# Chapter 9. A Comparison of The Object Oriented Approach and The Structured Approach

## Table of Contents



# Objectives

At the end of this chapter you should be able to:

- Describe the main techniques used for modelling dynamic behaviour of objects
- Explain, with examples, the main differences between the structured approach and the object-oriented approach
- Explain, with examples, the main similarities between the structured approach and the object-oriented approach

# Introduction

In previous units you have seen how object-oriented design can be initiated by means of Use-Case Diagrams and how the attributes and operations of objects can be identified in class definitions, and class diagrams were introduced. Additional notations exist which allow designers to document sequences of events that may occur while the application is running (i.e. dynamically); how individual objects must behave in response to these events and how they collaborate together. Once all these topics have been tackled, the traditional structured approach using the well-established methodology SSADM will be compared with the object-oriented approach using UML to consolidate the knowledge from the previous learning units.

### Read

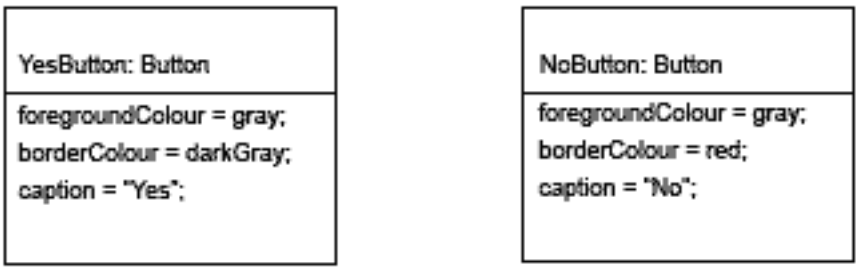
Read up about Object-oriented Systems Analysis and design using UML in your textbook and online resources.

# Characteristics of the Object-oriented Paradigm

## Encapsulation of data and functions

The object-oriented technology is a consequence of the constant gradual development of programming languages, making it possible to implement better encapsulated modules/types, and overcome the artificial separation of data from functions that perform operations on the data which was imposed by the implementation limitations of languages in the past. The encapsulation of data also called attributes and operations makes it possible to view an object in a less artificial way more suitable for expressing event driven applications. The emphasis is on the object behaviour - what it can do, how it can be used Structured programming based on main program control and subroutines / modules did not offer a suitable way for designing event driven applications. If we have another look at the Button example :


The class diagrams above define the name, attributes and operations of a class of object. An object is an instance of the class. An object's attributes have some specific value also called the state. For example: instances of Button class YesButton and NoButton. The instances are shown as boxes where the top part has the object name: class name, the bottom part shows the object state i.e. the attributes and their values.




A Button can be pushed, and we can set the foreground and border colours. An object can be used only

through its interfaces - the operations that can be performed on it. The interfaces available for external use are public. Everything else remains hidden, and thus safe, which means cannot be accidentally corrupted or destroyed.

The advantages this offers are manifold. One is that the object can be used as a component, a black box. The user of the object does not need to know anything about the internal definitions of the object, but only its interfaces: operation names and required arguments. Executing a method is called sending a message to the object. To send a message to an object we need to use the object name and method name with the argument(s) it may require. In the Button example for the

```
setForegroundColour
```

we need to supply an argument e.g.blue. The operation push needs no arguments.

The following is a message to push the recipient OkButton, and change the border colour to blue.

```
OkButton.push();
```

- recipient: OkButton
- operation: push
- arguments: none

```
OkButton.setBorderColour(blue);
```

- recipient: OkButton
- operation: setBorderColour
- arguments: blue

## Review Question 1

Define object, class and instance.

Answer to this question can be found at the end of this chapter.

## Review Question 2

How does the object-oriented concept of message passing help to encapsulate the implementation of an object, including its data?

Answer to this question can be found at the end of this chapter.

## Review Question 3

What is the difference between generalisation and specialisation?

Answer to this question can be found at the end of this chapter.

## Review Question 4

What rules describe the relationship between a subclass and its superclass?

Answer to this question can be found at the end of this chapter.

### Review Question 5

Why is encapsulation important to creating reusable components?

Answer to this question can be found at the end of this chapter.

# Relationships among classes and objects

When a system is being developed using the object-oriented paradigm, the description of the functionality of the system is expressed using Use Case modelling as explained in a previous unit, after which the problem the application is expected to solve is expressed in terms of class diagrams. The class diagram shows the main entities of the problem as classes and the relationships among them. Different designers are likely to produce different class diagrams for any given application. .However, each class diagram will represent a workable solution to the problem, and the differences between diagrams represent different design decisions. It is important therefore to understand what different relationships may exist between classes appearing in the class diagram.

## Associations

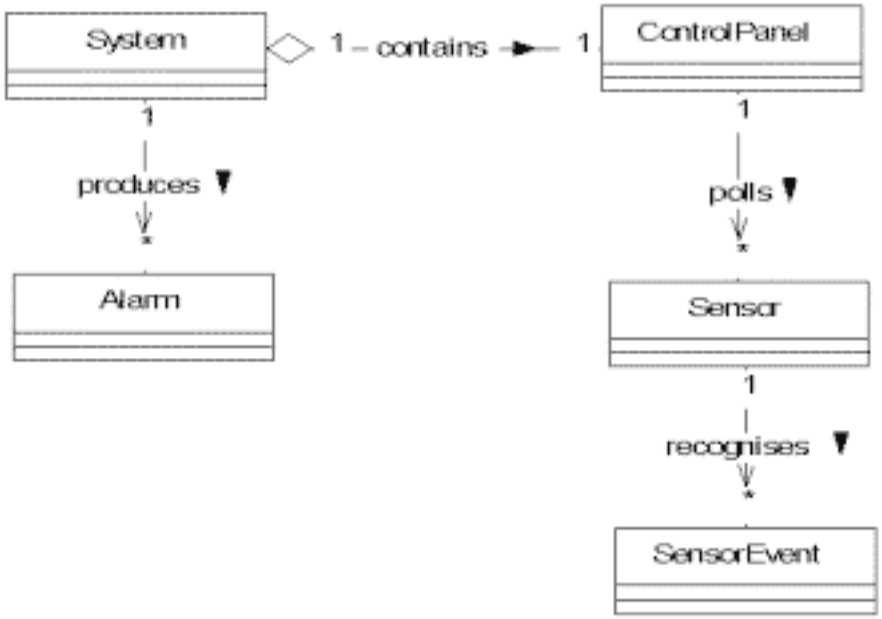
The first type of relationship that may exist between two classes in the class diagram is the association. An association between two classes indicates that their respective instance objects need to co-operate to achieve some desired functionality. Booch describes an association as a ‘uses’ relationship. Generally objects of different classes send each other messages. In order to be able to do this we have to provide links between objects, which allow them to communicate. Associations can be :

- one to one: one object of a class has a link to one other object of another class
- one to many: one object of a class has links with many objects of a particular class
- many to many: many objects of one class have links with many objects of a particular class

Associations may also occur between different objects of the same class. Associations should have a text label to describe the relationship and a small arrowhead to indicate the direction of the relationship. Relationships may also be bi-directional.

Consider for example the design of a Home Security system The home security system monitors a number of sensors connected to it. The system has a control panel with a keypad, a number of function keys through which the user interacts with the system, and a LCD display. The system is password protected. When a sensor event occurs, the system invokes the attached alarm. The system can be configured to dial the telephone number of a monitoring service. (See Use Case )

The following is an example of a class diagram for the home Security system. This class diagram has three associations: In each case the relationship is one to zero or more denoted with 1 and * , and one aggregation.




# Aggregations

An aggregation is a different type of relationship between classes. Aggregations can take several forms - the most obvious is when there are clear parts/whole has a, has or contains relationships. The Home Security system example shown above has one aggregation: the System contains a ControlPanel.

## Containers and Containment

Containers are objects that are able to contain other objects. A container exists independently of whether or not it contains any objects at a particular time. An aggregation expresses the relationship between a container and its contents. A composition or containment is a special case of an aggregation. A composition hierarchy defines how an object is composed of other objects. For example a car has an engine compartment with an engine and a boot. The engine is an essential part of a car since a car (it is a component) since a car cannot behave like a car without one. In contrast the integrity of a car object is not affected by what is in its boot which is a container and exists independently of its components.

Properties of aggregations that distinguish them from associations :

- Transitivity: If A is part of B, and B is part of C then A is part of C.
- Antisymmetry: if A is part of B, then B is not part of A - more unequal then associations
- Propagation: The environment of the parts is the same as that of the assembly - for example a car and its engine and boot. With associations objects are independent of each other and this does not apply.

Aggregations can be fixed, variable or recursive

- Fixed: the number and type of components is fixed

- Variable: the number of levels of aggregation is fixed, but the number of parts may vary
- Recursive: Object contains components of its own type like a Russian doll each one contains a smaller doll.

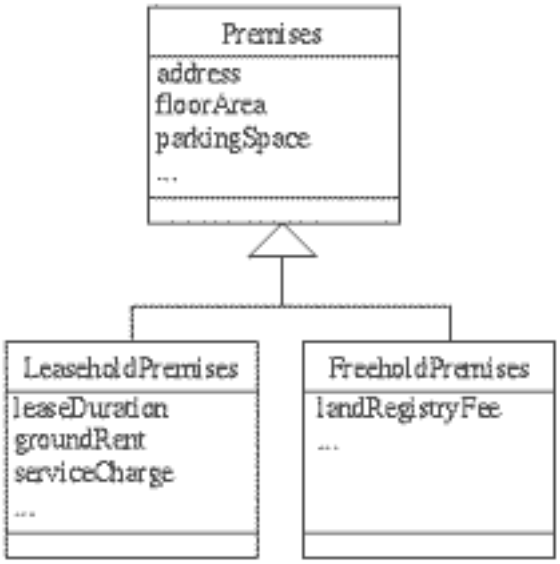
# Inheritance

## Generalization/specialization

The final kind of relationship is inheritance. The process of discovering shared features among a group of related classes is known as generalisation and the opposite specialisation. The object-oriented paradigm supports generalisation/specialisation among classes through the mechanism of inheritance. Conceptually, inheritance models the is-a(n) relationship

- Generalization: Merging two or more classes into one according to their similarities
- Specialization: Splitting a class into two or more according to their differences

The Premises example from a previous unit

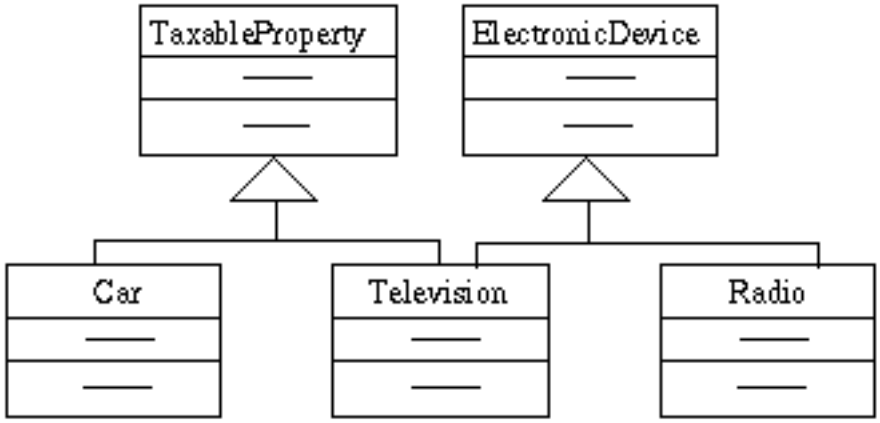



Inheritance is one of the most significant features of object-orientation. Classes can be organised into inheritance hierarchies with an ancestor at its root, and the ultimate descendants at the leaves. The inheritance rule says: A class only need describe what makes it different from its ancestors Otherwise a class inherits common parts of its description from its ancestors.

Inheritance hierarchies may come in the form of

- Single: where the hierarchy is a tree, where every class has at most one immediate parent

- Multiple: where the hierarchy is a graph, where every class may have one or more immediate parent classes




In the example from Activity 7 of the previous Unit, shown on the above diagram Television inherits from TaxableProperty and ElecronicDevice. Languages such as Java and Smalltalk support only single inheritance, while C++ supports multiple inheritance.

When defining a new class, you need only specify how it is different from its ancestors Decide on the feature of the new class Find an existing class with some of these Relate the two classes by inheritance Specify only the new, different features

It is best to introduce a new feature at the most general level, as high in the hierarchy as possible, so that as many classes may inherit it. This will reduce duplication, and increase functionality.

## Inheritance of State

In practice considering a class's state i.e. its data, a child class inherits the attributes of its parent class and may add some of its own. A class gradually accumulates named storage cells from all its ancestors. The attributes from the ancestor classes i.e. the shared attributes have still only one copy. Shared attributes -to prevent child classes from updating the shared attributes languages such as Java enables them to be specified as private, or to make them constants that are initialised only once declares them as static.

For example the Java class Math has attributes E, and PI declared as final and static to prevent any possibility of having them ever changed or redefined.

There are two viewpoints either shared data represents data that is permanently true, or what is currently true for a given sub-tree of classes.

## Inheritance of Behaviour

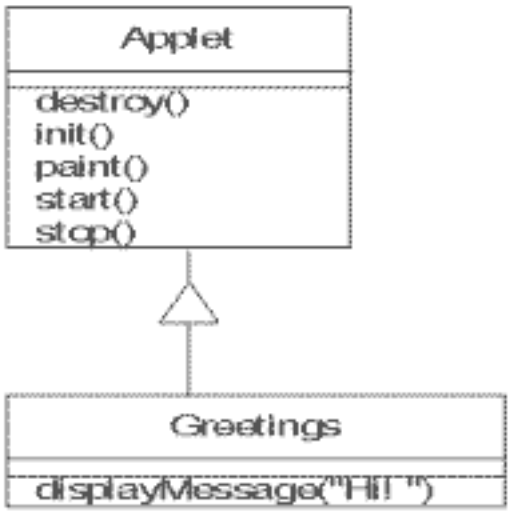
A child class is a subtype of its parent class and can do all the things its ancestors could do, and possibly more. Objects can use operations for any super type (any more general type) as well as their own. Operations can be safely applied to the sub-types because they only access the part of the subtype instance that was defined in the super type..

The ability to share code (behaviours) is a very powerful abstraction mechanism it allows us to charac-

terise very general types by a few functions that are common to many types. This ability to share code encourages you to factor out common behaviour and promotes re-use by inheritance giving maximum economy, maximum functionality..

For example an applet in Java is always derived from the Applet class which means that it is guaranteed to inherit all the functionality of an Applet. An applet can be loaded and instantiated in a browser or appletviewer, initialised, init() start running, start()repaint itself, paint()stop, stop() and destroy itself , destroy()..

Applet is the generalisation of all the behavour common to all applets.




For each applet we create some additional attributes/operations. The Greetings applet has an additional operation displayMessage.

## Inheritance and Encapsulation

Inheritance is also a sharing of name spaces. Typically a child class can see all the information as its ancestors. This mechanism is in addition to the usual notion of scope–visibility. Descendant classes may refer directly to attributes and methods of their ancestors without having to use their public interfaces to do so.

Java and C++ provide three levels of visibility:

- private: names visible only to the class itself
- protected: visible to a class and its descendants
- public: names visible to any class anywhere

Does inheritance break encapsulation? This depends on whether a class is viewed more like an extens-

ible type or a closed module. From the module point view a child class is a separate module from its parent(s) and it breaks the abstraction barrier around the parent when it uses any inherited features. From the type point of view a child class is a true extension of its parent(s) and can be seen as if all inherited features have been defined locally anyway. The only serious case to consider is when shared data can be modified.

Using the Premises example, a LeaseholdPremises object would be able to use attributes: address, floorArea, parkingSpace, unless these were specified as private, in addition to: leaseDuration, groundRent, serviceCharge

To prevent this in Java one can define a class, as finalwhich will prevent the derivation of any new classes from it.

To prevent a change to the meaning and use of some fundamental classes from the jdk possible classes are specified as final. Examples are classes Color and String.

## The open/closed principle

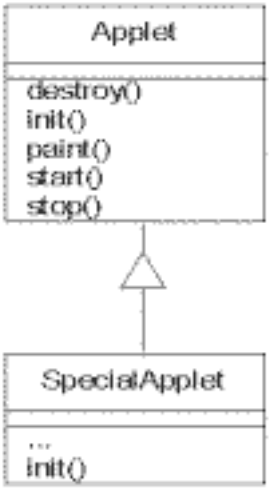
Classes provide modules that are both opened and closed. With inheritance both the original and the derived class can be used independently as two separate entities. This provides a very important facility for software engineering - a class can be extended later through inheritance, without invalidating applications that already use the class.

## Inheritance and Overriding

As well as adding new features to classes, you may modify inherited features - known as overriding or redefining. This may be done for a number of reasons:

- To restrict the type of attribute of operation
- To replace an operation by a more efficient one, specific to the subclass.

For example we may use specific initialisation for our applet. By rewriting the operation init() in the derived class the new overridden version will be the one used for our applet.



In Java to prevent overriding an attribute or operation can be also be specified as final.

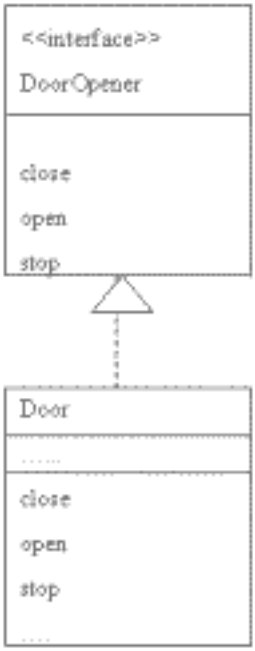
## Deferred Classes

This facility enables the specification of common behaviour, and can be achieved in a number of ways. It involves the declaration of methods in the superclass that will be supplied subsequently in the sub-classes.

Abstract classes are one way of defining the generic characteristics. They are called abstract because such classes cannot have objects, the classes are not complete, and some extra coding will be necessary in those classes, which inherit from the abstract class. Examples of such classes in Java are:

```
Component, Container, Graphics
```

Class Door implements interface DoorOpener, which means that it will be able to execute methods open, close, stop and the code for the methods will be specified in class Door.




Implementing an interface in Java means agreeing to define the methods that the interface requires. All classes that implement the same interface will conform to the same behaviour and in that sense be of the same type without belonging to the same class. Interfaces are simply pure abstract classes.

Interfaces is a simpler way of specifying common behaviour because there is no inherited code, the code has to be specified in the class that implements it. In Java only single inheritance can be used, but a class can have multiple interfaces.

## Note

Some examples of inheritance, abstract classes and interfaces were from the Java API. This is because of

the assumption that you may be familiar with them already.

### Review Question 6

What does object-orientation offers that helps to create reusable components?

Answer to this question can be found at the end of this chapter.

### Review Question 7

Distinguish composition from aggregation.

Answer to this question can be found at the end of this chapter.

### Review Question 8

Why are operations sometimes redefined in a subclass?

Answer to this question can be found at the end of this chapter.

### Review Question 9

What is an abstract class?

Answer to this question can be found at the end of this chapter.

### Review Question 10

Why is generalisation important for creating reusable components?

Answer to this question can be found at the end of this chapter.

### Review Question 11

When should you not use generalisation in a model?

Answer to this question can be found at the end of this chapter.

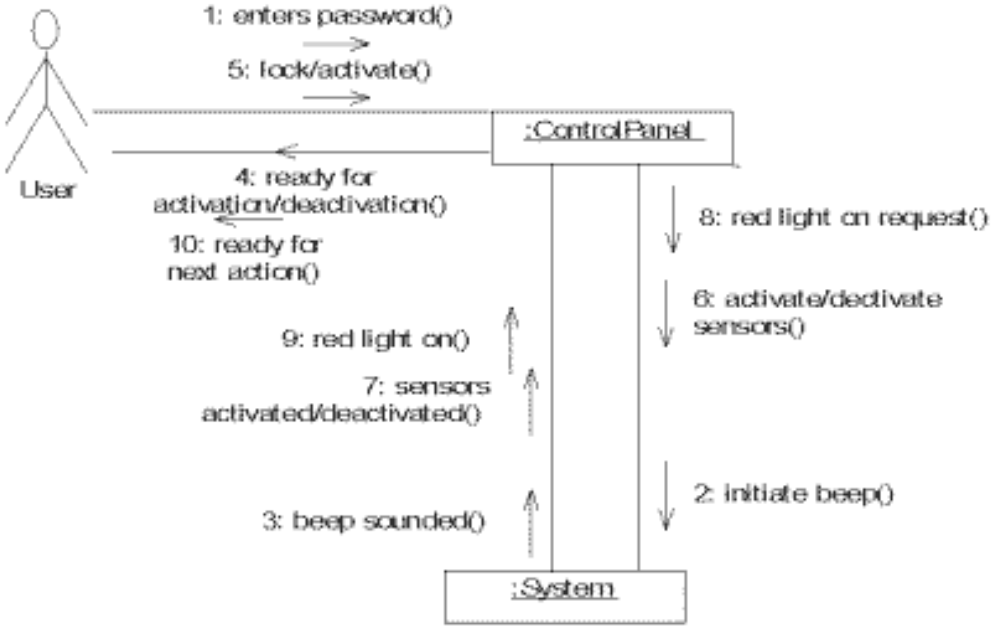
# Dynamic Modelling Using objects

Class diagrams describe the static relationship among classes. The dynamic behaviour showing the interactions among objects and sequencing of events and actions related to an object is modelled in UML using sequence or collaboration diagrams and statecharts.

## Sequence and Collaboration Diagrams

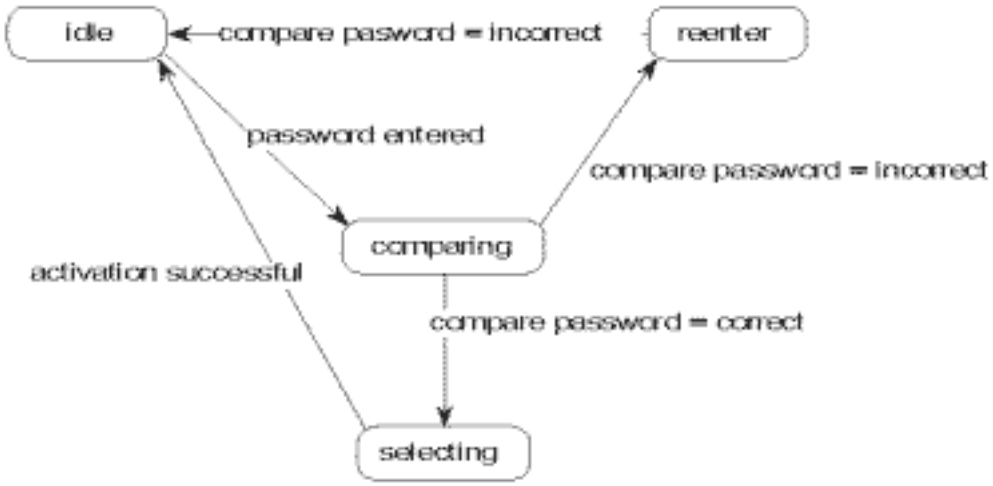
Sequence diagrams document object interaction by highlighting the time ordering of method invocations. Objects that participate in the collaboration are represented as columns. The object that initiates the interaction is usually placed in the leftmost column with the other objects shown on its right. A vertical dashed line shows the lifetime of an object in this particular collaboration. Horizontal arrows indicate the invocation and return of operations. The order of operations is shown in ascending order of time vertically.

Collaboration diagrams, as the one below, show the same information in a different format, with less emphasis on the time ordering. (See Use Case for points of information exchange and class diagram)




## State Chart

State charts describe the flow of control using the concept of state and transitions. A state is a condition or situation in the life of an object during which it satisfies a certain condition, performs some actions, or waits for some events. A transition is a relationship between two states indicating that an object in the first state (the source state) will perform certain actions and enter a second state (the destination state) when a specified event occurs and certain conditions are satisfied. An event is an occurrence of a stimulus that will trigger the transition. Conditions known as guards can be attached to transitions. The following state chart represents the active state transitions for the ControlPanel object:



# Using and re-using objects

An object is viewed in terms of how it can be used, or how it works. It should be designed to be as self contained as possible, have all data necessary for performing its operations, and no other data. An object should be expected only to perform the operations that achieve the required functionality. For example a problem domain object would not be expected to perform operations such as interaction with the user, data storage, or database retrieval. These operations would be performed by specialised classes.

The object-oriented approach promotes reusability. The development process will therefore normally include the reuse of some previously developed classes. Some organisations have libraries of reusable classes that can be used in many different applications. Re-use can be considered at the analysis stage of development as well as later as part of design and implementation.

The previous discussion has shown how encapsulation, inheritance, composition and conformance (implementing an interface) can contribute to re-use.

# Object-oriented Analysis and Design

The result of analysis is a specification of the requirements for the proposed system. Design is the process of producing the solution to meet these requirements. In essence object-oriented analysis is a classification activity concerned mainly with the business requirements sometimes called the problem domain. Objects, their relationships and behaviour are identified. Object-oriented design specifies how the relationships among objects are achieved, how behaviour will be implemented. Design traditionally includes system design and detailed design.

System design specifies the overall architecture of the system - the structure and organisation of the main software components . This would include the global control structures, protocols for communication, synchronisation, data access, assignment of functionality to design elements, composition of design elements, physical distribution, scaling and performance, selection among design alternatives.

Once the decisions and architectural choices have been made, detailed design will develop the classes to handle the different aspects of the system. It is not always possible to make a clear distinction between object-oriented analysis and object-oriented design.

Some aspects of the design are dependent of the choice of platform. These will affect the system architecture, the design of objects and the interfaces with various components of the system. Examples are data storage for example whether or not a file handling system or database management system will be used, and which one, the choice of the Graphical User Interface for developing the HCI, communication with other systems or devices etc.

The generic subsystems are problem domain (also called business objects), human interaction, data management and system interaction. One of the reasons for this choice of subsystems is the fact that each of the subsystems will re-use a specific specialised set of classes.

If a GUI is used for the human interaction subsystem then the HCI will re-use classes from the particular implementation environment e.g. in the case of Java the jdk awt. For the data management subsystem which will handle the persistence problem - the permanent storage of data and retrieval where multi-user access is required and a DBMS is used, specialised class libraries will be used for accessing the DBMS, e.g. in Java the ODBC/JDBC. If the application is distributed then the system interaction subsystem will involve the re-use of networking and communication classes, e.g. in Java the RMI from the java.net package.

## Review Question 12

What are the differences between system design and detailed design?

Answer to this question can be found at the end of this chapter.

### Review Question 13

What aspects of the system are added to the class diagram(s) in object-oriented detailed design?

Answer to this question can be found at the end of this chapter.

### Review Question 14

Why is it sometimes necessary to design information systems that have explicitly concurrent behaviour?

Answer to this question can be found at the end of this chapter.

### Review Question 15

What facilities are typically offered by a DBMS?

Answer to this question can be found at the end of this chapter.

# Programming, Design and Analysis

It has been claimed in the past that analysis and design methods are independent of the implementation language. This is a myth. A programming paradigm has specific characteristics, abstraction mechanisms, and program structuring facilities, which have potential strengths and weaknesses A design method is developed which modifies conceptual structures according to the abstraction mechanisms so that they map onto the program structures provided by the programming paradigm. An analysis methodology is developed which tackles unstructured requirements and moulds them into the conceptual structures needed in the design. Every programming paradigm brings with it a paradigm for design and analysis.

## Limitations of the traditional structured approach

- They take no account of evolutionary change
- They impose a separation of data from the operations on the data. (This was imposed by the 3GL, 4GL languages used to implement the systems.
- They provide an artificial way of viewing a system, which makes it progressively more difficult to trace real world, objects in the design and implementation of the system.
- Changes to an existing system are more difficult to implement because of the nature of the implemented software.
- The development process is often viewed as a ‘one-off’ rather than a continuous process.
- They do not encourage reusability

# Comparison of the Object-oriented and the Structured Analysis and Design

## The object-oriented stages of development

The development process is both top-down and bottom-up. The problem is partitioned in terms of objects and classes, which is a top down activity. Re-use is considered at all points, during analysis, design and implementation. Existing designs, frameworks, patterns, components, class libraries are considered for re-use. This is a bottom up activity. The life cycle does not consist of entirely distinct stages. This is

especially true of the transition from design to implementation. The design and implementation will differ only in the level of detail.

According to Meyer3 the effort required with the Object-oriented approach compared to the traditional approach reflects the differences:

| **Life Cycle Stage** | **Traditional** | **Object-oriented** |
|---|---|---|
| Requirements Specification | 20 | 20 |
| Design | 30 | 50 |
| Coding | 35 | 20 |
| Testing | 15 | 10 |

Design is much more complex than with traditional development, because of re-use, but coding requires less effort so does testing. UML may be used in all phases of development and for all types of software systems. Views show different aspects of the system. There are nine diagram types combined in different ways to provide the views.

| **Major Area** | **View** | **Diagrams** | **Main Concepts** |
|---|---|---|---|
| Structural | static view | class diagram | class, association, generalisation, dependency, realisation, interface |
| Structural | use case view | use case diagram | use case, actor, association, extend, include, use case generalisation |
| Structural | implementation view | component diagram | component, interface, dependency, realisation |
| Structural | deployment view | deployment diagram | node, component, dependency, location |
| Dynamic | state machine view | statechart diagram | state, event, transition, action |
| Dynamic | activity view | activity diagram | state, activity, completion transition, fork, join |
| Dynamic | interaction view | sequence diagram | interaction, object message, activation |
| Dynamic | interaction view | collaboration diagram | collaboration, interaction, collaboration role, message |
| Model management | model management view | class diagram | package, subsystem, model |
| Extensibility | all | all | constraint, stereotype, tagged values |

# Analysis - Requirements Specification

## Comparison of Context diagrams, DFSs and Use Cases

The Data Flow Diagrams (DFDs) contain much of the information of the Use Case diagrams. Other similarities are that both specify what is inside and what outside the system boundary. Use case diagrams identify the main functions of a system and are a purely logical representation, without trying to define

in any specific way how the functions they detail will be implemented.

A use case is a specification of a function of the system, while a DFD specifies processes. There is no flow of data between use cases, where DFDs show the flow of data between processes. It is claimed that use cases are more use friendly, and can be used throughout the development process, finally as a blue print for testing. DFDs are more difficult to trace in the process definitions.

### Activity 1

Create a use case diagram for the Estate Agent case study.

**Estate agency case study**

Clients wishing to put their property on the market visit the estate agent, who will take details of their house, flat or bungalow and enter them on a card which is filed according to the area, price range and type of property.

Potential buyers complete a similar type of card, which is filed by buyer name in an A4 binder.

Weekly, the estate agent matches the potential buyer' requirements with the available properties and sends them the details of selected properties.

When a sale is completed, the buyer confirms that the contracts have been exchanged, client details are removed from the property file, and an invoice is sent to the client. The client receives the top copy of a three-part set, with the other two copies being filed.

On receipt of the payment the invoice copies are stamped and archived. Invoices are checked on a monthly basis and for those accounts not settled within two months a reminder (the third copy of the invoice) is sent to the client.

A discussion on this activity can be found at the end of this chapter.

## Comparison of ERDs and Class Diagrams

Entity Relationship Diagrams (ERDs) contain most of the information of the Class diagrams. The main entities of a system that represent the data about which information needs to be kept are usually also classes of the problem domain.

The main difference is that classes also define functionality. The Object-oriented paradigm has additional structuring/abstraction facilities of inheritance and polymorphism. ERD are a modelling technique designed for 4GL development using relational databases.

### Activity 2

Create a class diagram for the Estate Agent System.

A discussion on this activity can be found at the end of this chapter.

## Comparison of ELH and sequence/collaboration and statechart diagrams

Both approaches need to incorporate the modelling of event handling, and timing. Entity Life Histories (ELHs) contain information similar to statecharts. Dynamic modelling is of essential importance in object-orientation. This is because there is no hierarchical control of execution of the software, but rather objects interacting with each other through messages, event driven. To make sure the system will work and that objects will be able to communicate, a vital part of the development process is executing scenarios. This is modelled in UML by using sequence or collaboration diagrams and statecharts.

Objectory mentioned in earlier units describes a systematic approach to this activity as part of design/

implementation.

1. A sequence diagram is designed for each use case. The sequence diagram will have all the objects involved in the use case. This may include HCI objects such as menus or data management objects as well as problem domain objects.

2. A statechart is produced for each object/class.

3. All the interactions an object may participate in will be checked against each sequence diagram, and the statechart.

4. All this information will be taken into account for the final specification of the class.

### Activity 3

Create a sequence or collaboration diagram one for each use case specified in activity 1

A discussion on this activity can be found at the end of this chapter.

## Advantages of the structured approach to analysis and design

The structured approach is well established. There is a lot of expertise and CASE tools exist to support the development. Most development projects set their own standards - which is a selection of techniques that are adopted for analysis and design, and most CASE tools can be configured, to support a chosen set of standards.

The distinct stages make it easier to schedule, distribute work among a number of people. It is easier to express a system in terms of its functions than its data. The structured methods are based on functional decomposition expressed using DFDs.

Class diagrams are more similar to ERDs, which are more difficult to model. The iterative approach to development does not provide the clear cut stages at the end of which one can deliver part of the system, get management commitment before continuing with the development. Most applications need to handle persistent data in a way that will provide multi-user access, and provide all the facilities for security, transaction management, concurrency, etc. This would best be handled using a DBMS. The most widely used are relational DBMSs. There is no natural way in which an object-oriented system can be mapped onto a relational database. Object-oriented databases are not yet widely used or as reliable.

## Towards a Methodology

There are several Object-oriented methodologies, and CASE tools that support them. In the early 1990s most of these methodologies were largely based on the structured methodologies, and did not deliver the expected benefits. The object-oriented technology has not been adopted as widely as was initially expected at the time.

Object-orientation has reached a state of maturity and some useful techniques have been standardised which has resulted in the specification of the UML. This is not a methodology because it does not prescribe a process and stages in which different tasks should be performed. It offers a set of techniques and expects the developers, like good craftsmen, to choose the best tools of the trade for the particular problem they are expected to solve.

Because much of the implementation will inevitably involve re-use of code at different levels: patterns, classes, and frameworks the implementation will influence the design and the designer will need to understand more about the implementation environment. It is also important that the designer understands

what the object-oriented paradigm has to offer both in terms of its strengths and weaknesses.

### Discussion Topic

Compare the structured models of the Estate Agency system with the object-oriented ones you have created, and with the models your colleagues.

**Contribution to discussion:** Has anyone used inheritance in the class diagrams? How many of you have created the same class diagram? What additional work is required to complete the development process?

# Answers and Discussions

### Answer to Review Question 1

Objects and classes have two different aspects: their representation in the object-oriented model and their interpretation in the real world. A class represents a set of objects with similar characteristics and behaviour. These objects are called instances of the class. A class is represented in the model by the structure of states and behaviours that are shared by all instances. An object represents anything in the real world that can be distinctly identified. An object has unique identity, a state and behaviour.

### Answer to Review Question 2

A message consists of the receiving object, the operation to be invoked, and (optionally) the argument(s) to the operation. A set of operations to which an object can respond are called the object interface. The object data can only be accessed by the object's own operations.

### Answer to Review Question 3

Generalization: Merging two or more classes into one according to their similarities

Specialization: Splitting a class into two or more according to their differences

### Answer to Review Question 4

A subclass inherits the characteristics of all its ancestors. The definition of the subclass contains at least one characteristic not included in its ancestors. A superclass contains the common characteristics of all it s subclasses.

### Answer to Review Question 5

Encapsulation provides self-contained modules that can be used as black boxes through their interfaces.

### Answer to Review Question 6

Encapsulation facilities, inheritance, polymorphism, composition all help re-use.

### Answer to Review Question 7

Aggregation is a whole part relationship, which can be a composition or containment. Composition is a kind of aggregation where the whole and its parts they have the same lifetime. The composite cannot be formed without all its components.

### Answer to Review Question 8

Usually to add some specific customising features.

### Answer to Review Question 9

It is a class that specifies common behaviour, but as not all the operations are fully defined it cannot be instantiated. The subclasses will contain all the code for the operations from the abstract class that were not fully defined.

### Answer to Review Question 10

It helps build structures that make explicit the degree of similarity or difference between classes.

## Answer to Review Question 11

If all the characteristics of the superclass do not apply to the subclass. The subclasses must be fully consistent with its superclass.

## Answer to Review Question 12

System design is mainly concerned with the design of overall software architecture of the system while detailed design consists of designing in detail the implementation of the architecture.

## Answer to Review Question 13

Classes that will handle the interface with the user, interface with other systems, data management will be added, and the class diagrams further refined based on the refined sequence and collaboration diagrams ensuring the message passing mechanisms for implementing the associations are correct etc.

## Answer to Review Question 14

Some systems need to respond concurrently to different events, or to allow many tasks to be performed simultaneously because of multi-user access to data. This may be because operations need to co-operate in the execution of a particular task for example some distributed application involving or to compete for resources e.g. multi-user access to a shared database involving updates.

## Answer to Review Question 15

DBMS typically offers support for:

- Different views of the data by different users
- Control of multi-user access
- Distribution of data over different platforms
- Security
- Enforcement of integrity constraints
- Access to data by various applications
- Data recovery
- Portability across platforms
- data access via query languages
- query optimisation

## Discussion of Activity 1

Compare the use case diagram with the context and level one DFD describing the same problem. Make a list of similarities and differences.

## Discussion of Activity 2

Compare the class diagram to the ERD. Have you used any specific OO structuring facilities such as inheritance? Does you choice of objects differ greatly from the choice of entities. How about the attributes, and operations?

Discuss how relationships may be established in OO? In ERD relationships are based on equal values of attributes, and artificial introduction of foreign keys? There is no need for objects to introduce duplica-

tion of data. How about object identity?

## Discussion of Activity 3

After completing the dynamic modelling of the Estate agency case study you should be able to iterate through your class diagram and refine it. This may result in adding attributes, operations, moving them from one class to another, so as to achieve the required functionality. Try following the Objectory technique of collecting the information concerning an object from all the use cases it may be involved in and the sequence diagrams to specify the class.

Choose the most active object for this exercise.