application complex , numerical analysis procedures; process control; and many other engineering and scientific software applications fall into this category. Data flow oriented design techniques are also applicable in data processing applications and can be effectively applied even when hierarchical data structures do exist.

There are cases, however, in which a consideration of data flow is at best a side issue. In such applications (e.g., database systems, expert systems, object oriented interfaces), other design methods may be more appropriate.

20.3 The Architectural Design Process

Data flow oriented design is an architectural design method that allows a convenient transition from the analysis model to a design description of program structure. The transition from information flow (represented as a data flow diagram) to structure is accomplished as part of a five step process

1. The type of information flow is established
2. Flow boundaries are indicated
3. The DFD is mapped into program structure
4. Control hierarchy is defined by factoring
5. The resultant structure is refined using design measures and heuristics.

The information flow type is the driver for the mapping approach required in step 3. In the following sections we examine two flow types.

Transform Flow

In the fundamental system model (level 0 data flow diagram), information must enter and exit software in an “external world” form, for example, data typed on a keyboard, tones on a telephone line, and pictures on a computer graphics display are all forms of external world information. Such externalized data must be converted into an internal form for processing. The time history of data can be illustrated in figure 20.1. Information enters the system along paths that transform external data into an internal form and will be identified as incoming flow. At the kernel of the software, a transition occurs. Incoming data are passed through a transform enter and begin to move along paths that now lead “out” of the software. Data moving along these paths are called outgoing flow. The overall flow of data occurs in a sequential manner and follows one, or only a few, “straight line” paths. When a segment of a data flow diagram exhibits these characteristics, transform flow is present.

Transaction Flow

The fundamental system model implies transform flow; therefore, it is possible to characterize all data flow in this category. However, information flow is often characterized by a single data item, called a transaction, that triggers other data flow along one of many paths. When a DFD takes the form shown in fig 14.2, transaction flow is present.

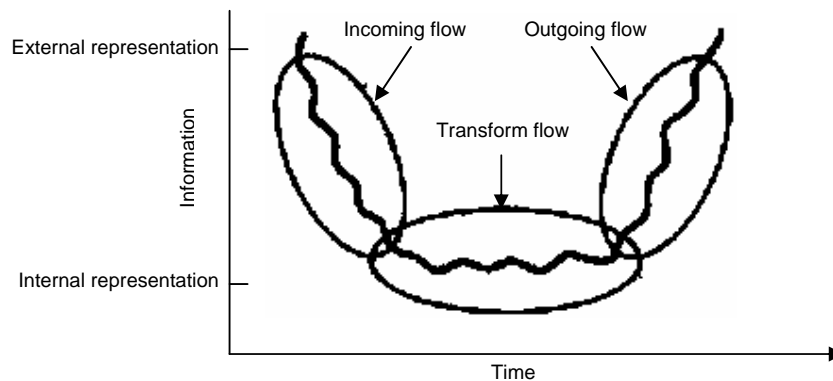


Figure 20.1. Flow of information

Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated, and based on its value, flow along one of many action paths is initiated. The hub of information flow from which many action paths emanate is called a transaction center.

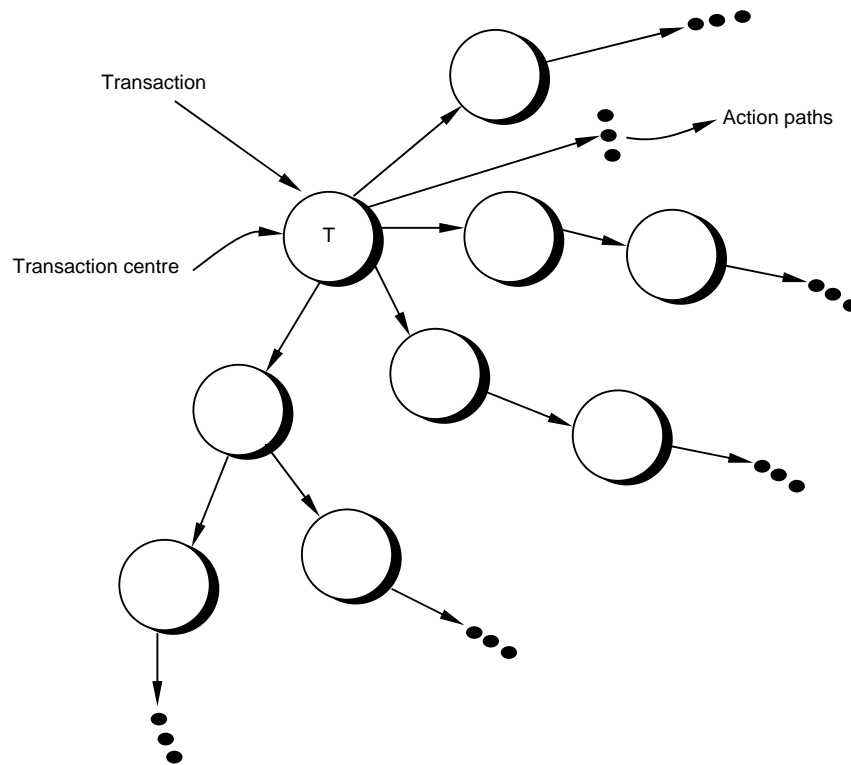


Figure 20.2 Transaction flow

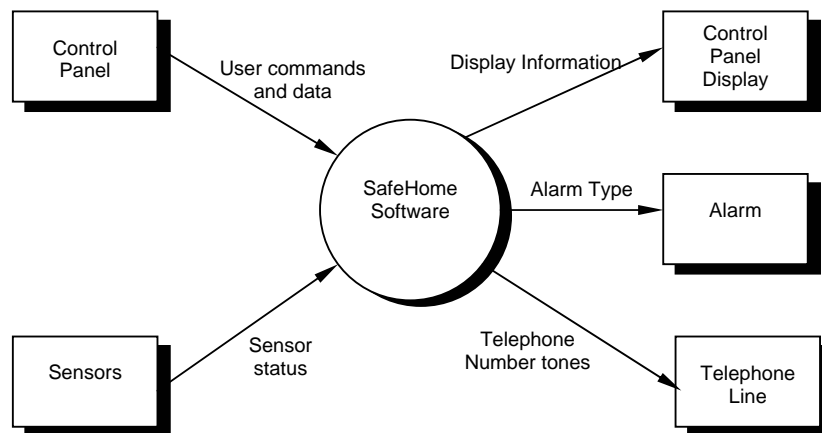


Figure 20.3 Context-level DFD for SafeHome

It should be noted that within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction oriented flow, information flow along an action path may have transform flow characteristics.

20.4 Transform Mapping

Transform mapping is set of design steps that allows a DFD with transform flow characteristics to be mapped into a predefined template for program structure. In this section transform mapping is described by applying design steps to an example system-a portion of the SafeHome security software presented in earlier chapters.

An example

the SafeHome security system introduced earlier in this book, is representative of many computer based products and systems in use today. The product monitors the real world and reacts to changes that it encounters. It also interacts with a user through a series of typed inputs and alphanumeric displays. The level 0 data flow diagram for SafeHome, reproduced from previous lecturer, is shown in 20.3.

During requirements analysis, more detailed flow models would be created for SafeHome. In addition, control and process specifications, a data dictionary, and various behavioral models would also be created.

Design Steps

The above example will be used to illustrate each step in transform mapping the steps begin with a re-evaluation of work done during requirements analysis and then move to the development of program structure.

Step 1. review the fundamental system model. The fundamental system model encompasses the level 0 DFD and supporting information. In actuality the design step begins with an evaluation of both the system specification and the software requirements specification. Both documents describe information flow and structure at the software interface. Figure 20.3, 20.4 depict level 0 and level 1 data flow for the SafeHome software.

Step 2. Review and refine data flow diagrams for the software. Information obtained from analysis models contained in the software requirements specification is refined to produce greater detail. For example, the level DFDs for monitor sensors are examined, and a level 3 data flow diagram is derived as shown in figure 20.6. At level3, each transform in the data flow diagram exhibits relatively high cohesion(in previous chapters). That is, the process implied by a transform performs a single, distinct function that can be implemented as a module in the SafeHome software. Therefore, the DFD in figure 20.6 contains sufficient detail for a “first cut” at the design of program structure for the monitor sensors subsystem, and we proceed without further refinement.

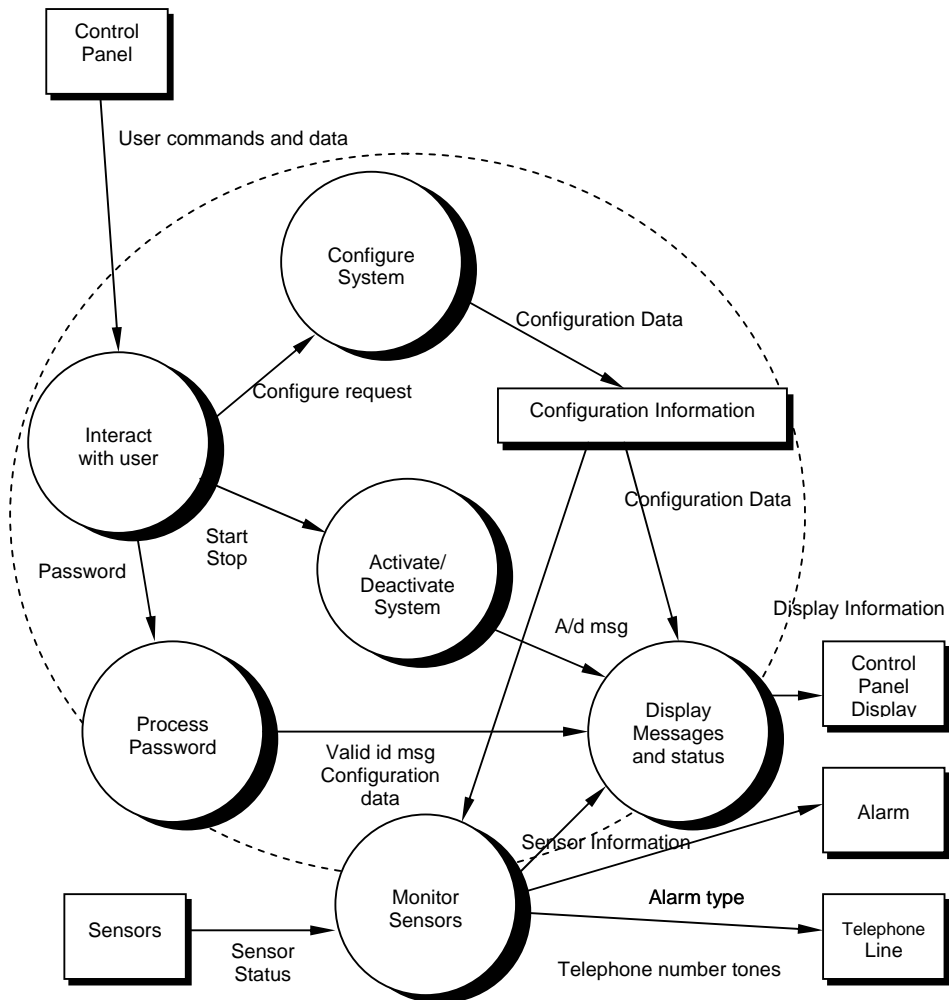


Figure 20.4 Level 1 DFD for SafeHome

Step 3. Determine whether the DFD has transform or transaction flow characteristics. In general, information flow with a system can always be represented as a transform. However, when an obvious transaction characteristic is encountered, a different design mapping is recommended. In this step the designer selects a global (software-wide) flow characteristic based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These subflows can be used to refine program structure derived from a global characteristic described above. For now, we focus our attention only on the monitor sensors subsystem data flow depicted in Figure 20.6.

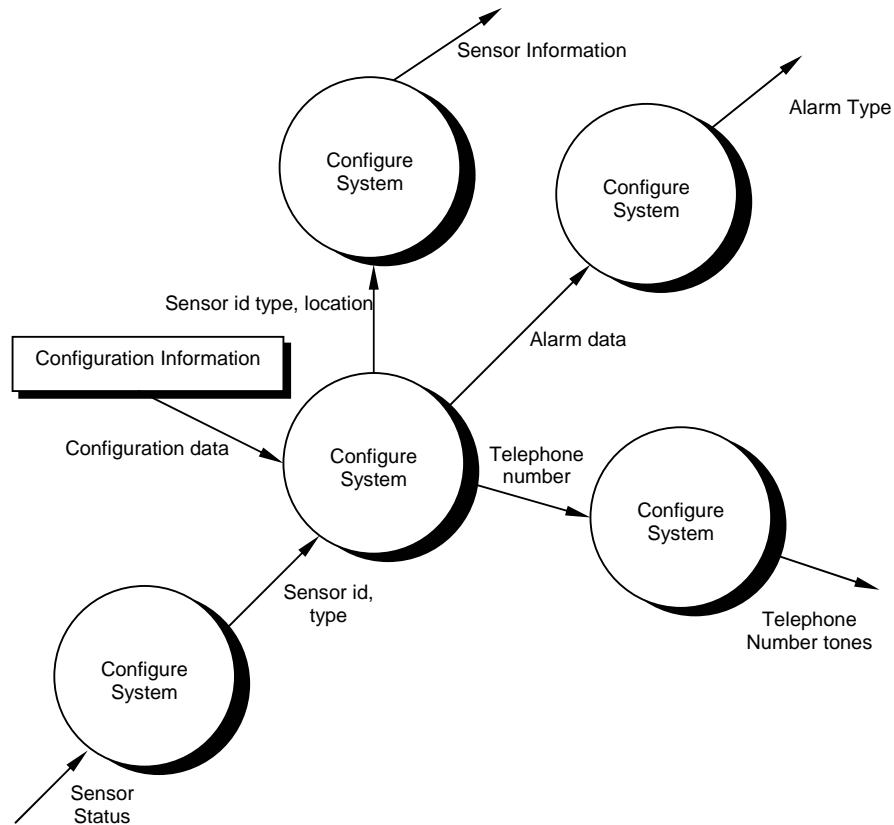


Figure 20.5 Level 2 DFD that refines the monitor sensors process

Evaluating the DFD we see data entering the software along one incoming paths and exiting along three outgoing paths. No distinct transaction center is implied (although the transform acquire alarm conditions could be perceived as such). Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow. the transforms (bubbles) that comprise the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary(e.g., an incoming flow boundary separating read sensors and acquire response info could be proposed). The emphasis in

this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

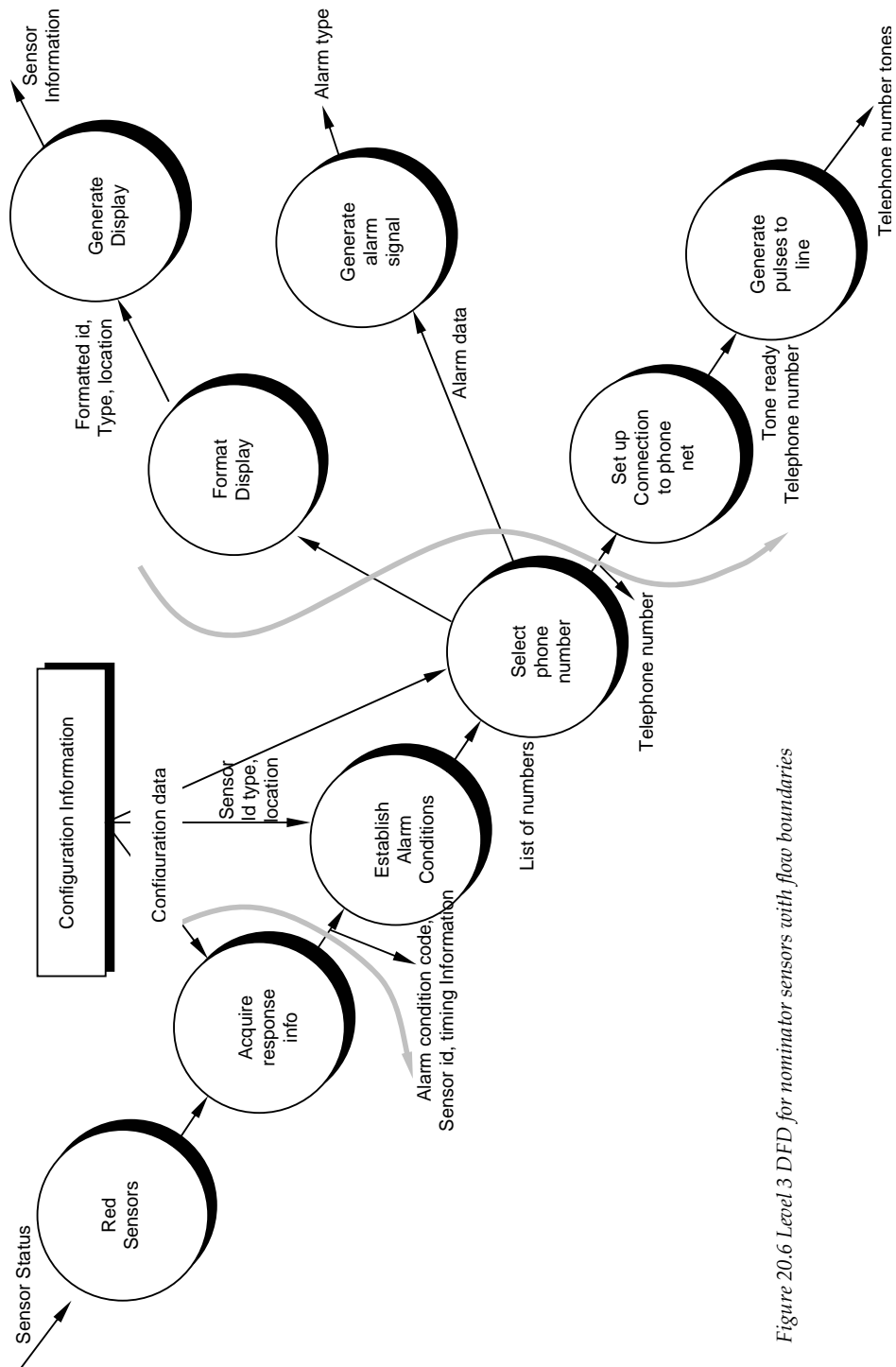


Figure 20.6 Level 3 DFD for nominator sensors with flow boundaries

Step 5. perform “first level factoring”. Program structure represents a top down distribution of control. Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computational, and output work. Middle-level modules perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the monitor sensors subsystem is main controller (called monitor sensors executive) resides at the top of the program structure and serves to coordinate the following subordinate control functions;

an incoming information processing controller, called sensor input controller, coordinated receipt of all incoming data;

A transformation flow controller, called alarm conditions controller, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures;

An outgoing information processing controller, called alarm output controller, coordinated production of output information.

Although a three pronged structure is implied by figure 20.7 complex flows in large systems may dictate two or more control modules for each of the generic control functions described above. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good coupling and cohesion characteristics.

Step 6. Perform “second level factoring.” Second level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the program structure. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second level factoring for the SafeHome data flow is illustrated in figure 20.8.

Although figure 20.8 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one module (recalling potential problems with cohesion), or a single bubble may be expanded to two or more modules. Practical considerations and measures of design quality dictate the outcome of second level factoring. Review and refinement may lead to changes in this structure, but it can serve as a first design iteration.

Second level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of monitor sensors subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first iteration program structure is shown in figure 20.9.

The modules mapped in the manner described above and represent an initial design of program structure. Although modules are named in a manner that implies function, a brief processing narrative (adapted from the PSPEC created during analysis modeling) should be written for each.

The narrative describes:

- ⊥ Information that passes into and out of the module (an interface description);
- ⊥ Information that is retained by a module, e.g., data stored in a local data structure;
- ⊥ A procedural narrative that indicated major decisions points and tasks; and
- ⊥ A brief discussion of restrictions and special features (e.g., file I/O, hardware dependent characteristics, special timing requirements).

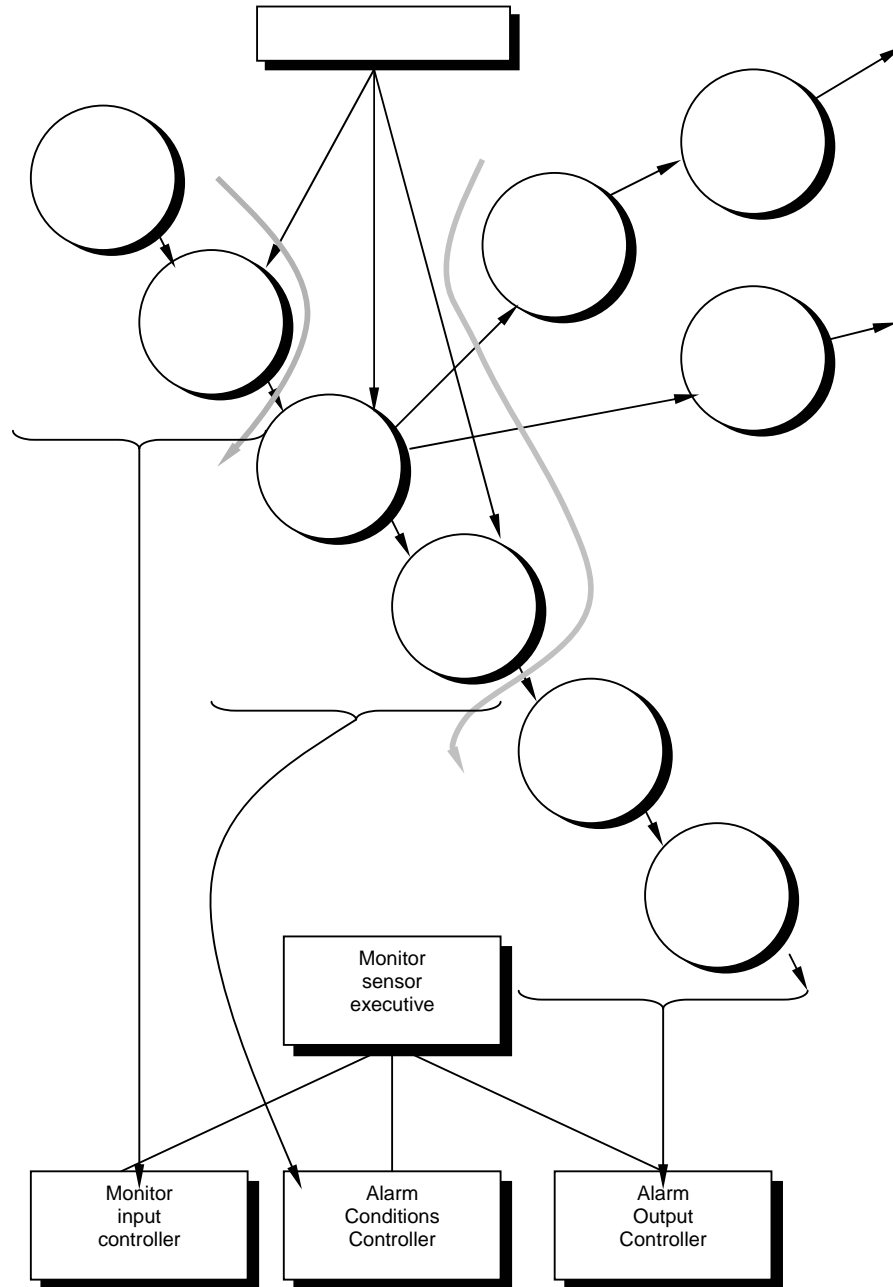
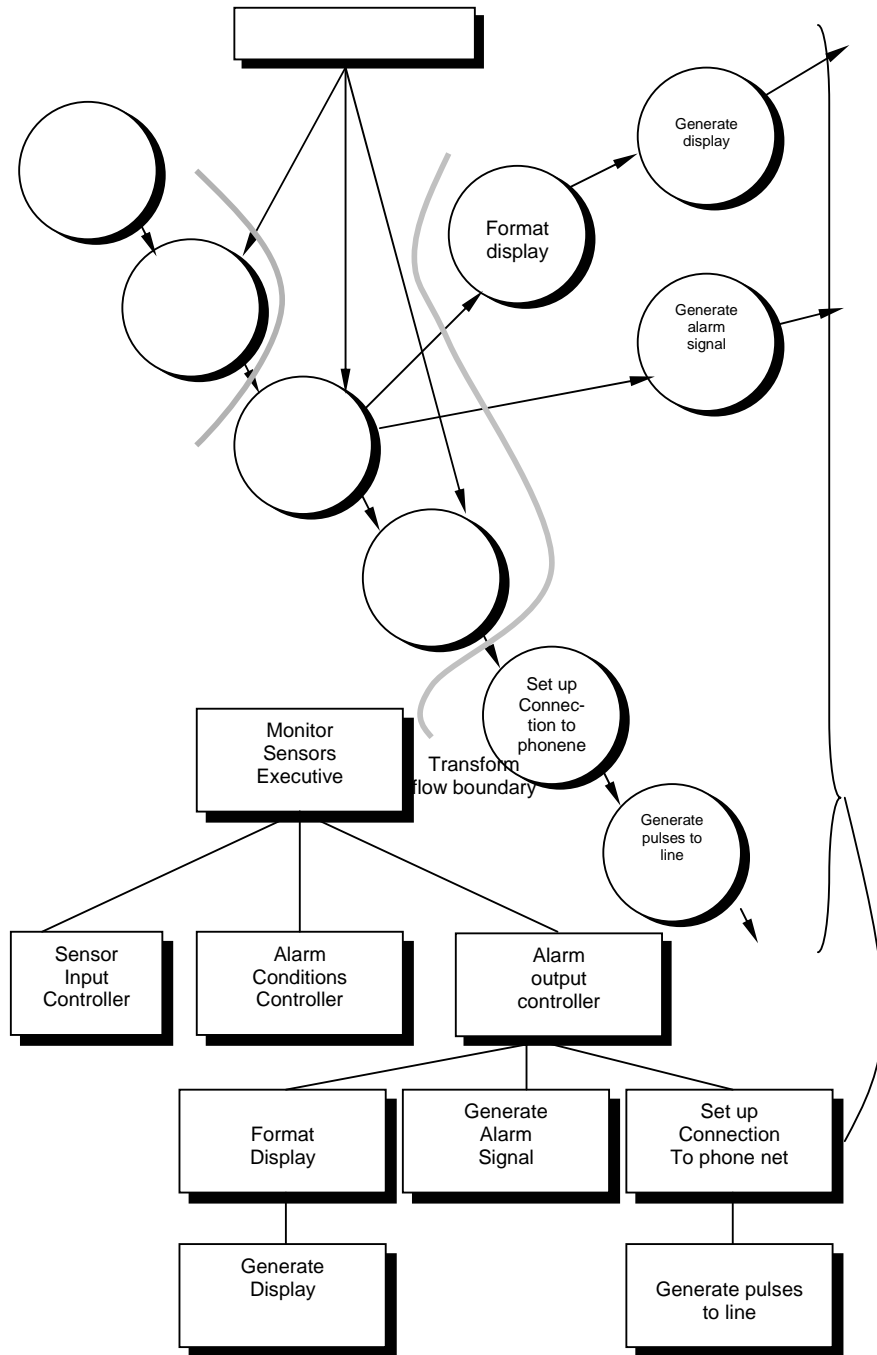


Figure 20.7 First-level factoring for monitor sensors

The narrative serves as a first generation design specification. However, further refinement and additions occur regularly during this period of design.



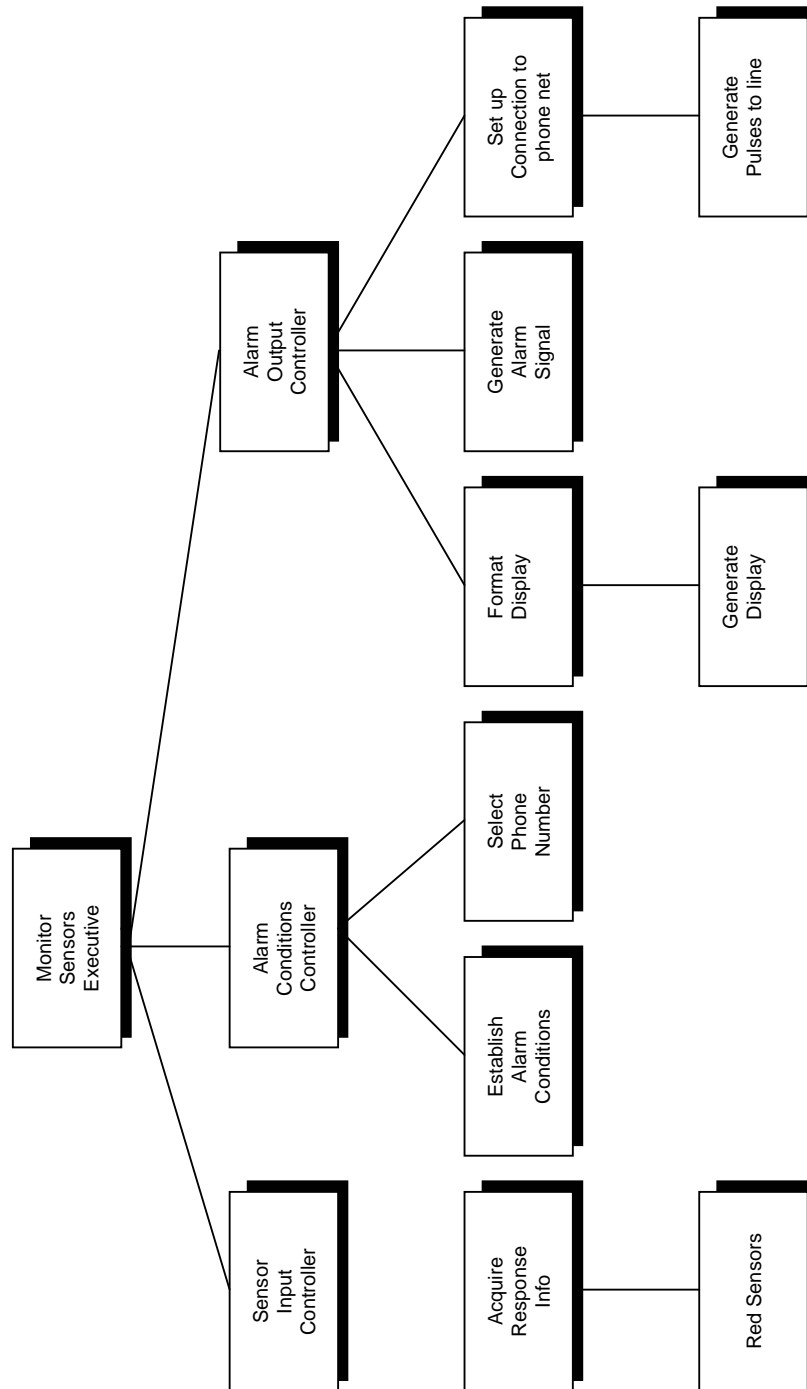


Figure 20.9 "Firstcut" program structure for monitor sensors

Refinements are dictated by practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a module that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved. Software requirements coupled with human judgment is the final arbiter.

Many modifications can be made to the first iteration structure developed for the SafeHome monitor sensors subsystem: (1) The incoming controller can be removed because it is unnecessary when a single incoming flow path is to be managed. (2) The sub substructure generated from the transform flow can be imploded into the module establish alarm conditions (which will now include the processing implied by select phone number). The transform controller will not be needed and the small decrease in cohesion is tolerable. (3) The modules format display and generate display can be imploded (we assume that display formatting is quite simple) into a new module called produce display. The refined software structure for the monitor sensors subsystem is shown in Figure 20.10.

The objective of the preceding seven steps is to develop a global representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modification made at this time require little additional work, yet can have a profound impact on software quality and maintainability.

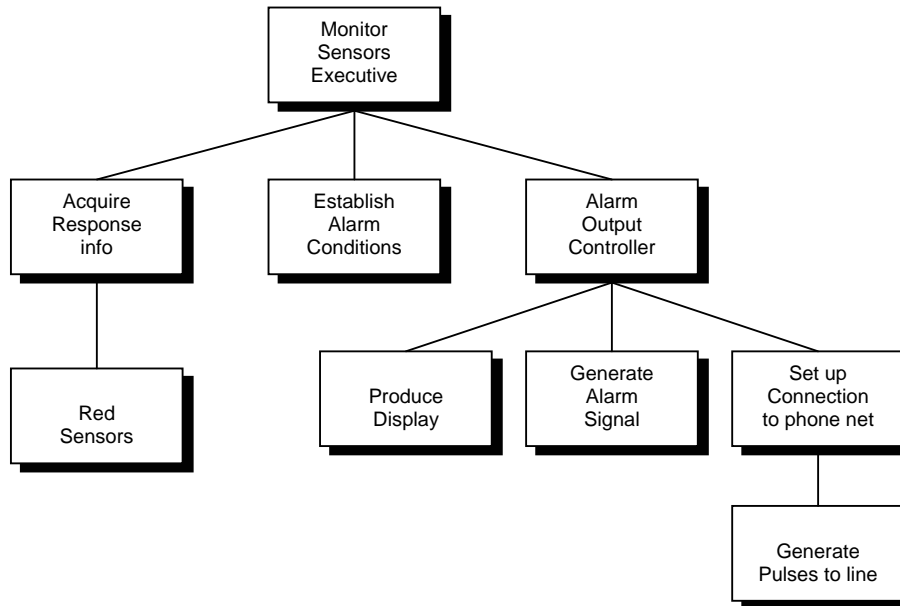


Figure 20.10 Refined program structure for monitor sensors

The reader should pause for a moment and consider the difference between the design approach described above and the process of “writing programs”. If code is the only representation of software, the developer will have great difficulty evaluating or refining at a global or holistic level and will, in fact, have difficulty “seeing the forest for the trees”.

20.5 Transaction Mapping

In many software applications, a single data item triggers one or a number of information flows that effect a function implied by the triggering data item,. The data item, called a transaction, and its corresponding

flow characteristics were discussed in previous section we consider design steps used to treat transaction flow.

An example

Transaction mapping will be illustrated by considering the user interaction subsystem of the SafeHome software. Level 1 data flow for this subsystem is shown as part of figure 20.4. Refining the flow, a level 2 data flow diagram (a corresponding data dictionary, CSPEC, and PSPECs would also be created) is developed and shown in figure 20.11.

As shown in figure, user commands flows into the system and results in additional information flow along one of three action paths. A single data item, command type, causes the data flow to fan outward from a hub. Therefore, the overall data flow characteristic is transaction-oriented.

It should be noted that information flow along two of the three action paths accommodate additional incoming flow (e.g., system parameters and data are input on the “configure” action path. Each action path flows into a single transfor, display messages and status.

Design Steps

Design steps for transaction mapping are similar and in some cases identical to steps for transform mapping. A major difference lies in the mapping of the DFD to software structure.

Step 1. Review the fundamental system model.

Step 2. review and refine data flow diagrams for the software.

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Steps 1,2, and 3 are identical to corresponding steps in transform mapping. The DFD shown in figure 20.11 has a classic transaction flow characteristic. However, flow along two of the action paths emanating from the invoke command processing bubble appears to have transform flow characteristics. Therefore, flow boundaries must be established for both flow types.

Step 4. Identify the transaction center and the flow characteristics along each of the action paths. The location of the transaction center can be immediately discerned from the DFD. The transaction center lies at the origin of a number of actions paths that flow radically from it. For the flow shown in figure 20.11 the invoke command processing bubble is the transaction center.

The incoming path (i.e., the flow path along which a transaction is received) and all action paths must also be isolated. Boundaries that define a reception path and action paths are also shown in the figure. Each action path must be evaluated for is individual flow characteristic. For example, the “password” path (shown enclosed by a shaded area in figure 20.11) has transform characteristics. Incoming, transform, and outgoing flow are indicated with boundaries.

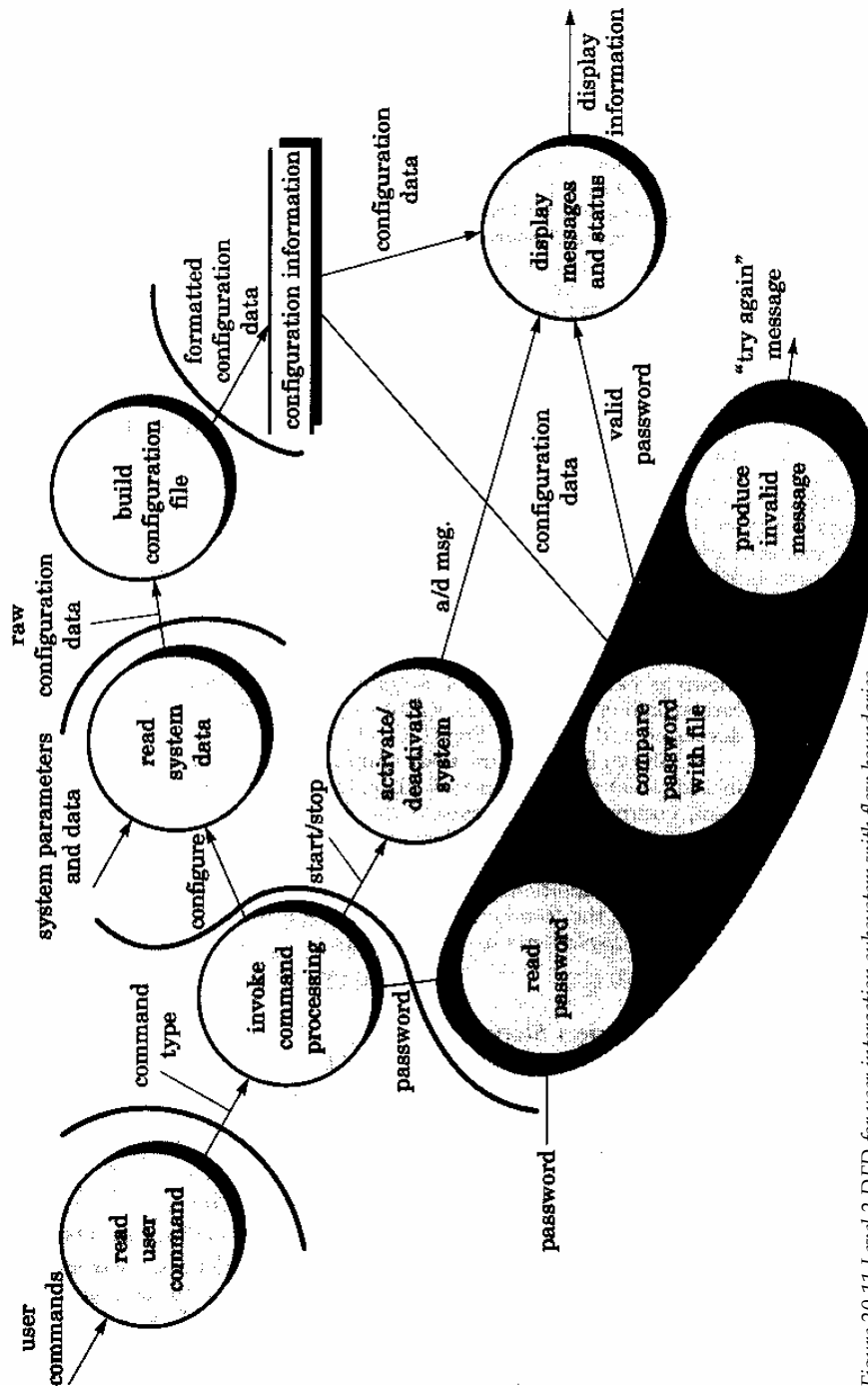


Figure 20.11 Level 2 DFD for user interaction subsystem with flow boundaries

Step 5. map the DFD in a program structure amenable to transaction processing. Transaction flow is mapped into a program structure that contains an incoming branch and a dispatch branch. Structure for the incoming branch is developed in much the same way as transform mapping. Starting at the transaction center, bubbles along the incoming path are mapped into modules. The structure of the

dispatch branch contains a dispatcher module that controls all subordinate action modules. Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics. This process is illustrated in figure 20.12.

Considering the user interaction subsystem data flow, first level factoring for step 5 is shown in figure 20.13. the bubbles read user command and activate/deactivate system map directly into the program structure without the need for intermediate control modules. The transaction center, invoke command processing, maps directly into a dispatcher module of the same name. Controllers for system configuration and password processing are mapped as indicated in figure 20.12.

Step 6. Factor and refine the transaction structure and the structure of each action path. Each action path of the data flow diagram has its own information flow characteristics. We have already noted that transform or transaction flow may be encountered. The action path-related “substructure” is developed using the design steps discussed in this and the preceding section.

As an example, consider the password processing information flow shown (inside shaded area). The flow exhibits classic transform characteristics. A password is input (incoming flow) and transmitted to a transform center where it is compared against stored passwords. An alarm and warning message (outgoing flow) are produce (if a match is not obtained). The “configure” path is drawn similarly using the transform mapping. The resultant program structure is shown in figure 20.14.

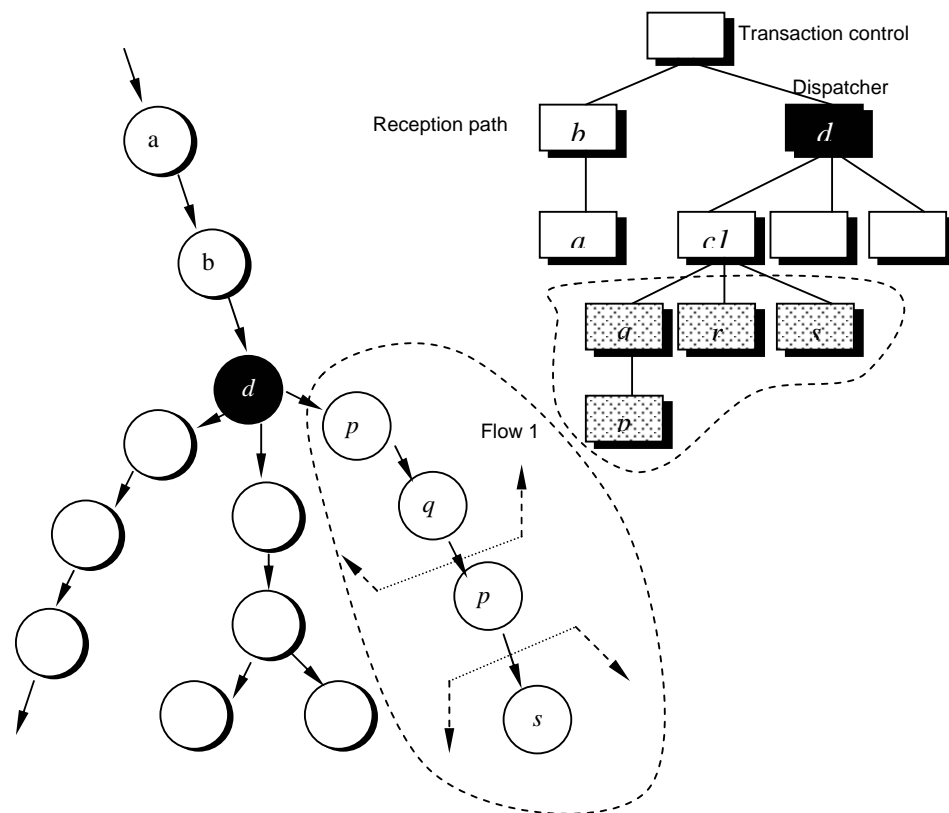


Figure 20.12 Transaction Mapping

Step 7. Refine the first iteration program structure using design heuristics for improved software quality. This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality (efficacy of implementation and test), and maintainability must be carefully considered as structural modifications are proposed.

20.6 Design Postprocessing

Successful application of transform or transaction mapping is supplemented by additional documentation that is required as part of architectural design. After structure has been developed and refined, the following tasks must be completed :

- A processing narrative must be developed for each module.
- An interface description is provided for each module.
- Local and global data structures are defined.
- All design restrictions/limitations are noted.
- A design review is conducted.
- “Optimization “ is considered (if required and justified).

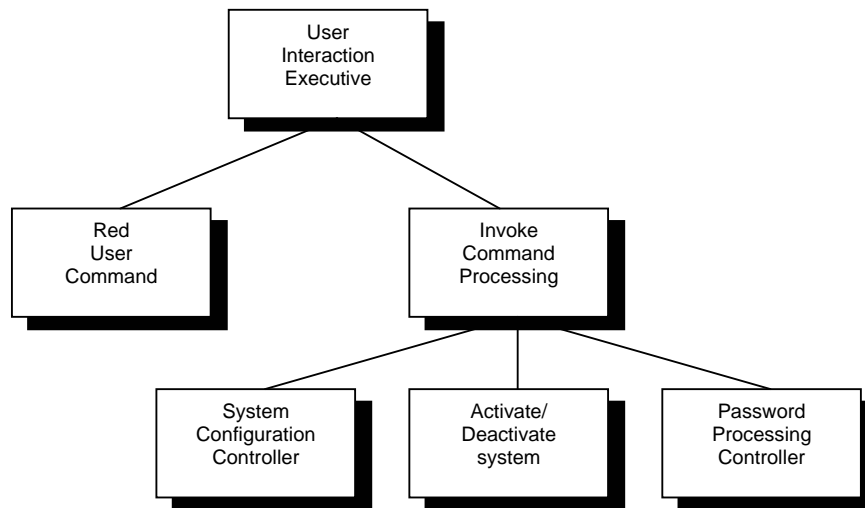


Figure 20.13 First level factoring for user interaction

A processing narrative is (ideally) an unambiguous, bounded description of processing that occurs within a module. The narrative describes processing tasks, decisions, and I/O.

The interface description requires the design of internal module interfaces, external system interfaces and the human-computer interface.

The design of data structures can have a profound impact on program structure and the procedural details for each module. Techniques described in previous chapter establish the basic data model and identify all important data objects. These then serve as the basis for the design of both local and global data structures.

Restrictions and/or limitations for each module are also documented. Typical topics for discussion include restriction of data type or format, memory or timing limitations, bounding values or quantities of data structures, special purpose of a restrictions/limitations section is to reduce the number of errors introduced because of assumed functional characteristics.

Once design documentation has been developed for all modules, a design review is conducted. The review emphasizes trace ability to software requirements, quality of program structure, interface descriptions, data structure descriptions, implementation and test practicality, and maintainability.

20.7 Short Summary

- Software design encompassed four distinct but interrelated activities: data design, architectural design interface design, and procedural design. When each of these design activities has been completed, a comprehensive design model exists for the software.
- Data design translates the data objects defined in the analysis model into data structures that reside within the software. The attributes that describe data objects, the relationships between data objects, and their use within the program all influence the choice of data structures.
- The architectural design method presented in this chapter use information flow characteristics described in the analysis model to derive program structure.
- A data flow diagram is mapped into program structure using one of two mapping approaches – transform mapping and/or transaction mapping. Transform mapping is applied to an information flow that exhibits distinct boundaries between incoming and outgoing data.
- The DFD is mapped into a structure that allocates control to input, processing, and output along three separately factored module hierarchies.
- Transaction mapping is applied when a single information item causes flow to branch along one of many paths.
- The DFD is mapped into a structure that allocates control to a substructure that acquires and evaluates a transaction. Another substructure controls all potential processing actions based on a transaction.

20.8 Brain Storm

1. Explain briefly about Architectural Design ?
2. What is Transform Mapping ? Give a brief note on it ?
3. Describe about Transaction Mapping ?
4. What is SignPostprocessing ?



Lecture 21

Discussion

Lecture 22

Design Methods

Objectives

In this lecture you will learn the following

- ✧ About Architectural Design Optimization
- ✧ About Interface Design
- ✧ About Procedural Design

Coverage Plan

Lecture 22
22.1 Snap Shot
22.2 Architectural Design Optimization
22.3 Interface Design
22.4 Human-Computer Interface Design
22.5 General Interaction
22.6 Procedural Design
22.7 Short Summary
22.8 Brain Storm

22.1 Snap Shot

In this lecture, we focus on Architectural Design Interface Design, General Interaction and Procedural Design.

22.2 Architectural Design Optimization

Any discussion of design optimization should be prefaced with the following comment: "Remember that an 'optimal design' that doesn't work has questionable merit." The software designer should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design quality measures.

Refinement of program structure during the early stages of design is to be encouraged. Alternative representation may be derived, refined and evaluated for the best approach. This approach optimization is one of the true benefits derived from developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design optimization should strive for the smallest number of modules that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

For performance critical applications it may be necessary to "optimize" during later design iterations and possibly during coding. The software engineer should note however that a relatively small percentage of a program often accounts for a large percentage of all processing time. It is not unreasonable to propose the following approach for performance critical software:

1. Develop and refine program structure without concern for performance critical optimization.
2. Use CASE tools that simulate run time performance to isolate areas of inefficiency.
3. During late design iterations select modules that are suspect "time hogs" and carefully develop procedures for time efficiency.
4. Code in an appropriate programming language.
5. Instrument the software to isolate modules that account for heavy processor utilization.
6. If necessary redesign or recode in machine-dependent language to improve efficiency.

This approach follows a dictum that will be further discussed in a later chapter: "Get it to work, then make it fast."

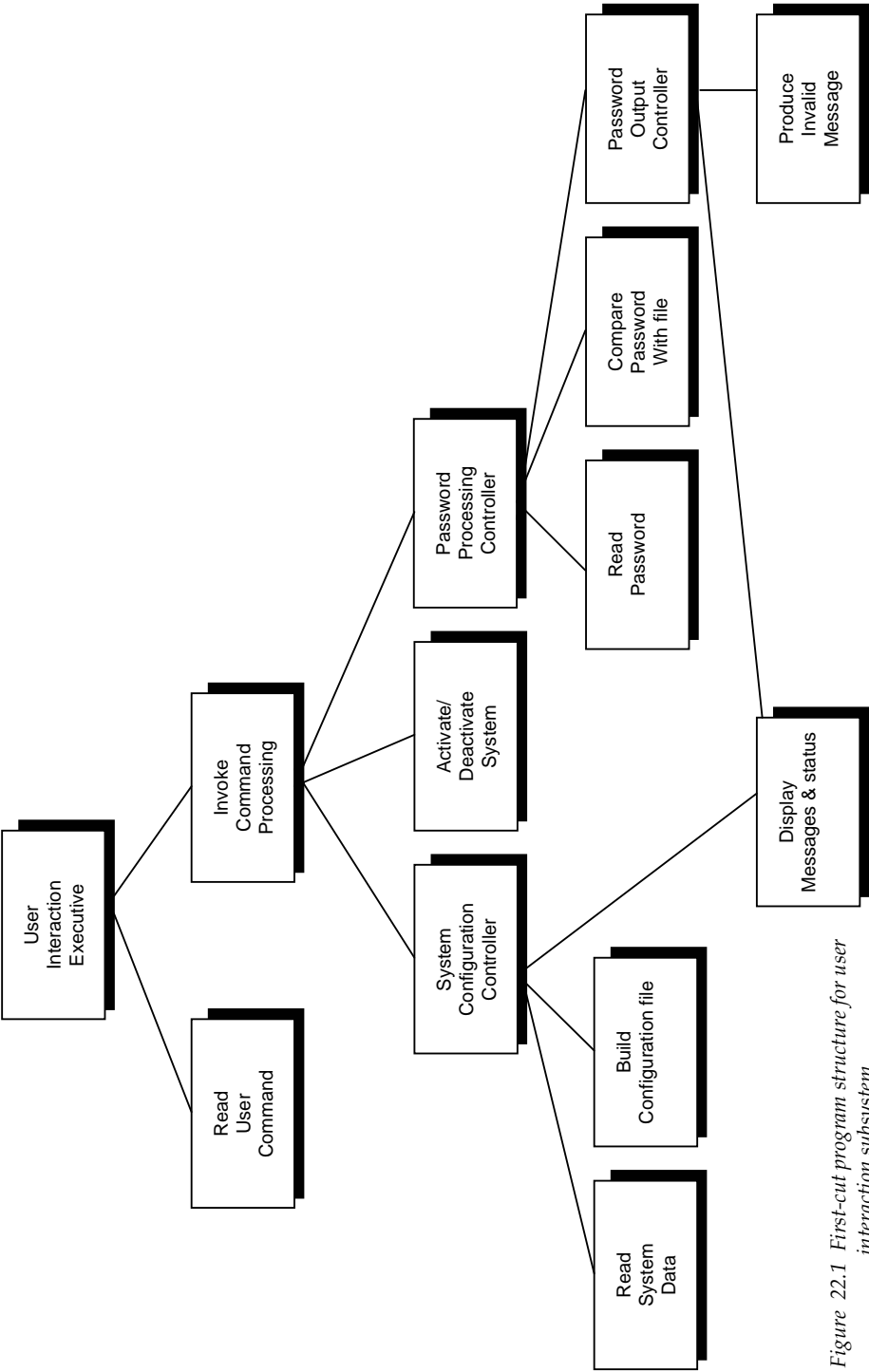


Figure 22.1 First-cut program structure for user interaction subsystem

22.3 Interface Design

The architectural design provides a software engineer with a picture of the program structure like the blue print for a house the overall design is not complete without a representation of doors windows and utility connections for water electricity and telephone. The "doors, windows, and utility connections" for computer software comprise the interface design of a system.

Interface design focuses on three areas of concern : (1) the design of interfaces between software modules ; (2) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities); and (3) the design of the interface between a human (i.e., the user) and the computer.

Internal and External Interface Design

The design of internal program interfaces sometimes called intermodular interface design is driven by the data that must flow between modules and the characteristics of the programming language in which the software is to be implemented. In general the analysis model contains much of the information required for intermodular interface design. The data flow diagram describes how data objects are transformed as they move through a system. The transforms of the DFD are mapped into modules within the program structure. Therefore the arrows flowing into and out of each DFD transform must be mapped into a design for the interface of the module that corresponds to that DFD transform.

External interface design begins with an evaluation of each external entity represented in the DFDs of the analysis model. The data and control requirements of the external entity are determined and appropriate external interfaces are designed. For example the SafeHome software discussed earlier in this chapter requires interfacing with a variety of external security sensors. The design of the external interface for each sensor is predicated on the specific data and control items required for the sensor.

Both internal and external interface designs must be coupled with data validation and error handling algorithms within a module. Because side effects propagate across program interfaces it is essential to check all data flowing from module to module to ensure that the data conform to bounds established during requirement analysis.

User Interface Design

In the preface to his book on user interface design, Ben Shneiderman states: Frustration and anxiety are part of daily life for many users of computerized information systems. They struggle to learn command language or menu selecting systems that are supposed to help them do their job. Some people encounter such serious cases of computer shock terminal or network neurosis that they avoid using computerized systems.

The problem to which Shneiderman alludes are real. We have all encountered "interfaces" that are difficult to learn difficult to use confusing unforgiving and in many cases , totally frustrating. Yet someone spent time and energy building each of these interfaces and it is likely that the builder did not create these problems purposely.

User interface design has as much to do with the study of people as it does with technology issues. Who is the user? How does the user learn to interface with a new computer-based system? How does the user interpret information produced by the system? What will the user expect of the system? These are only a few of the many questions that must be asked and answered as part of user interface design.

22.4 Human-Computer Interface Design

The overall process for designing a user interface design with the creation of different models of system function. The human and computer-oriented tasks that are required to achieve system function are then delimited; design issues that apply to all interface designs model and the result is evaluated for quality.

Interface Design Models

Four different models come into play when a human computer interfaces is to be designed. The software engineer creates a design model, a human engineer establishes a user model the end user develops a mental image that is often called the user's model or the system perception and the implementers of the system create a system image. Unfortunately these models may differ significantly. The role of interface designer is to reconcile these differences and derive a consistent representation of the interface.

A design model of the entire system incorporates data architectural interface and procedural representations of the software. The requirements specification may establish certain constraints that help to define the user of the system but the interface design is often only incidental to the design model.

The user model depicts the profile of end users of the system. To build an effective user interface "all design should begin with an understanding of the intended user including profile of their age, sex , physical abilities, education, cultural, or ethnic background motivation goals and personality" In addition users can be categorized as:

- novices --no syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general;
- Knowledgeable intermittent users-- reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface; and
- Knowledgeable, frequent users -- good semantic and syntactic knowledge that often leads to the "power-user syndrome" that is individuals who look of shortcuts and abbreviated modes of interaction.

The system perception is the image of the system that an end user carries in his or her head. For example if the user of a particular word processor were asked to describe its operation the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands word processors fully but has only worked with the specific word processor once, might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

The system image couples the outward manifestation of the computer based system (the look nad feel of the interface) with all supporting information (books, manuals, video tapes) that describe system syntax and semantics .When the system image and the system image and the system perception are coincident , users generally feel comfortable with the software and use it effectively. To accomplish this "melding " of the models the design model must have been developed to accommodate the information contained in the user model ant he system image must accurately reflect syntactic and semantic information about the interface. The interrelationship among the models is shown if Figure 22.2.

The models described in this section are "abstractions of what the user is doing or thinks he is doing or what somebody else thinks he ought to be doing when he uses an interactive style" In essence these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "Know the user, know the tasks."

Task Analysis and Modeling

Task analysis and modeling can be applied to understand the tasks that people currently perform (when using a manual or semi automated approach) and then map these into a similar (but not necessarily identical)set of tasks that are implemented in the context of the HCI. This can be accomplished by observation or by studying an existing specification of a computer- based solution and deriving a set of user tasks that will accommodate the user model the design model and the system perception.

Regardless of the overall approach to task analysis the human engineer must first define and classify tasks. One approach is stepwise elaboration . For example assume that a small software software company want to build a computer aided design system explicitly for interior designers. By observing a designer at work the engineer notices that the interior design is comprised of a number of major activities : furniture layout fabric and material selection wall and window covering selection presentation (to the customer), costing and shopping. Each of these major tasks can be elaborated into subtasks. For example , furniture

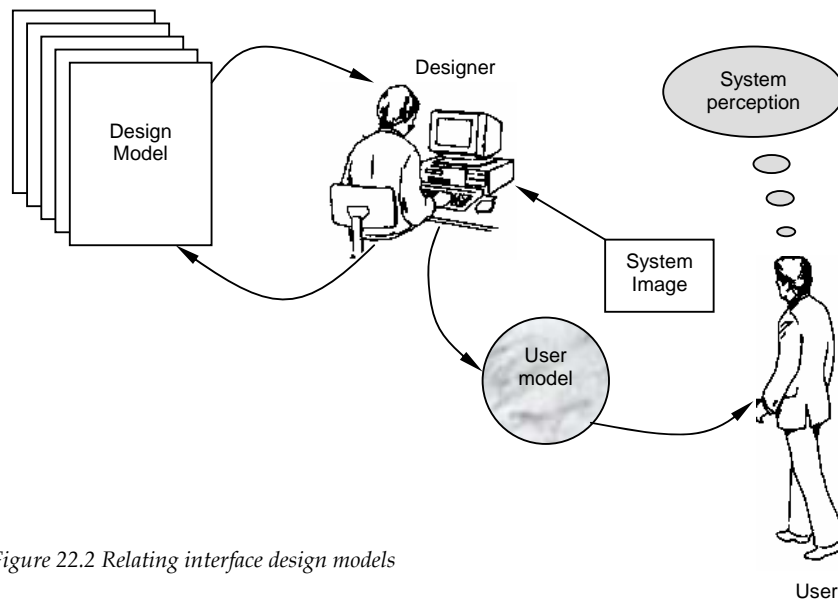


Figure 22.2 Relating interface design models

layout can be refined into the following tasks: (1)Draw floor plan based on room dimensions; (2)place windows and doors at appropriate locations; (3)use furniture templates to draw scaled furniture outlines on floor plan;(4) move furniture outline to get best placement; (5) label all furniture outline; (6)draw dimensions to show location and (7)draw perspective view for customer. A similar approach could be used for each of the other major tasks.

Subtasks 1 to 7 can each be refined further. Subtasks 1 to 6 will be performed by manipulation information and performing actions with the user interface. On the other hand subtask 7 can be performed automatically in software and will result in little direct user interaction. The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a

"typical" interior designer) and system perception (what the interior designer expects from a automated system).

An alternative approach to task analysis takes an object-oriented point of view. The human engineer observes the physical objects that are used by the interior designer and the actions that are applied to each object. For example interior the furniture template would be an object in this approach to task analysis The interior designer would select the appropriate template move it to a position on the floor plan trace the furniture outline and so fourth. The design model for the interface would not describe implementation details for each of these actions but it would define user tasks that accomplish the end result (drawing furniture outline on the floor plan).

Once each task or action has been defined interface design begins. The first steps in the interface design process can be accomplished using the following approach:

1. Establish the goals and intentions for the task
2. Map each goals intention to a sequence of specific actions
3. Specify the action sequence s it will be executed at the interface level.
4. Indicate the state of the system ; i.e., what does the interface look like at the time that an actionist eh sequence is performed?
5. Define control mechanism i.e., the devices and action available to the user to alter the system state.
6. Show how control mechanism affect the state of the system.
7. Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues

As the design of a user interface evolves four common design issues almost always surface system response time user help facilities error information handling and command labeling . unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first linking of a problem doesn't occur until an operational prototype is available) Unnecessary interaction project delays and customer frustration 2almost always result. it is far better to establish was as a design issue to be considered at the beginning of software design when changes are easy and costs are low.

System response time is the primary complaint for many interactive systems. In general system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristic: length and variability. If the length of time for system response time is too long user frustration and stress is the inevitable result. However a very brief response time can also be detrimental if the user is being paced by the interface. A rapid response may force the user to rush and therefore make mistakes.

Variability refers to he deviation from average response time and in many ways it is the more important or the response time characteristics. Low variability enables the user to establish a rhythm even if response time is relatively long. For example one second response to a command is preferable to a response that

varies from 0.1 to 2.5 seconds. The user is always off balance always wondering whether something "different" has occurred behind the scenes.

Almost every user of an interactive, computer-based system requires help now and then, in some cases a simple question addressed to a knowledgeable colleague can do the trick. In others detailed research in a multivolume set of user manuals may be the only option. In many cases however modern software provides on line help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

Two different types of help facilities are encountered integrated and add on. An integrated help facility is designed into the software from the beginning. It is often context sensitive enabling the user to select from those topics that are relevant to the actions currently being performed. Obviously, this reduces the time required for the user to obtain help and increases the "friendliness " of the interface. An add-on help facility is added to the software after the system has been built. In many ways, it is really an on-line user's manual with limited query capability. The user may have search through a list of hundreds of topics to find appropriate guidance often making many false starts and reviewing much irrelevant information . There is little doubt that the integrated help facility is preferable to a the add-on approach.

A number of design issues must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction ? Options include help only for a subset of all functions and actions and help for all functions.
- How will the user request help? Options include data help menu a special function key and a HELP command .
- How will help be represented ? Options include a separate window a reference to a printed document and a one or two line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include a turn button displayed on the screen and a function key or control sequence.
- How will help information be structured? Options include a "flat" structure in which all information is accessed through a keyword a layered hierarchy of information that provides increasing detail as the user proceeds into the structure and the use of hypertext.

Error messages and warning are "bad news " delivered to user's of interactive systems when something has gone awry. At their worst error messages and warning impact useless or misleading information and serve only to increase user frustration. Few computer users have not encountered an error of the form;

Severe System Failure –14A

Sometimes an explanation for error 14A must exist; otherwise why would the designers have added the identification? Yet the error message provides no real indication of what is wrong or where to look to get additional information. An error message presented in the manner shown above does nothing to assuage user anxiety or to help correct the problem.

In general every error message or warning produced by an interactive system should have the following characteristics:

- The message should describe the problem in jargon that the user can understand.
- The messages should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have)
- The message should be accompanied by an audible or visual cue. That is a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color".
- The message should be "nonjudgmental." That is the wording should never place blame on the user.

Because no one really likes bad news few users will like an error message no matter how well it is designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

The typed command was once the most common mode of interacting between user and system software and was commonly used for application of every type. Today the use of window-oriented point and pick interfaces has reduced reliance on typed commands, but many power-users continue to prefer a command-oriented mode of interacting. In many situations the user can be provided with an option software functions can be selected from a static or pull down menu or invoked through some keyboard command sequence.

A number of design issues arise when commands are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., ^P), function keys and a typed word.
- How difficult will it be to learn and remember the command? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?

In a growing number of applications interface designers provide a commands under a user-defined name. Instead of each command being typed individually the command macro is typed and all commands implied by it are executed in sequence.

In an ideal setting conventions for command usage should be established across all applications. It is confusing and often error-prone for a user to type ^D when a graphics object is to be duplicated in one application and ^D when a graphics object is to be deleted in another. The potential for error is obvious.

Implementation Tools

The process of user interface design is interactive. That is a design model is created implemented as a prototype examined by users (who fit the user model described earlier) and modified based on their comments. To accommodate this interactive design approach a broad class of interface design and prototyping tools has evolved. Called user interface toolkits or user interface development systems (UIDS) these tools provide routines or objects that facilitate creation of windows, menus, device interaction, error messages, commands and many other elements of an interactive environment.

Using prepackaged software that can be used directly by the designers and implementer or a user interface, a UIDS provides built in mechanism for:

- Managing input devices (such as the mouse or keyboard);
- Validating user input;
- Handling errors and displaying error messages;
- Providing feedback (e.g., automatic input echo);
- Providing help and prompts;
- Handling windows and fields, scrolling within windows;
- Establishing connections between application software and the interface;
- Insulating the application from interface management functions; and
- Allowing the user to customize the interface

The functions described above can be implemented using either a language based or a graphical approach.

Design Evaluation

Once an operational user interface prototype has been created it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive" in which a user provides imprompt feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.

The user interface evaluation cycle takes the form shown in Figure 22.3. After the preliminary design has been completed a first level prototype is created. The prototype is evaluated by the user who provides the designer with direct comments about the efficacy of the interface. In addition if formal evaluation techniques are used (e.g., questionnaires rating sheets) the designer may extract information from this information (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. But is it possible to evaluate the quality of a user interface before a prototype is built? If potential problems can be uncovered and corrected early the number of loops through the evaluation cycle will be reduced and development time will shorten.

When a design model of the interface has been created a number of evaluation criteria can be applied during early design reviews:

1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of commands or actions specified and the average number of arguments per command or individual operations per action provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, commands and system states indicated by the design model indicate the memory load on users of the system.

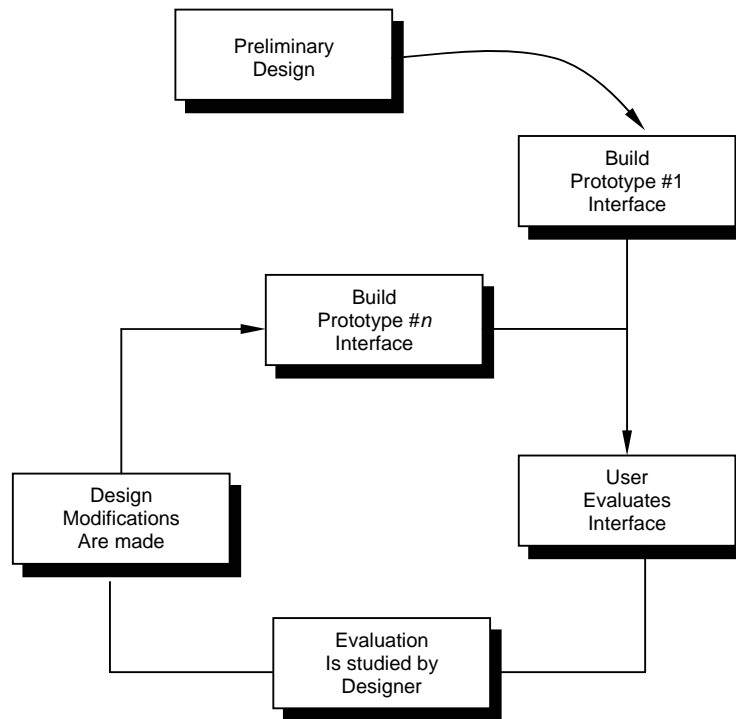


Figure 22.3 The interface design evaluation design

4. Interface style help facilities and error handling protocols provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data questionnaires can be distributed to user of the prototype can be (1) simple yes/no (2) numeric (3) scaled (subjective), (4)percentage (subjective). Examples are:

1. Were the commands easy to remember?
2. How many different commands did you use?
3. How easy was it to learn basic system operations?
4. Compared to other interfaces you've used, how would this rate? (top 1%,top10%, top 25%, top 50%, bottom 50%)

If qualitative data are desired a form of time study analysis can be conducted. Users are observed during interaction and data such as number of tasks correctly completed over a standard time period, frequency of command use , sequence of commands, time spent "looking " at eh display, number of errors types of error and error recovery time, time spent using help and number of help references per standard time period are collected and used as a guide for interface modification

A complete discussion of user interface evaluation methods is beyond the scope of this book. For further information see [LEA88].

22.5 General Interaction

Guidelines for general interaction often cross the boundary into information display data entry and overall system control. They are therefore all-encompassing and are ignored at great risk. The following guidelines focus on general interaction:

Be Consistent. Use a consistent format for menu selection command input data display and the myriad other functions that occur in a HCI.

Offer meaningful feedback Provide the user with visual and auditory feedback to ensure that two way communication (between user and interface) is established .

Ask for verification Of any nontrivial destructive action If a user requests the deletion of a file indicates that substantial information's to be overwritten or asks for the terminating of a program an "Are you sure...?" message should appear.

Permit easy reversal of most actions. UNDO or REVERSE functions have saved tens of thousands of end users from millions of hours of frustration . Reversal should be available in every interactive application.

Reduce the amount of information that must be memorized between actions. The user should not be expected to remember a list of numbers or names so that he or she can reuse them in a subsequent function. memory load should be minimized.

Seek efficiency in dialog, motion, and thought. Keystroke should be minimized the distance a mouse must trace between picks should be considered in designing screen layout, the user should rarely encounter a situation where he or she asks , " Now what does this mean?"

Forgive mistakes. The system should protect itself from errors that might cause it to fail.

Categorize activities by function and organize screen geography accord singly. One of the key benefits of the pull-down menu is the ability to organize command by type. In essence the designer should strive for "cohesive" placement of commands and actions.

Provide help facilities that are context sensitive.

use simple auctioneers or short verb phrases to name commands. A lengthy command name is more difficult to recognize and recall. It may also take up unnecessary space in menu lists.

Information Display

If information presented by the HCI is incomplete, ambiguous or unintelligible , the application will fail to satisfy the needs of a user, Information is "displayed " in many different ways: with text pictures and sound; by placement, motion, and size; and using color resolution and even omission. The following guidelines focus on information display:

Display only that information that is relevant to the current context. The user should not have to wade through extraneous data, menus and graphics to obtain information relevant to a specific system function.

Don't bury the user with data use a presentation format that enables rapid assimilation of information. Graphs or charts should replace voluminous tables.

Use consistent labels, standard abbreviations and predictable colors. The meaning of a display should be obvious without reference to some outside source of information.

Allow the user to maintain visual context. If graphical representations are scaled up and down, the original image should be displayed constantly (in reduced form at the corner of the display) so that the user understands the relative location of the portion of the image that is currently being viewed.

Produce meaningful error messages.

Use upper and lower case, identification and text grouping to aid in understanding. Much of the information imparted by a HCI is textual and the layout and form of the text has a significant impact on the ease with which information is assimilated by the user.

Use windows to compartmentalize different types of information. Windows enable the user to "keep" many different types of information within easy reach.

Use 'analog' displays to represent information that is more easily assimilated with this form of representation. For example a display of holding tank pressure in an oil refinery would have little impact if a numeric representation were used. However if a thermometer like display were used vertical motion and color changes could be used to indicate dangerous pressure conditions. This would provide the user with both absolute and relative information.

Consider the available geography of the display screen and use it efficiently. When multiple windows are to be user, space should be available to show at least some portion of each. In addition screen size (a system engineering issue) should be selected to accommodate the type of application that is to be implemented.

Data Input

Much of the user's time is spent picking commands typing data and otherwise providing system input. In many application the keyboard remains in the primary input medium, but the mouse, digitizer and even voice recognition systems are rapidly becoming effective alternatives. The following guidelines focus on data input:

Minimize the number of input actions required of the user. Above all reduce the amount of typing that is required. This can be accomplished by using the mouse to select from predefined sets of input using "sliding scale" to specify input data across a range of values and using macros that enable a single keystroke to be transformed into a more complex collection of input data.

Maintain consistency between information display and data input. The visual characteristics of the display (e.g., text size, color, and placement) should be carried over to the input domain.

Allow the user to customize input. An expert user might decide to create custom commands or dispense with some types of warning messages and action verification. The HCI should allow this.

Interaction should be flexible but also tuned to the user's preferred mode of input. The user model will assist in determining which mode of input is preferred. A clerical worker might be very happy with keyboard input while a manager might be more comfortable using a point and pick device such as a mouse.

Deactivate commands that are inappropriate in the context of current actions. This protects the user from attempting same action that could result in an error.

Let the user control the interactive flow. The user should be able to jump unnecessary actions, change the order of required actions (when possible in the context of an application) and recover from error conditions without exiting from the program.

Provide help to assist with all input actions.

Eliminate "Mickey mouse" input. Do not require the user to specify units for engineering input (unless there may be ambiguity). Do not require the user to type .00 for whole number dollar amounts, provide default values whenever possible and never require the user to enter information that can be acquired automatically or computed within the program.

22.6 Procedural Design

Procedural design occurs after data, architectural and interface design have been established. In an ideal world the procedural specification required to define algorithmic details would be stated in a natural language such as English. After all members of a software development organization all speak a natural language people outside the software domain could readily understand the specification and no new learning would be required.

Unfortunately there is one small problem. Procedural design must specify procedural detail unambiguously and a lack of ambiguity in a natural language is not natural. Using a natural language we can write a set of procedural steps in too many different ways. We frequently rely on context to get a point across. We often write as if a dialog with the reader were possible. For these and many other reasons a more constrained mode for representing procedural detail must be used.

Structured Programming

The foundation of procedural design were formed in the early 1960s and were solidified with the work of Edgar Dijkstra and his colleagues. In the late 1960s Dijkstra and others proposed the use of a set of existing logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain" That is each construct has a predictable logical structure was entered at the top and exited at the bottom enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition and repetition. Sequence implements processing steps that are essential in the specification of any algorithm condition provides the facility for selected processing based on some logical occurrence and repetition provides for looping. These three constructs are fundamental to structured programming an important procedural design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable operations. Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call

chunking. To understand this process consider how you are reading this page. You do not read individual letters; rather you recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow reader to recognize procedural elements of a module rather than read the design or code line by line. Understanding is enhanced when readily recognizable logical forms are encountered.

Any program regardless of application area or technical complexity can be designed and implemented using only the three structured constructs. It should be noted however that dogmatic use of only these constructs can sometimes cause practical difficulties.

Graphical Design Notation

"A picture is worth a thousand words" but it's rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the flowchart or box diagram, provide excellent pictorial patterns that readily depict procedural detail. However if graphical tools are misused, the wrong picture may lead to the wrong software.

The flowchart was once the most widely used graphical representation for procedural design. Unfortunately it was the most widely abused method as well.

The flowchart is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition and arrows show the flow of control. Figure 22.4 illustrates the three structured constructs. Sequence is represented as two processing boxes connected by a line of control. Condition also called if-then-else is depicted as a decision diamond which if true causes then part processing to occur and if false invokes else-part processing. Repetition is represented using two slightly different forms. The do-while tests a condition and executes a loop task repetitively as long as the condition holds true. A repeat-until executes the loop task first then tests a condition and repeats the task until the condition fails. The selection construct shown in the figure is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

The structured constructs may be nested within one another as shown in Figure 22.5. In the figure a repeat-until forms the then part of an if-then-else (shown enclosed by the outer dashed boundary). Another if-then-else forms the else-part of the larger condition. Finally, the condition itself becomes a second block in a sequence. By nesting constructs in this manner, a complex logical schema may be developed. It should be noted that any one of the blocks in Figure 22.5 could reference another module, thereby accomplishing procedural layering implied by program structure.

In general the dogmatic use of only the structured constructs can introduce inefficiency when an escape from a set of nested loops or nested conditions is required. More important, additional complication of all logical tests along the path of escape can cloud software control flow, increase the possibility of error and have negative impact on readability and maintainability. What can we do?

The designers left with two options: (1) The procedural representation is redesigned so that the "escape branch" is not required at a nested location in the flow of control; or (2) the structured constructs are violated in a controlled manner; that is a constrained branch out of the nested flow is designed. Option 1 is obviously the ideal approach but option can be accommodated without violating of the spirit of structured programming.

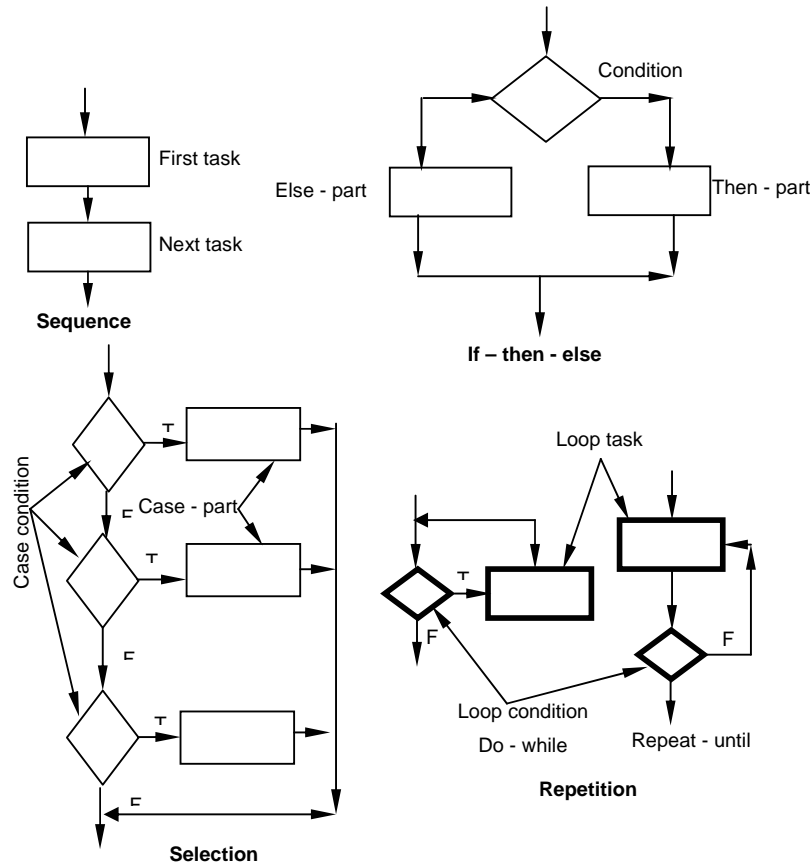


Figure 22.4 Flowchart constructs

Another graphical design tool, the box diagram evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs. Developed by Nassi and Shneiderman and extended by Chapin the diagrams (also called Nassi-Shneiderman charts, N-S charts or Chapin charts) have the following characteristics: (1) functional domain (that is the scope of repetition or an if-then-else) is well defined and clearly visible as a pictorial representation; (2) arbitrary transfer of control is impossible; (3) the scope of local and/or global data can be easily determined and (4) recursion is easy to represent.

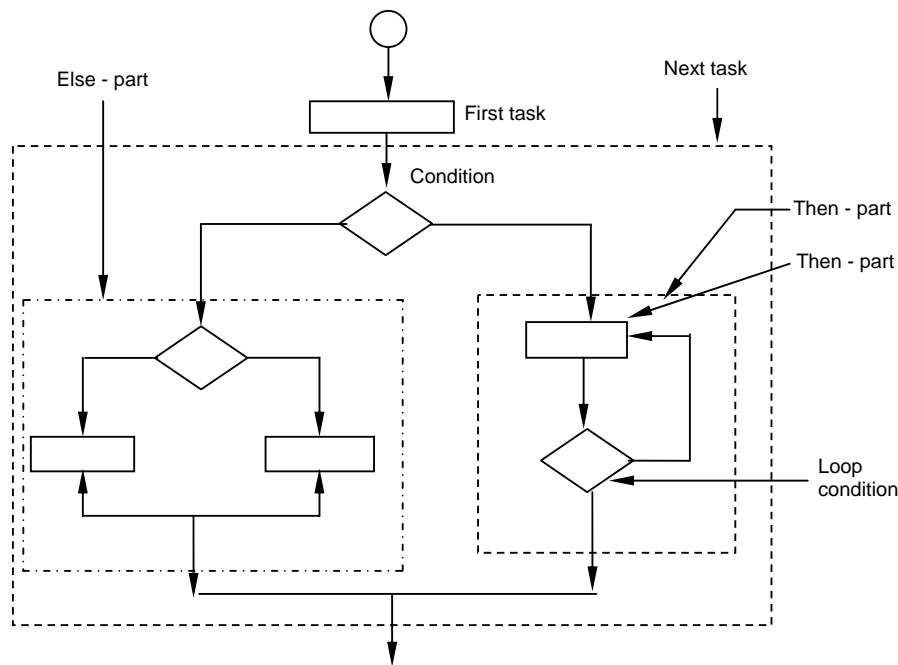


Figure 22.5 Nesting constructs

The graphical representation of structured constructs using the box diagram is illustrated in Figure 22.6. The fundamental element of the diagram is a box. To represent sequence two boxes are connected bottom to top. To represent an if-then-else a condition box is followed by a then-part box and else-part box. Repetition is depicted with a bounding pattern that encloses the process to be repeated. Finally selections represented using the graphical form shown at the bottom right of the figure.

Like flowcharts a box diagram is layered on multiple pages a processing elements of a module are refined. A "call" to a subordinate module can be represented by a box with the module name enclosed by an oval.

In many software applications a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions into a tabular form. The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm. In a comprehensive treatment of this design tool Ned Chapin states [HUR83]

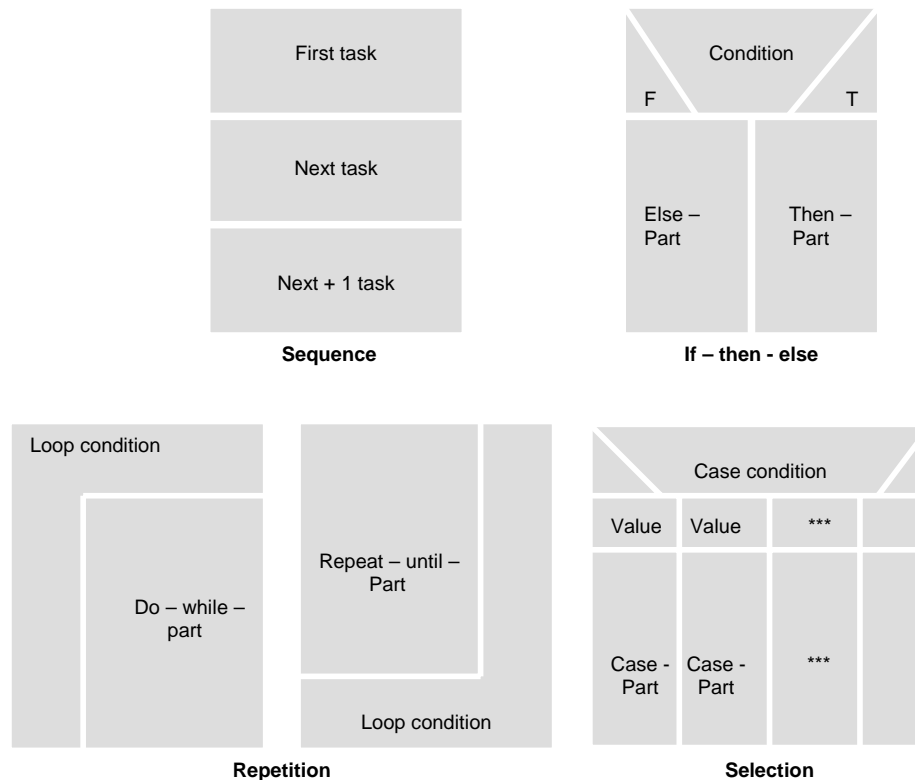


Figure 22.6 Box diagram constructs

Some old software tools and techniques mesh well with new tools and techniques of software engineering. Decision tables are an excellent example. Decision tables preceded software engineering by nearly a decade but fit so well with software engineering that they might have been designed for that purpose.

Decision table organization is illustrated in Figure 22.7. The table is divided into four sections. The upper left hand quadrant contains a list of all conditions. The lower left hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right hand quadrants form a matrix that indicates conditions, combinations and the corresponding actions that will occur for a specific combination. Therefore each column of the matrix may be interpreted as a processing rule.

The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or module).
2. List all conditions (or decisions made) during execution of the procedure.
3. Associate specific sets conditions with specific actions, elimination impossible combinations of conditions; alternatively develop every possible permutation of conditions.
4. Define rules by indicating what action or actions occur for a set of conditions.

To illustrate the use of a decision table consider the following excerpt from a processing narrative for a public utility billing system:

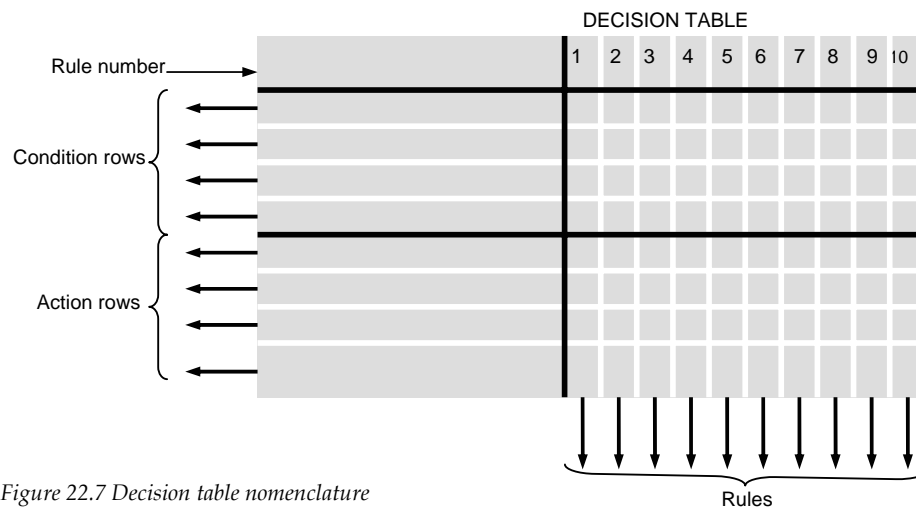


Figure 22.7 Decision table nomenclature

[I]f the customer account is billed using a fixed rate method, a minimum monthly charge is assessed for consumption of less than 100 kWh . Otherwise computer billing applies a Schedule A rate structure. However if the account is billed using a variable rate method, a Schedule A rate structure will apply to consumption below 100 kWh with additional consumption billed according to Schedule B.

Figure 22.8 illustrates a decision table representation of the preceding narrative . Each of the five rules indicates one of five viable conditions (e.g., a “T” (true) in both fixed rate and variable rate account makes no sense in the context of this procedure) As a general rule the decision table can be effectively used to supplement other procedural design notation.

Program Design Language

Program Design Language (PDL) also called structured English or pseudocode, is “a pidgin language in that it uses the vocabulary of one language(i.e., English)and the overall syntax of another (i.e., a structured programming language)” In this chapter PDL is used as a generic reference for a design language.

At first glance PDL looks something like any modern programming language. The difference between PDL and a modern programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements. Because narrative text is embedded directly into a syntactical structure, PDL cannot be compiled . However PDL “processors” currently exist to translate PDL into a graphical representation (e.g., a flowchart) of design and produce nesting maps a design operation index cross reference tables and a variety of other information.

		1	2	3	4	5
Conditions	Fixed rate account	T	T	F	F	F
	Variable rate account	F	F	T	T	E
	Consumption <100 KWH	T	F	T	F	
	Consumption ≥ 100 KWH	F	T	F	T	
Actions	Minimum monthly charge	X				
	Schedule A billing		X	X		
	Schedule B billing				X	
	Other treatment					X

Figure 22.8 Resultant decision table

A program design language may be a simple transposition of a language such as Ada or C. Alternatively it may be a product purchased specifically for procedural design. Regardless of origin a design language should have the following characteristics:

- A fixed syntax of keywords that provide for all structured constructs data declarations and modularity characteristics.
- A free syntax of natural language that describes processing features;
- Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures; and
- Subprogram definition and calling techniques that support various modes of interface description. Today a high order programming language is often used as the basis for a PDL. For example, Ada-PDL is widely used in the Ada community as a design definition tool. Ada language constructs and format are “mixed” with English narrative to form the design language.

A basic PDL syntax should include constructs for subprogram definition interface description and data declaration; and techniques for block structuring condition constructs repetition constructors and I/O constructs. The format and semantics for some of these PDL constructs are presented in the section that follows.

It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling interprocesses, synchronization and many other features. The application design for which PDL is to be used should dictate the final form for the design language.

A PDL Example

To illustrate the use of PDL, we present an example of a procedural design for the SafeHome security system software introduced in earlier chapters. The SafeHome system in question monitors alarms for fire, smoke, burglars, water (flooding), and temperature (e.g. furnace breaks while home owner is away during winter); produces an alarm signal; and calls a monitoring service, generating a voice synthesized message. In the PDL that follows, we illustrate some of the important.

Recall that PDL is not a programming language. The designer can adapt as required without worry of syntax errors. However, the design the monitoring software would have to be reviewed (do you see any problems) and further refined before code could be written. The following PDL defines an elaboration of the procedural design for the security monitor procedure.

22.7 Short Summary

- Interface design encompasses internal and external program interfaces and the design of the user interface. Internal and external interface design are guided by information obtained from the analysis model.
- The user interface design process begins with task analysis and modeling, a design activity that defines user tasks and actions using either a elaborative or object-oriented approach.
- Design issues such as response time, command structure, error handling, and help facilities are considered, and a design model for the system is refined.
- A variety of implementation tools are used to build a prototype for evaluation by the user.
- A set of generic design guidelines govern general interaction information display , and data entry.
- Design notation, coupled with structured programming concepts, enables the designer to represent procedural detail in a manner that facilitates translation to code. Graphical, tabular, and textual notations are available.
- Data structure is developed, program architecture is established, modules are defined, and interfaces are established. This blueprint for implementation forms the basis for all subsequent software engineering work.

22.8 Brain Storm

1. Explain briefly about Architectural Design Optimization ?
2. What is Interface Design ?
3. Discuss about Human – Computer Interface design ?
4. What is Design Evaluation ?
5. Explain briefly about Procedural Design ?
6. Give a Short Note on Program Desing Language ?

☞☞☞

Lecture 23

Discussion

Lecture 24

Design for Real Time - I

Objectives

In this lecture you will learn the following

- ✎ About System Consideration
- ✎ About Real Time System

Coverage Plan

Lecture 24
24.1 Snap Shot
24.2 System Consideration
24.3 Real Time System
24.4 Short Summary
24.5 Brain Storm

24.1 Snap Shot

Like any computer-based system, a real-time system must integrate hardware, software, human and database elements to properly achieve a set of functional and performance requirements. In lecture 12, we examined the allocation task for computer-based systems, indicating that the system engineer must allocate function and performance among the system elements. The problem for real time systems is proper allocation. Real-time performance is often as important as function, yet allocation decisions that relate to performance are often difficult to make with assurance. Can a processing algorithm meet severe timing constraints, or should we build special hardware to do the job? Can an off-the-shelf operating system meet our need for efficient interrupt handling, multi-tasking and communication or should we build custom executive? Can specified hardware coupled with proposed software meet performance criteria? These and many other questions must be answered by the real-time system engineer.

A comprehensive discussion of all elements of real-time systems is beyond the scope of this book. Among a number of good sources of information are [SAV85],[ELL94] and [sel94]. However it is important that we understand each of the elements of a real-time system before discussing software analysis and design issues.

Everett[EVE95] defines three characteristics that differentiate real-time software development from other software engineering efforts:

- The design of a real-time system is resource constrained. The primary resource for a real-time system is time. It is essential to complete a defined task within a given number of CPU cycles. In addition, other system resources such as memory size, may be traded against time to achieve system objectives.
- Real-time systems are compact yet complex. Although a sophisticated real-time system may contain well over a million lines of code, the time-critical portion of the software typically represents a very small percentage of the total. It is this small percentage of code that is the most complex (from an algorithmic point of view)
- Real-time systems often work without the presence of a human user. Therefore, real-time software must detect problems that lead to failure and automatically recover from these problems before damage to data and the controlled environment occurs.

In the section that follows, we examine some of the key attributes that differentiate real-time systems from other types of computer software.

24.2 Real-Time Systems

Real-time systems generate some action in response to external events. To accomplish this function, they perform high-speed data acquisition and control under severe time and reliability constraints. Because these constraints are so stringent, real-time systems are frequently dedicated to a single application.

Real-time systems are used widely for diverse applications that include military command and control systems, consumer electronics, process control, industrial automation, medical and scientific research, computer graphics, local and wide area communication, aerospace systems, computer-aided testing and a vast array of industrial instrumentation.

Integration and Performance Issues

Putting together a real-time system presents the system engineer with difficult hardware and software decisions. (The allocation issues associated with hardware for real-time systems are beyond the scope of this book; see [SAV85] for additional information). Once the software element has been allocated, detailed software requirements are established and a fundamental software design must be developed. Among many real-time design concerns are coordination between the real-time tasks, processing of system interrupts, I/O handling to ensure that no data are lost, specifying the system's internal and external timing constraints and ensuring the accuracy of its database.

Each real-time design concern for software must be applied in the context of system performance. In most cases, the performance of a real-time system is measured as one or more time related characteristics, but other measures such as fault-tolerance may also be used.

Some real-time systems are designed for applications in which only the response time or the data transfer rate is critical. Other real-time applications require optimization of both parameters under peak loading conditions. What's more real-time systems must handle their peak loads while performing a number of simultaneous tasks.

Since the performance of a real-time system is determined primarily by the system response time and its data transfer rate, it is important to understand these two parameters. System response time is the time within which a system must detect an internal or external event and respond with an action. Often event detection and response generation are simple. It is the processing of the information about the event to determine the appropriate response that may involve complex, time-consuming algorithms.

Among the key parameters that affect the response time are context switching and interrupt latency. Context switching involves the time and overhead to switch among tasks, and interrupt latency is the time lag before the switch is actually possible. Other parameter that affects response time are the speed of computation and the speed of access to mass storage.

The data transfer rate indicates how fast serial or parallel data as well as analog or digital data must be moved into or out of the system. Hardware vendors often quote timing and capacity values for performance characteristics. However hardware specifications for performance are usually measured in isolation and are often of little value in determining overall real-time system performance. Therefore I/O device performance, and a host of other factors although important are only part of the story of real-time system design.

Real-time systems are often required to process a continuous stream of incoming data. Design must ensure that data are not missed. In addition, a real time system must respond to events that are asynchronous. Therefore, the arrival sequence and data volume cannot be easily predicated in advance.

The need for reliability however, has spurred an ongoing debate about whether on-line systems, such as airline reservation systems and automatic bank tellers also qualify as real-time. On one hand such on-line systems must respond to external interrupts within prescribed response times on the other of one second. On the other hand nothing catastrophic occurs if an on-line system fails to meet response requirements; instead, only system degradation results.

Interrupt Handling

One characteristic that serves to distinguish real-time systems from any other type is interrupt handling. A real-time system must respond to external stimuli-interrupts-in a time frame dictated by the external world. Because multiple stimuli(interrupts) are often present, priorities and priority interrupts must be established. In other words, the most important task must always be serviced within predefined time constraints regardless of other events.

Interrupts handling entails not only storing information so that the computer can correctly restart the interrupted task, but also avoiding deadlocks and endless loops. The overall approach to interrupt handling is illustrated in figure 24.1. Normal processing flow is “interrupted” by an event that is detected by processor hardware. An event is any occurrence that requires immediate service and may be generated by either hardware or software. The state of the interrupted program is saved (i.e., all register contents, control blocks, etc. are saved) and control is passed to an interrupt service routine that branches to appropriate software for handling the interrupt. Upon completion of interrupt servicing, the state of the machine is restored and normal processing flow continuous.

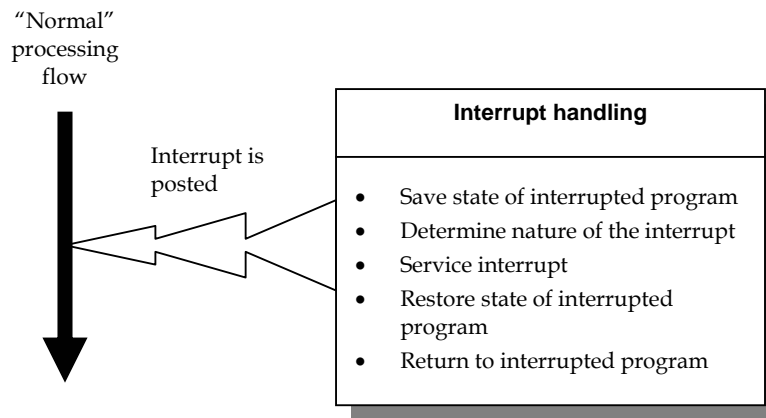


Figure 24.1 Interrupts

In many situations, interrupt servicing for one event may itself be interrupted by another, higher-priority event. Interrupt priority levels (Figure 24.2) may be established. If a lower-priority process is accidentally allowed to interrupt a higher-priority one, it may be difficult to restart the processes in the right order and an endless loop may result.

To handle interrupts and still meet the system time constraints, many real-time operating systems make dynamic calculations to determine whether the system goals can be met. These dynamic calculations are based on the average frequency of occurrence of events, the amount of time it takes to service them(if they can be serviced), and the routines that can interrupt them and temporarily prevent their servicing.

If the dynamic calculations show that it is impossible to handle the events that can occur in the system and still meet the time constraints, the system must decide on a plan of action. One possible approach involves buffering the data so that it can be processed quickly when the system is ready.

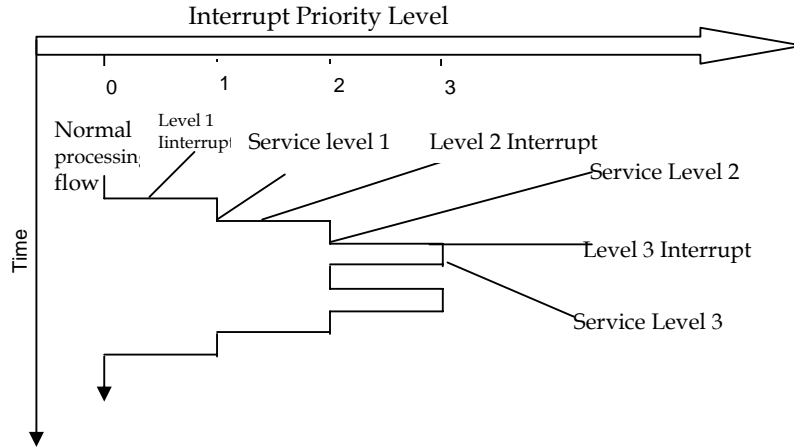


Figure 24.2 An example of interrupt priority levels

Real-Time Data Bases

Like many data processing systems, real-time systems often are coupled with a database management function. However, distributed databases would seem to be a preferred approach in real-time systems because multitasking is commonplace and data are often processed in parallel. If the database is distributed, individual tasks can access their data faster and more reliably and with fewer bottlenecks than with a centralized database. The use of a distributed database for real-time applications divides input/output “traffic” and shortens queues of tasks waiting for access to a database. Moreover, a failure of one database will rarely cause the failure of the entire system, if redundancy is built in.

The performance efficiencies achieved through the use of a distributed database must be weighed against potential problems associated with data partitioning and replication. Although data redundancy improves response time by providing multiple information sources, replication requirements for distributed files also produce logistical and overhead problems, since all the files copies must be updated. In addition, the use of distributed databases introduces the problem of concurrency control involves synchronizing the database so that all copies have the correct, identical information free for access.

The conventional approach to concurrency control is based on what are known as locking and time stamps. At regular intervals, the following tasks are initiated: (1) the database is “locked” so that concurrency control is assured; no I/O is permitted; (2) updating occurs as required; (3) the database is unlocked; (4) files are validated to ensure that all updates have been correctly made; (5) the completed update is acknowledged. All locking tasks are monitored by a master clock (i.e., time stamps). The delays involved in these procedures, as well as the problems of avoiding inconsistent updates and deadlock, militate against the widespread use of distributed databases.

Some techniques, however, have been developed to speed updating and to solve the concurrency problem. One of these, called the exclusive-writer protocol maintains the consistency of replicated files by allowing only a single, exclusive writing task to update a file. It therefore eliminates the high overhead of locking or time stamp procedures.

Real-Time Operating Systems

Some real-time operating systems (RTOS) are applicable to a broad range of system configurations, while others are geared to a particular board or even microprocessor, regardless of the surrounding electronic environment. RTOS achieve their capabilities through a combination of software features and (increasingly) a variety of micro-coded capabilities implemented in hardware.

Today two broad classes of operating systems are used for real-time work; (1) dedicated RTOS designed exclusively for real-time applications and (2) general-purpose operating systems that have been enhanced to provide real-time capability. The use of a real-time executive makes real-time performance- faster and more efficiently than the general-purpose operating system.

All operating systems must have a priority scheduling mechanism, but RTOS must provide a priority mechanism that allows high-priority interrupts to take precedence over less important ones. Moreover, because interrupts occur in response to asynchronous, nonrecurring events, they must be serviced without first taking time to swap in a program from disk storage. Consequently, to guarantee the required response time, a real-time operating system must have a mechanism for memory locking- that is locking at least some programs in main memory so that swapping overhead is avoided.

To determine which kind of real-time operating system best matches an application, measures of RTOS quality can be defined and evaluated. Context switching time and interrupt latency determine interrupt handling capability, the most important aspect of a real-time system. Context switching time is the time the operating system takes to store the state of the computer and the contents of the registers so that it can return to a processing task after servicing the interrupt.

Interrupt latency, the maximum time lag before the system gets around to switching a task, occurs because in an operating system there are often non-reentrant or critical processing paths that must be completed before an interrupt can be processed.

The length of these paths (the number of instructions) before the system can service an interrupt indicates the worst-case time lag. The worst case occurs if a high-priority interrupts is generated immediately after the system enters a critical path between an interrupt and interrupt service. If the time is too long, the system may miss data that are unrecoverable. It is important that the designer know the time lag so that the system can compensate for it.

Many operating systems perform multitasking [WOO90] or concurrent processing, another major requirement for real-time systems. But to be viable for real-time operation, the system overhead must be low in terms of switching time and memory space used.

Real-Time Languages

Because of the special requirements for performance and reliability demanded of real-time systems, the choice of a programming language is important. Many general-purpose programming languages (e.g., C, Fortran, Modula-2) can be used effectively for real-time applications. However, a class of so-called “real-time languages” (e.g., Ada, Jovial, HAL/S, Chill and others) is often used in specialized military and communications applications.

A combination of characteristics makes a real-time language different from a general-purpose language. These include the multitasking capability, constructs to directly implement real-time functions and

modern programming features that help ensure program correctness.

A programming language that directly supports multitasking is important because a real-time system must respond to asynchronous events. Although many RTOS provide multitasking capabilities, embedded real-time software often exists without an operating system. Instead, embedded applications are written in a language that provides sufficient run-time support for real-time program execution. Run-time support requires less memory than an operating system, and it can be tailored to an application, thus increasing performance.

Task Synchronization and Communication

A multitasking system must furnish a mechanism for the tasks to pass information to each other as well as to ensure their synchronization. For these functions, operating systems and languages with run-time support commonly use queuing semaphores, mailboxes or message systems. A semaphore enables concurrent tasks to be synchronized. It supplies synchronization and signaling but contain no information. Messages are similar to semaphores except that they carry the associated information. Mailboxes, on the other hand, do not signal information but instead contain it.

Queuing semaphores are software primitives that help manage traffic. They provide a method of directing several queues – for example, queues of tasks waiting for resources, database access and devices as well as queues of the resources and devices. The semaphore coordinate(synchronize) the waiting tasks with whatever they are waiting for without letting tasks or resources interfere with each other.

In a real-time system, semaphores are commonly used to implement and manage mailboxes. Mailboxes are temporary storage places (also called a message pools or buffers) for message sent from one process to another. One process produces a piece of information, puts it in the mailbox and then signals a consuming process that there is a piece of information in the mailbox for it to use.

Some approaches to real-time operating systems or run-time support systems view mailboxes as the most efficient way to implement communications between processes. Some real-time operating systems furnish a place to send and receive pointers to mailbox data. This eliminates the need to transfer all of the data – thus saving time and overhead.

A third approach to communication and synchronization among processes is a message system. With a message system, one process sends a message to another. The latter is then automatically activated by the run-time support system or operating system to process the message. Such a system incurs overhead because it transfers the actual information, but it provides greater flexibility and ease of use.

24.3 Short Summary

- The design of real time software encompasses all aspects of conventional software design while introducing a new set of design criteria and concerns. Because real time software must respond to real world events in a time frame dictated by those events, all classes of design become more complex.
- It is difficult, and often impractical, to divorce software design from larger system oriented issues. Because real time software is either clock or event driven, the designer must consider function and performance of hardware and software.

- Interrupt processing and data transfer, rate distributed databases and operating systems, specialized programming languages and synchronization methods are just some of the concerns of the real time system.
- The analysis of real time systems encompasses both mathematical modeling and simulation. Queuing and network models enable the system engineer to assess overall response time, processing rate and other timing and sizing issues. Formal analysis tools provide a mechanism for real time system simulation.

24.4 Brain Storm

1. Discuss about Real Time System ?
2. What is Real Time Database ?
3. Short Note on Task Synchronization Communication ?



Lecture 25

Design for Real Time - II

Objectives

In this lecture you will learn the following

- ✎ About Analysis and Simulation of Real Time Systems
- ✎ About Real Time Design

Coverage Plan

Lecture 25
25.1 Snap Shot
25.2 Analysis and simulation of Real Time System
25.3 Real Time Design
25.4 Short summary
25.5 Brain storm

25.1 Snap Shot

In this lecture we are going to learn about Analysis and Simulation of RealTime and Systems and RealTime Design.

25.2 Analysis and Simulation of Real-Time Systems

In the preceding section, we discussed a set of dynamic attributes that cannot be divorced from the functional requirements of a real-time system:

- Interrupt handling and context switching
- Response time
- Data transfer rate and throughput
- Resource allocation and priority handling
- Task synchronization and intertask communication

Each of these performance attributes can be specified, but it is extremely difficult to verify whether system elements will achieve desired response, system resources will be sufficient to satisfy computational requirements or processing algorithms will execute with sufficient speed.

The analysis of real-time systems requires modeling and simulation that enables the system engineer to assess “timing and sizing” issues. Although a number of analysis techniques have been proposed in the literature (e.g., [LIU90], [WIL90] and [ZUC89]), it is fair to state that analytical approaches for the analysis and design of real-time systems are still in their early stages of development.

Mathematical Tools for Real-Time System Analysis

A set of mathematical tools that enable the system engineer to model real-time system elements and assess timing and sizing issues has been proposed by Thomas McCabe [MCC85]. Based loosely on data flow analysis techniques, McCabe’s approach enables the analyst to model both hardware and software elements of a real-time system; represent control in a probabilistic manner; and apply network analysis, queuing and graph theory and a Markovian mathematical model [GRO85] to derive system timing and resource sizing. Unfortunately the mathematics involved is beyond the scope of this book, making a detailed explication of McCabe’s work difficult. However, an overview of the technique will provide a worthwhile view of an analytical approach to the engineering of real-time systems.

McCabe’s real-time analysis technique is predicated on a data flow model of the real-time system. However, rather than using a DFD in the conventional manner, McCabe [MCC85] contends that the transforms of a DFD can be represented as process states of a Markov chain and the data flows themselves represent transitions between the process states. The analyst can assign transitional probabilities to each data flow path. As shown in figure 25.1 a value $0 < p_{ij} \leq 1.0$ may be specified for each flow path, where p_{ij} represents the probability that flow will occur between process i and process j . The processes correspond to information transforms (bubbles) in the DFD.

Each process in the DFD-like model can be given a “unit cost” that represents the estimated (or actual) execution time required to perform its function and an “entrance value” that depicts the number of system interrupts corresponding to the process. The model is then analyzed using a set of mathematical tools that compute (1) the expected number of visits to a process, (2) the time spent in the system when processing begins at a specific process and (3) the total time spent in the system.

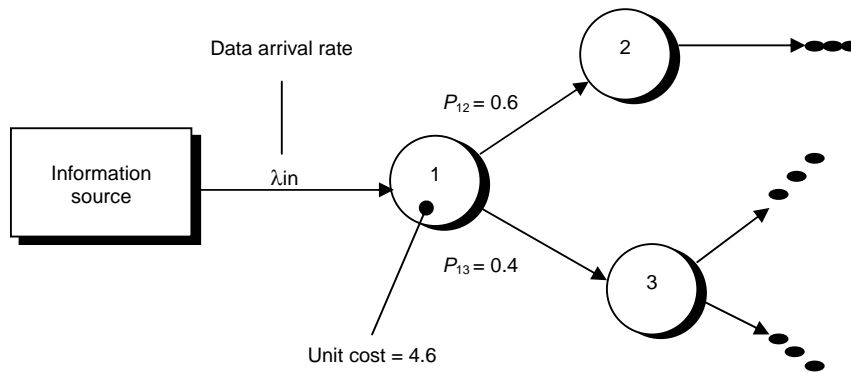


Figure 25.1 DFDs as a queuing network model

To illustrate the McCabe technique on a realistic example, we consider a DFD for an electronic counter measures system shown in figure 25.2. The data flow diagram takes the standard form, but data flow identification has been replaced by p_{ij} . The values λ_i correspond to the arrival rate (arrivals per second) at each process. Depending on the type of queue encountered, the analyst must determine statistical information such as the mean service rate (mean run time per process), variance of service rate, variance of arrival rate, and so forth.

The arrival rates for each process are determined using the flow path probabilities, p_{ij} and the arrival rate into the system, λ_{in} . A set of flow balance equations are derived and solved simultaneously to compute the flow through each process. For the example the following flow balance equations result [MCC85]:

$$\begin{aligned}\lambda_4 &= p_{64}\lambda_6 \\ \lambda_5 &= p_{25}\lambda_2 = \lambda_3 \\ \lambda_6 &= \lambda_5 \\ \lambda_7 &= p_{67}\lambda_6\end{aligned}$$

for the p_{ij} shown and an arrival rate, $\lambda_{in} = 5$ arrivals per second, the above equations can be solved [MCC85] to yield:

$$\begin{aligned}\lambda_1 &= 8.3 \\ \lambda_2 &= 5.8 \\ \lambda_3 &= 5.4 \\ \lambda_5 &= 8.3 \\ \lambda_6 &= 8.3 \\ \lambda_7 &= 5.0\end{aligned}$$

Once the arrival rates have been completed, standard queuing theory can be used to compute system timing. Each subsystem (a queue, Q and a server, S) may be evaluated using formulas that correspond to the queue type. For (m/m/1) queues [KLI75]:

$$\begin{aligned}\text{utilization: } \rho &= \lambda / \mu \\ \text{expected queue length: } N_q &= \rho^2 (1 - \rho) \\ \text{expected number in subsystem: } N_s &= \rho_1 (1 - \rho) \\ \text{expected time in queue } T_q &= \lambda / (\mu(\mu - \lambda)) \\ \text{expected time in subsystem: } T_s &= 1 / (\mu - \lambda)\end{aligned}$$

$$\begin{aligned}\lambda_1 &= \lambda_{in} + \lambda_4 \\ \lambda_2 &= p_{12}\lambda_1 \\ \lambda_3 &= p_{13}\lambda_1 + p_{23}\lambda_2\end{aligned}$$

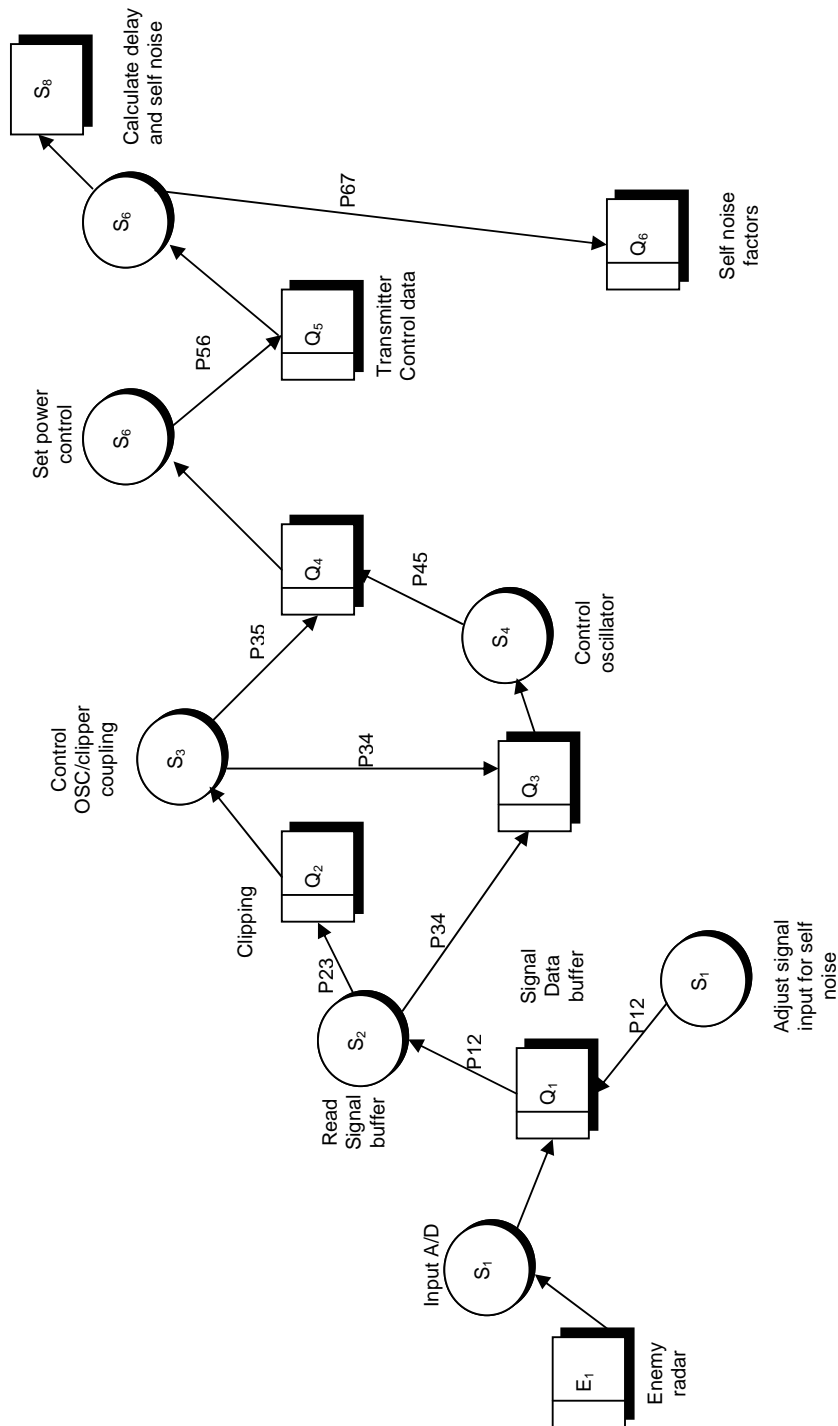


Figure 25.2 Example DFD for real time analysis.

where μ is completion rate (completions/sec). Applying standard queuing network reduction rules the original network derived from the data flow diagram can be simplified by applying the steps shown in figure 25.4. The total time spent in the system is 2.37 seconds.

Obviously, the accuracy of McCabe’s analysis approach is only as good as estimates for flow probability, arrival rate and completion rate. However, significant benefit can be achieved by taking a more analytical view of real-time systems during analysis. To quote McCabe [MCC85]:

By changing such variables as arrival rates, interrupt rates, splitting probabilities, priority structure, queue discipline, configurations, requirements, physical implementation and variances we can easily show the program manager what affect it will have on the system at hand. These iterative methodologies are necessary to fill a void in real-time specification modeling.

Simulation and Modeling Techniques

Mathematical analysis of a real-time system represents one approach that can be used to understand projected performance. However, a growing number of real-time software developers use simulation and modeling tools that not only analyze a system’s performance, but also enable the software engineer to build a prototype, execute it and thereby gain an understanding of a system’s behavior.

The overall rationale behind simulation and modeling for real-time systems is discussed [ILO89] by i-Logix (a company that develops tools for systems engineers):

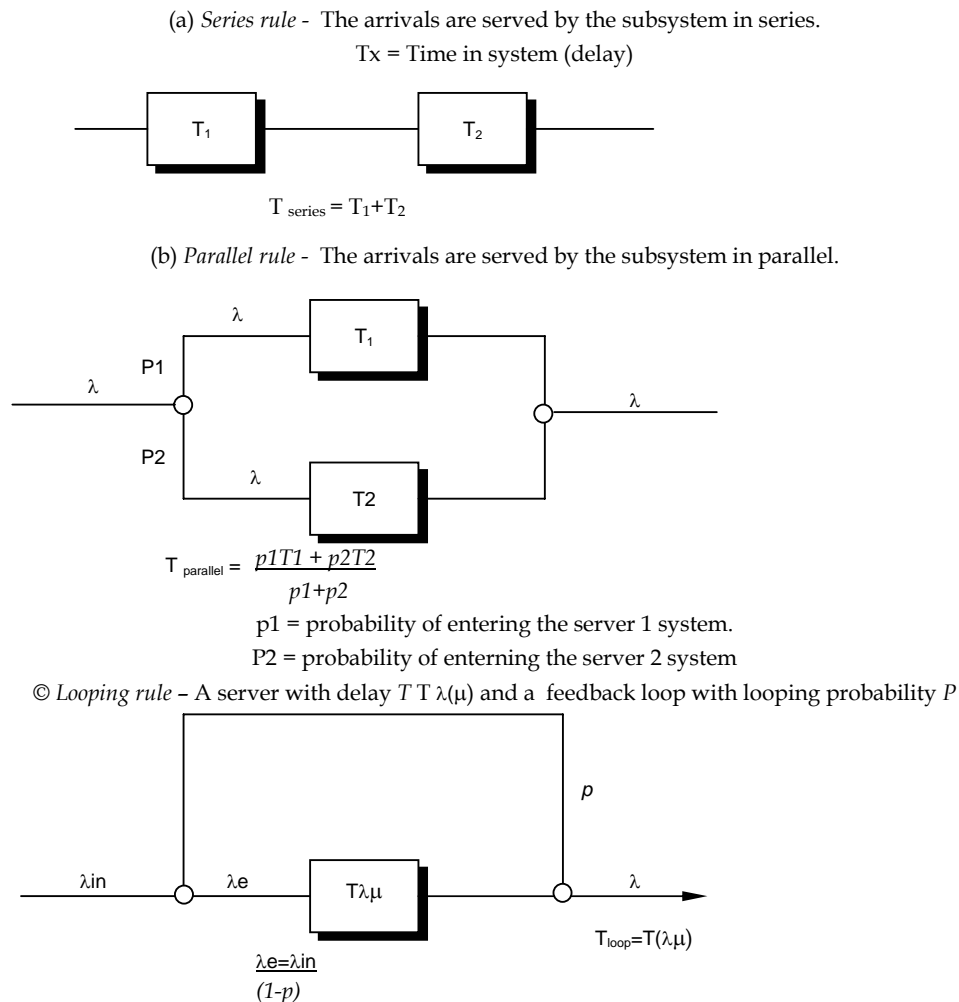
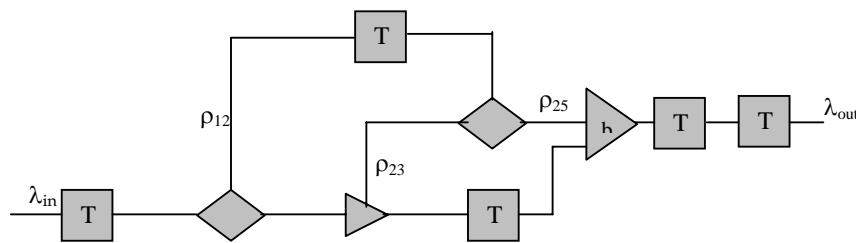


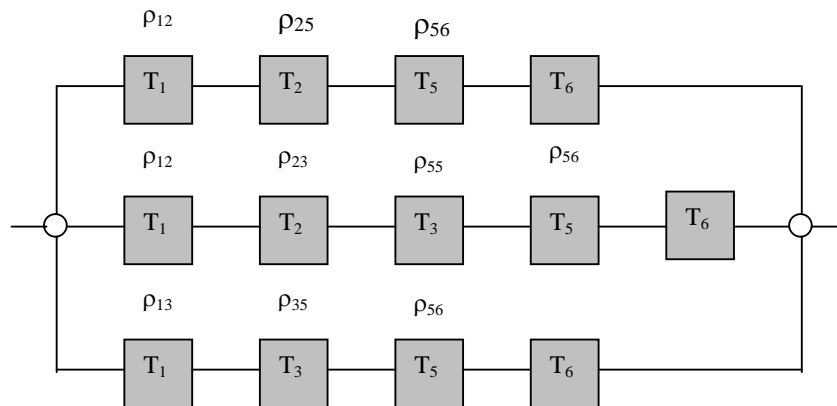
Figure 25.3 Queing network reduction rules

The understanding of a system's behavior in its environment over time is most often addressed in the design, implementation and testing phases of a project, through iterative trial and error. The Statemate [a system engineering tool for simulation and modeling] approach provides an alternative to this costly process. It allows you to build a comprehensive system model that is accurate enough to be relied on and clear enough to be useful. The model addresses the usual functional and flow issues, but also covers the dynamic, behavioral aspects of a system. This model can then be tested with the Statemate analysis and retrieval tools, which provide extensive mechanisms for inspecting and debugging the specification and retrieving information from it. By testing the implementation model, the system engineer can see how the system as specified would behave if implemented.

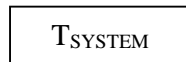
The i-Logix approach [HAR90] makes use of notation that combines three different views of a system: the activity-chart, the module-chart and the state-chart. In the paragraphs that follow the i-Logix approach to real-time system simulation and modeling is described.



Step 1. Further abstracted queuing network



Step 2. Equivalent queuing network showing all possible paths through the network



Step 3. Final reduction

Figure 25.4 Simplifying the queuing network

The Conceptual View

Functional issues are treated using activities that represent the processing capabilities of the system. Dealing with a customer's confirmation request in an airline reservation system is an example of an activity, as is updating the aircraft's position in an avionics system. Activities can be nested, forming a hierarchy that constitutes a functional decomposition of the system. Items of information such as the distance to a target or a customer's name, will typically flow between activities and might also be kept in data stores. This functional view of a system is captured with activity-charts which are similar to conventional data flow diagrams.

Dynamic behavioral issues, commonly referred to as control aspects, are treated using statecharts, a notation developed by Harel and his colleagues [HAR88], [HAR92]. Here states can be nested and linked in a number of ways to represent sequential or concurrent behavior. An avionics mission computer, for example, could be in one of three states: air-to-air, air-to-ground, or navigation. At the same time it must be in the state of either automatic or manual flight control. Transitions between states are typically triggered by events, which may be qualified by conditions. Flipping a certain switch on the throttle, for example, is an event that will cause a transition from the navigate state to the air-to ground state, but only on condition that the aircraft has air-to-ground ammunition available. As a simple example consider the digital watch shown in figure 25.5. The Statechart for the watch is shown in figure 25.6.

These two views of a system are integrated in the following way. Associated with each level of an activity-chart, there will usually be a statechart, called a control activity, whose role is to control the activities and data flows of that level [this is similar in some ways to the relationship between flow models and CSPEC described in chapter 12]. A statechart is able to exercise control over the activities. For example it can instruct activities to start and stop and to suspend and resume their work. It is able to change the values of variables and thus to influence the processing carried out by the activities. It is also able to send signals to other activities and thus cause them to change their own behavior. In addition to being able to generate actions, a controlling statechart is able to sense such actions being carried out by other statecharts. For example, if one statechart starts an activity or increments the value of variable another can sense that event and use it, say to trigger a transition.

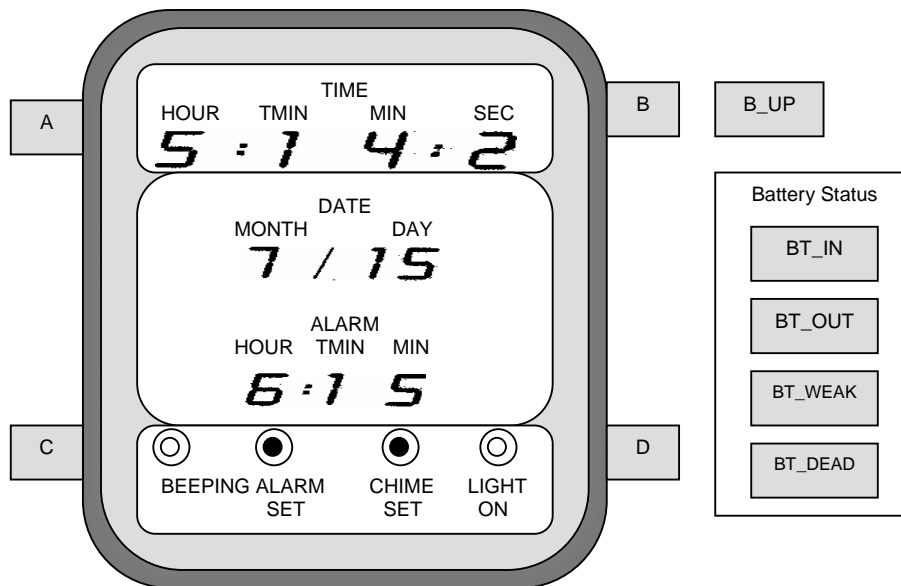


Figure 25.5 Digital watch prototype (courtesy I-Logix)

It is important to realize that activity-charts and statecharts are strongly linked, but they are not different representations of the same thing. Activity-charts on their own are incomplete as a model of the system, since they do not address behavior. Statecharts are also incomplete, since without activities they have nothing to control. Together, a detailed activity-chart and its controlling statecharts provide the conceptual model. The activity-chart is the backbone of the model; its decomposition of the capabilities of the system is the dominant hierarchy of the specification, and its controlling statecharts are the driving force behind the system's behavior.

The Physical View

A specification that uses activity-charts and statecharts in the form of a conceptual model is an excellent foundation, but it is not a real system. What is missing is a means for describing the system from a physical(implementation) perspective and a means to be sure that the system is implemented in a way that is true to that specification. An important part of this is describing the physical decomposition of the system and its relationship to the conceptual model.

The physical aspects are treated in Statemate using the language of module-charts. The terms “physical” and “module” are used generically to denote components of a system, whether hardware, software or hybrid. Like activities in an activity-chart, modules are arranged in a hierarchy to show the decomposition of a system into its components and subcomponents. Modules are connected by flow lines, which one can think of as being the carriers of information between modules.

Analysis and Simulation

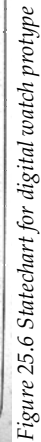
Once we have constructed a conceptual model, consisting of an activity-chart and its controlling statecharts, it can be thoroughly analyzed and tested. The model might describe the entire system, down to the lowest level of detail, or it might be only a partial specification.

We must first be sure that the model is syntactically correct. This gives rise to many relatively straightforward tests: for example, that the various charts are not blatantly incomplete e.g., missing labels or names, dangling arrows); that the definitions of nongraphical elements, such as events and conditions, employ legal operations only, and so on. Syntax checking also involves more subtle tests, such as the correctness of inputs and outputs. A example of this is a test for elements that are used in the statechart but are neither input nor affected internally such as a power-on event that is meant to cause a transition in the statechart but is not defined in the activity-chart as an input. All of these are usually referred to as consistency and completeness tests, and most of them are analogous to the checking carried out by a compiler prior to the actual compilation of a programming language.

Running Scenarios

A syntactically correct model accurately describes some system. However, it might not be the system we had in mind. In fact, the system described might be seriously flawed – syntactic correctness does not guarantee correctness of function or behavior. The real objective of analyzing the model is to find out whether it truly describes the system that we want. The analysis should enable us to learn more about the model that has been constructed, to examine how a system based on it would behave, and to verify that it indeed meets expectations. This requires a modeling language with more than a formal syntax. It requires that the system used to create the model recognize formal semantics as well.

If the model is based on a formal semantics, the system engineer can execute the model. The engineer can create and run a scenario that allows him to “press buttons” and observe the behavior of the model before the system is actually build. For example to exercise a model of an automated teller machine (ATM) the following steps occur: (1) a conceptual model is created; (2) the engineer plays the role of the customer and the bank computer, generating events such as insertion of a bank card, buttons being pressed and new balance information arriving; (3) the reaction of the system to these events is monitored and (4) inconsistencies in behavior are noted ; (5) the conceptual model is modified to reflect proper behavior and (6) iteration occurs until the system that is desired evolves.



Programming Simulations

One of the simplest things that can be done with SCL is to read lists of events from a batch file. This means that lengthy scenarios or parts of them can be prepared in advance and executed automatically. These can be observed by the system engineer. Alternatively, the system engineer can program with SCL to set break points and to monitor certain variables, states or conditions. For example, running a simulation of an avionics system, the engineer might ask the SCL program to stop whenever the radar locks on target and switch to interactive mode. Once “lock on” is recognized, the engineer takes over interactively so that this state can be examined in more detail.

The use of scenarios and simulations also enables the engineer to gather meaningful statistics about the operation of the system that is to be built. For example we might want to know how many times, in a typical flight of the aircraft, the radar loses a locked-on target. Since it might be difficult for the engineer to put together a single, all-encompassing flight scenario, a programmed simulation can be developed using accumulated results from other scenarios to obtain average- case statistics. A simulation control program generates random (say, seat ejection in a fighter aircraft) can be assigned very low probabilities while others are assigned higher probabilities, and the random selection of events thus becomes realistic. In order to be able to gather the desired statistics, we insert appropriate break points in the SCL program.

Automatic Translation into Code

Once the system model has been built, it can be translated in its entirety into executable code using a prototyping function. Activity-charts and their controlling statecharts can be translated into a high-level programming language, such as Ada or C. Today, the primary use of the resulting code is to observe a system perform under circumstances that are as close to the real world as possible. For example the prototype code can be executed in a full-fledged simulator of the target environment or in the final environment itself. The code produced by such CASE tools should be considered to be “prototypical”. It is not production or final code. Consequently it might not always reflect accurate real-time performance of the intended system. Nevertheless, it is useful for testing the system’s performance in close to real circumstances.

25.3 Real-Time Design

The design of real-time software must incorporate all of the fundamental concepts and principles associated with high-quality software. In addition, real-time software poses a set of unique problems for the designer.

- Representation of interrupts and context switching
- Concurrency as manifested by multitasking and multiprocessing
- Intertask communication and synchronization
- Wide variations in data and communication rates
- Representation of timing constraints
- Asynchronous processing
- Necessary and unavoidable coupling with operating systems, hardware and other external system elements

It is worthwhile to address a set of specialized design principles that are particularly relevant during the design of real-time systems. Kurki-Suono [KUR93] discuss the design model for real-time software;

All reasoning, whether formal or intuitive, is performed with some abstraction. Therefore it is important to understand which kinds of properties are expressible in the abstraction in question. In connection with

reactive systems, this is emphasized by the more stringent need for formal methods, and by the fact that no general consensus has been reached about the models that should be used. Rigorous formalisms for reactive systems range from process algebras and temporal logics to concrete state-based models and Petri nets, and different schools keep arguing about their relative merits.

He then defines a number of modeling principles that should be considered in the design of real-time software [KUR93]:

Explicit atomicity. It is necessary to define “atomic actions” explicitly as part of the real-time design model. An atomic action or event is a well-constrained and limited function that can be executed by a single task or executed concurrently by several tasks. An atomic action is invoked only by those tasks (“participants”) that require it, and the results of its execution affect only those participants; no other parts of the system are affected.

Interleaving Although processing can be concurrent, the history of some computation should be characterized in a way that can be obtained by a linear sequence of actions. Starting with an initial state, a first action is enabled and executed. As a result of this action the state is modified and a second action occurs. Because several actions can occur in any given state, different results(histories) can be spawned from the same initial state. “This non determinism is essential in interleaved modeling of concurrency” [KUR93].

Nonterminating histories and fairness. The processing history of a reactive system is assumed to be infinite. By this we mean that processing continues indefinitely or “stutters” until some event causes it to continue processing. Fairness requirements prevent a system from stopping at some arbitrary point.

Closed system principle. A design model of a real time system should encompass the software and the environment in which the software resides. “Actions can therefore be partitioned into those for which the system itself is responsible, and to those that are assumed to be executed by the environment”

Structuring of state. A real-time system can be modeled as a set of objects, each of which has a state of its own.

The software engineer should consider each of the concepts noted above as the design of real time system evolves.

Over the past two decades, a number of real time software design methods have been proposed to grapple with some or all of the problems noted above. Some approaches to real time design extend the design methods discussed in chapter 14 and 21 (e.g. data flow [WAR85], [HAT87] data structure [JAC83]; or object-oriented [LEL90] methods. Others introduce an entirely separate approach using finite state machine models or message passing systems. Petrinets, or a specialized language as a basis for design. A comprehensive discussion of software design for real time systems is beyond the scope of this book. For further details, the reader should refer to [LEV90],[SHU92],[SEL 94] and [GOM 95].

25.4 Short Summary

- Software design for real time systems can be predicated on a conventional design methodology that extends data flow oriented or object oriented design by providing a notation and approach that addresses real-time system characteristics. Alternatively, design methods that make use of unique notation or specialized languages can also be applied.

- Software design for real time systems remains a challenge. Progress has been made; methods do exist, but a realistic assessment of the state of the art suggests much remains to be done.

25.5 Brain Storm

1. Give a brief note on Analysis and Simulation of Real - Time systems ?
2. Explain briefly about Real - Time Design ?

☞ Best of Luck ☞

MS3.1 Software Engineering - Concepts and Implementation

Syllabus

Lecture 1

Software characteristics – Software Components – Software Applications.

Lecture 2

Software Engineering – A layered technology – Software process – Software process models – the linear sequential model – Evolutionary software process models – the incremental model.

Lecture 3

Project management concepts – the management spectrum – people ,problem, process.

Lecture 4

Software project planning – project planning objectives – software scope - resources - software project estimations.

Lecture 5

Decomposition techniques – empirical estimation models – the make buy decision – automated estimation tools.

Lecture 6

Risk management – reactive Vs proactive risk strategies – software risks - risk identification – risk projection – risk mitigation and management – safety risks and hazards.

Lecture 7

Project scheduling and tracking - Basic concepts- the relationship between people and effort – defining a task set for the software project.

Lecture 8

Selecting software engineering tasks – refinement of major tasks – defining a task network – scheduling – the project plan.

Lecture 9

Quality concepts - the quality movement – software quality assurance – software reviews

Lecture 10

Formal technical reviews – formal approaches to SQA - the SQA plan – the ISO 9000 quality standards.

Lecture 11

Software configuration management – the SCM process – identification of objects in the software configuration – version control – change control – configuration audit – status reporting – SCM standards.

Lecture 12

System engineering - Computer based systems – the system engineering hierarchy - information engineering.

Lecture 13

Information strategy planning – business area analysis.

Lecture 14

Product engineering – modeling the system architecture – system modeling and simulation – system specification.

Lecture 15

Analysis concepts and principles - Requirements analysis – communication techniques – analysis principles.

Lecture 16

Software prototyping – specification – specification review.

Lecture 17

Analysis modeling - The elements of the analysis model – data modeling – functional modeling and information flow.

Lecture 18

Behavioral modeling – the mechanics of structured analysis

Lecture 19

The data dictionary - an overview of other classical analysis methods.

Lecture 20

Design concepts and principles - Software design and software engineering – the design process – design principles – design concepts.

Lecture 21

Effective modular design – design heuristics for effective modularity – the design model – design documentation.

Lecture 22,23

Design methods - Data design – architectural design – the architectural design process – transform mapping – transaction mapping – design postprocessing.

Lecture 24,25

Architectural design optimization – interface design – human computer interface design – interface design guidelines – procedural design.

Lecture 26

Design for real time systems - System considerations – real time systems .

Lecture 27

Analysis and simulation of real time systems – real time design.

Lecture 28

Software testing methods - Software testing fundamentals – test case design – white box testing – basis path testing –

Lecture 29

Control structure testing - testing for specialized environments.