UNIT 1 INTRODUCTION TO OBJECT ORIENTED MODELING

Structure **Page Nos.**

1.0 Introduction 7
1.1 Objectives 7
1.2 Object Oriented Modeling 8
1.3 Basic Philosophy of Object Orientation 10
1.4 Characteristics Object Oriented Modeling 11
1.4.1 Class and Objects
1.4.2 Links and Association
1.4.3 Generalization and Inheritance
1.5 An Object Model 16
1.6 Benefits of OO Modeling 17
1.7 Introduction to OOA& Design Tools 17
1.8 Summary 19
1.9 Solutions/Answers 19

1.0 INTRODUCTION

Object oriented design methods emerged in the 1980s, and object oriented analysis methods emerged during the 1990s. In the early stage, object orientation was largely associated with the development of graphical user interfaces (GUIs), and a few other applications became widely known. In the 1980s, Grady Booch published a paper on how to design for Ada, and gave it the title, *Object Oriented Design*. In 1991, Booch was able to extend his ideas to a genuinely object oriented design method in his book with the same title, revised in 1993 (Booch, 1994) [*sic*].

The Object Modeling Technique (OMT) covers aspects of object oriented analysis and design. OOT provides a very productive and practical way of Software development. As object oriented Technology (OOT) is not language dependent, there is no need for considering a final implementation language, during Object Oriented Modeling (OOM). OOT combine structural, control and functional aspects of the system. We will discuss the structural, control and functional aspects of the systems in great detail in block 3 of this course.

In this unit, we will discuss the basic notions of object orientation. This unit will cover discussion on objects, classes, links, association, generalization, and inheritance. We will discuss the basics of an object model with the help of an example. Towards the end of this unit we will cover the benefits of OOM. In this unit, you will also be introduced to some OOAD tools.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- explain basics of object oriented Modeling;
- define Objects and Classes;
- explain the concepts of links and Associations;
- explain the concept of Generalization and Inheritance;
- describe benefits of Object Oriented Modeling, and
- explain the use of some OOAD tools.

1.2 OBJECT ORIENTED MODELING

Object oriented modeling is entirely a new way of thinking about problems. This methodology is all about visualizing the things by using models organized around real world concepts. Object oriented models help in understanding problems, communicating with experts from a distance, modeling enterprises, and designing programs and database. We all can agree that developing a model for a software system, prior to its development or transformation, is as essential as having a blueprint for large building essential for its construction. Object oriented models are represented by diagrams. A good model always helps communication among project teams, and to assure architectural soundness.

It is important to note that with the increasing complexity of systems, importance of modeling techniques increases. Because of its characteristics Object Oriented Modeling is a suitable modeling technique for handling a complex system. OOM basically is building a model of an application, which includes implementation details of the system, during design of the system.

Brooks observes that the hard part of software development is the manipulation of its essence due to the inherent complexity of the problem, rather than the accidents of its mapping into a particular language, which are due to temporary imperfections in our tools that are rapidly being corrected (Brooks-87).

As you know, any system development refers to the initial portion of the software life cycle: analysis, design, and implementation. During object oriented modeling identification and organization of application with respect to its domain is done, rather than their final representation in any specific programming language. We can say that OOM is not language specific. Once modeling is done for an application, it can be implemented in any suitable programming language available.

OOM approach is a encouraging approach in which software developers have to think in terms of the application domain through most of the software engineering life cycle. In this process, the developer is forced to identify the inherent concepts of the application. First, developer organize, and understood the system properly and then finally the details of data structure and functions are addressed effectively.

In OOM the modeling passes through the following processes:

- System Analysis
- System Design
- Object Design, and
- Final Implementation.

System Analysis: In this stage a statement of the problem is formulated and a model is build by the analyst in encouraging real-world situation. This phase show the important properties associated with the situation. Actually, the analysis model is a concise, precise abstraction and agreement on how the desired system must be developed. You can say that, here the objective is to provide a model that can be understood and criticized by any application experts in the area whether the expert is a programmer or not.

System Design: At this stage, the complete system architecture is designed. This is the stage where the whole system is divided into subsystems, based on both the system analysis model and the proposed architecture of the system.

Object Design: At this stage, a design model is developed based on the analysis model which is already developed in the earlier phase of development. The object design decides the data structures and algorithms needed to implement each of the classes in the system with the help of implementation details given in the analysis model.

Final Implementation: At this stage, the final implementation of classes and relationships developed during object design takes place a particular programming language, database, or hardware implementation (if needed). Actual implementation

should be done using software engineering practice. This helps to develop a flexible and extensible system.

Whole object oriented modeling is covered by using three kinds of models for a system description. These models are:

- object model,
- dynamic model, and
- functional model.

Object models are used for describing the objects in the system and their relationship among each other in the system. The dynamic model describes interaction among objects and information flow in the system. The data transformations in the system are described by a functional model. **All three models are applicable during all stages of development**. These models bear the responsibility of acquiring implementation details of the system development. It is important to note that you cannot describe a system completely until unless all three modes are described properly. In block 3 of this course, we will discuss these three models in detail.

Before we discuss the characteristics of object oriented modeling, let us see how object oriented development is different from structured development of the system. In the structured approach, the main emphasis is **on specifying and decomposing system functionality**. Structured approach is seen as the **most direct way of implementing a desired goal**. A structured approach has certain basic problems, such as, if the requirements of system change then a system based on decomposing functionality may require massive restructuring, and, the system gradually become unmanageable. In contrast to the structured approach, the basic focus of object-oriented approach is to **identify objects** from the application domain, and then to **associate procedures** (methods) around these identified objects.

You can say that object oriented development is an indirect way of system development because in this approach **a holistic view of application domain is considered, and objects are identified** in the related problem domain. A historic view of application helps in realizing the situations and characteristics of the system. Taking a **holistic view of the problem domain rather than considering functional requirements of a single problem give an edge to object oriented development**. Once the objects are created with the needed characteristics, they communicate with each other **by message passing** during problem solving.

☞ Check Your Progress 1

1) What is OOM?

……..

……..

……..

2) List different steps involved in OOM process.

……..

……..

……..

3) Differentiate OO development from structured development.

……..

……..

……..

1.3 BASIC PHILOSOPHY OF OBJECT ORIENTATION

There are several characteristics of object-oriented technology. Some of these characteristics have been discussed in course MCS-024. We have also implemented some of them in Java programming language, although these characteristics are not unique to object-oriented systems in the sense that they vary from object based systems to object oriented systems. However, most of the properties are particularly well supported in object oriented systems.

Now, let us discuss about the basic characteristics around which object oriented systems are developed.

Abstraction

Abstraction is one of the very important concepts of object oriented systems Abstraction focues on the essential, inherent aspects of an object of the system. It does not represent the accidental properties of the system. In system development, abstraction helps to focus on what an object is supposed to do, before deciding how it should be implemented. The use of abstraction protects the freedom to make decisions for as long as possible, by avoiding intermediate commitments in problem solving. Most of the modern languages provide data abstraction. With the abstraction, ability to use inheritance and ability to apply polymorphism provides additional freedom and capability for system development. When you are using abstraction during analysis, you have to deal with application-domain concepts. You do not have to design and make implementation decisions at that point.

Encapsulation

Encapsulation, or information hiding, is the feature of separating the external aspects of an object, from the internal implementation details of that object. It helps in hiding the actual implementation of characteristics of objects. You can say that encapsulation is hiding part of implementation that do internal things, and these hidden parts are not concerned to outside world. Encapsulation enables you to combine data structure and behaviour in a single entity. Encapsulation helps in system enhancement. If there is a need to change the implementation of object without affecting its external nature, encapsulation is of great help.

Polymorphism

Class hierarchy is the deciding factor in the case of more than one implementation of characteristics. An object oriented program to calculate the area of different Figures would simply call the Find_ Area operation on each figure whether it is a circle, triangle, or something else. The decision of which procedure to use is made implicitly by each object, based on its class polymorphism makes maintenance easier because the calling code need not be modified when a new class is added.

Sharing of Structure and Behaviour

One of the reasons for the popularity of object-oriented techniques is that they encourage sharing at different levels. Inheritance of both data structure and behaviour allows common structure (base class) to be used in designing many subclasses based on basic characteristics of base class, and develop new classes with less effort. Inheritance is one of the main advantages of any object oriented language, because it gives scope to share basic code.

In a broader way we can say that object oriented development not only allows information sharing and reuse within an application, but also, it gives a base for project enhancement in future. As and when there is a need for adding new characteristics in the system, they can be added as an extension of existing basic

features. This can be done by using inheritance, and that too, without major modification in the existing code. But be aware that just by using object orientation you do not get a license to ensure reusability and enhancement. For ensuring reusability and enhancement you have to have a more general design of the system. This type of design can be developed only if the system is properly studied and features of proposed system are explored.

Emphasis on Object Structure, not on Operation Implementation

In object orientation the major emphasis is on specifying the characteristics of the objects in a system, rather than implementing these characteristics. The uses of an object depend highly on the facts of the application and regular changes during development. As requirements extend, the features supplied by an object are much more stable than the ways in which they are used, hence software systems built on object structure are more secure.

While developing a system using the object oriented approach, main emphasis is on the essential properties of the objects involved in the system than on the procedure structure to be used for implementation. During this process what an object is, and its role in system is deeply thought about.

1.4 CHARACTERISTICS OF OBJECT ORIENTED MODELING

In object oriented modeling objects and their characteristics are described. In any system, objects come into existence for playing some role. In the process of defining the roles of objects, some features of object orientation are used. In this section we will discuss these features, which include:

- Class and Objects
- Links and Association
- Generalization and Inheritance

Let us start our discussion with Class and Objects.

1.4.1 Class and Objects

A *class* is a collection of things, or concepts that have the same characteristics. Each of these things, or concepts is called an *object.*

We will discuss, in the next unit of this block, that the class is the most fundamental construct within the UML.

Classes define the basic words of the **system being modeled**. Using a set of classes as the core vocabulary of a software project tends to greatly facilitate understanding and agreement about the meanings of terms, and other characteristics of the objects in the system.

Classes can serve as the foundation for data modeling. In OOM, the term classes is usually the base from which visual modeling tools−such as Rational Rose XDE, Visual Paradigm function and design the model of systems.

Now, let us see how the characteristics that classes share are captured as attributes and operations. These terms are defined as follows:

- *Attributes* are named slots for data values that belong to the class. As we have studied in MCS-024, different objects of a given class typically have at least some differences in the values of their attributes.
- *Operations* represent services that an object can request to affect the behaviour of the object or the system itself.

In unit 3 of this block, we will cover the standard UML notation for OOM in detail. Here, we will mention about standard notation of class. The notation for a class is **a box with three sections**. The top section contains the name of the class in boldface type, the middle section contains the attributes that belong to the class, and the bottom section contains the class's operations as you can see in *Figure 1.*

Figure 1: Class notation

You can, also show a class without its attributes or its operations, or the name of the class can appear by itself as shown in *Figure 2.*

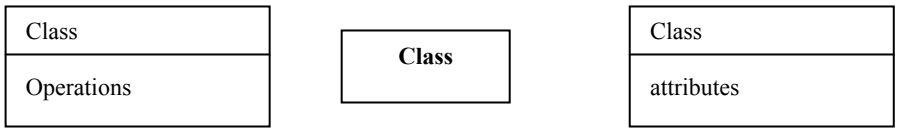


Figure 2: Alternate class notations

The naming convention for classes are as follow:

- Class names are simple nouns or noun phrases.
- Attribute names in a class are simple nouns or noun phrases. The first word is not capitalized, but subsequent words may be capital.
- Operation names are simple verbs. As with attributes, the first word is not capitalized and subsequent words may be capital.

Objects

The notation for an object is the same in basic form as that for a class. There are three differences between the notations, which are:

- Within the top section of the class box, the name of the class to which the object belongs appears after a **colon**. The object may have a name, which appears before the colon, or it may be anonymous, in which case nothing appears before the colon.
- The contents of the top compartment are underlined for an object.
- Each attribute defined for the given class has a specific value for each object that belongs to that class.

You can see the notion of an object you can see in *Figure 3.*

Figure 3*:* Notation of object

If you look around you will find many examples of real world objects such as your books, your desk, your television, etc.

Everything that the software object knows (state) and can do (behaviour) is expressed by the variables and the methods within that object. In other words, all the objects share states and behaviour. Let us say that a software object that models your real-world bicycle would have variables that indicated the bicycle's current state: its speed is 20 mph, and its current gear is the 3rd gear, etc.

Communication by Message Passing

You will agree that a single object alone is generally not very useful. Objects usually appear as a components of a larger program or a system. Through the interaction of these objects, functionality of systems are achieved. Software objects interact and communicate with each other by *message passing* to each other. When object X wants object Y to perform one of methods of object Y, object X sends a message to object Y. Message passing provide two significant benefits:

- An object's characteristics are expressed through its methods, so message passing supports all possible interactions between objects.
- It closes the gap between objects. Objects do not need to be in the same process, or even on the same machine, to send and receive messages back and forth to each other.

1.4.2 Links and Association

Links and associations are the basic means used for establishing relationships among objects and classes of the system. In the next subsection we will discuss links and associations which are used for representing relationship.

General Concepts

A link is a physical or conceptual connection between objects for example, a student, **Ravi study in IGNOU**. Mathematically, you can define a link as a tuple that is an ordered list of objects. Further, a link is also defined as an instance of an association. In other words you can say that an association is a group of links with a common structure and common meanings, for example, a student study in a university. All the links in an association connects objects from the same classes. A link is used to show a relationship between two (or more) objects.

Association and classes are similar in the sense that classes describe objects, and association describe **links**. *Figure 4a* shows us how we can show the association between Student and University

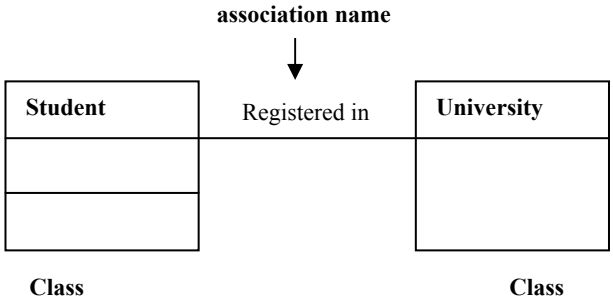


Figure 4a: Association

Note that every association has roles. For example, in *Figure 4b* you can see that two classes, Student and University, have their defined roles. Here you can also see that binary association has two roles, one from each class.

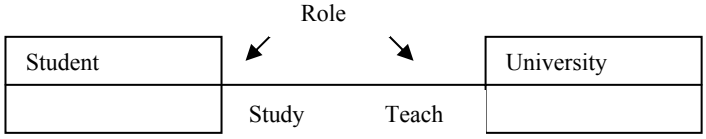


Figure 4b: Roles in association

Associations may be binary, ternary, or have higher order. In exercise, the vast majority of association are binary or ternary associations. But a ternary association is formed compulsion; they cannot be converted into binary association. If a ternary association is decomposed in some other association, some information will be lost. In *Figure 5* you can see a ternary association.

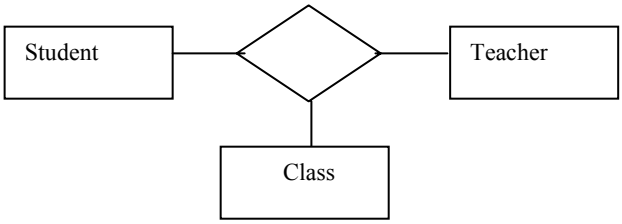


Figure 5: Ternary association

Multiplicity

Multiplicity in an association specifies how many objects participate in a relationship. Multiplicity decides the number of related objects. Multiplicity is generally explained as "one" or "many," but in general it is a subset of the non-negative integers.

Table 1: Multiplicity Indicators.

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
1..n	One to *n* (where *n* > 1)

In associations, generally movement is in both the directions of the relationships but if you want to be specific in any particular direction, you have to mark it by an arrow as given in *Figure 6.*

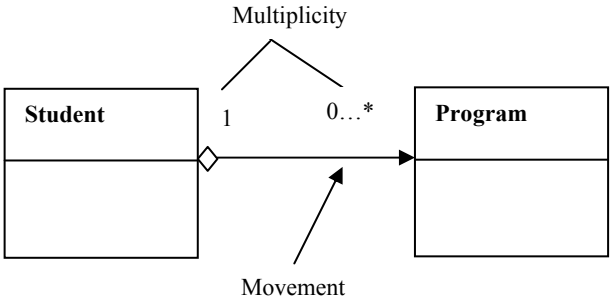


Figure 6: Association and movement

Aggregation

Aggregation is a special form of association, which models the "part-whole" or "a-part-of" relationship as an aggregate (the whole) and parts. The most considerable property of aggregation is transitivity, that is, if X is part of Y and Y is part of Z, then X is part of Z. Aggregation is seen as a relationship in which an assembly class is related to component class. In this component objects are not having separate existence, they depend on composite objects as you can see in *Figure 7* Exam Schedule is not having separate existence.

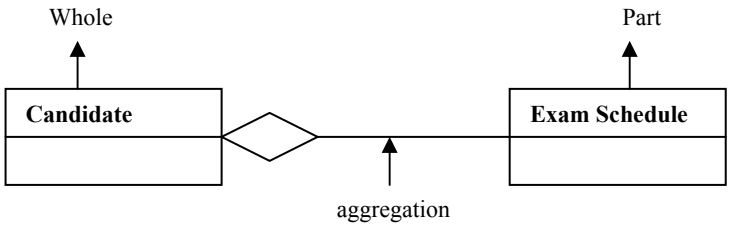


Figure 7: Association and whole-part relationship

☞ Check Your Progress 2

1) What is abstraction?

……

……

2) What is association? Give example of association.

……

……..

3) What is multiplicity in associations? Give example to explain multiplicity?

……

……

1.4.3 Generalization and Inheritance

In this section we will discuss the concepts of generalization, inheritance, and their uses in OOM.

Generalization

Generalization and inheritance are powerful abstractions for sharing the structure and/or behaviour of one or more classes.

Generalization is the relationship between a class, and it defines a hierarchy of abstraction in which subclasses (one or more) inherit from one or more superclasses.

Generalization and inheritance are transitive across a subjective number of levels in the hierarchy. Generalization is an "is-a-kind of" relationship, for example, Saving Account is a kind of Account, PG student is kind of Student, etc.

The notation for generalization **is a triangle connecting a super class to its subclasses**. The superclass is connected by a line to the top of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of the triangle. Generalization is a very useful construct for both abstract modeling and implementation. You can see in *Figure 8*, a generalization of Account class.

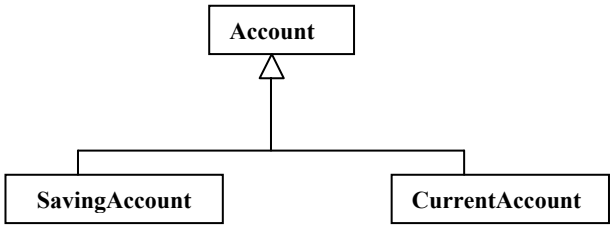


Figure 8: Generalization of account class

Inheritance

Inheritance is taken in the sense of code reuse within the object oriented development. During modeling, we look at the resulting classes, and try to group similar classes together so that code reuse can be enforced. Generalization, specialization, and inheritance have very close association. Generalization is used to refer to the relationship among classes, and inheritance is used for sharing attributes and operations using the generalization relationship. In respect of inheritance, generalization and specialization are two phases of a coin in the sense that if a

subclass is seen from a superclass the subclass is seen as a specialized version of superclass and in, reverse, a superclass looks like general form of subclass.

During inheritance, a subclass may override a superclass feature by defining that feature with the same name. The overriding features (the subclass feature with the same names of superclass features) refines and replaces the overridden feature (the superclass feature).

Now let us look at the diagram given in *Figure 9***.** In this diagram, Circle, Triangle, and Square classes are inherited from Shape class. This is a case of single inheritance because here, one class inherits from only one class.

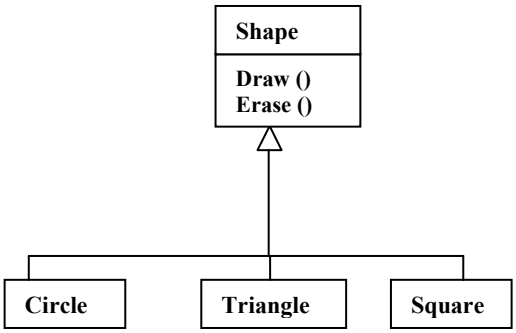


Figure 9: Single inheritance

Multiple inheritance is shown in *Figure 10*. Here, one class is inherited from more than one class.

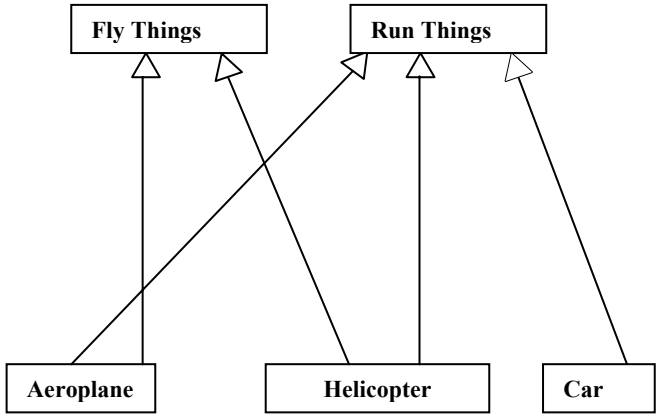


Figure 10: Multiple inheritance

1.5 AN OBJECT MODEL

In object oriented modeling system understanding and on the basis of that identification of classes. Establishing relationship among different classes in the system are the first and foremost activity. Here, we have a simple model of a University System with respect to different levels of courses offered by the University. As you can see in *Figure11*, we have given the basic classes of this system.

This diagram covers different levels of students in the hierarchy. Similarly, for other classes, such as Administration and Faculty, hierarchy level can be drawn to give a broader view of whole system.

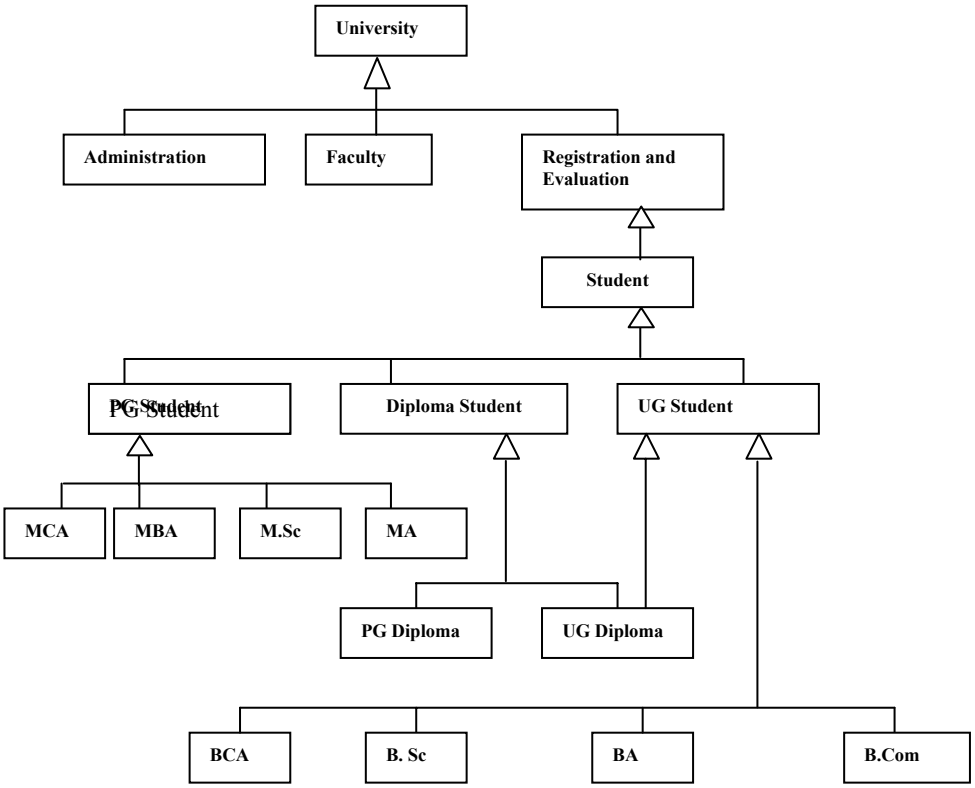


Figure 11: Object model for university system

1.6 BENEFITS OF OBJECT ORIENTED MODELING

There are several advantages and benefits of object oriented modeling. Reuse and emphasis on quality are the major highlights of OOM. OOM provides resistance to change, encapsulation and abstraction, etc. Due to its very nature, all these features add to the systems development:

- Faster development
- Increased Quality
- Easier maintenance
- Reuse of software and designs, frameworks
- Reduced development risks for complex systems integration.

The conceptual structure of object orientation helps in providing an abstraction mechanisms for modeling, which includes:

- Classes
- Objects
- Inheritance
- Association etc.

1.7 INTRODUCTION TO OBJECT ORIENTED ANALYSIS & DESIGN: TOOLS

Unified Modeling Language (UML) is a well accepted language for OOAD. It is used for visualizing, specifying, constructing, and in final documentation. The basic building blocks of UML used for OOAD are things, relationships, and diagrams. Basically, the *Unified Modeling Language* focuses on the concepts of Booch, OMT, and Object Oriented Software Engineering (OOSE). The result of these concepts is a single, common, and widely usable modeling language for users of these and other

methods. The *Unified Modeling Language also* promotes the concept of what can be done with existing methods.

Many modern applications are being developed based on object oriented principles such as classes, methods, and inheritance. To fulfil the needs of such developments, CASE tools offer many benefits for developers building large-scale systems. CASE tools enable us to abstract away from the mess of source code, to a level where design and propose become more clear and easier to understand and modify.

For making and representing these features of the systems to be developed, some tools are used. These tools support UML features and building blocks. Object modeling CASE tools provide support for object oriented modeling notations and methodologies, and they also generate parts of object oriented applications. New versions of many object oriented CASE tools are beginning to address new languages such as Java. Many of these object modeling CASE tools also support relational databases by performing arts of logic, and in some cases, physical database modeling and design, including schema generation and reverse engineering of RDBMS tables, and other elements.

Here, we have tried to give a list collected from many sites and individual searches, for UML modeling tools. Since UML is a fast growing engineering this list may keep on changing all the time.

Action Semantics: This is a group of firms that have responded to the OMG's RFP to define the action semantics for UML.

ArgoUML: This is a domain-oriented design environment that provides cognitive support of object oriented design. ArgoUML provides some of the same automation features of a commercial CASE tool.

ARTiSAN Software: This provides a variety of UML based CASE tools, including a real time modeling tool.
BridgePoint : This provide features of a real time UML modeling tool.

GDPro : this is a full suite of UML and code management tools.

MagicDraw UML: This has Full support for all UML diagrams: MagicDraw RConverter allows you to convert these UML diagrams into MagicDraw: Activity, Class, Collaboration, Component, Deployment, Sequence,State chart, Three-tiered, and Use Case diagrams.

Rational Rose: IBM Rational RequisitePro is a powerful and easy-to-use tool for requirements and use case management.

Visio 2000 Enterprise: It contains a UML suite that can build diagrams within Visio.

Visual Paradigm: Visual Paradigm for the Unified Modeling Language (VP-UML) is a UML CASE suite. The suite of tools is designed for a wide range of users, including Software Engineers, System Analysts, for building large scale software systems reliably through the use of the object oriented approach.

☞ Check Your Progress 3

1) What is inheritance?

……..…

……..….

2) Give an example of multiple inheritance.

……...…

…….

3) Explain the benefit of OOM.

……..…

……..…

……..

1.8 SUMMARY

In this unit we have discussed the basic notions of object orientation, and the need for object oriented modeling. It is the basic characteristic of object orientation which makes it possible to develop systems in such a way that the system is open for reusability. In this unit, concepts of abstraction, encapsulation, polymorphism and sharing of structure and behaviour are discussed.

Further, in this unit, we have discussed notions of class and object. We saw that how inheritance, generalization/specialization, and associations are represented. In this unit, a hierarchy of classes representing different levels of students in a University system is representated, reuse and quality are mentioned as benefits of OOM, and in the last section, some tools, which support UML designs are mentioned.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Object oriented modeling is a approach through which modeling of the systems are done by visualizing the system based on the real world concepts. Object oriented modeling is language independent and generally used for complex system development.

2) Steps involve in OOM are:

   System Analysis

   System Design

   Object Design and

   Final implementation.

3) Structured approach of problem solving is based on the concept of decomposition of system in to subsystem. In this approach of system development readjustment of some new changes in the system is very difficult. On the other hand in object oriented approach holistic view of application domain is considered and related object are identified. Further classification of objects are done. Object oriented approach give space for further enhancement of the system without too much increase in systems complexity.

Check Your Progress 2

1) Abstraction in object orientation is a concept which provide opportunity to express essential properties of the object without providing much details of implementation of these properties.

2) Association is used for establishing relationships between classes. Association describe links between (among) classes. For example, if a professor works in a university then it can be represented as

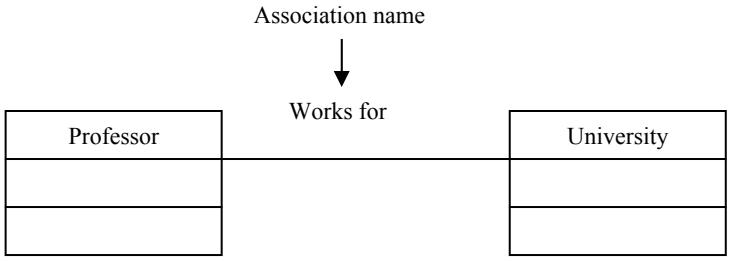


Figure 12: Association of professor and university

3) Multiplicity in an association indicate the number of objects participate in a relationship. For example in the association given in *Figure 13* you can see that one player can play for one team at a time so here multiplicity is 1.

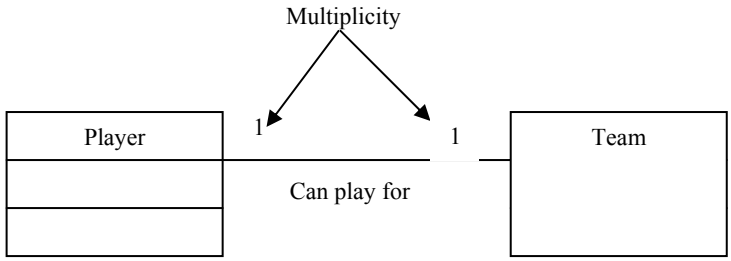


Figure 13: Multiplicity in association

Check Your Progress 3

1) Inheritance is an object orientation concept which allow reusability of design/code. Basic meaning of inheritance is that if one class is already defined than another class which also passes the property of existing class can be defined and inherit the property of existing class. For example, if a class named student is defined and another class for Post Graduate students is to be defined then PG Student class can inherit student class.

2) One example of multiple inheritance is a committee for students affair which consist of faculty and administrative staff member.

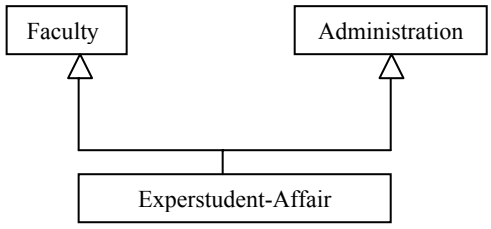


Figure 14: Multiple inheritance

3) Major benefits of object oriented modeling is that development of the system become fast, quality of the system is increase. It give freedom of use of existing design and code. Help in development of complex system with less risk due to the basic properties of object orientation which include class, objects and inheritance.

UNIT 2 OBJECT ORIENTED ANALYSIS

Structure **Page Nos.**

2.0 Introduction 21
2.1 Objectives 21
2.2 Object Oriented Analysis 22
2.3 Problem Statement: An Example 25
2.4 Differences between Structured Analysis and Object Oriented Analysis 26
2.5 Analysis Techniques 27
2.5.1 Object Modeling
2.5.2 Dynamic Modeling
2.5.3 Functional Modeling
2.6 Adding Operations 34
2.7 Analysis Iteration 36
2.7.1 Refining the Ratio Analysis
2.7.2 Restating the Requirements
2.8 Summary 36
2.9 Solutions/Answers 37

2.0 INTRODUCTION

Object oriented analysis (OOA) is concerned with developing software engineering requirements and specifications that expressed as a system's object model composed of a population of interacting objects. The concept of abstraction in object oriented analysis (OOA) is important. Abstraction may be defined as: the characteristics of an object which make it unique and reflect an important concept. Analysis is a broad term, best qualified, as in *requirements analysis* (an investigation of the requirements) or *object oriented analysis* (an investigation of the objects of the problem domain).

OOA views the world as objects with data structures and behaviors and events that trigger operations, for object behavior changes. The idea is to see system as a population of interacting objects, each of which is an atomic bundle of data and functionality, (the foundation of object technology) and provides an attractive alternative for the development of complex systems.

The problem statement is important for any analysis. It is a general description of the user's difficulties, and desires. The purpose of problem statement is to identify the general domain in which you will be working.

In this Unit you will study various analysis techniques: object modeling, dynamic modeling and functional modeling. You will also learn how add operations in system and how to do refining of the analysis model.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- define the concepts of the objects in the system;
- express required system behaviour in terms of business objects in the system, and actions that the user can perform on them;
- understand how to define and analyze the problem statement;
- explain the purpose of object modeling;
- explain dynamic modeling;
- describe the event flow between objects and how to draw state diagrams of real world problems;
- explain and understand the importance of functional model;
- explain how operations can be added in various analysis techniques, and
- explain the importance of iterating or refining the analysis model.

2.2 OBJECT ORIENTED ANALYSIS

In this section we will discuss basics of object oriented analysis with the help of object oriented features.

Analysis

Analysis is not a solution of the problem. Let us see what actually object oriented (OO) analysis. Analysis emphasizes an ***investigation*** of the **problem** and **requirements,** for example, if a new online trading system is desired by a trader, there may be questions such as, how will it be used? What are its functions?

If you start a project the first question about that project' will be how do we get started? Where do we begin?

The starting point for object oriented analysis is to identify candidate objects and their relationships.

The first stage can be a simple brain-storming of the possible objects. Then one method is to go through all the nouns in any documentation about the world you are analysing, and considering these as candidate objects of the system. Also, you can use the alternative technologies for developing software engineering requirements and specifications that include functional decomposition, essential systems analysis, and structured analysis.

The Process of Development

The approach to development can be an iterative one. It involves repeated refinement of the object model. The process needs to be controlled by an appropriate project management process, involving reviews and check pointing at the levels of:

- Analysis
- Design
- Implementation
- Post Implementation Review

Object Orientation and Analysis

An ***Object*** is something that exists within the problem domain that can be identified by data and/or behaviour. An example of an object is a car. The data of a car could be the wheel, brake, seat, etc. The behaviour of the car would be to drive on roads, its speed, etc.

Object oriented analysis is the concept which actually forces you to think in terms of the application domain when its behaviour and data known to you.

In OOA the primary focus on identifying objects from the application domain, then fitting procedures around them.

For example, in the case of the flight information system, the objects would include *Plane, Flight,* and *Pilot,* etc.

The object model has many aspects, which are associated with OO concepts.
Now we will discuss the following principle of OO.

**Abstraction, Encapsulation, Identity, Modularity, Hierarchy,
Typing, Concurrency, and Persistence**

Abstraction

You understand the term object. Now, let us see how problems are seen as objects, and find their associated data and behaviour. You will notice that an object is a real life entity, or abstraction.

In our daily life we deal with complexity by abstracting details away.

Let us see this with an example of:

Driving a car does not require knowledge of internal combustion engine. It is sufficient to think of a car as simple transport.

In simple term, abstraction means to focus on the essential, inherent aspects of an entity, ignoring its accidental properties. Abstraction is a normal process that we do everyday. When you view the world, you can not possibly take in every minute detail, so you concentrate on the aspects that are important.

An abstraction is a simplified description of a system that captures the essential elements of that system (from the perspective of the needs of the modeler), while suppressing all other elements.

Encapsulation

This separates the interface of an abstraction from its implementation. Taking the above example of a car, we can now categorize as:

Abstraction	Car
Interface	Steering, pedals, controls
Implementation	Generally, you don't know

Encapsulation also means **data hiding**, which consists of separating the external aspects of an object, which are accessible to other objects.

Now let us take a example of the Stack.

A Stack abstraction provides methods like push (), pop (), isEmpty(), isFull(). The Stack can be implemented as a singly linked list, or a doubly linked list, or an array, or a binary search tree. This feature is called encapsulation. It hides the details of the implementation of an object.

The Benefits of Encapsulation

To hide the details of a class, you can declare that data or implementation in its private part so that any other class clients will not be able to know about. It will have the ability to change the representation of an abstraction (data structures, algorithms) without disturbing any of its clients. The major benefit of encapsulation is that you.

By Encapsulation

You can delay the resolution of the details until after the design.
Keep your code modular.

Object Identity

Object identity is a property of an object that distinguishes the objects from all other objects in the applications. With object identity, objects can contain, or refer to other objects. You can create an object identity in three ways:

1) You can refer as memory address in programming languages.

2) Assign identifier keys in the database.

3) By user-specified names, used for both programming and database.

In a complete object oriented system each object is given an identity that will be permanently associated with the object irrespective of the object's structural or state transitions. The identity of an object is also independent of the location, or address of the object. Object identity provides the most natural modeling primitive to allow the "same object to be a sub-object of multiple parent objects".

Modularity

Modularity is closely tied to encapsulation; you may think of modularity as a way of mapping encapsulated abstractions into real, physical modules. It is a property of a system that has been decomposed into cohesive and loosely coupled modules.

Cohesion and coupling gives two goals for defining modules. You should make a module cohesive (shared data structures, similar classes) with an interface that allows for minimal inter-module coupling.

It is important to remember that the decisions concerning modularity are more physical issues, whereas the encapsulation of abstractions is logical issues of design.

Hierarchy

This is ranking or ordering of abstraction.
Hierarchy is decided by using the principle of 'divide and conquer'. You can describe complex objects in terms of simpler objects. This is the property of object oriented analysis (OOA) which enables you to *reuse the code.*

You can place an existing class directly inside a new class. The new class can be made up of any number and type of other objects, in any combination that is needed to achieve the desired functionality. This concept is called *composition* (or more generally, *aggregation*). Composition is often referred to as *a "has-a"* relationship *or "part-of"* relationship, for example, in "automobile has an engine" or "engine is part of the automobile."

Typing

This enforces object class such that objects of different classes may not be interchanged. In other words, class can be visualized as a type. Remember that there are two kinds of typing. This typing does not mean the way you usually type the letters.

Strong Typing: When any operation upon an object (which is defined) can be checked at compile time, i.e., type is confirmed forcefully.

Weak Typing: Here, operations on any object can be performed, and you can send any message to any class. Type confirmation is not essential, but in these type of language more errors at execution time may occur.

Concurrency

The fundamental concept in computer programming is the idea of handling more than one task at a time. Many programming problems require that the program be able to:

1) Stop what it's doing

2) Currently deal with some other problem, and

3) Return to the main process. There is a large class of problems in which you have to partition the problem into separately running pieces so that the whole program can be more responsive. Within a program, these separately running pieces are called threads, and the general concept is called *multithreading.*

Currently, if you have more than one thread running that is expecting to access the same resource then there is a problem. To avoid this problem, a thread locks a resource, completes its task, and then releases the lock so that someone else can use the resource. It can lock the memory of any object so that only one thread can use it at a time. It is important to handle concurrent running programs/threads properly.

Persistance

When you create an object, it exists for as long as you need it, but under no circumstances do object exist when the program terminates. While this makes sense at first, there are situations in which it would be incredibly useful if an object could exist

and hold its information even while the program is not running. When, next time you start the program, the object would be there and it would have the same information it had the previous time the program was running. Of course, you can get a similar effect by writing the information to a file or to a database, but in the spirit of making everything an object it would be quite convenient to be able to declare an object persistent and have all the details taken care of for you.

2.3 PROBLEM STATEMENT: AN EXAMPLE

You must understand here that you are looking for a statement of needs, not a proposal for a solution. OOA specifies the structure and the behaviour of the object, that comprise the requirements of that specific object. Different types of models are required to specify the requirements of the objects. These object models contain the definition of objects in the system, which includes: the object name, the object attributes, and the objects relationships to other objects.

As you know, an object is a representation of a real-life entity, or an abstraction. For example, objects in a flight reservation system might include: an airplane, an airline flight, an icon on a screen, or even a full screen with which a travel agent interacts. The behaviour, or state model describes the behavior of the objects in terms of the states of the object and the transitions allowed between objects, and the events that cause objects to change states.

These models can be created and maintained using CASE tools that support the representation of objects and object behaviour.

You can say that the problem statement is a general description of the user's difficulties, desires, and its purpose is to identify the general domain in which you will be working, and give some idea of why you are doing OOA.

It is important for an analyst to separate the true requirements from design and implementation decisions.

Problem statement should not be incomplete, ambiguous, and inconsistent. Try to state the requirements precisely, and to the point. Do not make it cumbersome.

Some requirements seem reasonable but do not work. These kind of requirements should be identified. The purpose of the subsequent analysis is to fully understand the problem and its implications, and to bring out the true intent of the Client.

☞ Check Your Progress 1

1) Give two benefits of Reuse of Code.

……

…..……...

.....

2) Give an example of enforcement in Typing.

……

…..……...

……

3) What are the benefits of OOA technology?

……

…..……..

……

4) Briefly explain what is to be done while defining the problem statement.

……

…..……...…

……

You are already familiar with structured analysis. Now, this is the appropriate time to have a comparison of OOA and Structured analysis.

2.4 DIFFERENCES BETWEEN STRUCTURED ANALYSIS AND OBJECT ORIENTED ANALYSIS

Object oriented analysis design (OOAD) is basically a bottom up approach which supports viewing the system as a set of components (objects) that can be logically kept together to form the system.

Advantages and Disadvantages of Object Oriented Analysis and Design

Advantages:

The OO approach inherently makes each object a stand alone component that can be reused not only within a specific stat problem domain, but also is completely different problem domains, having the requirement of similar objects.

The other main advantage of object oriented (OO) is the focus on data relationships. You cannot develop a successful system where data relationships are not well understood. An OO model provides all of the insight of an ER diagram and contains additional information related to the methods to be performed on the data. We will have more detailed discussion on this aspects in Block 3 of this Course.

Disadvantages:

You know that OO methods only build functional models within the objects. There is no place in the methodology to build a complete functional model. While this is not a problem for some applications (e.g., building a software toolset), but for large systems, it can lead to missed requirements. You will see in Unit 3 of this course. “Use cases” addresses this problem, but since all use cases cannot be developed, it is still possible to miss requirements until late in the development cycle.

Another disadvantage of the object oriented analysis design (OOAD) is in system modeling for performance and sizing. The object oriented (OO) models do not easily describe the communications between objects. Indeed, a basic concept of object oriented (OO) is that the object need not know who is invoking it. While this leads to a flexible design, performance modeling cannot be handled easily.

The object oriented (OO) analysis design itself does not provide support for identifying which objects will generate an optimal system design. Specifically, there is no single diagram that shows all of the interfaces between objects. You will study object oriented analysis design (OOAD) diagrams in Unit 3 of this course. As you know, coupling is a major factor in system complexity, not having this information makes architecture component selection a hit or miss proposition.

Advantages and Disadvantages of Structured Analysis

With experience, you will come to know that most customers understand structured methods better than object oriented (OO) methods. Since one of the main reasons of modeling a system is for communication with customers and users, there is an advantage in providing structured models for information exchange with user groups or customers.

In fact, specifications are typically in the form of a simple English language statement of Work and Requirement Specification. Therefore, the system to be built, must be understood in terms of requirements (functions the system must perform), that is why this naturally leads to a structured analysis, at least at the top level. Specifically structured methods (functional decomposition) provide a natural vehicle for discussing, modeling, and deriving the requirements of the system.

The disadvantage with structured methods is that they do not readily support the use of reusable modules. The top down process works well for new development, but does not provide the mechanisms for "designing in" the use of existing components. The top down process of functional decomposition does not lead to a set of requirements which map well to existing components.

When the requirements do not map cleanly, you have two choices: either you do not use the existing components, or force fit the requirements to the existing components and "somehow" deal with the requirements which are only partially covered by the existing components, which does not lead to a good successful system. Now, we will discuss how actually object oriented analysis (OOA) system is performed.

2.5 ANALYSIS TECHNIQUES

In this section we will see how classes re identified, and their applications with the help of basic modeling techniques.

2.5.1 Object Modeling

Object modelling is very important for any object oriented development, object modeling shows the static data structure of the real world system. Basically, object modeling means identifying objects and classes of a system, or you can say that it describes real world object classes and their relationships to each other. You can develop an object model by getting information for the object model from the problem statement, expert knowledge of the application domain, and general knowledge of the real world. Object model diagrams are used to make easy and useful communications between computer professionals and application domain experts.

To develop an object model first identify the classes and their associations as they affect the overall problem structure and approach. Then prepare a data dictionary.

i) Identify associations between objects.
ii) Identify attributes of objects and links.
iii) Organise and simplify object classes using inheritances.

Then you verify that access paths exist for likely queries

iv) Iterate and refine the model, and
v) Group classes into modules.

Identifying Object Classes

You should always be careful while identifying relevant object classes from the application domain. Objects include physical entities in a system like buildings, employees, department, etc. Classes must make sense in the application domain. At this stage you should avoid computer implementation constructs, such as linked lists and subroutines.

Some classes are implicit in the application domain, so you need to find out by understanding the problem well.

Action-object matrix: A matrix showing how update actions affect objects. It may be considered to be part of the user object model, as it summarizes user object action definitions in a tabular view.

Process of this whole activity is like:

Check for multiple models

- Identify objects
- Create user object model diagram
- Define user object attributes
- Define user object actions
- Create action-object matrix
- Check for dynamic behavior
- Review glossary.

Some notations for user object model:

Notation Meaning

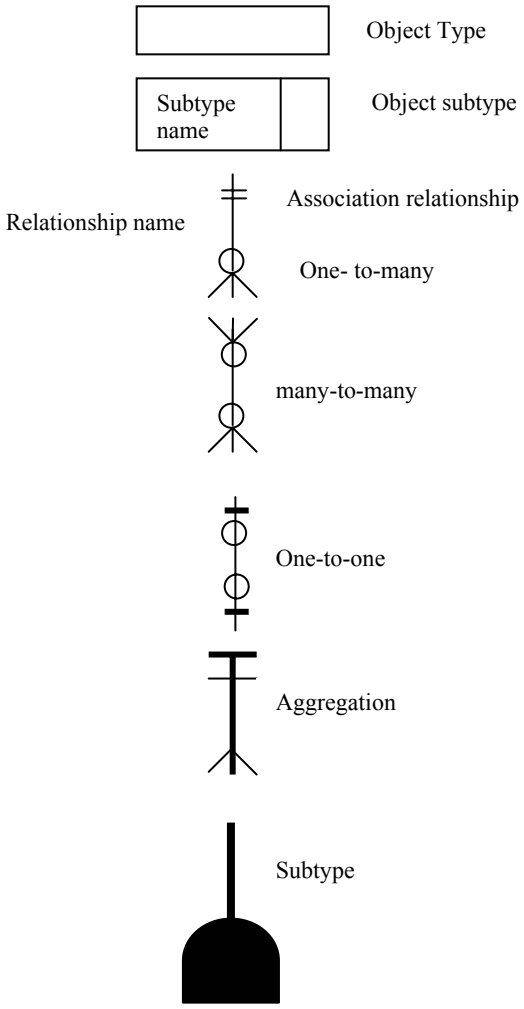


Figure 1: User Object Model Notations

Let us see the example in which we will discuss how concepts are implemented:

To illustrate the object modeling technique to you by taking the very common example of the tasks associated with using a telephone set:

Answer the “phone	
Hear the “phone ringing.	[Ringer]
Identify the ringing “phone.	[Receiver Unit]
Pick up the handset.	[Handset]
Talk to the caller.	[Other party]
Replace the handset.	[Handset]
Make a ‘phone call.	
Pick up the handset.	[Handset]
Listen for the dialing tone.	[Exchange]

'Dial' the number. [Dial]
Listen for the ringing tone. [Receiver Unit]
Talk to the person answering. [Other party]
Replace the handset. [Handset]

On the basis of above information, the **objects/actions** that may be identified from this are:

Receiver Unit
Identify
Ringer
Hear
Handset
Pick up
Replace
Other party
Talk with
Exchange
Listen to
Dial
Enter number

The attributes of these identified objects are identified as:

Receiver Unit
Identity (phone number)
Status (ringing, engaged)
Ringer
Ringing (true, false)
Handset
Hook (on, off)
Other party
Status (caller, answered)
Exchange
Tone (dead, dialing, other)
Dial
Status (empty, number entered)

You can see the User Object Model diagram produced from this given in *Figure 2.*

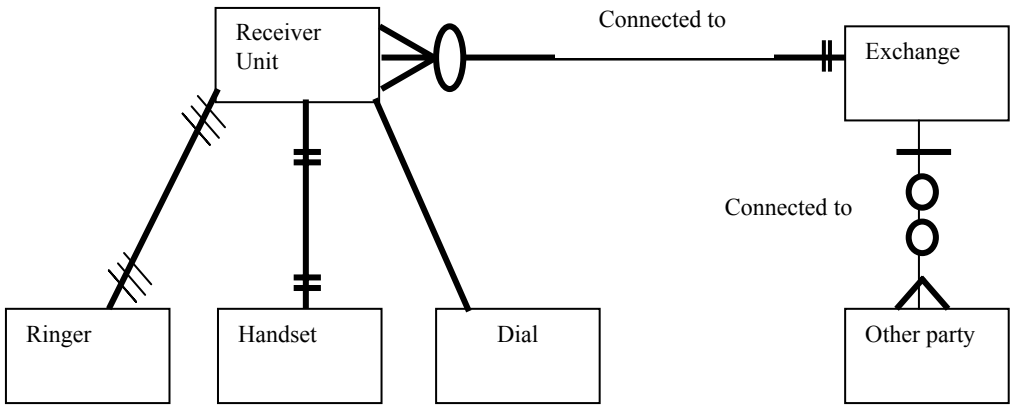


Figure 2: User object Model for Telephone set

2.5.2 Dynamic Modeling

You know that computer systems are built from objects which respond to events. External events arrive at the boundary of the system; you understand the concept of events.

For example, whenever you click with your mouse on the desktop or the canvas area some action is triggered (you get some response back).

In Dynamic modeling you examine "**a way of describing how an individual object responds to events, either internal events triggered by other objects or external events triggered by the outside world**".

Dynamic modeling is elaborated further by adding the concept of time: new attributes are computed as a function of attribute changes over time.

Before you define the dynamic behaviour of user objects. You should know the following:

The Inputs are:

User object model which comprises objects, attributes, actions and relationships.
Task model: from which significant object states may be identified.

Products

Dynamic model: A model of the dynamic behaviour of a user object. It defines significant states of the user object, the way that actions depend on the state, and affect the state.

The dynamic model consists of a dynamic model diagram, showing states and transitions and supplementary notes, specifying states and actions in more detail.

Process of Dynamic modeling:

- Analyse applicability of actions
- Identify object states
- Draw dynamic model diagram
- Express each state in terms of object attributes
- Validate dynamic model
- Concepts.

Dynamic modeling: state diagrams

We will study this in detail in Block 3 of this course.

Generally, a state diagrams allows you to further explore the operations and attributes that need to be defined for an object. They consist of sets of states which an object is in, and events which take the object from one state to another.

State: An object may have one or more states−stable points in its life, expressed by the object's attributes and relationships.

Event/action: Something that happens to an object. Atomic, in that it either has happened or it hasn't. An event causes an action.

Transition:

A jump between states, labeled with the corresponding action.

Notations for state diagram are shown below in *Figure 3*:

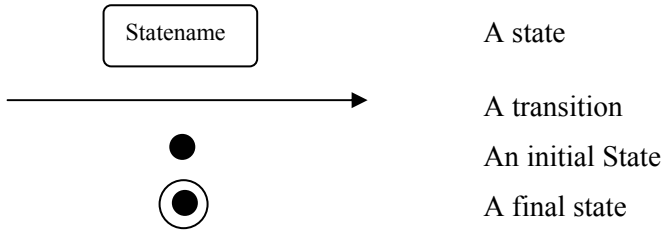


Figure 3: Notation for State Diagram

A simple example using these notation is shown below in Figure 4.

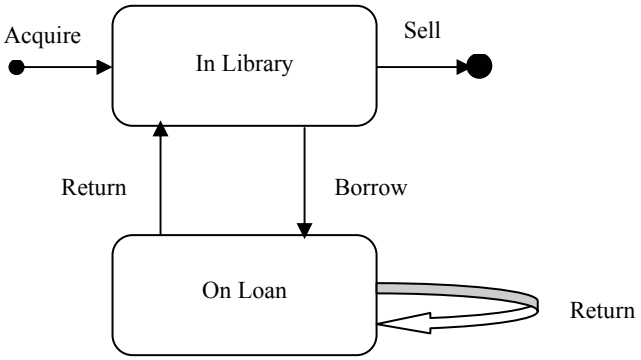


Figure 4: A simple State Diagram

States may be seen as composite states: a collection of states treated as one state at a higher level on analysis. In the example above, the state "In Library" is actually composed of a sequence of sub-states which the library staff, if not borrowers would need to know about.

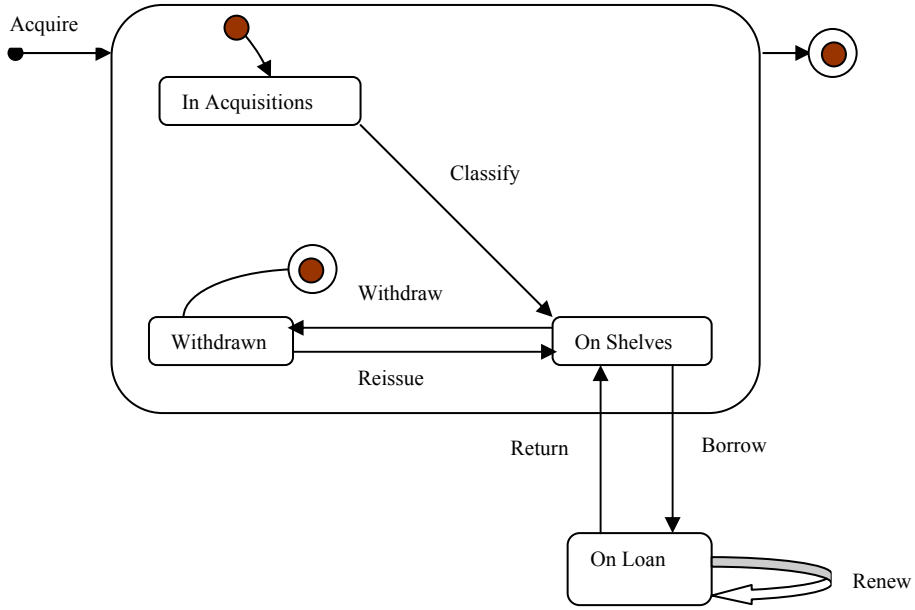


Figure 5: Composite States

Basically any system which you can reasonably model on a computer jumps from state to state.

The level of state decomposition must be decided by judgement. A too fine grained model is inappropriate; for example, modeling all the possible ages of an employee as individual states. Also, a gross decomposition is useless; for example, modeling an employee as employed or not.

You can ask whether all objects of world have behaviour which can be modeled.

Of course, not all objects have behavior worth modeling. Consider our example from the field of object oriented nursery rhymes, the sad story of Humpty Dumpty. Clearly,

you would not like to model such trivial examples.

Basically you need to construct a state transition diagram for every object with significant behaviour. You need not construct one for anything with trivial behaviour.

The reason for doing this is to identify further operations and attributes, to check the logical consistency of the object and to more clearly specify its behaviour.

All the best notations can be used to describe the process they facilitate.

Now we have the basic ideas of object models and dynamic models, our approach to analysis and design (so far) can be summarised as:

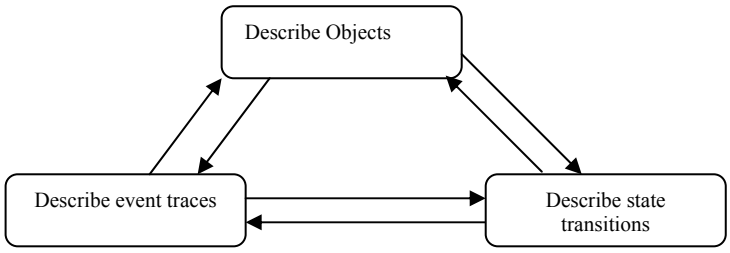


Figure 6: Objects and State transitions

Note: You should also match the events between state diagrams to verify consistency. Till so far, you have been introduced to object modeling, and dynamic modeling. Now, we will see what Function modeling is:

2.5.3 Functional Modeling

You know that Data flow modeling is a common technique used for the analysis of a problem in software engineering. It is exceptionally useful for analyzing and describing systems where there is a large amount of calculation involved.

Data flow models consist of a number of processes which exchange information. A process transforms information it receives, and passes the transformed information on to other processes or to objects in the system. Data flow models can be used to uncover new operations and new attributes for the object model. Sometimes, new objects can be discovered too.

Basically you can state that the functional model shows how values are computed. It describes the decisions or object structure without the regard for sequencing.
It gives you dependency between the various data and the functions that relate them, giving the flow of data.

Each process needs to be implemented as an operation in one or more of the objects. Each data item arising from an object must have a corresponding attribute, or set of attributes in the source object. The data flow diagram (DFD) corresponds to activities or actions in the state diagrams of the classes.

That is why it is suggested to construct the functional model after the object and dynamic models.

Now, let us discuss about the creation of the DFD and the functional model.

The approach to data flow diagramming should be as follows:

- Create a data flow diagram for each of the major outputs of the system
- Work back from the outputs to the inputs to construct the diagram
- Add new objects where necessary to the object model as you discover the need for them in the data flow modeling add new operations and attributes to the object model as you discover the need for them in the data flow modeling.

One thing you have to remember is that the data flow diagram is not used as a basis for devising the structure of the system.

Steps in constructing a Functional Model

- Identify input and output values
- Build data flow diagrams showing functional dependencies
- Describe functions
- Identify constraints
- Specify optimisation criteria.

Identifying Input and Output Values

First, identify what data is going to be used as input to the system, and what will be the output from the system. Input and output values are parameters of events between the system and the outside world. You must note that Input events that only affect the flow of control, such as cancel, terminate, or continue. They do not supply input values. For example, supplier code, name, product description, rate per unit, etc. are the inputs to a sales system.

Build data Flow diagrams showing functional dependencies

Data flow diagrams are useful for showing the functional dependencies. In data flow diagrams processes are drawn as bubbles, each bubble containing with the name of the process inside. Arrowhead lines are used to connect processes to each other, and to objects in the system. The lines are label with the information that is being passed. Objects are drawn as rectangular boxes, just as in the object model, but usually with just the name of these objects and not the attributes and operations.

i)

Sales
Quantity delivered
Calculate quantity delivered to customer

iii)

Product
name price vat rate

ii)

Customer
Product line discount quantity discount
Calculate product discount calculate quantity discount

iv)

Invoice
Calculate line item discount Calculate total Deduct quantity discount Print

Figure 7: Some objects in figure 7: a simple DFD is given for sales system

Let us look at a simple example:

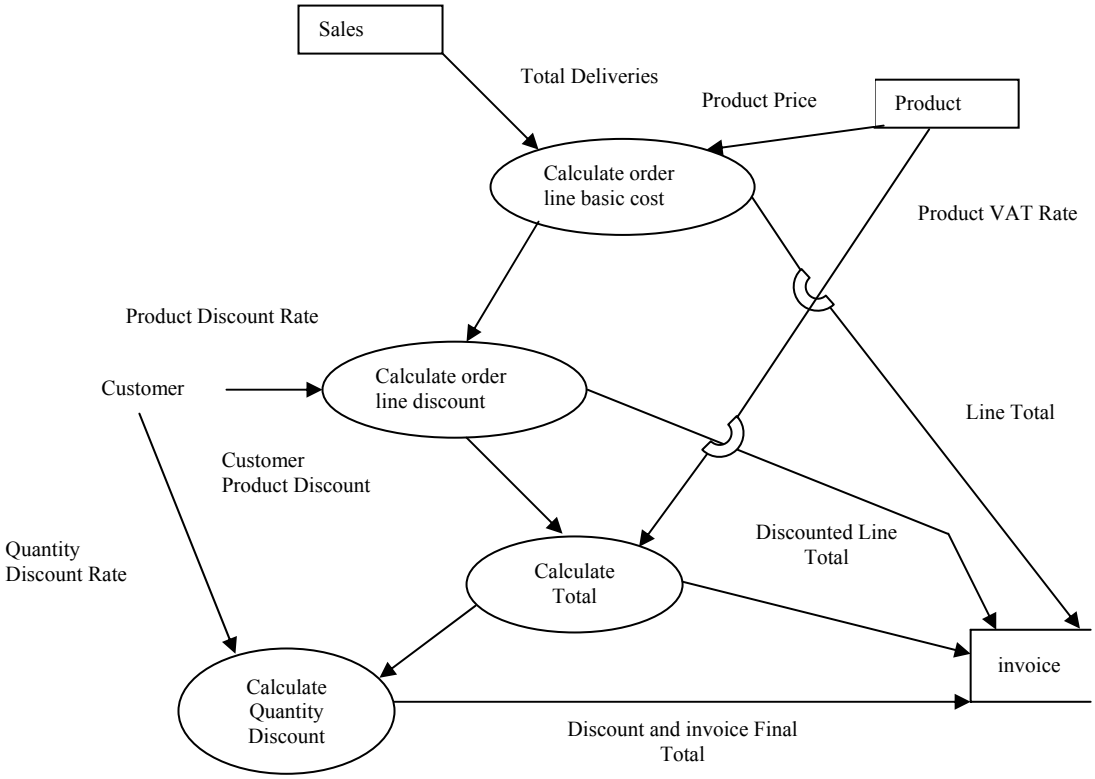


Figure 8: DFD for Sales System

The next stage is to devise operations and attributes to support the calculations.

Describe functions

After you have roughly designed the data flow diagram, you can write a description of each function, and you can describe the function in any form such as, mathematical equations, pseudo code, decision tables, or some other appropriate form. You need not know the implementation of the function, but what it does. The description can be declarative or procedural.

A declarative description specifies the relationship between the input and output values, and relationship among the output values.

A procedural description specifies a function by giving an algorithm to compute it. The purpose of the algorithm is only to specify what the function does.

Identifying Constraints Between Objects

In a system, there are some boundaries to work on. Boundaries or Constraints are the functional dependencies between objects which are not related by an input-output dependency. You have to find out the constraints between the objects. Constraint has different behavior at different times. It can be on two objects at the same time, between instances of the same object at different times (an invariant), or between instances of different objects at different times.

You can define constraints as Preconditions (input values) and PostConditions (output values). Preconditions on functions are constraints that the input values must satisfy. Post conditions are constraints that the output values must satisfy to hold. State the conditions under which constraints hold.

Specifying Optimisation Criteria

Specify values to be maximized, minimized, or optimized. You can understand it as the way you normalize the data in the database. For example, you should minimize the time an account is locked for concurrency reasons. If locks are needed, then it is extremely important to minimize the time that an entire bank is locked for concurrency reasons.

☞ Check Your Progress 2

1) Explain how you can define an object model of a system.

……

….……...…

……...

2) Show the basic dynamic model of telephone.

……

….……

2.6 ADDING OPERATIONS

Whenever you look at the operations in OOPs you find queries about attributes or associations in the object model (such as student.name), to events in the dynamic model (such as ok, cancel), and to functions in the functional model (such as update, save).

You can see operations from the object model. It includes reading and writing attribute values and association links. In the object model operations are presented with an attribute. During analysis nothing is private, you assume that all attributes are accessible.

Accessing from one object to another through the object model can be referred to as "pseudo-attribute" like Account.Bank, where account and bank are two separate objects of their respective classes.

Operations from events

During analysis, events which are sent to the target objects. An operation on the object are presented as labels on transitions and should not be explicitly listed in the object model.

Events can be expressed as explicit methods.

You can also implement events by including event handler as part of the system substrate.

Operations from State Action and Activities

You must see that State actions and activities are actually functions, which can be defined as the operations on the object model.

Operations from Functions

As you know, function are actually operations on object. These functions should be simple and summarized on the object model. Organise the functions into operations on objects. For example, the select operations are really path traversals in the object model. The operations like withdrawal-money, verify-password are the operations on class Account of Bank Management system.

You can write as:
account: withdraw (code, amount)->status
account: deposit (code, amount)->status.

☞ Check Your Progress 3

1) Describe how you can simplify Operations.

……

……

……

2) Give an example of operations from State Actions and Activities.

……

……

……

3) Give an example of Operations from functions of a bank.

……

……

……

4) How do you see the final model after iterative analysis?

……

……

……

5) Why is iterative analysis of any problem needed?

……

……

……

Iteration of analysis is important. Let us see how OOA apply iteration

2.7 ANALYSIS ITERATION

To understand any problem properly you have to repeat the task which implies that analysis requires repetition. First, just get the overview of the problem, make a rough draft, and then, iterate the analysis according to your understanding. Finally, you should verify the final analysis with the client or application domain experts. During the iteration processes refining analysis and restating of requirement takes place.

2.7.1 Refining the Ratio Analysis

Basically, refinement leads to purity. So to get a cleaner, more understandable and coherent design, you need to iterate the analysis process.

- Reexamine the model to remove inconsistencies, and imbalances with and across the model.
- Remove misfit, and wrong concepts from the model.
- Refining sometimes involves restructuring the model**.**
- Define constraints in the system in a better way.
- You should keep track of generalizations factored on the wrong attributes.
- Include exceptions in the model, many special cases, lack of expected symmetry, and an object with two or more sets of unrelated attributes, or operations.
- You should remove extra objects or associations from the model. Also, take care to remove redundancy of objects and their extra attributes.

2.7.2 Restating the Requirements

To have clarity of the analytical model of the system you should state the requirements specific performance constraints with the optimization criteria in one document verify in the other document. You can state the method of a solution.

Verify the final model with the client. The requirement analysis should be confirmed and clearly understood by the client**.**

The impractical, or incorrect, or hypothetical objects that do not exist in the real world should be removed from the proposed system.

You must note that the analysis model is the effective means of communication with application experts, not computer experts. In summary, the final model serves as the basis for system architecture, design, and implementation.

2.8 SUMMARY

In this Unit you have learned that the goal of analysis is to understand the problem and the application domain, so that you come out with a well cohesive design. There is no end or border line between the analysis and the design phase in software engineering. There are three objectives of the analysis model:

Object oriented analysis (OOA) describes what the customer requires, it establishes as basis for the creation of a software design, and it defines a set of requirements that can be validated.

You also remember that the requirement analysis should be designed in such a way that it should tell you what to be done, not how it is implemented.

The object model is the principal output of an analysis and design process.

Dynamic modeling is elaborated further by adding the concept of time: new attributes are computed as a function of attribute changes over time. In this unit, after defining the **scenario** of **typical** and **exceptional sessions**, identify events followed by the building of state diagrams for each active object showing the patterns of events it

receives and sends, together with actions that it performs. You should also match the events between state diagrams to verify consistency.

You have learned that a functional model shows how values are computed; it describes the decisions, or object structure without regard for sequencing.

It gives you dependency between the various data and the functions that relate them, giving the flow of data. Here you construct the data flow diagram which interacts with internal objects and serves as, data stores between iterations. You also specify the required constraints, and the optimisation criteria.

You have seen that refining and restating will give you more clarity of the analytical model of the system.

2.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) **Reusing the implementation**. Place an existing class directly inside a new class. The new class can be made up of any number and type of other objects, in any combination that is needed to achieve the desired functionality. This concept is called *composition* (or more generally, *aggregation*). Composition is often referred to as a "has-a" relationshp or "part-of" relationship, as in "automobile has an engine" or "engine is part of the automobile."

   Reusing the interface. Take an existing class and make modifications or additions to achieve desired functionality. This concept is called *inheritance*. The original class is called the Base class or Parent class and the modified class is called the *Derived* or *Inherited* or *Sub* or *Child* class.

2) You can understand the concept of enforcement as it make sure objects of different classes may not be interchanged as follows:

   Example: Vegetable v; Fruit f; Mango m;

   This implies 'v' is variables of class Vegetable, 'f' of class Fruit and 'm' of class Mango. Typing ensures that value of 'f' cannot be assigned to 'v'. However, if Mango extends Fruits, then 'f' can be assigned a value of 'm'. The variable of a sub class can be assigned to variable of super class, but not the other way around, and 'm' cannot be assigned a value of 'f'.

3) Using the OOA technology can yield many benefits, such as:

   i) Reusability of code
   ii) Productivity is gained gains through direct mapping
   iii) Maintainability, through this simplified mapping to the real world is possible.

4) To define the Problem Statement of a system, define what is to be done, and how you are going to implement that. Your statement should include the mandatory features, as well the optional features of the system to be developed.

   You should include:
   What is the problem and its scope,
   What is needed
   Application context
   Assumptions
   Performance needs.

Check Your Progress 2

1) A list of terms which will be used by end users to describe the state and behaviour of objects in the system.

Different user classes may require different mental models of the objects in the system. This includes:
What type of objects there are (user objects).
What information the user can know about an object of a particular type (user object attributes).
How the objects may be related to other objects (relationships).
Object types with 'subtypes' which have additional specialised actions or attributes, i.e., User object, Container objects, User object action, User object subtype.
A model of the business objects which end users believe interact with in a GUI system.

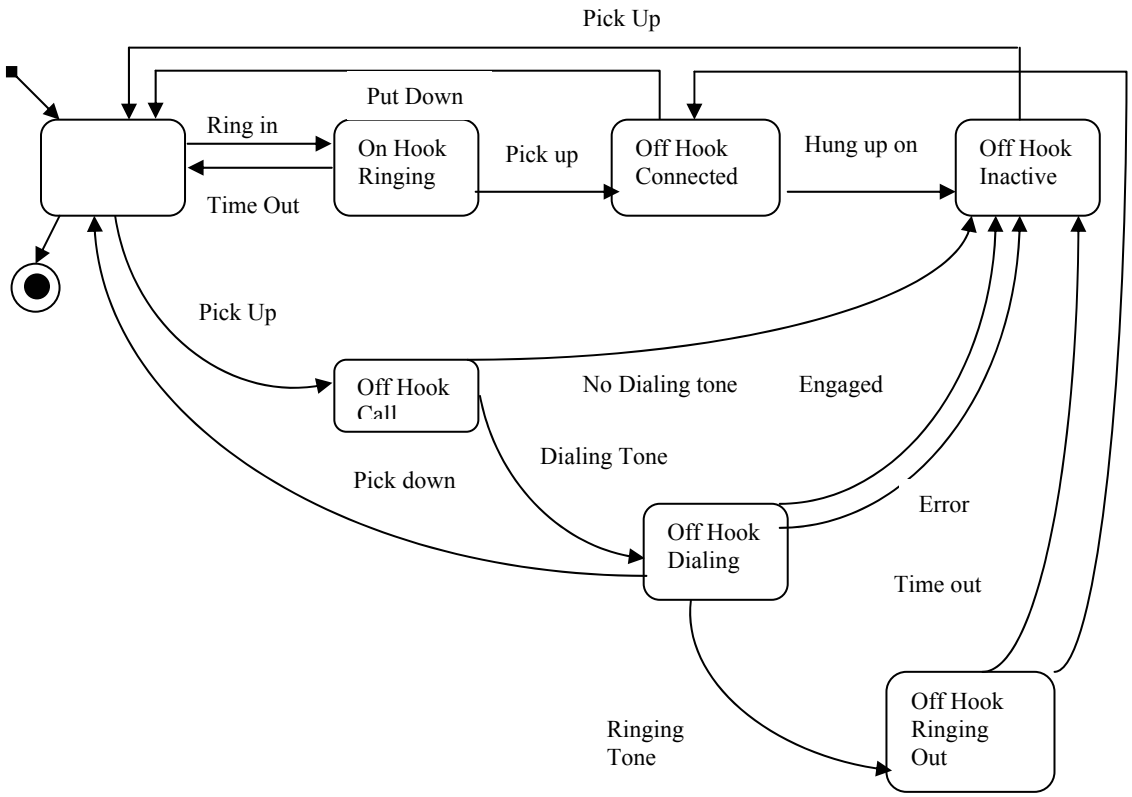


Figure 9: Dynamic Model of Telephone

Check Your Progress 3

1) To simplify the operation, one should use inheritance, where possible, to reduce the number of distinct operations. Introduce new superclasses as needed to simplify the operations, but they should not be forced or unnatural.
Locate each operation at the correct level in the class hierarchy.

2) For example, in the bank the activity, *verify account code* and *verify password*

3) For the creation of a saving account, you may write:
bank::: create –savings-account (customer)->account.
For the operation of checking account, you can write:
bank:: :create-checking-account

4) The final model serves as the basis for system architecture, design, and implementation.

5) The iterative analysis is required to get cleaner, more understandable, and coherent design.

UNIT 3 USING UML

Structure **Page Nos.**

3.0 Introduction 39
3.1 Objectives 40
3.2 UML: Introduction 40
3.3 Object Modeling Notations: Basic Concepts 41
3.4 Structural Diagram 47
3.4.1 Class Diagram
3.4.2 Object Diagram
3.4.3 Component Diagram
3.4.4 Deployment Diagram
3.5 Behavioral Diagrams 50
3.5.1 Use Case Diagram
3.5.2 Interaction Diagram
3.5.3 Activity Diagram
3.5.4 Statechart Diagram
3.6 Modeling with Objects 55
3.7 Summary 56
3.8 Solutions/Answers 56

3.0 INTRODUCTION

One of the major issues in software development today is quality. Software needs to be properly documented and implemented. The notion of software architecture was introduced for dealing with software quality. For successful project implementation the three essential components are: process, tools and notations. The notation serves three roles:

- as the language for communication,
- provide semantics to capture strategic and tactical decisions,
- to offer a form that is concrete enough to reason and manipulate

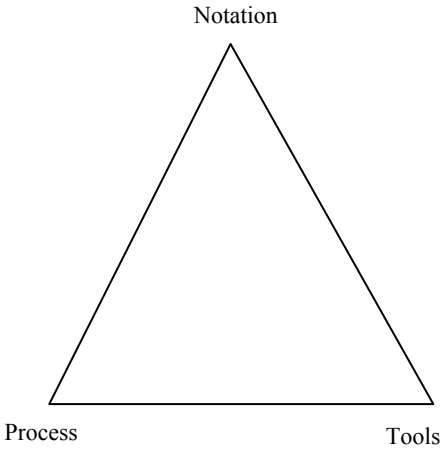


Figure 1: Components of project implementation

Architectural description languages (ACLs) have been developed for architectural description in analysis and design process. Important architectural description languages are Rapide, Unicorn, Asesop, Wright, ACME, ADML and UML. Currently, Universal Modeling Language (UML) is a *de facto* standard for design and description of object oriented systems, and includes many of the artifacts needed for architectural description, such as like processes, nodes, views, etc.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- trace the development of UML;
- identify and describe the notations for object modeling using UML;
- describe various structural and behavioral diagrams;
- list the characteristics of various static and dynamic diagrams, and
- understand the significance of different components of UML diagrams.

In this Unit we will discuss object modeling notations, structured diagrams and behavioral diagrams of systems.

3.2 UML: INTRODUCTION

The Unified Modeling Language (UML) is used to express the construct and the relationships of complex systems. It was created in response to a request for proposal (RFP) from the Object Management Group (OMG). Earlier in the 1990s, different methodologies along with their own set of notations were introduced in the market. The three prime methods were OMT (Rumbaugh), Booch and OOSE (Jacobson). OMT was strong in analysis, Booch was strong in design, and Jacobson was strong in behavioral analysis. UML represents unification of the Booch, OMT and OOSE notations, as well as are the key points from other methodologies. The major contributors in this development shown in *Figure 2*.

UML is an attempt to standardize the artifacts of analysis and design consisting of semantic models, syntactic notations and diagrams. The first draft (version 0.8) was introduced in October 1995. The next two versions, 0.9 in July 1996 and 0.91 in October 1996 were presented after taking input from Jacobson. Version 1.0 was presented to Object Management Group in September 1997. In November 1997, UML was adopted as standard modeling language by OMG. The current version while writing this material is UML 2.0.

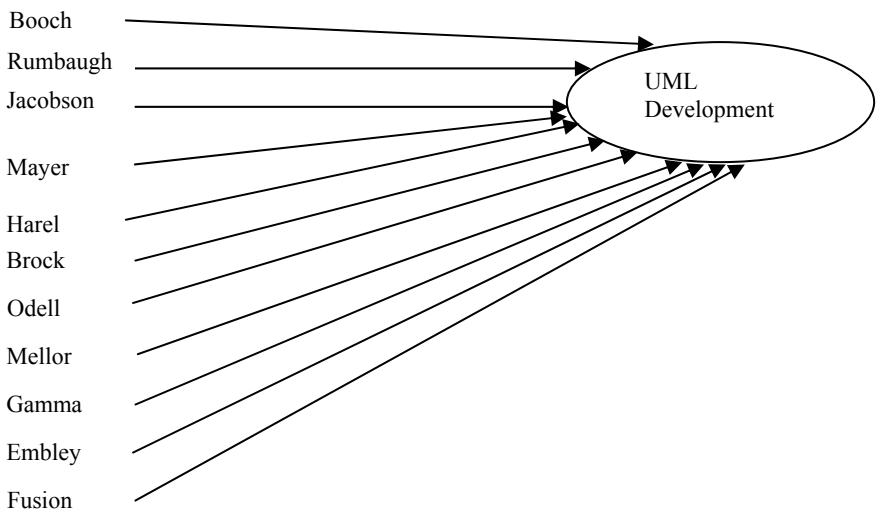


Figure 2: The Input for UML development

The major features of UML are:

- defined system structure for object modeling
- support for different model organization
- strong modeling for structure and behavior
- clear representation of concurrent activities
- support for object oriented patterns for design reuse.

The model for the object oriented development could be shown as in *Figure 3*. It could be classified as static/dynamic and logical/physical model.

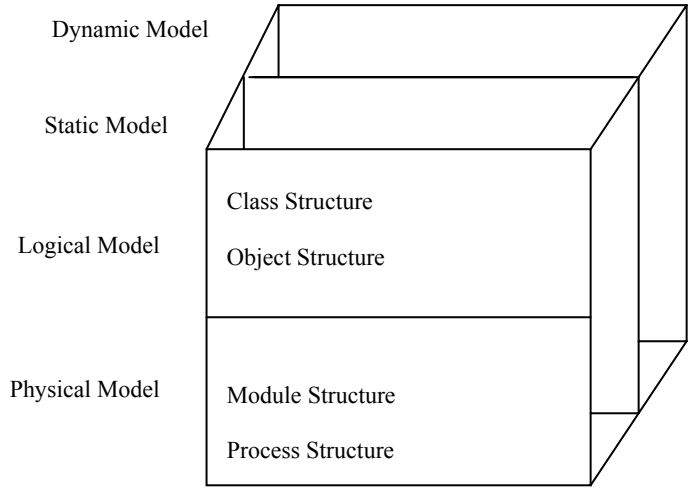


Figure 3: The Model for Object oriented development

The Logical view of a system serves to describe the existence and meaning of the key abstractions and the mechanism that form the problem space, or that define the system architecture.

The Physical model describes the concrete software and hardware components of the system's context or implementation.

UML could be used in visualizing, specifying, constructing and documenting object oriented systems. The major building blocks of UML are structural, behavioral, grouping, and annotational notations. Let us discuss these blocks, one by one.

a. **Structural Notations:** These notations include static elements of a model. They are considered as **nouns** of the UML model which could be conceptual or physical. Their elements comprises class, interface, collaboration, use case, active class, component, and node. It also includes actors, signals, utilities, processes, threads, applications, documents, files, library, pages, etc.

b. **Behavioral Notations:** These notations include dynamic elements of a model. Their elements comprises interaction, and state machine. It also includes classes, collaborations, and objects.

c. **Grouping Notations:** These notations are the boxes into which a model can be decomposed. Their elements comprises of packages, frameworks, and subsystems.

d. **Annotational Notations:** These notations may be applied to describe, illuminate, and remark about any element in the model. They are considered as explanatory of the UML. Their elements comprised of notes which could be used for constraints, comments and requirements.

UML is widely used as it is expressive enough, easy to use, unambiguous and is supported by suitable tools.

3.3 OBJECT MODELING NOTATIONS: BASIC CONCEPTS

A **system** is a collection of subsystems organised to accomplish a purpose and described by a set of models from different viewpoints.

A **model** is a semantically closed abstraction of a system which represents a complete and self-consistent simplification of reality, created in order to better understand the system.

A **view** is a projection into an organisation and structure of a system's model, focused on one aspect of that system.

A **diagram** is a graphical presentation of a set of elements.

A **classifier** is a mechanism that describes structural and behavioral features. In UML the important classifiers are class, interface, data type, signals, components, nodes, use case, and subsystems.

A **class** is a description of a set of objects that share the same attribute, operations, relationships, and semantics. In UML, it is shown by a rectangle.

An **attribute** is a named property of a class that describes a range of values that instances of the property may hold. In UML, they are listed in the compartment just below that class name.

An **operation** is the implementation of a service that can be requested from any object of the class to affect behavior. In UML, they are listed in the compartment just below that class attribute.

The notation for class, attribute, and operations is shown as:

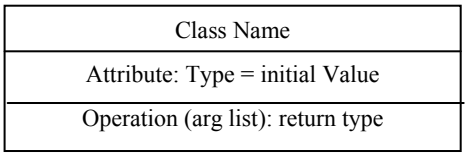


Figure 4: Class with attributes and operations

An **interface** is a collection of operations that are used to specify a service of a class or a component.

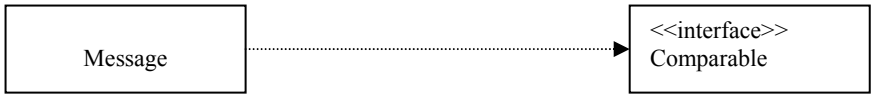


Figure 5: Realizing an interface

A **signal** is the specification of an asynchronous stimulus communicated between instances.

A **component** is a physical and replaceable part of the system that confirms to, and provides the realization of a set of interfaces. In UML, it is shown as a rectangle with tabs. The notation for component and interface is shown as:

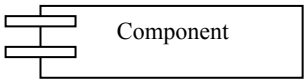


Figure 6: Component

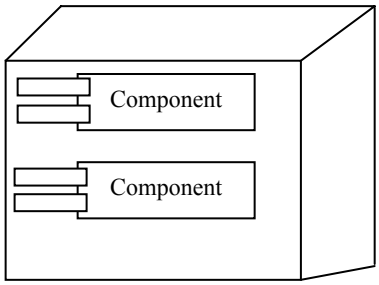


Figure 7: Components and Server

A **node** is a physical element that exists at runtime, and represents a computational resource generally having a large memory and often process capability. In UML, it is shown as a cube. The notation for node is shown as:

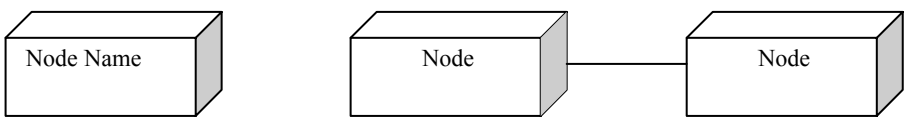


Figure 8: Node and relationship between nodes

A **use case** is a description of a set of sequence of actions that a system performs to yield an observable result that is of a value to an actor. The user is called actor and the process is depicted by use case. The notation for use case is shown as:

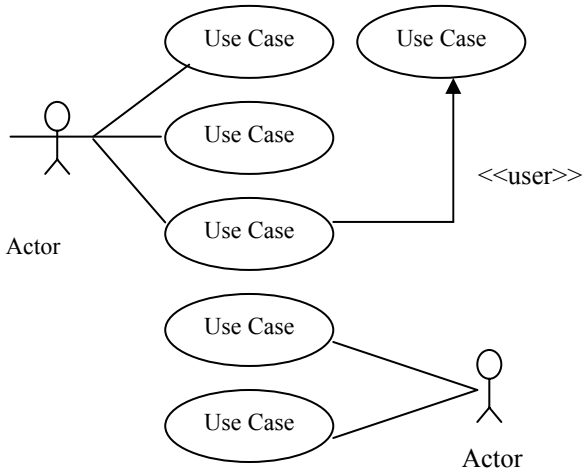


Figure 9: Relationship between actor and use case

A **subsystem** is a grouping of elements of which some constitute a specification of the behavior offered by other contained elements.

An **object** is an instance of a class. The object could be shown as the instance of a class, with the path name along with the attributes. Multiple objects could be connected with links. The notation for unnamed and named objects, object with path name, active objects, object with attributes, multiple objects, and self linked object is shown as:

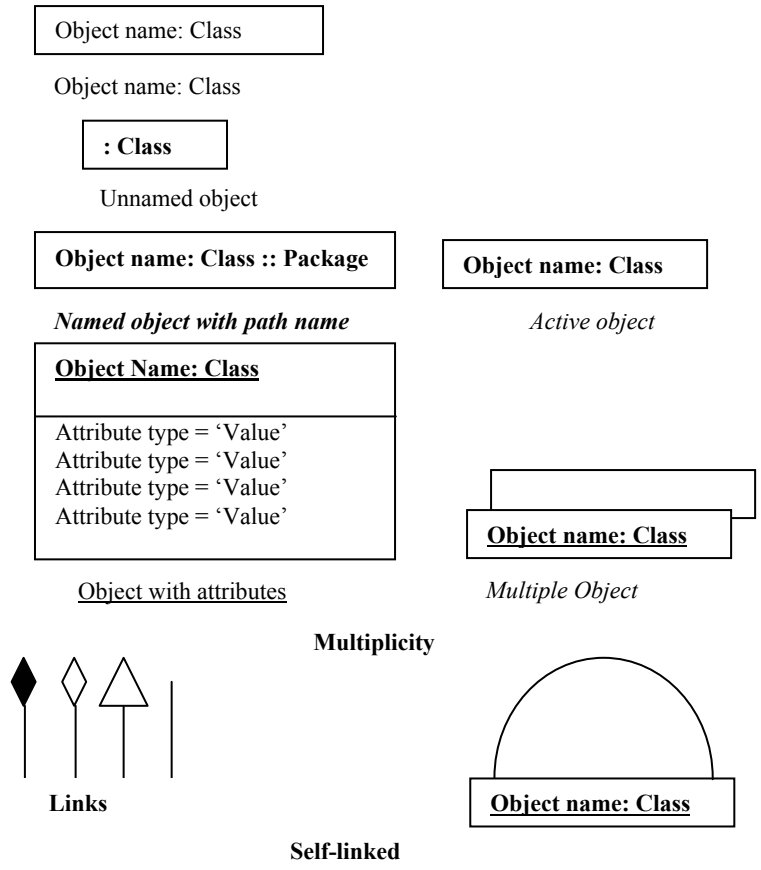


Figure 10: Different types of objects

A **package** is a general purpose mechanism for organising elements into groups. It can also contain other packages. The notation for the package shown contains name

and attributes as shown in *Figure11*. Packages are used widely in a Java based development environment. You may refer to the Unit 3 of Block 2 of the MCS-024 course for more details about packages.

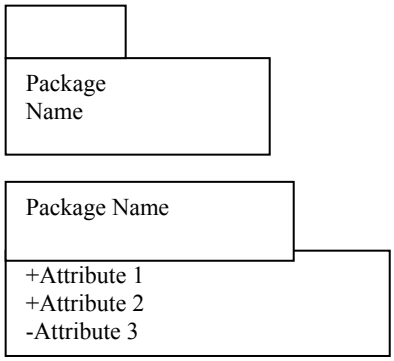


Figure 11: Package Diagram

A **collaboration** is a society of classes, interfaces and other elements that work together to provide some cooperative behavior that is bigger than the sum of its parts.

A **relationship** is a connection among things. In object models, the common types of relationships are inheritance, dependency, aggregation, containment, association, realisation, and generalisation. The notation for relationship between use cases is shown as:

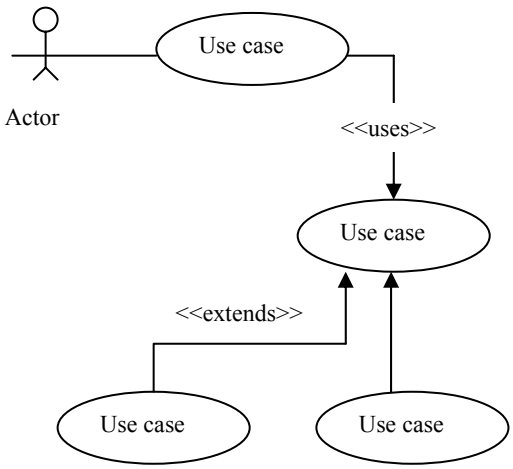


Figure 12: Relationship between different use case

Relationships exists in many forms. The notation for different form of relationship is shown as:

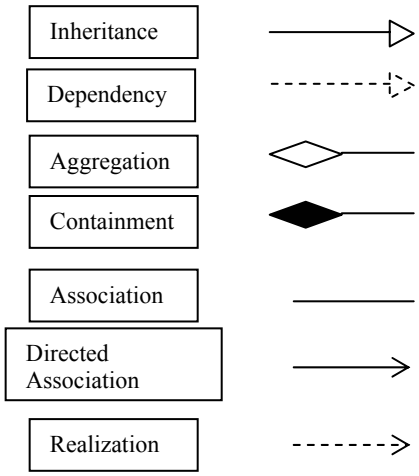


Figure 13: Common relationship types

A **dependency** is a relationship that states, that a change in specification of one thing may affect another thing, but not necessarily the reverse. In UML, it is shown as a dashed directed line. The notation for dependency between components and packages is shown as:

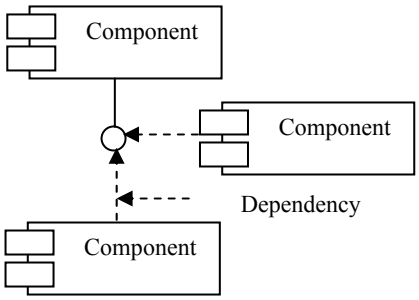


Figure 14: Dependency between components

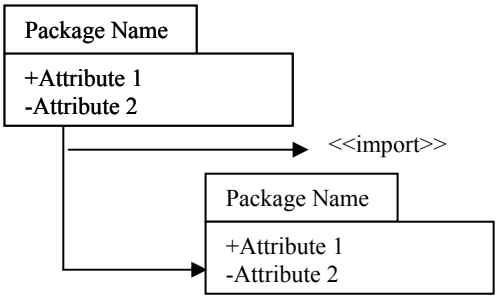


Figure 15: Dependency between packages

A **generalization** is a relationship between a general thing and a specific kind of thing. It is also called "is-a-kind-of" relationship. Inheritance may be modeled using generalization. In UML, it is shown as a solid directed line with a large open arrow pointing to the parents.

An **association** is a structural relationship that specifies that the objects of one thing are connected with the objects of another. In UML, it is shown as a solid line connecting same or different class. The notation for association between nodes is shown as:

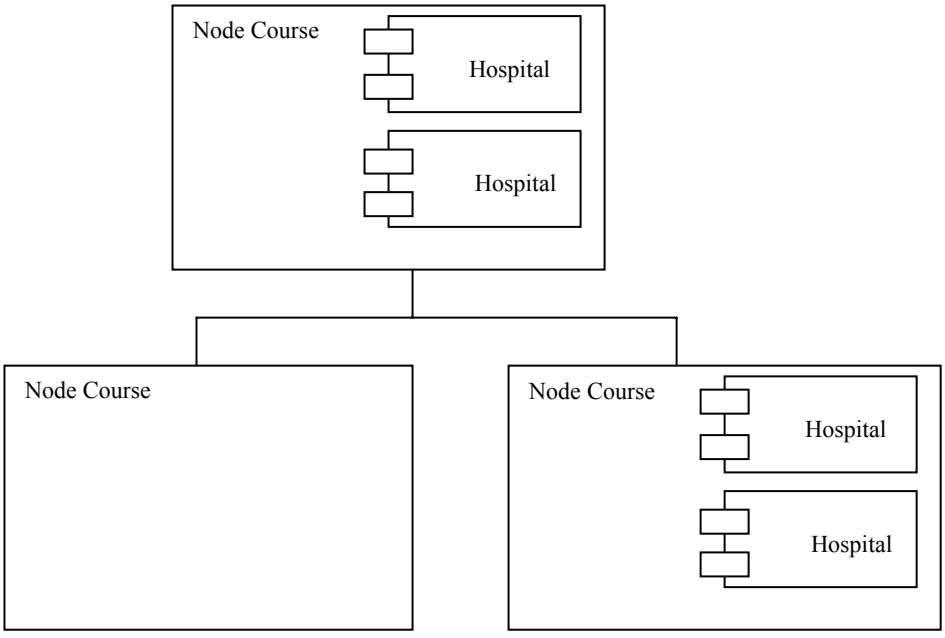


Figure 16: Association between nodes

The four enhancements that apply to association are name, role, multiplicity, and aggregation. Each class participating in an association has a specific role which is specified at the rear end of the association.

Multiplicity specifies how many objects may be connected across an instance of an association which is written as a range of values (like 1..*). The notation for roles and multiplicity between classes is shown as:

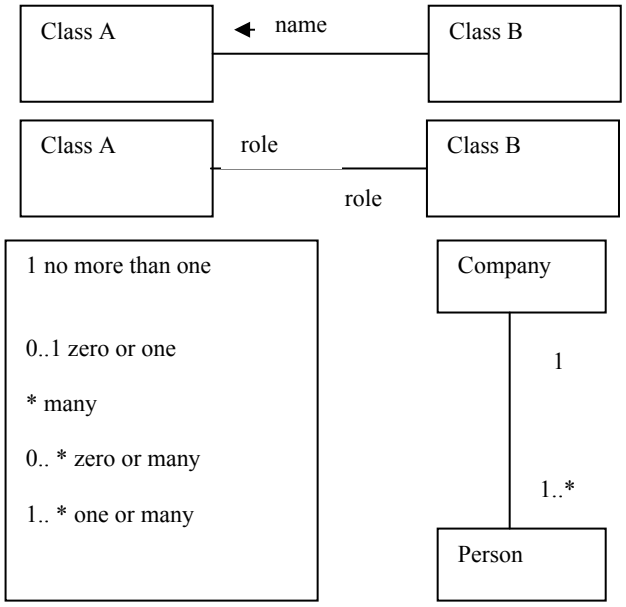


Figure 17: Various roles and multiplicity defined with association

An **aggregation** is a structural relationship that specifies that one class represents a large thing which constitute of smaller things. It represents "has-a" relationship. In UML, it is shown as association with an open diamond at the large end. The notation for aggregation is shown as:

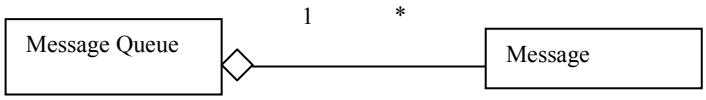


Figure 18: Aggregation

A **state** encompasses all the properties of the object along with the values of each of these properties.

An **instance** is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effect of the operation.

A **transition** is a relationship between two states indicating that an object in the first state will perform certain action and enter the second state when a specific event occurs and specific conditions are satisfied.

A **note** is a graphical symbol for rendering constraints or comments attached to an element or collection of elements.

☞ Check Your Progress 1

1) Which of the following is not a valid Architectural Definition language?

   a) Rapide b) ACME
   c) UML d) Pascal

   ……

   ……...

2) OMT is

a) Object Methodology Gateway
b) Objective Methodology Gateway
c) Object Management Gateway
d) Object Management Group

……

……..……..

3) Object Oriented Software Engineering is given by

a) Booch
b) Rumbaugh
c) Jacobson
d) None of these

……

…..……..

4) Booch was strong in

a) Analysis
b) Design
c) Implementation
d) Software engineering

……

5) Which of the following is not a valid UML notations?

a) Behavioral
b) Grouping
c) Transactional
d) Annotational

……

……

For object modeling some standard notations are used. Now let us discuss these basic notations. A well-defined logical notation is important in the software development process. It helps the software architect to clearly establish the software architecture and implement module integration. In order to define the commonly used diagrams in UML, it is essential to understand the basic concepts of the object modeling.

3.4 STRUCTURAL DIAGRAMS

The main purpose of structural diagram is to visualize, specify, construct and document the static aspects of a system. Their elements comprised of class, interface, active class, component, node, processes, threads, applications, documents, files, library, pages etc. The four main structural diagrams are class, object, component and deployment diagram.

3.4.1 Class Diagram

A class diagram is used to support functional requirement of system. In a static design view, the class diagram is used to model the vocabulary of the system, simple collaboration, and logical schema. It contains sets of classes, interfaces, collaborations, dependency, generalization and association relationship. The notation for classes and the relationship between classes is shown as:

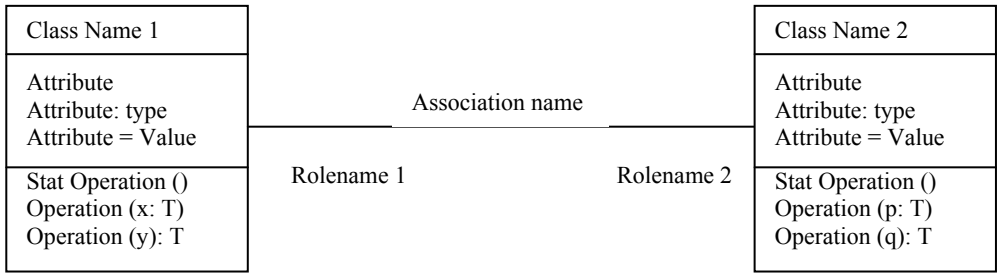


Figure 19: Relationship among classes

If in any college, there are limited classrooms that have to be allocated to different classes and instructors are fixed for all classes, then the class diagram for the allocation of classrooms and instructors is shown as:

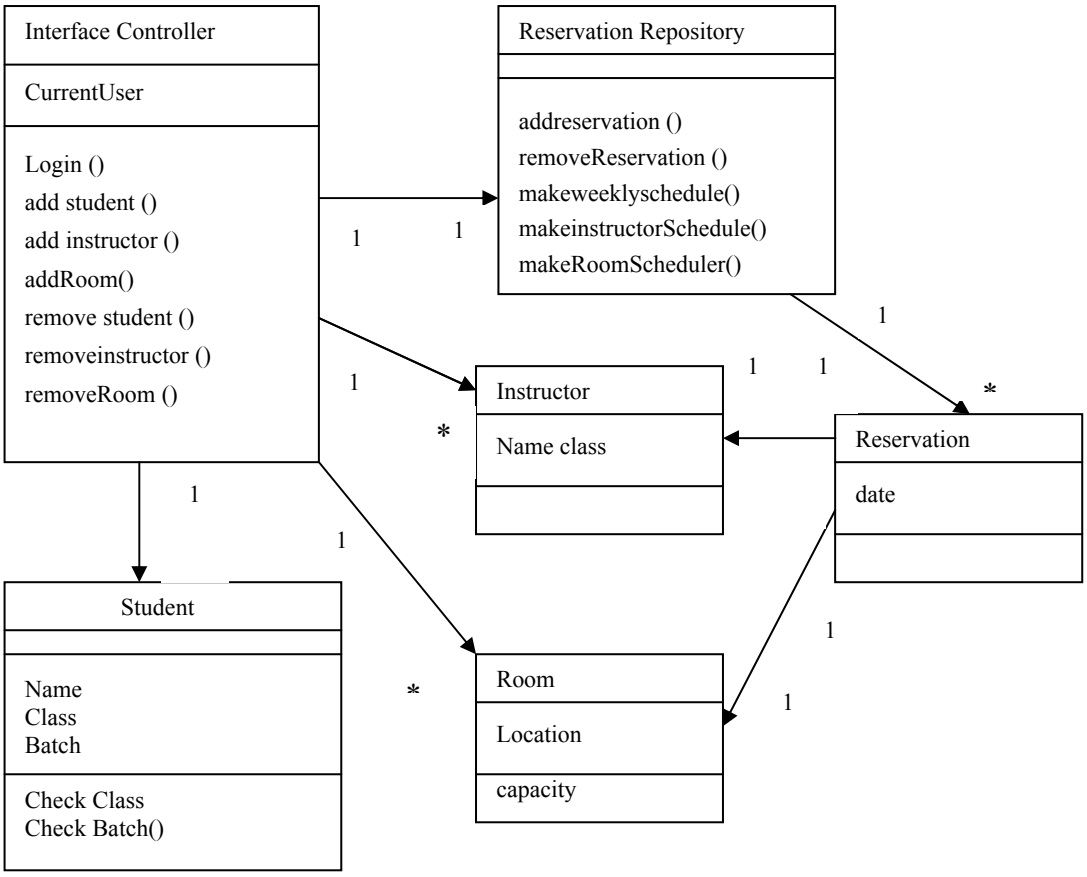


Figure 20: Class diagram for a class room scheduling system

3.4.2 Object Diagram

An object diagram shows a set of objects and their relationships at a point of time. In a static view, the object diagram is used to model interactions that consist of objects that collaborate the without any message passed among them. It contains name, graphical contents, notes, constraints, packages and subsystems. The notation for objects and the relationship between objects is shown as:

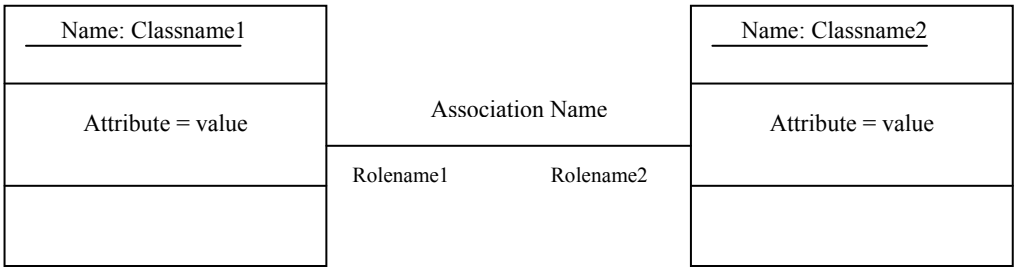


Figure 21: Relationship among objects

3.4.3 Component Diagram

A component diagram shows a set of component and their relationships. In a dynamic model, the component diagram is used to model physical components such as executable releases, libraries, databases, files or adaptable systems. It contains components, interfaces, packages, subsystems, dependency, generalization, association, and relationship. The notation for components and relationship between components is shown as:

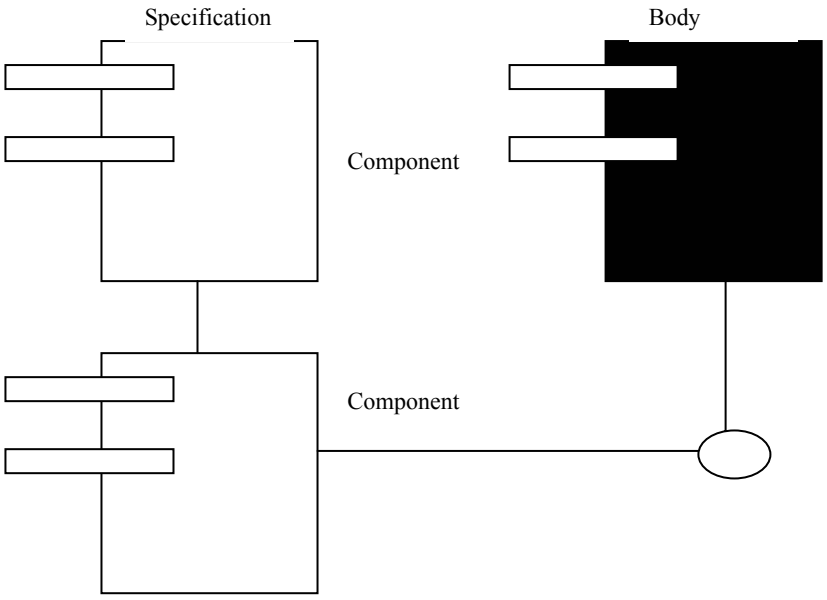


Figure 22: Relationship among components

The component diagram for ATM is shown as:

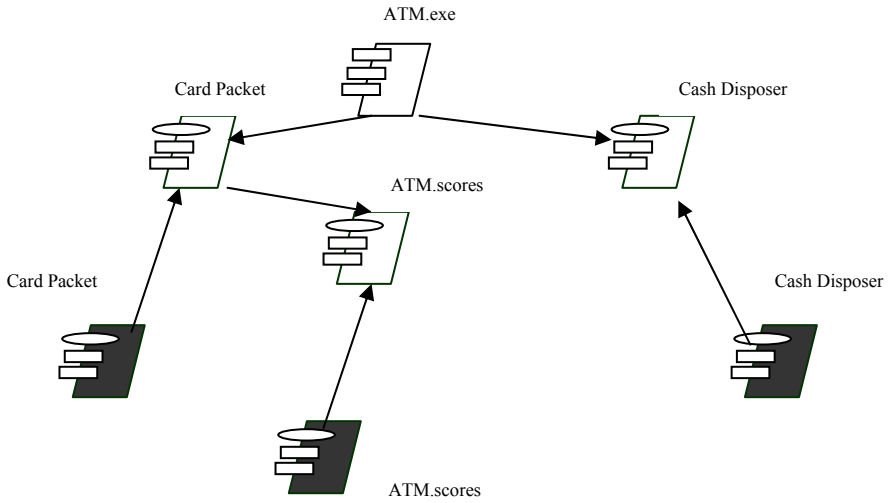


Figure 23: Component diagram for ATM

3.4.4 Deployment Diagram

A deployment diagram shows all the nodes on the network, their interconnections, and processor execution. In a dynamic model, a deployment diagram is used to represent computational resources. The notation for nodes and relationship between processors and devices is shown as:

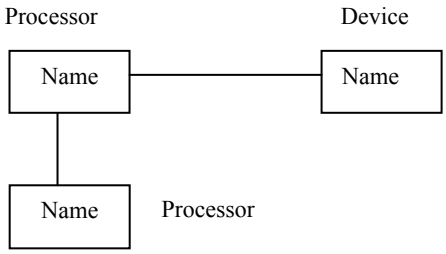


Figure 24: Relationship among nodes

The deployment diagram for student administration is shown as:

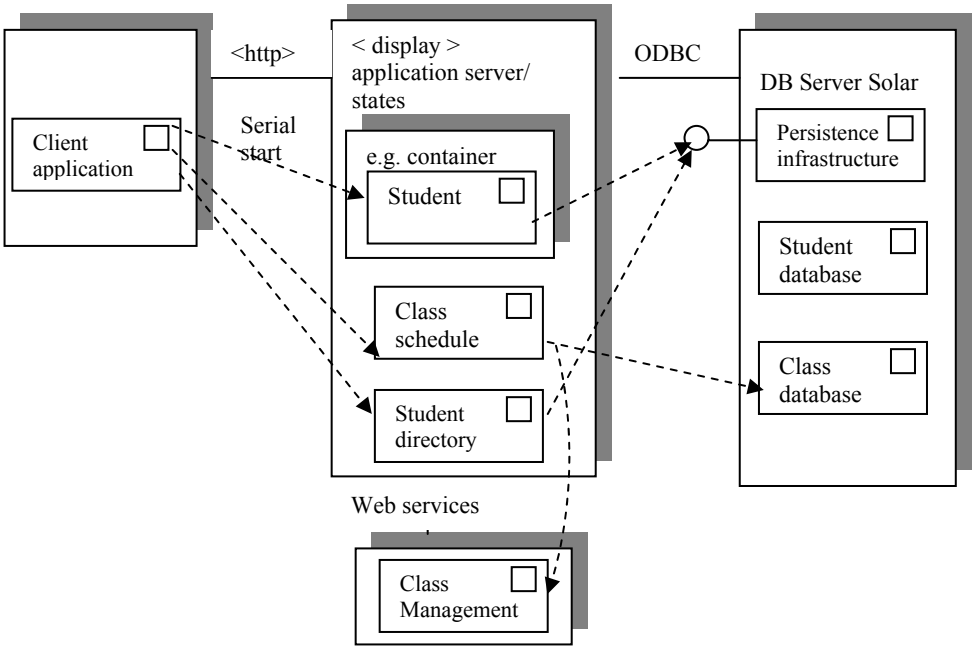


Figure 25: Deployment diagram for Student Administration

Now you are familiar with structured diagram. To represent dynamic aspect of the structured system, behavioral diagrams are used. In the next section, we will study various behavioral diagrams.

3.5 BEHAVIORAL DIAGRAMS

The main purpose of behavioral diagrams is to visualize, specify, construct, and document the dynamic aspects of a system. The interaction between objects indicating the flow of control among them is shown using these diagrams. The flow of control may encompass a simple, sequential thread through a system, as well as complex flows that involves branching, looping, recursion and concurrency. They could model time ordering (sequence diagram) or sequence of messages (collaboration diagram). The four main behavioral diagrams are: use case, interaction, activity and statechart diagram.

3.5.1 Use Case Diagram

A use case diagram shows a set of use cases, actors, and their relationships. These diagrams should be used to model the context or the requirement of a system. It contains use cases, actors, dependency, generalization, association, relationship, roles, constraints, packages, and instances. The use case diagram makes systems, subsystems, and classes approachable by presenting an outside view of how the elements may be used in context. The notation for use cases and relationship among use cases, base cases and extend cases is shown as:

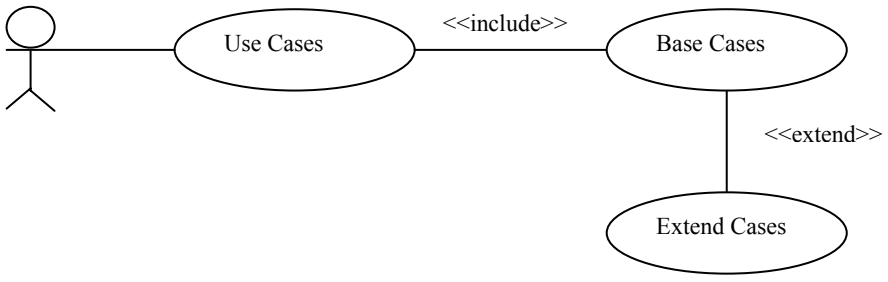


Figure 26: Relationship among use cases

3.5.2 Interaction Diagram

An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. These diagrams should be used to model the dynamic aspect of the system. It includes sequence diagrams and collaboration diagrams. Here we will discuss two interaction diagrams, sequence diagrams and collaboration diagrams.

3.5.2.1 Sequence Diagrams

A sequence diagrams are interaction diagrams that emphasize the time ordering of messages. In UML it is shown as a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. It has a global life line and the focus of control. An object life line is the vertical dashed line that represents existence of an object over a period of time. The focus of control is tall and thin rectangle that shows the period during which an object is performing an action. The notation for depiction of sequence among objects with certain conditions is shown as:

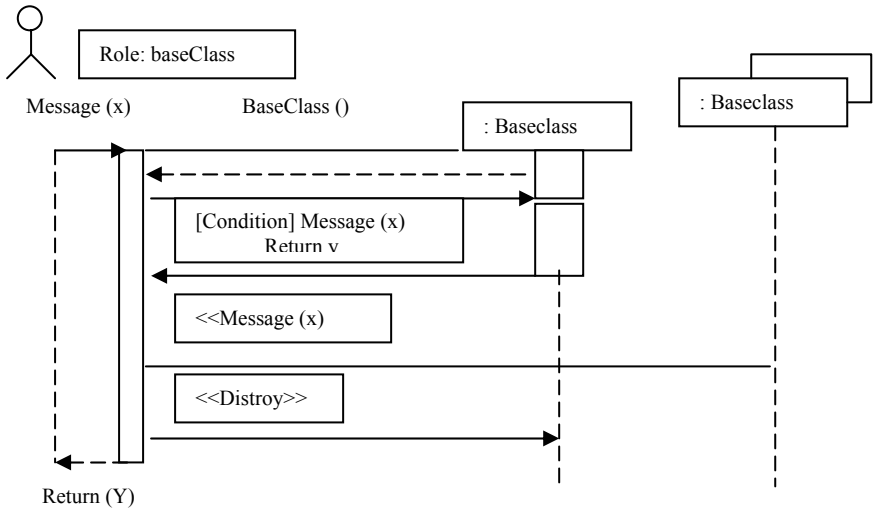


Figure 27: Sequence diagram

The sequence diagram for sending the document along the network is shown as:

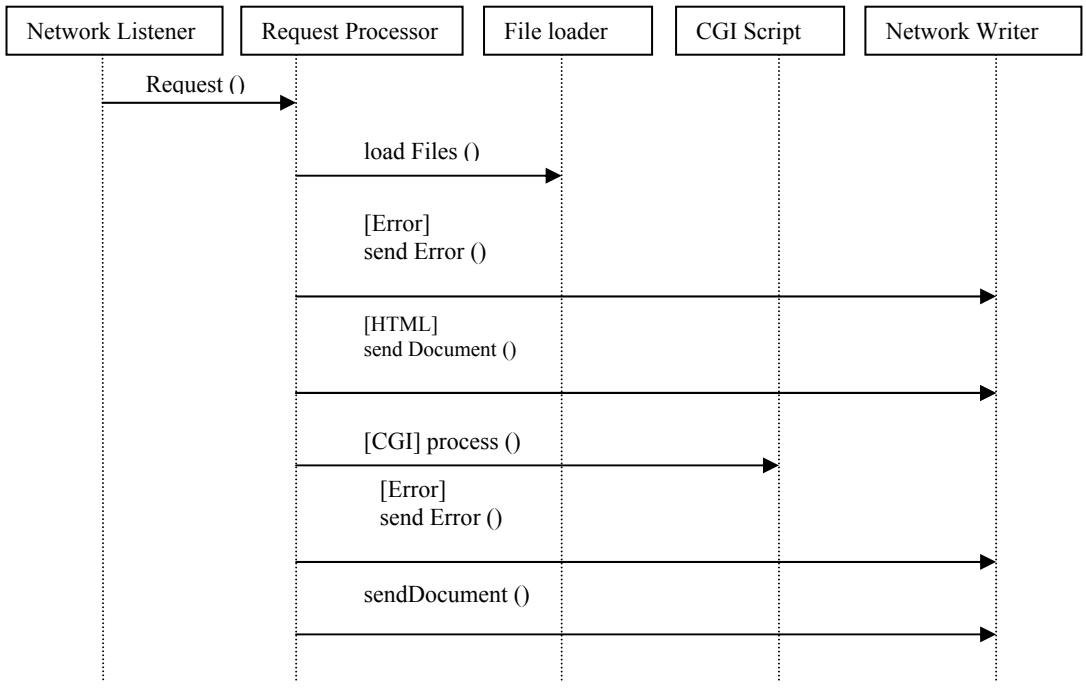


Figure 28: Sequence diagram for sending document

3.5.2.2 Collaboration Diagrams

Collaboration diagrams are interaction diagrams that emphasize the structural organisation of an object that send and receive messages. There is always a path in collaboration diagrams to indicate how one object is linked to another, and sequence numbers to indicate the time ordering of a message. The notation for depiction of a collaboration diagram is shown as:

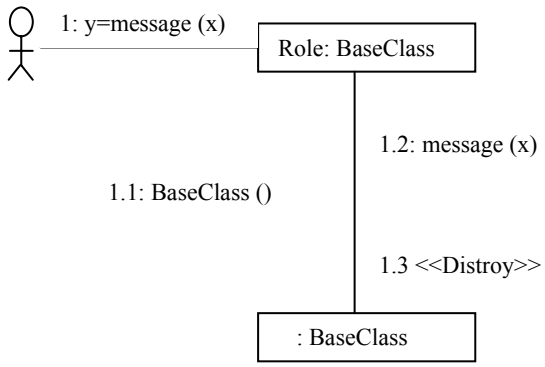


Figure 29: Collaboration diagram

The collaboration diagram for the execution using J2ME and EJB from the remote database is shown as:

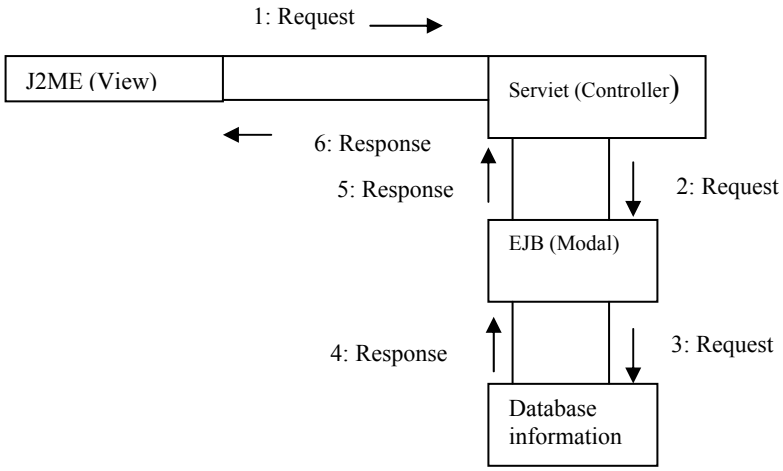


Figure 30: Collaboration diagram for execution using J2ME, Servlet and EJB

3.5.3 Activity Diagram

Activity diagrams show the flow from one activity to another. An activity is an ongoing non atomic execution within a state machine. Activity ultimately results in some action, which is made up of executable atomic computations that result in a change in state of the system, or the return of a value. It contains activity states, action states, transition states, and objects. The activity diagram for encryption of a message send through e-mail is shown as:

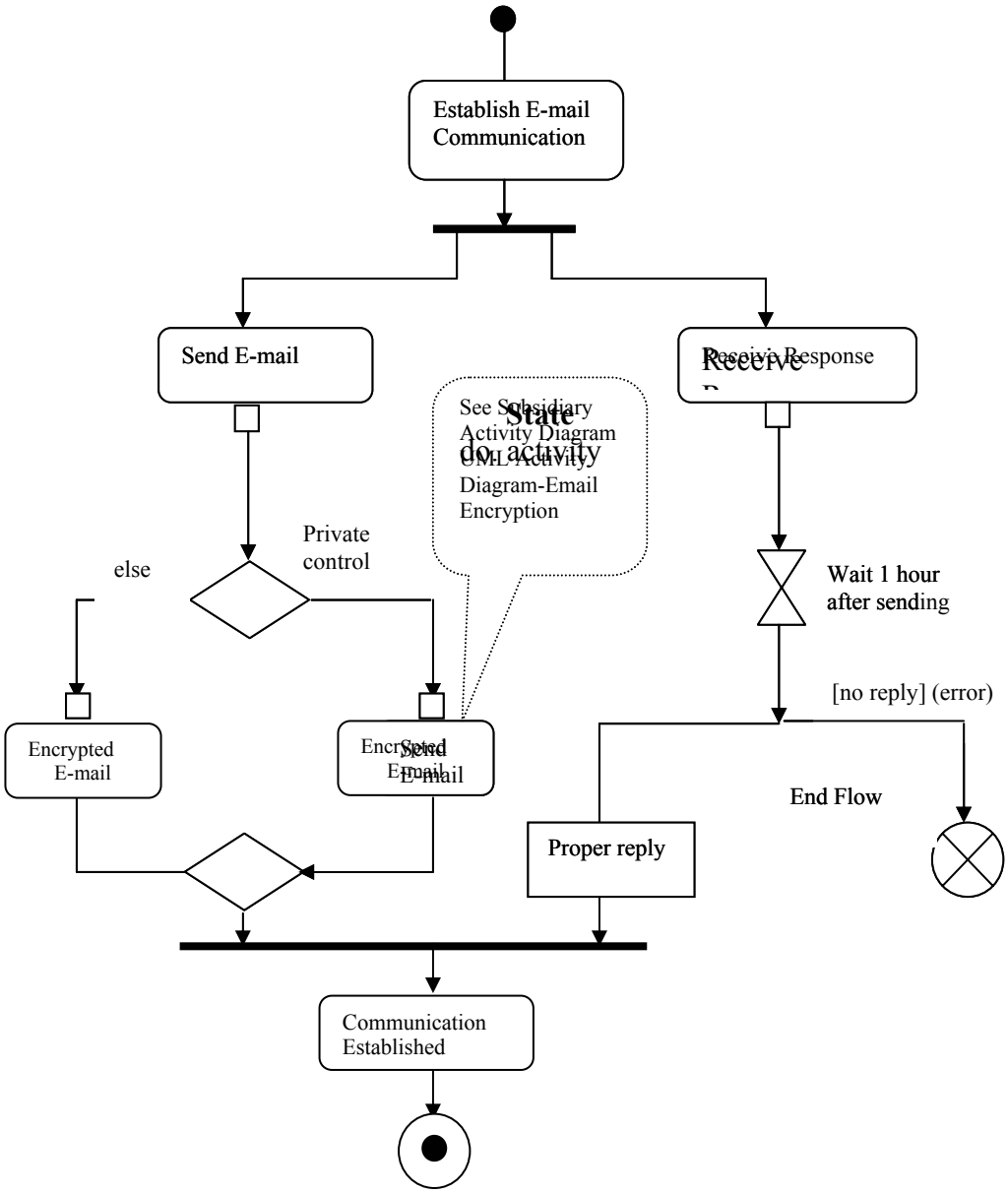


Figure 31: Activity diagram for E-mail encryption

3.5.4 Statechart Diagram

A state chart shows a state machine, emphasizing the flow of control from one state to another. A state machine is a behaviour that specifies the sequence of states that an object goes through during the life time in response to events together with its response to those events. A state is a condition/situation during the object's life which performs some activity, or waits for some event. It contains simple states, composite states, transitions, events, and actions. The notation for multiple states depending on events/action based on set of activities is shown as:

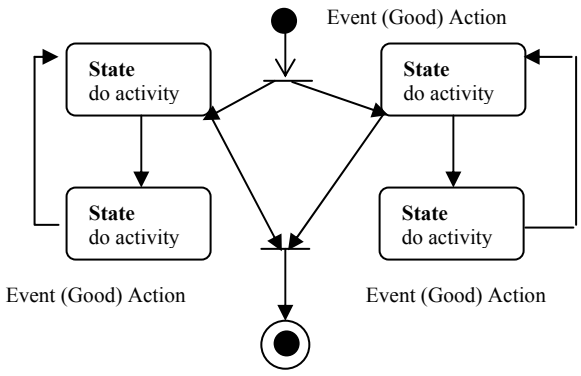


Figure 32: State diagram for multiple activities

☞ Check Your Progress 2

1) In a class diagram, a class is denoted by

a) rectangle b) circle
c) ellipse d) oval

……

……

2) An actor in use case diagrams is a

a) process b) subprogram
c) users d) comments

……

……

3) Which of the following diagram have to be numbered to understand their order?

a) Object b) Collaboration
c) Component d) Deployment

……

……

4) Which of the following diagram is used for physical layer?

a) class b) object
c) use case d) component

……

……

5) A particular state in a statechart diagram is denoted by

a) rectangle b) circle
c) ellipse d) oval

……

……

6) A bull's eye icon is used to represent _____ in a statechart diagram

a) initial state b) action
c) final state d) event

……

……

7) Which of the following diagram's is used for dynamic modeling?

a) class b) object
c) use case d) interaction

……

……

8) Which view plays a special role integrating the contents of other views?

a) Use case view b) Process view
c) Design view d) Implementation view

……

……

……

3.6 MODELING WITH OBJECTS

A model is an abstract representation of a specification, design or system from a particular view. A modeling language is a way of expressing the various models produces during the development process. It is a collection of model elements. It is normally diagrammatic. It has

syntax – the rules that determine which diagrams are legal
semantics – the rules that determine what a legal diagram means.

A model is a semantically closed abstraction of a system compose of elements. It could be visualized using any of the following five views:

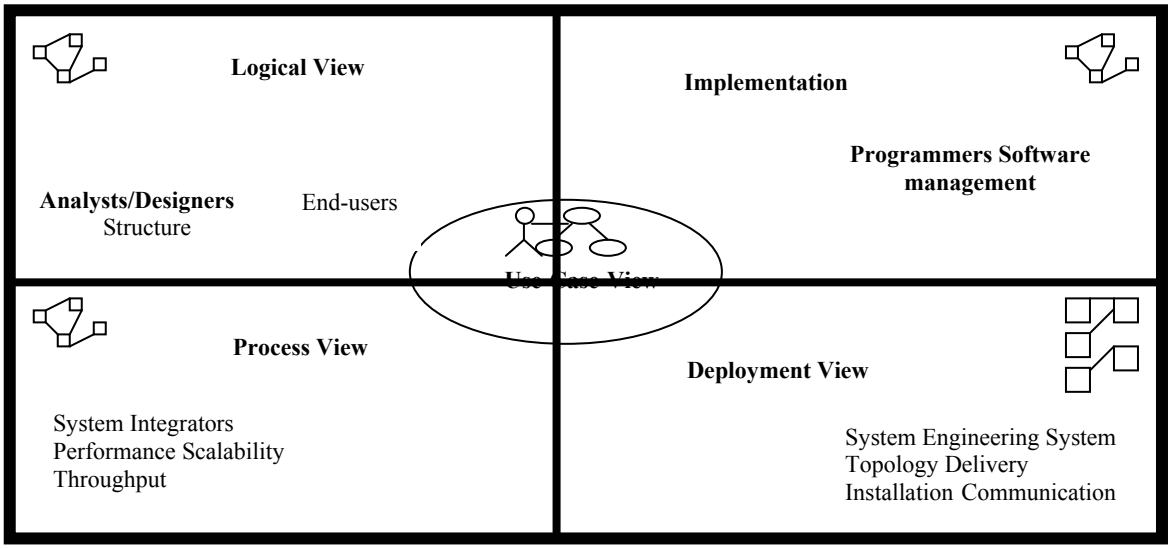


Figure 33: 4+1 View of Software Architecture

a. **Logical view** — This view is concerned with the functional requirements of the system. It is used early in the elaboration phase with the creation of class and packages, using a class diagram which may reflect the strategic dimension of the system.

b. **Implementation view** — This view focuses on the actual software module organisation within the developmental environment. It includes taking the derived requirement, software management, reuse, and constraints imposed by the program tools. The physical partitioning is done in this phase.

c. **Process view** — This involves the runtime implementation structure of the system. It contains requirements such as performance, reliability, scalability, integrity, synchronization, etc. Executable components are used here to show runtime components to map classes such as java applet, activeX component or DLL.

d. **Deployment view** — This view demonstrates mapping software to process nodes showing the configuration of runtime processing elements. It takes into account, requirements such as availability, reliability, performance and scalability. Major issues here are processor, architecture, speed, along with inter process communication, bandwidth and distributed facilities.

e. **Use case view** This view addresses and validates the logical, process, component, and deployment view.

For an iterative and incremental life cycle, the two criteria are time and process. The major components of showing a project development along with time scale, inception, elaboration, construction and transition. When the project is structured along with the process scale, then the major steps are business modeling, requirements, analysis and design, implementation, testing and deployment. The mapping between time and process scale can be shown diagrammatically for more clarity.

Using UML, it is possible to generate code in any programming language from UML model (called forward engineering) and reconstruct a model from an implementation into UML (called reverse engineering) as show in the *Figure 34* given below.

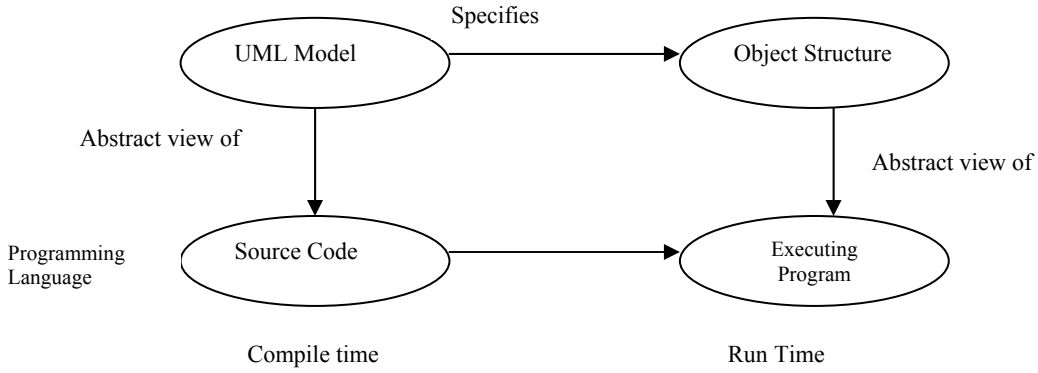


Figure 34: The relationship between models and code

☞ Check Your Progress 3

1) Create a use case diagram for a cell phonebook.

……

…..…….

2) Create a sequence diagram for a logon scenario.

……

…..…….

3.7 SUMMARY

This Unit provides an introduction to the basic concept of object modeling notations. The major diagrams used with UML have been discussed. An idea has been provided about different views and the corresponding diagrams. This Unit only contains the introduction of various diagrams, and the student is not expected to be an expert in designing every diagram.

3.8 SOLUTIONS /ANSWERS

Check Your Progress 1

1. d) 2. d) 3. c) 4. b)

Check Your Progress 2

1. a) 2. c) 3. b) 4. d)

5. d) 6 .c) 7.d) 8. a)

Check Your Progress 3

1)

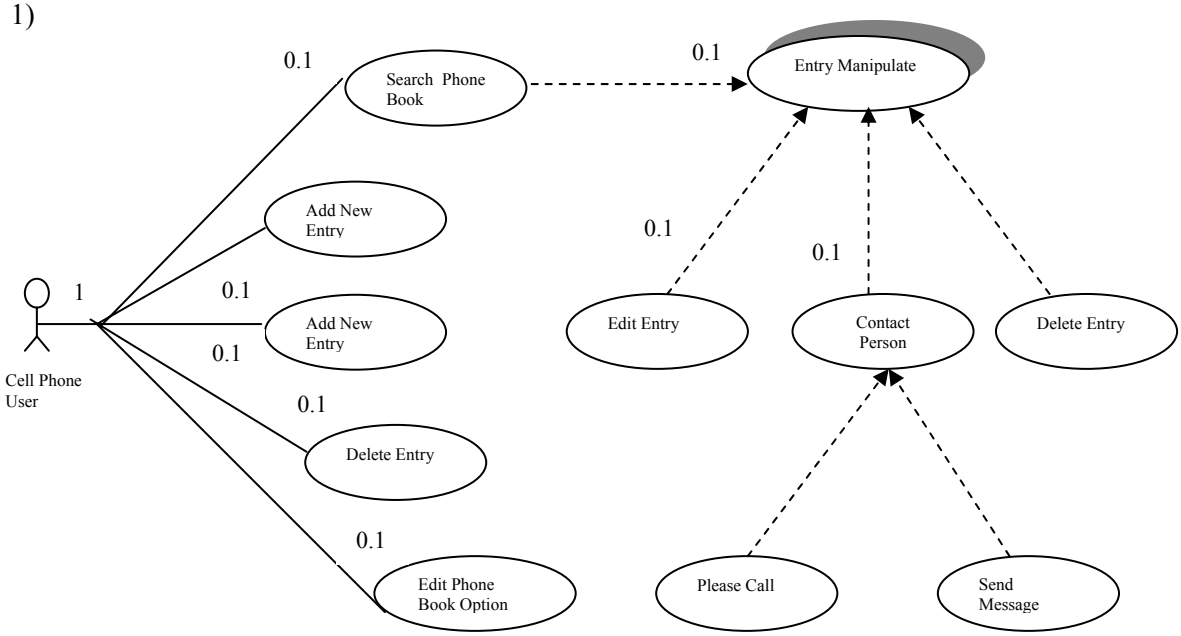


Figure 35: Use case diagram for a cell phonebook

2)

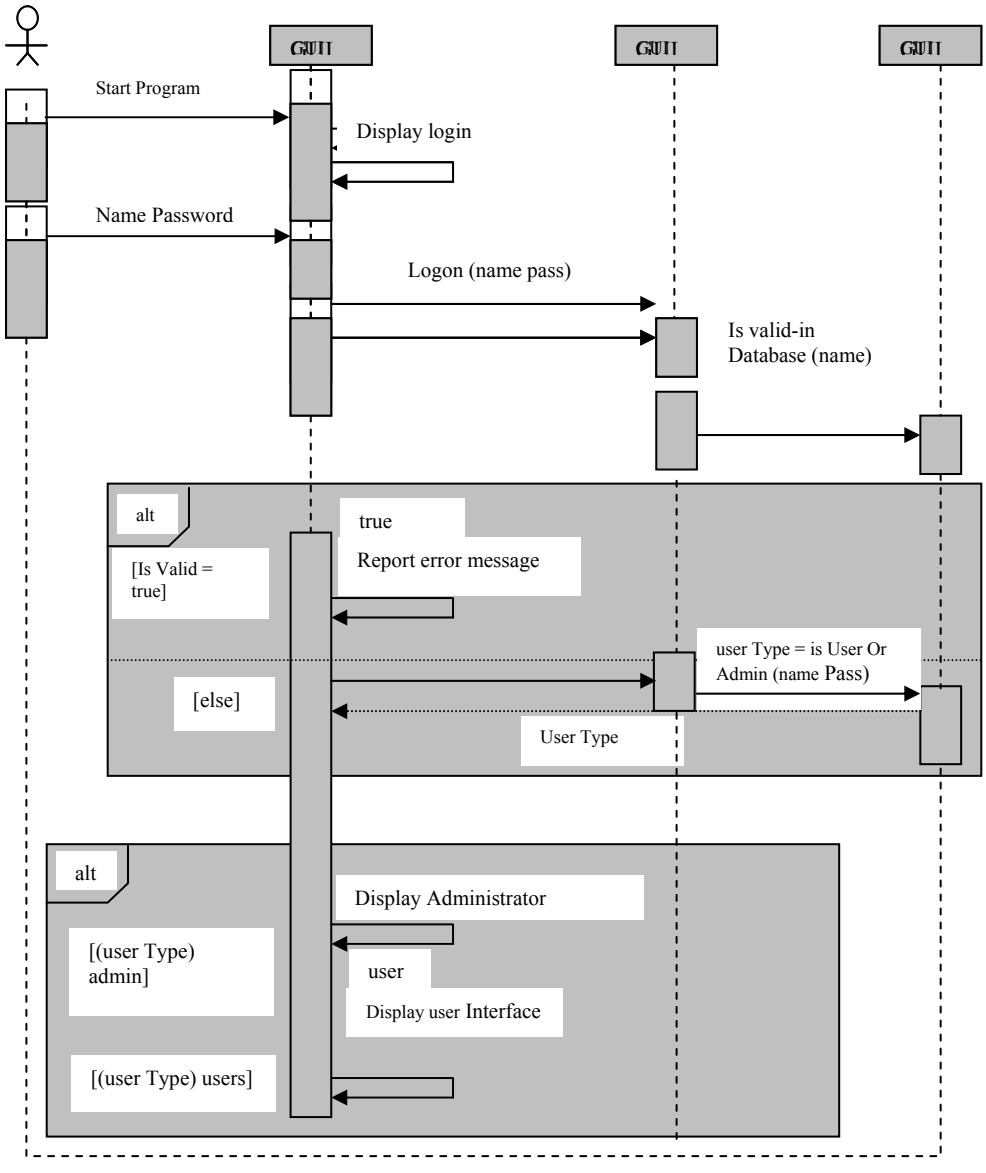


Figure 36: Sequence Diagram for logon scenario

UNIT 1 SYSTEM DESIGN

Structure **Page Nos.**

1.0	Introduction	5
1.1	Objectives	5
1.2	System Design: An Object Oriented Approach	6
1.3	Breaking into Subsystems	9
1.4	Concurrency Identification	11
1.5	Management of a Data Store	13
1.6	Controlling Events Between Objects	16
1.7	Handling Boundary Conditions	16
1.8	Summary	18
1.9	Solutions/Answers	18

1.0 INTRODUCTION

Object oriented design (OOD) is concerned with developing an object oriented model of a software system to implement the identified requirements. Many OOD methods have been described since the late 1980s. The most popular OOD methods include Booch, Buhr, Wasserman, and the HOOD method developed by the European Space Agency which can yield the following benefits: maintainability through simplified mapping to the problem domain, which provides for less analysis effort, less complexity in system design, and easier verification by the user. OOD also provides reusability, which saves time and costs, and productivity gains through direct mapping to features of Object-Oriented Programming Languages. Object Oriented Development (OOD) has been touted as the next great advance in software engineering. It promises to reduce development time, reduce the time and resources required to maintain existing applications, increase code reuse, and to provides a competitive advantage to organizations that use it. While the potential benefits and advantages of OOD are real, excessive hype has lead to unrealistic expectations among executives and managers. Even software developers often miss the subtle, but profound, differences between OOD and classic software development.

In this Unit we will discuss the basics of object oriented design. We will learn how systems are broken into subsystems, concurrency identification, and how data storage is managed.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- give an overview of object oriented design;
- breaking down system to subsystems;
- explain how a software design may be represented as a set of interacting objects that manage their states and operation;
- identify concurrent object, and explain how to handle concurrency;
- explain methods of storing data and comparison with conventional system design;
- control events between objects; and
- explain boundary condition.

1.2 SYSTEM DESIGN: AN OBJECT ORIENTED APPROACH

Any software systems always tends to change and evolve as technology and business rules change. The evolution of information systems is unavoidable, and it is a natural phenomenon. Organisations need to support systems evolution to take advantage of the new technology and to address the changing business rules. The evolutionary nature of software products requires us to maintain products with their continually changing nature. Not all software products developed are amenable to these fast changes. In this **ever-increasing** competitive business environment, there is hardly any option but *to* ***adopt technology*** that is adaptable to changes.

Software systems designed with **structured design methodology** do not support some of the desired quality attributes such as **reusability, portability** and mapping to the problem domain. Many large organisations find that systems designed with structured approaches are less reusable and less maintainable than those designed with object-oriented approaches.

OOD techniques are useful for development of **large, complex systems**. It has been observed that large projects developed using OOD techniques resulted in a 30% **reduction** in **development** times and a **20% reduction** in **development staff effort,** as compared to similarly sized projects using traditional software development techniques.

Although object oriented methods have roots that are strongly anchored back in the 60s, structured and functional methods were the first to be used. This is not very uncommon, since functional methods are **inspired directly by computer architecture** (a proven domain, well known to computer scientists). The separation of data and program as exists physically in the hardware was translated into **functional methods**. This is the reason why computer scientists got into the habit of thinking in terms of **system functions**. Later it was felt that hardware should act as the **servant** of the software that is executed on it rather than imposing architectural constraints on system development process.

Moving from a functional approach to an object oriented one requires a translation of the functional model elements into object model elements, which is far from being straightforward or natural. Indeed, there is no direct relationship between the two sets, and it is, therefore, necessary to break the model elements from one approach in order to create model element fragments that can be used by the other. In the initial phases of OO design a mixed approach was followed computer scientist tend to use functional approach in analysis phase and OO approach in design phase. The combination of a functional approach for analysis and an object-oriented approach for design and implementation does not need to exist today, as modern object-oriented methods cover the full software lifecycle.

Traditional System Analysis and Design: Traditional System Analysis and Design (SAD) has three fundamental life cycle models. A typical software lifecycle consists of following phases:

- Planning
- Development
- Maintenance

Key Criteria

- a well-defined methodology
- traceability among steps
- documentation
- control

Software Life Cycle

The software design consists of a number of stages, or phases

1. Requirement Phase– determine use need
2. Specification Phase– define requirement

3. Design Phase: design the system
4. Implementation Phase– fabricate the system
5. Testing and Integration Phase– testing the fabricated system
6. Maintenance phase
7. Retirement.

Shortcomings in the Structured approach

- Scalability: The product designed was not scalable
- Maintenance cost: high maintenance cost
- Data and program are separate and difficult to map a real life situation
- Structured methodology treat data and their behaviors (function) separately, this makes it more difficult to **isolate changes**. If certain changes requires then changes in both data structure and algorithms and to be done.

The Proliferation of Object-Oriented Methods

The first few years of the 1990s saw the blossoming of around **fifty different object oriented methods**. This proliferation is a sign of the great vitality of object oriented technology, but it is also the fruit of a multitude of interpretations of exactly what an object is. The drawback of this abundance of methodologies is that it **encourages confusion**, leading users to adopt a 'wait and see' attitude that limits the progress made by the methods. The best way of testing something is still to deploy it; methods are not cast in stone – they evolve in response to comments from their users. Fortunately, a close look at the dominant methods allows the extraction of a consensus around common ideas. The main characteristics of objects, shared by numerous methods, are articulated around the concepts of class, association (described by *James Rumbaugh*), partition into subsystems (*Grady Booch*), and around the expression of requirements based on studying the interaction between users and systems (***Ivar Jacobson's use case*s**).

Finally, well-deployed methods, such as Booch and OMT (Object Modeling Technique), were reinforced by experience, and adopted the methodology elements that were most appreciated by the users.

From SAD to OOAD (Structured Analysis and Design (SAD) to Object Oriented Analysis and Design (OOAD).

We will see here how we can map different models in SAD to different models in OOAD. We will a consider various levels of abstraction through which this is done. The data flow diagram (DFD) in SAD is mapped to Use Case diagram in OOAD. DFD represents a broader model of a system from the process point of view. Hence, DFD cannot be transformed as it is to equivalent representation in UML.

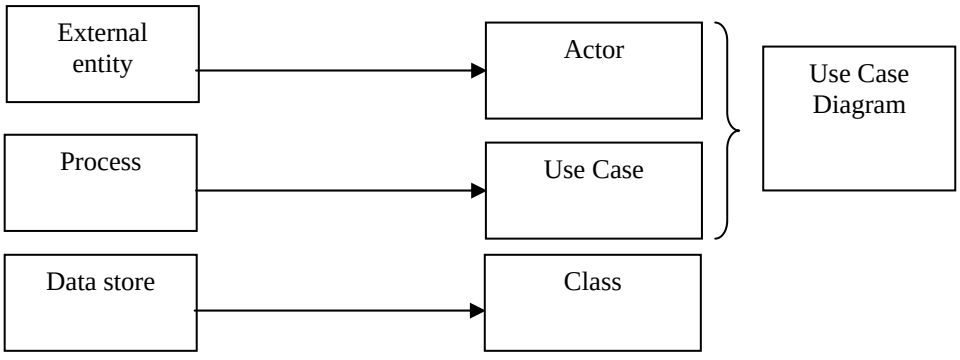


Figure 1: Transformation from SAD to OOAD

The processes are transformed those into Use Case, and the external entity in DFD has similar characteristics to of an actor in OOAD. The data store is transformed into class and part of the data store to attributes of class.

An object-oriented design process

1. Define the context and modes of use of the system

2. Designs the system architecture
3. Identifies the principal system objects
4. Identifies concurrency in the problem
5. Handling boundary conditions
6. Develops design models
7. Specifies object interfaces

Object oriented design is concerned with developing OO model of software systems to implement the identified requirement during the analysis phase.

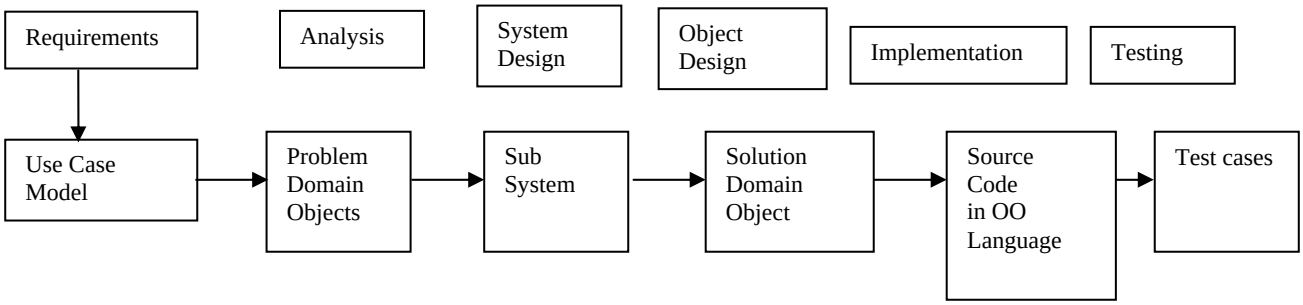


Figure 2: Software life cycle activity

Software developers, data base administrators (DBAs), need to be familiar with the basic concepts of object-orientation. The object-oriented (OO) paradigm is a development strategy based on the concept that **systems should be built from a collection of reusable components called objects**. Instead of separating data and functionality as is done in the structured paradigm, objects encompass both. While the object oriented paradigm sounds similar to the structured paradigm, as you will see in this course material it is actually quite different. A common mistake that many experienced developers make is to applying similar software-engineering principles to OO design. To succeed one must recognize that the OO approach is different than the structured approach.

☞ Check Your Progress 1

1) Why were functional methods popular in early days?

……

……..……

…..……

……

……

……

2) What are the shortcomings in structured approach?

……

……..……

…...

……

……..……

…...

……

…

3) Describe different steps of an object-oriented design process.

……

…..……

…..……

……

……

……

Breaking the system into subsystem involves breaking the problem in to logically independent and interacting subsystems. Now we will discuss how a system is broken into subsystems.

1.3 BREAKING INTO SUBSYSTEMS

Decomposition is an important technique for coping with complexity based on the idea of **divide and conquer**. In dividing a problem into a subproblem the problem becomes less complex and easier to overlook and to deal with. Repeatedly dividing a problem will eventually lead to subproblems that are small enough so that they can be conquered. After all the subproblems have been conquered and solutions to them have been found, the solutions need to be composed in order to obtain the solution of the whole problem. The history of computing has seen two forms of decomposition, **process-oriented** and **object-oriented decomposition.**

Process-oriented decompositions divide a complex process, function or task into simpler sub processes until they are simple enough to be dealt with. The solutions of these subfunctions, then, need to be executed in certain sequential or parallel orders, in order to obtain a solution to the complex process.

Object-oriented decomposition aims at identifying individual autonomous objects that encapsulate both a state and a certain behavior. Each major components of the system is called a subsystem. Then communication among these objects leads to the desired solutions. Decomposition of system in function-data model (SAD) and object oriented decomposition is given below. Although both solutions help deal with complexity, we have reasons to believe that an object-oriented decomposition is favourable because, the object-oriented approach provides for a **semantically richer framework** that leads to decompositions that are more closely related to **entities from the real** world. Moreover, the identification of **abstractions** supports heaving (more abstract) solutions to be reused, and the object-oriented approach supports the evolution of systems better, as those concepts that are more likely to change can be hidden within the objects.

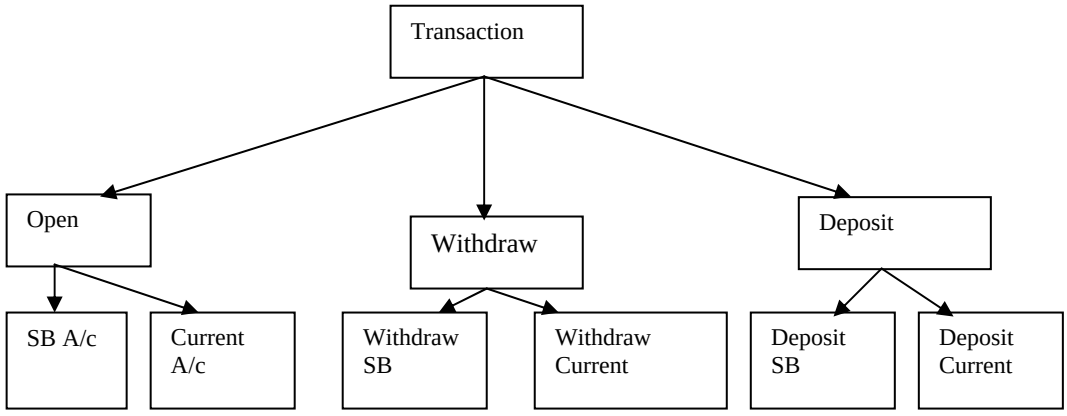


Figure 3: A Function data decomposition (SAD)

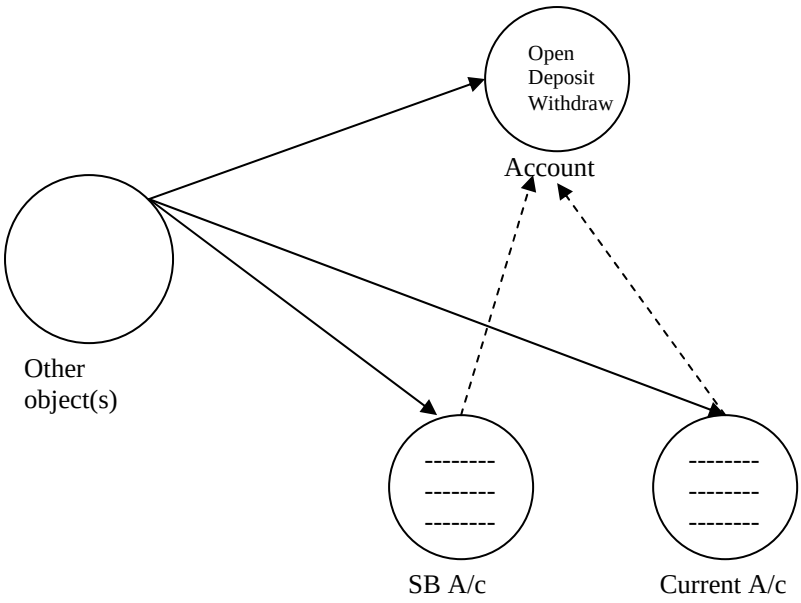
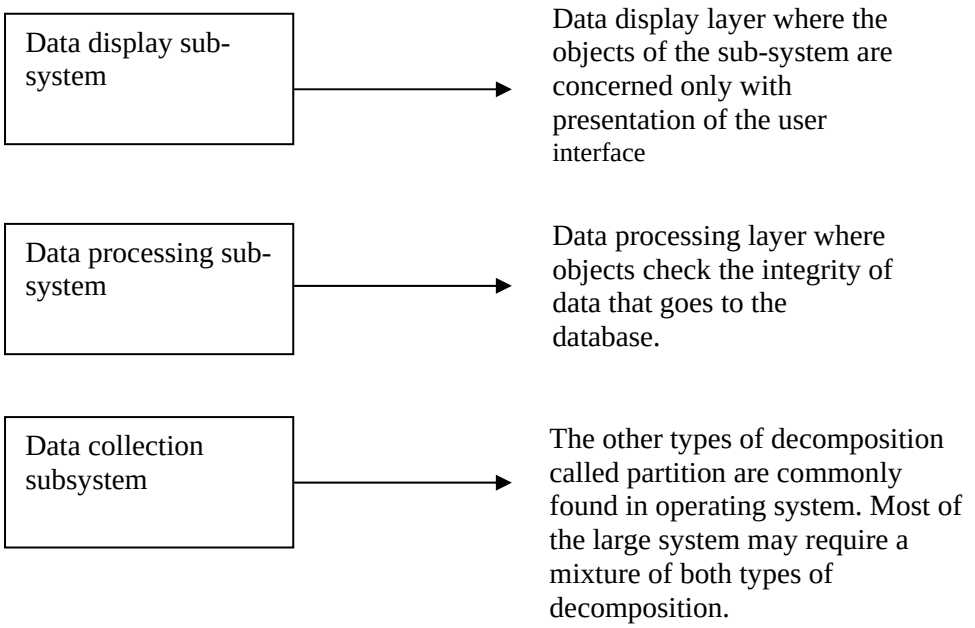


Figure 4: OO Decomposition (OOD)

Object-oriented decompositions of systems tend to be better able to cope with change. Each subsystem has a well-defined interface that communicate with rest of the system. Each of these interfaces defines all form of interaction that are required for proper functioning of the whole system, but the internal implementations are left to the sub-system itself. This is because they manage to encapsulate those items that are likely to change (such as functionality, sequence of behaviour and attributes) within an object and hide them from the outside world. The advantage is that the outside cannot see them, and therefore cannot be dependent on them and does not need to be changed if these items change. Also, object-oriented decompositions are closer to the problem domain, as they directly represent the real-world entities in their structure and behavior. The abstraction primitives built into reuse have a huge potential of
reuse as commonalities between similar objects can be factored out, and then, the solutions can be reused. Finally, object-orientation has the advantage of continuity throughout analysis, design implementation, and persistent representation.

- Object-oriented analysis, design and programming are related but distinct.
- OOA is concerned with developing an object model of the application domain.
- OOD is concerned with developing an object-oriented system model to implement requirements.
- OOP is concerned with realizing an OOD using an OO programming language such as Java or C++.
- Objects are abstractions of real-world/system entities.
- Objects manage themselves.
- Objects are independent and encapsulate state and representation.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated.
- Objects communicate by message passing.
- Objects may be distributed and may execute sequentially, or in parallel.

The system can be decomposed into two-layer architecture (referred as layer) vertical and decomposition (referred as partition).



Now, let us see the basic advantages of decomposition.

The Advantages of Decomposition

1. Separate people can work on each subsystem.
2. An individual software engineer can specialize in a domain.
3. Each individual component is smaller, and therefore easier to understand and manage.
4. Part of the subsystem can be replaced or changed without having to replace or extensively change other subsystems.

Concurrency identification is very challenging in nature. In the next section, we will discuss objects concurrency identification.

1.4 CONCURRENCY IDENTIFICATION

While designing the analysis, model we map the real world model into our analytical model. Real life objects are concurrent in nature, but all **design model objects are not concurrent in nature** as a **single process may support multiple objects.**

Let us see what concurrency actually is: **Concurrency in objects can be identified by the way they change state**. Current objects can change state **independently**. Aggregation implies concurrency. Concurrency in OOAD study is described and studied in **dynamic modeling**.

One of the important issues in system design is to find the concurrency in objects. Once we identify non-concurrent (mutually exclusive) objects, we can fold all the objects together in **one thread of control, or process**. On the other hand, if the objects are concurrent in nature we have to assign them to, **different thread of control**. For example, withdraw and deposit operations related to a bank account may be executed in parallel (concurrently).

- A thread of control is a path through a set of state diagrams on which a single object is active at a time.
- Objects are shared among threads, that is, several methods of the same object can be active at the same time.
- Thread splitting: Object sends a message but does not wait for the completion of the method.

Identification of concurrency: Concurrency is identified in a dynamic model. Two objects are said to be concurrent (parallel) if they can receive events **at the same time**. Concurrent objects are required to be assigned to different threads of control. We will see how if is used in this example:

Example:

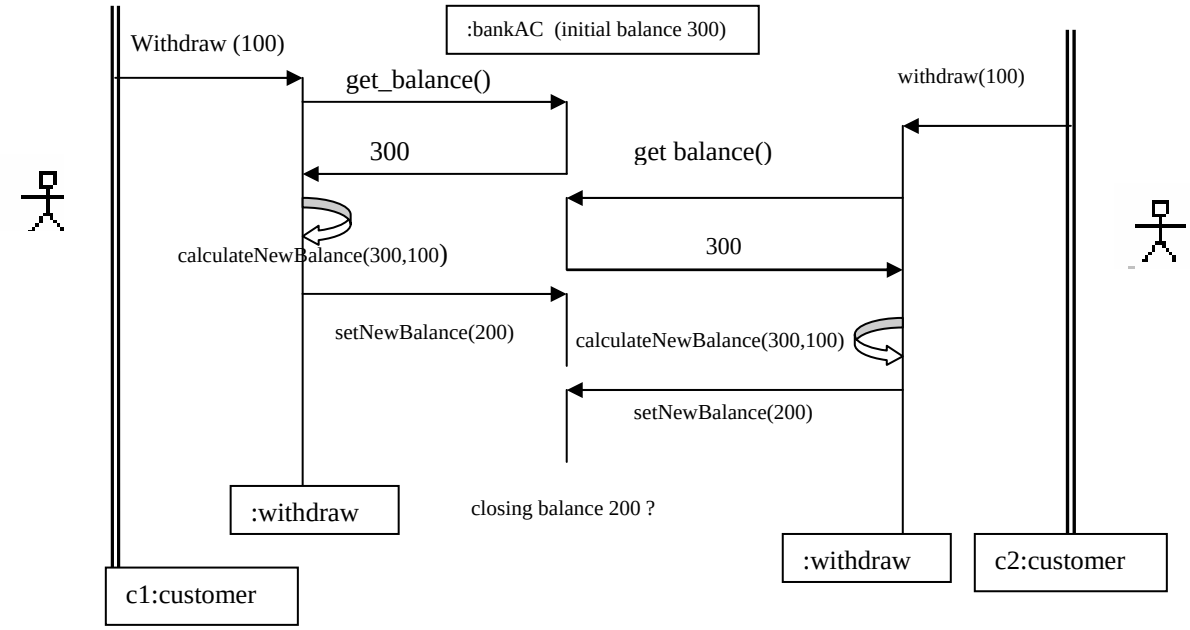


Figure 6: Concurrency without synchronization

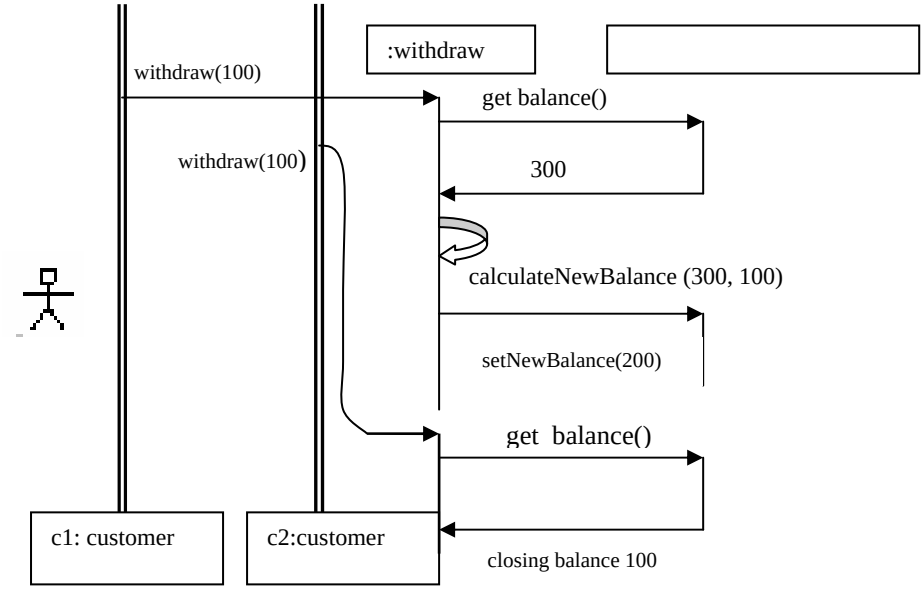


Figure 7: Concurrency with synchronization

Refer to the *Figure 7* if an object must perform two or more activities concurrently, then the internal steps of the process must be synchronized. Both the synchronized activity must complete before the object performing the concurrent activity can go to the next step.

Concurrency issues

- **Data integrity:** Threads accessing the same object need to be synchronized, for example: banking account.
- Deadlock: One or more threads in the system are permanently blocked
  Example: Thread A waiting on Thread B, which is waiting on Thread A.
- Starvation: A thread is not getting enough resources to accomplish its work

  Example: All requests from one user are being handled before another users requests.

How to handle concurrency*:*

Mechanisms

- Locks
- Semaphores
- Monitors
- Synchronized methods

Methods

- Deadlock avoidance
- Verification
- Simulation

Key:

1) Develop a clear strategy for dealing with all concurrency issues during system design.
2) Concurrency must be dealt with during the design process as dealing with concurrency after the system is implemented is difficult.

A detailed discussion on this topic will be there in MCS 041.

☞ Check Your Progress 2

1) What are the advantages of decomposing a system?
   ……
   ……
   ……
   ……
2) Differentiate between object oriented decomposition and structured decomposition?
   ……
   ……
   ……
   ……
3) How is concurrency identified?
   ……
   ……
   ……
   ……
   ……
   ……

Data storage is a very important stored aspect involve in any systems. In the next section we will discuss management of data.

1.5 MANAGEMENT OF DATA STORE

Every system irrespective of their nature of application needs to store permanent data for subsequent use in problem solving. Some objects in the models need to be **persistent**, to store the state of the object permanently in database. Most systems need **persistent data** which is not lost when the system closes down. These data are stored in file system, in a or database. Object-oriented applications often use relational databases to provide persistence.

Designer needs to:

1. Identify what data needs to be persistent
2. Design a suitable database schema for the database.

Persistent classes are shown using tagged value in UML diagram.

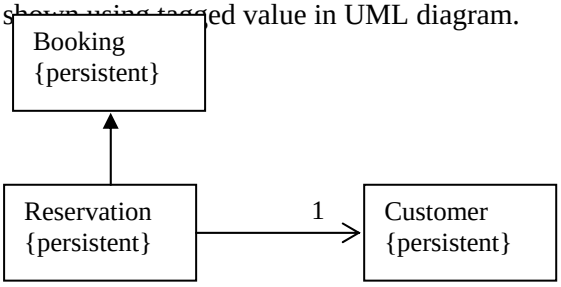


Figure 8: A UML diagram showing persistent class

For example, in reference to *Figure 8*, we must save all information related to customers and booking details.

In most of the cases, persistent class maps to one relational table (leaving aside the inheritance issue, for the moment). In the simplest model, a class from the logical model maps to a relational table, either in whole, or in part. The logical extension of this is that a single object (or instance of a class) maps to a single table row. Persistent object can be stored with one of the following:

Files

- Cheap, simple, permanent storage
- Low level (Read, Write)
- Applications must add code to provide a suitable level of abstraction.

Relational database

- Standardized, easy to port
- Supports multiple writers and readers
- Mature technology

Object database

- One-to-one mapping from the analysis model
- Associations are directly represented
- Slower than relational databases for complex queries

Accessing of value in data store

Updating of value in data store

Figure 9: Functional Model notation of data store

The advantage of using a Database Management System for a data store is that databases have mechanisms for **describing data, managing persistent storage and for providing a backup and recovery mechanism**. It also provides concurrent access to the stored data with an appropriate locking mechanism. Most of the **DBMS** contains information about data in the form of a data dictionary. Most commercial RDBMS come with an "**Object-Relational**" extension which implements an object database on top of a RDBMS.

Issues when Selecting a Database

- **Storage space requirement**: A database requires about triple the storage space of actual data.
- **Response time**: The response time of the database for I/O or communication bound (in case of distributed databases) request.
- **Locking modes pessimistic locking**: Lock before accessing an object and release when object access is complete.

- **Optimistic locking:** Read and writes may freely occur (high degree of concurrency).
- **Administration**: Modern DBMS requires specially trained support staff to set up security policies, manage the disk space, prepare backups, fine-tune the performance, and monitor performance.
- How often is the database accessed?
- What is the expected request (query) rate? In the worst case?
- What is the size of the typical request and of the worst case requests?
- Need for data to be archived.

Table 1

Relational Databases	Object-Oriented Databases
• Based on mathematical principles called relational algebra • Data are represented by a two dimensional table with columns and rows • Implements standard query language called SQL • Most RDBMS supports various constraints, like referential integrity.	• Supports all fundamental object modeling concepts: Classes, Attributes, Methods, Associations, Inheritance • Support for complex objects • Provides for mapping an object model to an OO-database • Determine which objects are persistent • Perform normal requirement analysis and object design • Create single attribute indices to reduce performance bottlenecks.

Most of the object oriented system use relational database to store persistent data. The advantages and disadvantages of OO database are compared in Table 2.

Table 2

Advantages of OO Database	Disadvantages OO Database
• Supports all fundamental object modeling concepts like Classes, Attributes, Methods, Associations, Inheritance • Maps an object model to an OO-databases • Determine which objects are persistents • Performs normal requirement analysis and object design • Creates single attribute indices to reduce performance bottlenecks • Supports complex objects. • Extensibility of data types. • Improves performance with efficient caching. • Versioning • Faster development and easy maintenance through inheritance and reusability.	• Strong opposition from the established players of relational database • Lacks rigorous theoretical foundation. • Retrogressive to the old pointer systems • Lacks standard ad hoc query language like SQL. • Lack of standards affects OO database design • Lack of business data design and management tools • Steep learning curve.

Other System Design Issue

- How to realise the subsystems: through hardware or software?
- How is the object model mapped on the chosen hardware software?
- Mapping objects onto hardware: processor, memory, I/O.
- Mapping associations onto networks: Connectivity.

Much of the difficulty of designing a system comes from fulfilling the restriction imposed by hardware and software constraints. This may include cases where certain throughput has to be guaranteed for a system, and where certain response time has to be guaranteed.

1.6 CONTROLLING EVENTS BETWEEN OBJECTS

Event is the specification of a significant occurrence that has a location in time and **space**.

Examples of events are mouse click and flight leaving from an airport. An event does not have a **fixed duration.** Each thing that happens modeled as an event. After an event, objects change their state, and these are represented by a state diagram.

Events are classified as four types in UML

1. Signals
2. Calls
3. Passing of Time
4. Change in State

Events also include inputs, decisions, interrupts and actions performed by users or any external device. Every event has a sender and receiver. In most of the cases the sender and receives are the same object. A state without a predecessor and successor are ambiguous, and care should be taken to represent initiations and termination of events. Events that have same effect on the control flow must be grouped together even if their value differs. The events are to be allocated to the object classes that send/receive it.

Most of the design issues of systems are concerned with steady-state behavior. However, the system design phase must also address the initiation and finalization of the system. This is addressed by a set of **new uses cases** called administration use cases. Now, let us discuss how these issues are handled.

1.7 HANDLING BOUNDARY CONDITION

These are some conditions which to be handled in any system initialization and termination

- Describes how the system is brought from a non-initialized state to steady-state (“startup use cases”). It describes normal operations like start-up, shutdown, reconfiguration, restart/reset, backup, etc.
- Describes what resources are cleaned up, and which systems are notified upon termination (“termination use cases”).

Configuration

- Describes how the system is adapted to a local installation.

Failure

- Unplanned termination of the system.

- Many possible causes: failure of system hardware, bugs, operator errors, external problems (power supply).
- Good system design foresees fatal failures ("failure use cases").

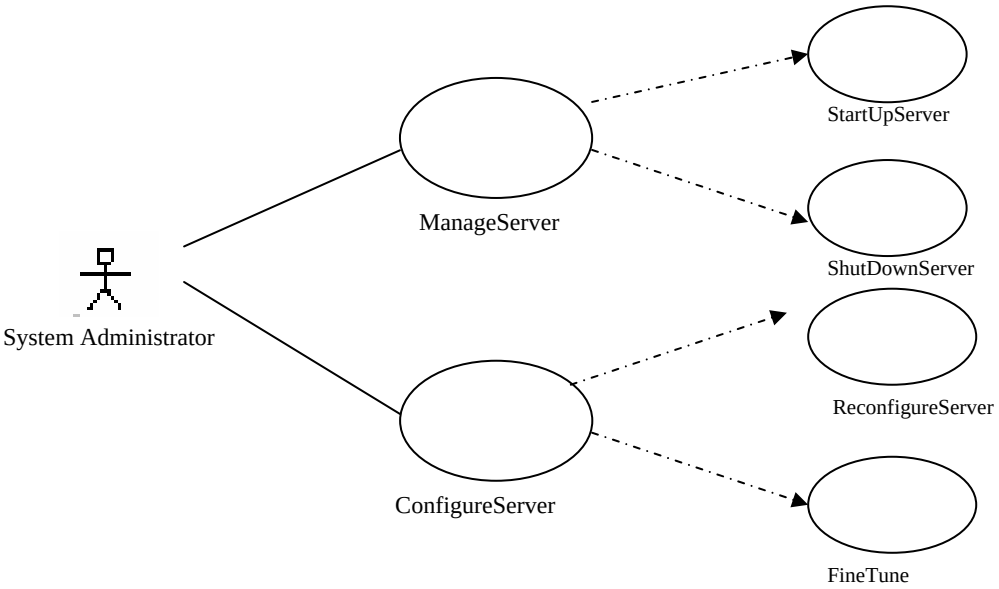


Figure 10: Boundary Use Case

Operation	Actor	Constraints
Start up	System Administrator / Operator	Availability of power, and no system fault
Shut down	System Administrator / Operator	No user is active, and data has been saved
Fine Tune	System Administrator	No user is active, database offline
Reconfigure	System Administrator	No user is active, system resources are available

While defining the boundaries of a system we must ascertain what system entities one does and does not have control over. The level of control is determined for all identified internal and external entities. The control status can be total, partial, or none.

☞ Check Your Progress 3

1) What is persistency?

……

……

……

……

2) Why, generally, does an object-oriented system use a relational DBMS?

……

……

……

……

3) Identify and name a few boundary processes.

……

……

……

……

4) Draw a use case diagram for a typical Flight Reservation System. Identify use case and actor.

……

……

……

…….

1.8 SUMMARY

OOD techniques are useful for the development of large, complex systems. Moving from a functional approach to an object-oriented one requires a translation of the functional model elements into object model elements, which is far from being straightforward. The high-level system design approach involves breaking down the system into simple and relatively independent sub systems for better manageability. Systems can be proportioned to horizontal or vertical partitions. Concurrency is inherent to objects, and concurrent objects cannot be folded into a single thread of control. Concurrency must be dealt with during the design process as dealing with concurrency after the system is implemented is difficult.

Most systems need **persistent data** which is not lost when the system closes down. Data can be stored in flat files or DBMS. Object-oriented databases provide support for all fundamental object modeling concepts like Classes, Attributes, Methods, Associations, and Inheritance. But unlike RDBMS, object-oriented databases lack theoretical foundation and standards. Although system design is concerned with the steady state behaviors of the system, the system must be designed to handle boundary conditions. Boundary use cases are useful to analyse boundary conditions.

1.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) Functional methods are inspired directly by computer architecture, and thus the popular among computer scientist in the early days. The separation of data and program as it exists physically in the hardware was translated into the functional methods.

2) Structured methodology treats data and their behaviors (functions) separately, and this makes it more difficult to isolate changes. If changes are required in the software then one has to change both the data structure and the algorithms and these changes subsequently require changes in the algorithms where this data structure is used as well.

3) The broad steps of an object-oriented design process are:

   a. Define the context and modes of use of the system

   b. Design the system architecture

   c. Identify the principal system objects

   d. Identification of concurrency in the problem

   e. Handling boundary condition

   f. Develop design models

   g. Specify object interfaces.

Check Your Progress 2

1) The advantages of decomposing a system into subsystems are that after decomposition, each individual components become smaller, and easy to manage. Changes in these subsystem can be effected without extensively

making changes in other subsystem. Decomposition it also allows software engineer to specialize in a particular domain of the system.

2) Process-oriented (structural) decomposition divides a complex process, function or task into simpler subprocesses until they are simple enough to be dealt with. The solutions of these subfunctions then need to be integrated and executed in certain sequential or parallel orders in order to obtain a solution to the large complex system. On the other hand object-oriented decomposition aims at identifying individual autonomous objects that encapsulate both a state and certain behavior. Each major component of the system is called subsystem. Then, communication among these objects leads to the desired solutions. Object-oriented decompositions of systems tend to be better able to cope with change. Each subsystem has well-defined interfaces that communicate with rest of the system

3) Concurrency in objects can be identified by the way **they change state**. Current objects can change state independently. Aggregation implies concurrency in the system.

Check Your Progress 3

1) Persistency ensures that data is stored, and that after the object is no longer available (program stops running), the data will be, available to other users, as and when needed.

2) The reasons for using RDBMS for OO systems could be any combination of the following:

   a. Many organizations have existing relational databases containing existing business data.

   b. Most commercial RDBMS come with an “Object-Relational” extension which implements an object database on top of a RDBMS.

   c. Purely object databases are too complicated to use, and lack standards like SQL.

3) System startup, shutdown, system failure.

4) Actors: Passenger, Ticket clerk.

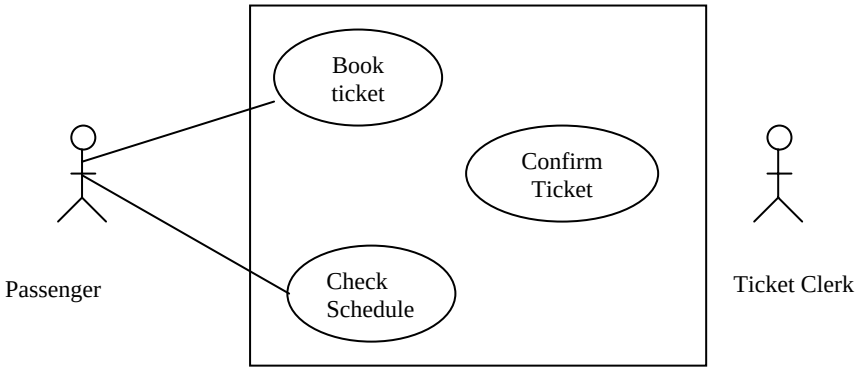


Figure 11: Use case for Book Ticket, check schedule and confirm Ticket

UNIT 2 OBJECT DESIGN

Structure **Page Nos.**

2.0 Introduction 20
2.1 Objectives 20
2.2 Object Design for Processing 20
2.3 Object Design Steps 21
2.4 Choosing Algorithms 23
  2.4.1 Selecting Data Structure
  2.4.2 Defining Internal Classes and Operations
  2.4.3 Assigning Responsibility for Operation
2.5 Design Optimization 24
2.6 Implementation of Control 26
  2.6.1 State as Location within a Program
  2.6.2 State Machine Engine
  2.6.3 Control as Concurrent Tasks
2.7 Adjustment of Inheritance 27
  2.7.1 Rearranging Classes and Operations
  2.7.2 Abstracting Out Common Behavior
2.8 Design of Associations 28
  2.8.1 Analyzing Association Traversal
  2.8.2 One-way Associations
  2.8.3 Two-way Associations
2.9 Summary 29
2.10 Solutions/Answers 29

2.0 INTRODUCTION

Strategies that are selected in system design are carried out in object-oriented design. In this process, objects that are identified during the analysis are implemented in a way that it minimizes memory, execution time and other associated costs. All this is done by the selection of appropriate algorithms, optimizations, and by enforcing proper. Controls.

In this unit, we will cover the concepts of object design for process, proper algorithm selections, design optimization and control implementation, with proper adjustment of inheritance.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the steps of object design;
- discuss algorithms that minimize costs;
- select appropriate data structure to the algorithm;
- define new internal classes and operations;
- assign responsibility for operation to the appropriate classes; and
- explain different types of associations.

2.2 OBJECT DESIGN FOR PROCESSING

The object design phase comes after the analysis and system design. The object design phase adds implementation details such as restructuring classes for efficiency, internal data structures and algorithms to implement operations, implementation of control,

implementation of associations and packaging into physical modules. Object design extends the analysis.

As you know, there are three models that define the operations on classes:

1) **Object model**: This describes the classes of objects in the system, including their attributes, and the operations that they support. The analysis object model information definitely exists in some form in design. For this, sometimes new redundant classes are added which increase efficiency.

2) **Functional model**: This defines the operation that the system must implement. For each operation, from the analysis model must be assigned an algorithms that implements clearly and efficiently, according to the optimization goals selected during system design. In this model, we map the logical structure of the analysis model into a physical organization of a program.

3) **Dynamic model**: The dynamic model describes how the system responds to external events. The implementation of control of flow in a program must be realized either explicitly or implicitly. Explicit means by the internal scheduler that recognizes events and map them into operation calls. Implicit means by choosing algorithms that perform the operations in a specified order.

Now let us see how object design is done for systems.

2.3 OBJECT DESIGN STEPS

Object design is a very iterative process in which several classes (maybe newly created), relationships between objects, are added when you move from one level to another level of the design.

There are certain steps to be followed in this design:

1) **Classify the operations on classes**

This step basically means all the three models, **functional, object** and **dynamic** (studied in last section) must be combined so as to know what operations are to be performed on objects.

We can make a state diagram describing the life history of an object. A transition is a change of state of object and it maps into an operation on the object. It helps in visualizing state changes. We can associate an operation with each event received by an object. Also, sometimes an event may represent an operation on another objects i.e., where one event triggers another event. Thus, in this case, the event pair must be mapped into an operation performing action and returning control, provided that the events are on a single thread of control passing from object to object.

Any action initiated by transition in a state diagram can be expanded into an entire data flow diagram in the functional model. The processes in a data flow diagram consist of sub-operations which may be operational on the original target object, or on other objects. We can determine the target object of a sub-operation as follows:

i) If the process extracts a value from **I/P flow**=> input, flow is target.

ii) If the process has the same type of input and output flow and **O/P** value is the updated version of **I/P** => **I/O,** flow is target.

iii) If the process has an **I/P** from or an **O/P** to a data source => data, source is a target of the process.

iv) If the process constructs **O/P** value from a number of inputs => operation is class operation on the output class.

2) **Design an algorithm to implement operations**

Each and every operation specified in a functional model should be formulated as an algorithm. The algorithm indicates how the operation is done rather than what it does, as in analysis specification.

The algorithm designer must:

i) Select the proper algorithm so as to minimize implementation cost

ii) Find the most appropriate data structure for the selected algorithm

iii) Define new internal classes and operations, if required

iv) Assign responsibility for operations to appropriate classes.

In the next section, we will discuss algorithm selection in more detail.

3) Optimization of data access paths

Optimization is a very important aspect of any design. The designer should do the followings for optimization:

i) Add redundant associations, or omit non-usable existing associations to minimize access cost and maximize convenience

ii) Rearrange the order of computational tasks for better efficiency

iii) Save derived attributes to **avoid re-computation** of complicated expressions.

4) Implementing software control

To implement software control the designer must redesign the strategy of the state event model that is present in the dynamic model.

Generally, there are three basic approaches to implement the dynamic model. These approaches are:

i) Storing state of program as location within a program, i.e., as a procedure driven system

ii) Direct implementation of a state machine mechanism i.e., event driver

iii) Using concurrent tasks.

5) Adjustment of inheritance

The inheritance can be increased as the object design progresses by changing the class structure. The designer should:

i) Adjust, or rearrange the classes and operations

ii) Abstract common behavior out of groups of classes

iii) Use delegation to share behavior when inheritance is semantically incorrect.

6) Design of associations

During the object design phase we must make a strategy to implement the associations. Association can be unidirectional or bi-directional. Whichever implementation strategy we choose, we should hide the implementation, using access operations to traverse and update the associations. This will allow us to change our decision with minimal effort. The designer should:

i) Analyze the path of associations.

ii) Implement every association either as a distinct object, or as a link to another object.

7) Determine object representation

As a designer, you must choose properly when to use primitive types in representing objects, and when to combine groups of related objects, i.e., what is the exact representation of object attributes.

8) Package classes and associations into models

Programs are made of discrete physical units that can be edited, complied, imported or otherwise manipulated. The careful partitioning of an implementation into package is important for group work on a program. Packaging involves the following issues:

a. Information hiding

b. Collection of entities

c. Constructing physical modules with strong coupling within each module.

☞ Check Your Progress 1

1) Describe briefly the models that define the operations on classes.

……

……

……

2) What is object design?

……

……

……

3) In object oriented design, what steps must the designer take to adjust inheritance?

……

……

……

Algorithm selection is a very important part of design. It reflects the happening in the system. In the next section we will discuss algorithm selection.

2.4 CHOOSING ALGORITHMS

In general, most of the operations are simple and have a satisfactory algorithm because the description of what is to be done also indicates how it is to be done. Most of the operations in the object link network simply traverse to retrieve or change attributes or links. Non-trivial algorithms are generally required for two reasons:

i) If no procedural specification is given for functions

ii) If a simple, but inefficient algorithm serves as a definition of function.

There are a number of metrics for selecting best algorithm:

i) **Computational complexity:** This refers to efficiency. The processor time increases as a function of the size of the data structure. But small factors of inefficiency are insignificant if they improve the clarity.

ii) **Ease of use:** A simple algorithm, which is easy to implement and understand, can be used for not very important operations.

iii) **Flexibility:** The fully optimized algorithm is generally less readable and very difficult to implement. The solution for this is to provide two implementations of crucial operations:

- A complicated but very efficient algorithm
- A simple but inefficient algorithm.

Now, let us see the basic activities that are involved in algorithm selection and expression.

2.4.1 Selecting Data Structure

Algorithms work on data structure. Thus, selection of the best algorithm means selecting the best data structure. The data structures never add any information to the

analysis model, but they organize it in a form that is convenient for the algorithms that uses it. Many such data structure include **arrays, lists, stacks, queues, trees**, etc. Some variations on these data structures are **priority queues, binary trees,** etc. Most object oriented languages provide various sets of generic data structures as part of their predefined class libraries.

2.4.2 Defining Internal Classes and Operations

When we expand algorithms, new classes can be added to store intermediate results. A complex operation can be looked at as a collection of several lower level operations i.e., a high level operation is broken into several low level operations. These lower level operations should be defined during the design phase.

2.4.3 Assigning Responsibility for Operation

Many operations may have obvious **target objects**, but some of these operations can be used at several places in an algorithm, **by one of several objects.** These operations are complex high level operations which may be overlooked in laying out object classes as they are not an inherent part of any one class.

Now, *the obvious question is*, how do you decide what class owns an operation? It is easy when only one object is involved in the operation: You are simply informing an object to perform the operation. But when more than one object is involved in an operation it becomes quite difficult. Thus, we must know which object plays the main role in the operation. For this, ask yourself the following questions:

- Is an object acted on when another object performs action? In general, we should associate the operation with the target of the operation, instead of the one initiating it.
- Whether an object is modified by the operation or when other objects are only performing query for getting some information from it. The object that is changed as in the whole process known as the target of the operation.
- Which class is the center of all classes and associations involved in the operation? If the classes and associations form a star around a single class, it is the target of the operation.
- If the object is some real world object represented internally, then what real object would you push, move, activate, or otherwise manipulate to initiate **the operation**?

2.5 DESIGN OPTIMIZATION

The inefficient but correct analysis model can be optimized to make implementation more efficient. To optimize the design, the following things should be done:

a) Adding Redundant Associations for Efficient Access

Redundant associations do not add any information, thus during design we should actually examine the structure of object model for implementation, and try to establish whether we can optimize critical parts of the completed system. Can new associations be added, or old associations be removed? The derived association need not to add any information to the network, they help increasing the model information in efficient manner.

We can analyze the use of paths in the association network as follows:

- Evaluate each operation
- Find associations that it must pass through to get information. Associations can be bi-directional (generally by more than one operation), or unidirectional, which can be implemented as pointers.

For each operation, we should know the followings:

- How frequently is the operation needed, and how much will it cost?
- What is the fan-out along a path through the network? To find fan-out of the complete path, multiply the average count of each "many" associations found in the path with individual fan-outs.
- What are the objects that satisfy the selection criteria (if specified) and are operated on? When most of the objects are rejected during traversal for some reason, then a simple nested loop may be inefficient at finding target objects.

b) Rearranging the Execution Order for Efficiency

As we already know algorithm and data structure are closely related to each other, but data structure is considered as the smallest but very important part of algorithm. Thus, after optimizing the data structure, we try to optimize the algorithm itself.

In general, algorithm optimization is achieved by removing dead paths as early as possible. For this, we sometimes reverse the execution order of the loop from the original functional model.

c) Saving Derived Attributes to Avoid Recomputation

Data which is derived from other data should be stored in computed form to avoid re-computation. For this, we can define new classes and objects, and obviously, these derived classes must be updated if any base object is changed.

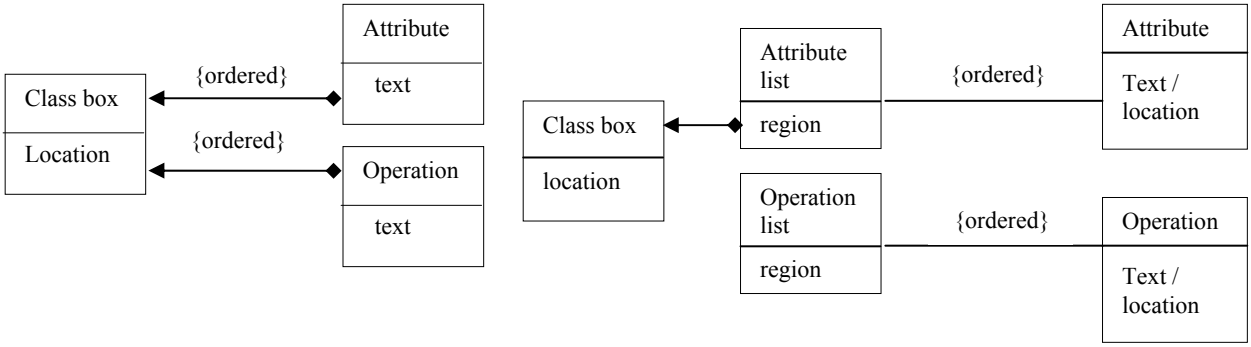


Figure 1: Derived attribute to avoid recomputation

Figure 1 shows a use of **derived object** and **derived attribute** in OOM. Each class box contains an ordered list of attributes and operations, each represented as a next string (*Figure 1(a)*). We can find the location of any attribute by adding the size of all elements in front of it, to the location of the class box itself [note: it is quite similar to the array address calculation]. If a new attribute string is added to the list, then the locations of the ones after it in the list are simply **offset by the size of the new element**. If an element is moved or deleted, the elements under it must be **redrawn**. Overlapping elements can be found by scanning all elements in front of the deleted element **in the priority** list for the sheet and comparing them to the deleted element. If the number of elements is large, this algorithm grows linearly in the number of elements.

As discussed earlier, the change in base object should update the related derived classes. There are three ways to know when an update is required: by **explicit code**, by **periodic recomputation**, or by **using active values**.

Explicit update: Every derived attribute is expressed as set of basic base objects. The designer finds the derived attributes that were affected by change in basic attributes. Then, he inserts code into the update operation on the base object to explicitly update the depending derived attributes.

Periodic re-computation: Generally, base values are updated in groups. Thus, all the derived attributes can be recomputed periodically, instead of after every base value change.

Active Value: An active value is a value that has **dependent values** and update operations. Each dependant value is registered to **an action value**. Updation operation of base **value triggers that updates** of all dependant values and the calling code need not explicitly invoke the updates.

☞ Check Your Progress 2

1) What are the metrics for choosing the best algorithm?

……

……

2) What are the ways of finding out whether an update is required or not for derived attributes?

……

……

3) Give an example of an active value.

……

……

Every program passes through several states, and these states are defined by implementing appropriate controls. In the next section, we will discuss implementation of controls.

2.6 IMPLEMENTATION OF CONTROL

As we know, controls are implemented around states and tasks (maybe concurrent tasks). Now, let us see three different implementation:

2.6.1 State as Location within a Program

In this traditional approach, the location of control within a program implicitly defines the program state. Any finite state machine can be implemented as a program (easily by using 'gotos').

One technique for converting a state diagram to code is as follows:

i) Identify the main control path. Starting with the first state, find a path from the diagram which corresponds to the expected sequence of events. Keep the names of states in a linear sequence that now forms a sequence of statements in the program.

ii) Identify alternate paths that branch off the main path and later joined again. These become conditional statements in the program.

iii) Identify loops i.e., the backward paths branching off from main path, and earlier. Multiple backward paths that not crossing over form nested loops.

iv) Left out states transitions correspond to exception conditions which can be tackled by exception handling, or error routines, or even by setting and testing of status flags.

2.6.2 State Machine Engine

The most direct approach to implement control is to have some way of explicitly representing and executing state machines. With this approach, we can make outline of the system where classes from object model are defined, state machines from dynamic model are given, and stubs of action routines can be created. **A stub is the minimal definition of a function of a subroutine without any internal code.** By using of object oriented language, the state machine mechanism can easily be created.

2.6.3 Control as Concurrent Tasks

As you know, any object can be implemented as a task in the programming language, or operating system. This is the most general approach because it keeps the **inherent concurrency of real objects**. Events are implemented as **inter-task** calls and as such, the task uses its location within the program to keep track of its state.

As we have discussed adjusting inheritance is one of the steps of object design. In the next section, we will discuss how inheritance is adjusted.

2.7 ADJUSTMENT OF INHERITANCE

During object design, inheritance is readjusted by rearranging classes and operations, and abstracting common behavior.

2.7.1 Rearranging Classes and Operations

The different, yet similar, operation of different classes can be slightly modified so that they can be covered by a single inherited operation. The chances of inheritance can be increased by the following kind of adjustments:

i) Some operations needs less arguments than other similar operations, like drawing an object, e.g., circle, rectangle, etc., with, or without, color fill. Thus, the attribute **color** can be accepted, or ignored for consistency with color displays.

ii) Some operations need less argument than other, because they are special case of general arguments. Thus, varied newer operation can be implemented by calling general operation and new argument values. For example, insertion in the beginning or end of the list are special cases of insertion in the list.

iii) Different classes can have similar attributes, but different names. Thus, they can be combined and placed in the base class so that the operation to access the attribute may match in different classes.

iv) Sometimes an operation is required by a subset of classes. In this case, declare the operation in base class, and all those derived classes that do not need it can be declared as no-operation.

2.7.2 Abstracting Out Common Behavior

Inheritance is not always recognised during the **analysis phase** of development, so it is necessary to re-evaluate the object model to find common operations between classes. Also, during design, new classes and operations may be added. If a set of operations and/or attributes seems to be repeated in two classes, then it indicates that the two classes are specialised variation of the same general class.

When common behavior is recognised, a **common super class** can be created which implements the shared features, leaving only the specialized features in the derived classes. This transformation of the object model is called abstracting out a common super class or a common behavior.

The creation of an abstract super class also **improves the extensibility of a software product**.

Associations are useful for finding access paths between objects in the system. During object design itself, we should implement associations. In the next section we will discuss the designing of a association.

2.8 DESIGN OF ASSOCIATIONS

Before designing associations, it is necessary to know the way they are used. For this, analysis of association traversals is necessary. It also important to find out whether the association is one-way association or two-way association.

2.8.1 Analyzing Association Traversal

Till now, we have assumed that associations are bi-directional. But in the case of traversal in only one direction in any application, the implementation becomes easier. However, for finding unidirectional associations we need to be extra cautious, as any new operation added later may be required to traverse the association in the opposite direction also. The bi-directional association makes modification, or expansion easier.

To change our decision of implementation strategy with minimum effort, we should hide the implementation, using access operations to traverse and update the association.

2.8.2 One-way Associations

When an association is traversed in only **one direction**, then it is implemented as a **pointer, i.e.,** an attribute that contains an **object reference**. If the multiplicity is '**one**' as shown in *Figure 2*, then it is simple pointer; otherwise it is a set of pointers.

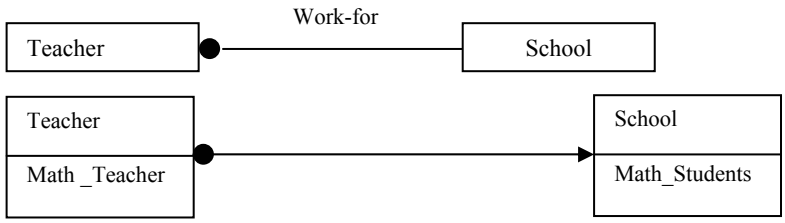


Figure 2: Implementation of a one-way association using pointers

2.8.3 Two-way Associations

Mostly, associations are traversed in **both directions**, although not usually with equal frequency. There are three approaches for implementation.

- In bi-directional associations, if one direction association is rarely used, then we should implement this as unidirectional association. Searching can perform the reverse association. This way, we can reduce the storage and update cost.
- Implementing it as a bi-directional association using the different techniques discussed in previous sections will turn up as shown in *Figure 3*. This approach gives faster access but also requires the updation of other attributes in case of any change a otherwise the link will become inconsistent.

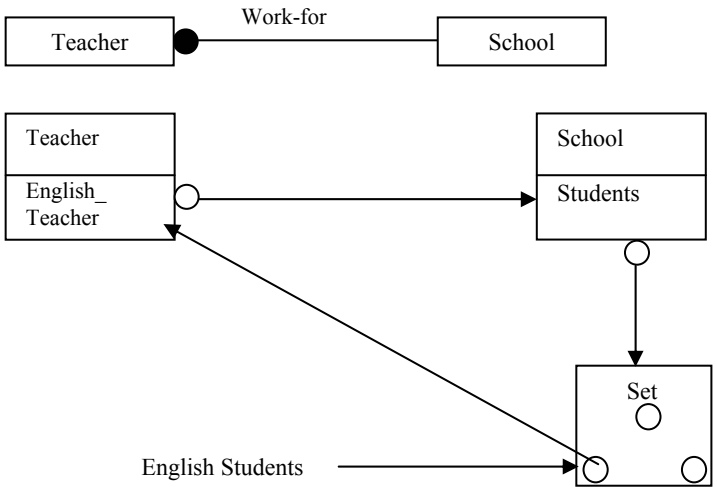


Figure 3: Implementation of two-way association, using pointer

- Implementing it as a distinct association object, i.e., independent of either class as shown in *Figure 4*. The association object may be implemented using two objects: one in forward direction, and the other in the reverse direction. This increases efficiency when hashing is used, instead of attribute pointer.

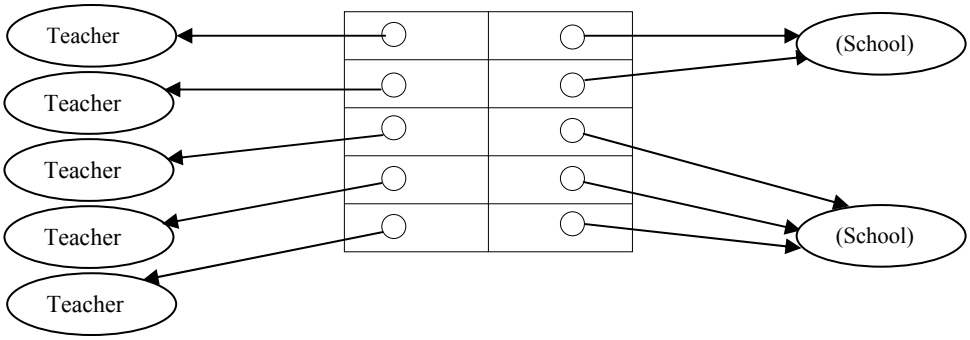


Figure 4: Implementation of association as an object

This approach is suitable in situations where modify action is minimal, or almost nil.

☞ Check Your Progress 3

1) List the steps for converting the state diagram to code.

……

……

2) What kinds of adjustments are required to increase the chances of inheritance.

……

……

3) What is the advantage of two-way association?

……

……

2.9 SUMMARY

This unit explains that the design model is driven by the relevance to computer implementation. The design model must be reasonably efficient and practical to encode. It consists of optimizing, refining and extending the object model, dynamic model and functional model until they are detailed enough for implementation.

In this unit, we have discussed the steps taken in object design: what the approach should be for algorithm selections, how design is optimized by providing efficient access and rearranging execution order, by avoiding recomputation. This unit also discussed controls implementations, and in the last section of this unit, issues related to design of associations were discussed.

2.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The three models that are used to define operations on classes are:

Object Model = Object model diagram + data dictionary

Dynamic Model = State diagrams + global event flow diagram
Functional Model = Data flow diagrams + constraints.

2) Object design is a very interactive process, which decides the relationship between objects, classifies operations on classes, designs algorithms associations, and determines overall system representation.

3) To readjust the inheritance the following steps should be taken:
   i) Rearrange and adjust classes operations to increase inheritance.
   ii) Abstract common behaviour out of groups of classes.

Check Your Progress 2

1) Matrices for choosing the best algorithms are:
   - Computational complexity
   - Ease of implementation and understandability
   - Flexibility.
2) The ways to find out whether an update is required or, not are:
   - Explicit update
   - Periodic recomputation
   - Active values
3) One example of active value is gross salary, which has values as TA, DA, HRA, etc.

Check Your Progress 3

1) a) Finding main control path
   b) Finding conditional statements
   c) Finding loops
   d) Finding exception handling, and error routines.
2) a) Some attributes can be added, or ignored in base class operation
   b) Some variations can be made in derived class from abstract classs.
   c) If an operation is not required by some classes in a group, then can it be declared as no-operation.
   d) Similar attributes can be combined in one abstract base class.
3) Two-way association has following advantages:
   a) Independent of classes.
   b) Useful for existing predefined classes which are not modified.

UNIT 3 ADVANCE OBJECT DESIGN

Structure **Page Nos.**

3.0 Introduction 31
3.1 Objectives 31
3.2 Control and its Implementation 32
3.2.1 Control as a State within Program
3.2.2 Control as a State Machine Engine
3.2.3 Control as Concurrent Task
3.3 Inheritance Adjustment 35
3.4 Association: Design 37
3.5 Object Representation 38
3.6 Design Optimization 39
3.7 Design Documentation 43
3.8 Summary 43
3.9 Solutions/Answers 45

3.0 INTRODUCTION

As discussed earlier analysis is the first step of the OMT methodology. It is concerned with devising a precise, concise, understandable and correct model of the realworld. For example, before building any complex thing, such as a house, a bridge, or a hardware-software system, the builder must understand the requirement of the user, and it is also necessary to know the realworld environment in which it will exist.

The advanced object design is a complex task. The objects discovered during analysis serve as a skeleton of the design. The operations identified during analysis should be expressed as algorithms. Advanced object oriented design is basically a process of refinement, or adding the details to the body of an object.

In this unit, you will learn how to design a formal and rigorous model of real-world problems by applying the findings of the analysis phase of OMT. The object design phase determines the complete definition of classes and associations used in the implementation. The advanced object design is a process to create architecture of the realworld problems. The advanced object design is analogous to the preliminarily design phase of the traditional software development cycle.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- combine the three OOAD models to obtain operations on classes;
- design algorithms to implement operations on classes;
- optimize access paths to data;
- implement control for external interactions;
- adjust class structure to increase inheritance;
- design association;
- determine object representation, and
- package classes and association into modules.

3.2 CONTROL AND ITS IMPLEMENTATION

In this unit, we will start our discussion with explanation of state-event models. Let us define an state-event model. "State-event model is a model which shows the sequence of events happening on an object, and due to which there are changes in the sate of an object". In the state-event model, the events may occur concurrently and control resides directly in several independent objects. As the object designer you have to apply a strategy for implementing the state event model. There are three basic approaches to implementing system design in dynamic models. These approaches are given below:

- Using the location within the program to hold state (procedure-driven system).
- Direct implementation of a state machine mechanism (event-driven system).
- Using concurrent tasks.

3.2.1 Control as State within Program

1. The term ***control*** literally means to check the effect of input within a program. For example, in *Figure1*, after the ATM card is inserted (as input) the control of the program is transferred to the next state (i.e., to request password state).

2. This is the traditional approach to represent control within a program. The location of control within a program implicitly defines the program state. Each state transition corresponds to an input statement. After input is read, the program branches depending on the input event produce some result. Each input statement handles any input value that could be received at that point. In case of highly nested procedural code, low-level procedures must accept inputs that may be passed to upper level procedures. After receiving input they pass them up through many levels of procedure calls. There must be some procedure prepared to handle these lower level calls. The technique of converting a ***state diagram*** to code is given as under:

a) Identify all the main control paths. Start from the initial state; choose a path through the diagram that corresponds to the normally expected sequence of events. Write the names of states along the selected path as a linear sequence. This will be a sequence of statements in the program.

b) Choose alternate paths that branch off the main path of the program and rejoin it later. These could be conditional statements in the program.

c) Identify all backward paths that branch off the main loop of the program and rejoin it earlier. This could be the loop in the program. All non-intersecting backward paths become nested loops in the program.

d) The states and transitions that remain unchecked correspond to exception conditions. These can be handled by applying several techniques, like error subroutines, exception handling supported by the language, or setting and testing of status flags.

To understand control as a state within a program, let us take the state model for the ATM class given below in *Figure 2* showing the state model of the ATM class and the pseudo code derived from it. In this process first, we choose the main path of control, this corresponds to the reading of a card querying the user for transaction information, processing the transaction, printing a receipt, and ejecting the card. If the customer wants to process for some alternates control that should be provided. For example, if the password entered by the customer is bad, then the customer is asked to try again.

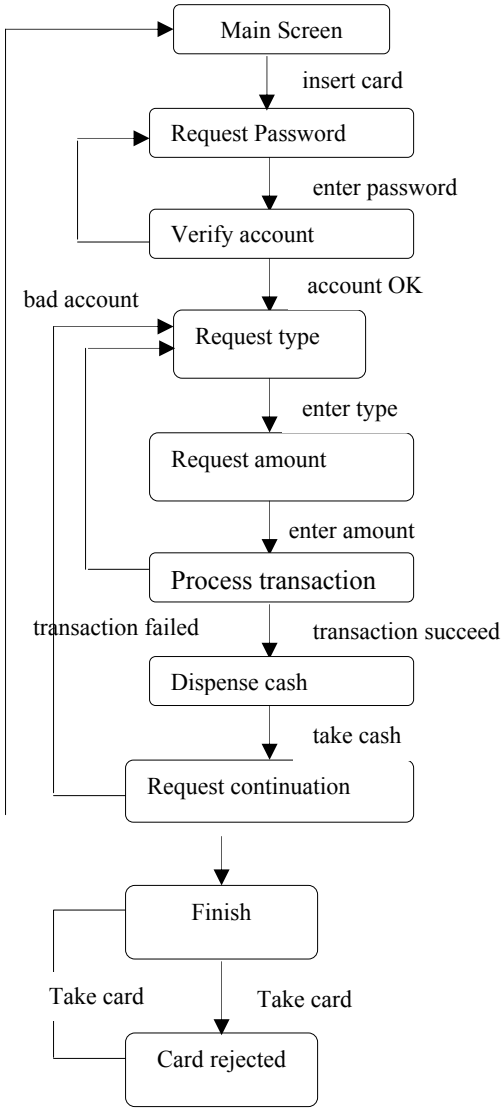


Figure 1: Control of states and events in ATM

Pseudocode of ATM control. The pseudocode for the ATM is given as under:

```
do forever
        display main screen
        read card
        repeat
                ask for password
                read password
                verify account
        until account verification is OK
        repeat
                repeat
                ask for type of transaction
                read type
                ask for amount
                read amount
                start transaction
                wait for it to complete
                until transaction is OK
                dispense cash
                wait for customer to take it
                ask whether to continue
        until user asks to terminate
        eject card
        wait for customer to take card
```

These lines are the pseudocode for the ATM control loop, which is another form of representation of *Figure 1*. Furthermore, you can add cancel event to the flow of control, which could be implemented as goto exception handling code. Now, let us discuss controls as a state machine engine.

3.2.2 Control as a State Machine Engine

First let us define state machine: ***"the state machine is an object but not an application object. It is a part of the language substrate to support the syntax of application object".*** The common approach to implement control is to have some way of explicitly representing and executing state machines. For example, a general "state machine engine" class could provide the capability to execute a state machine represented by a table of transitions and actions provided by the application. As you know, each object contains its own independent state variable and could call on the state engine to determine the next state and action.

This approach helps to quickly progress from the analysis model to a skeleton prototype of the system by defining classes from the object model, state machines from the dynamic model, and creating "stubs" of the action routines. A stub could be stated as *"the minimal definition of a function or subroutine without any internal code"*. Thus, if each stub-prints out its name, this technique allows you to execute the skeleton application to verify that the basic flow of control is correct or not.

State machine mechanisms can be created easily using an object oriented language.

3.2.3 Control as Concurrent Tasks

The term *control as concurrent task* means applying control for those events of the object that can occur simultaneously. An object can be implemented as a task in the programming language or operating system. This is the most general approach of concurrency controls. With this you can preserve the inherent concurrency of real objects. You can implement events as inter-task calls using the facilities of the language, or operating system.

As far as OO programming languages are concerned, there are some languages, such as Concurrent Pascal or Concurrent C++, which support concurrency, but the application of such languages in production environments is still limited. Ada language supports concurrency, provided an object is equated with an Ada task, although the run-time cost is very high. The major object oriented languages do not yet support concurrency.

☞ Check Your Progress 1

1) Briefly explain state diagram by taking one example.

……

……..

2) Explain concurrent task by taking a suitable example.

……

……..…

……..……

3) Explain the following terms.

Event, State, and Operation with respect to the advanced object modeling concept.

……

……

3.3 INHERITANCE ADJUSTMENT

As you know in object oriented analysis and design the terms inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes. As object design progresses, the definitions of classes and operations can often be adjusted to increase the amount of inheritance. In this case, the designer should:

- Rearrange and adjust classes and operations to increase inheritance
- Abstract common behavior out of groups of classes
- Use delegation to share behavior when inheritance is semantically invited.

Rearrange Classes and Operations

Sometimes, the same operation is defined across several classes and can easily be inherited from a common ancestor, but more often operations in different classes are similar, but not identical. By slightly modifying the definitions of the operations or the classes, the operations can often be made to match so that they can be covered by a single inherited operation. The following kinds of adjustments can be used to increase the chance of inheritance:

- You will find that some operations may have fewer arguments than others. The missing arguments can be added but ignored. For example, a draw operation on a monochromatic display does not need a color parameter, but the parameter can be accepted and ignored for consistency with color displays.
- Some operations may have fewer arguments because they are special cases of more general arguments. In this case, you may implement the special operations by calling the general operation with appropriate parameter values. For example, appending an element to a list is a special case of inserting an element into list; here the insert point simply follows the last element.
- Similar attributes in different classes may have different names. Give the attributes the same name and move them to a common *ancestor class*. Then operations that access the attributes will match better. Also, watch for similar operations with different names. You should note that a consistent naming strategy is important to avoid hiding similarities.
- An operation may be defined on several different classes in a group, but be undefined on the other classes. Define it on the common ancestor class and declare it as a no-op on the classes that do not care about it. For example, in OMTool the begin-edit operation places some figures, such as class boxes, in a special draw mode to permit rapid resizing while the text in them is being edited. Other figures have no special draw mode, so the begin-edit operation on these classes has no effect.

Making Common Behavior Abstract

Let us describe abstraction ***"Abstraction means to focus on the essential, inherent aspects of an entity and ignoring its accidental properties"***. In other words, if a set of operations and/or attributes seems to be repeated in two classes. There is a scope of applying inheritance. It is possible that the two classes are really specialised variations of the something when viewed at a higher level of abstraction.

When common behavior has been recognised, a common super class can be created that implements the shared features, leaving only the specialised features in the subclasses. This transformation of the object model is called abstracting out a common super class or common behavior. Usually, the resulting super class is abstract, meaning that there are no direct instances of it, but the behavior it defines belongs to all instances of its subclasses. For example, again we take a draw operation

of a geometric figure on a display screen requires setup and rendering of the geometry. The rendering varies among different figures, such as circles, lines, and spines, but the setup, such as setting the color, line thickness, and other parameters, can be inherited by all figure classes from abstract class figure.

The creation of abstract super classes also improves the extensibility of a software product, by keeping space for further extension on base of abstract class.

Use Delegation to Share Implementation

As we now know, inheritance means the sharing of to the behavior of a super class by its subclass. Let us see how delegation could be used for this purpose. Before we use delegation, let us try to understand that what actually delegation can do.

The term delegation **"*Delegation consists of catching an operation on one object and sending it to another object that is part, or related to the first object. In this process, only meaningful operations are delegated to the second object, and meaningless operations can be prevented from being inherited accidentally*"**. It is true that Inheritance is a mechanism for implementing generalization, in which the behavior of super class is shared by all its subclasses. But, sharing of behavior is justifiable only when a true generalization relationship occurs, that is, only when it can be said that the subclass ***is a form of*** the super class.

Let us take the example of implementation of inheritance. Suppose that you are about to implement a Stack class, and you already have a List class available. You may be tempted to make Stack inherit from List. Pushing an element onto the stack can be achieved by adding an element to the end of the list and popping an element from a stack corresponds to removing an element from the end of the list. But, we are also inheriting unwanted list operations that add or remove elements from arbitrary positions in the list.

Often, when you are tempted to use inheritance as an implementation technique, you could achieve the same goal in a safer way by making one class an attribute or associate of the other class. In this way, one object can selectively invoke the desired functions of another class, by using delegation rather than applying inheritance.

A safer implementation of Stack would delegate to the List class as shown in *Figure 2*. Every instance of Stack contains a private instance of List. The Stack :: push operation delegates to the list by calling its last and add operations to add an element at the end of the list, and the pop operation has a similar implementation using the last and remove operations. The ability to corrupt the stack by adding or removing arbitrary elements is hidden from the client of the Stack class.

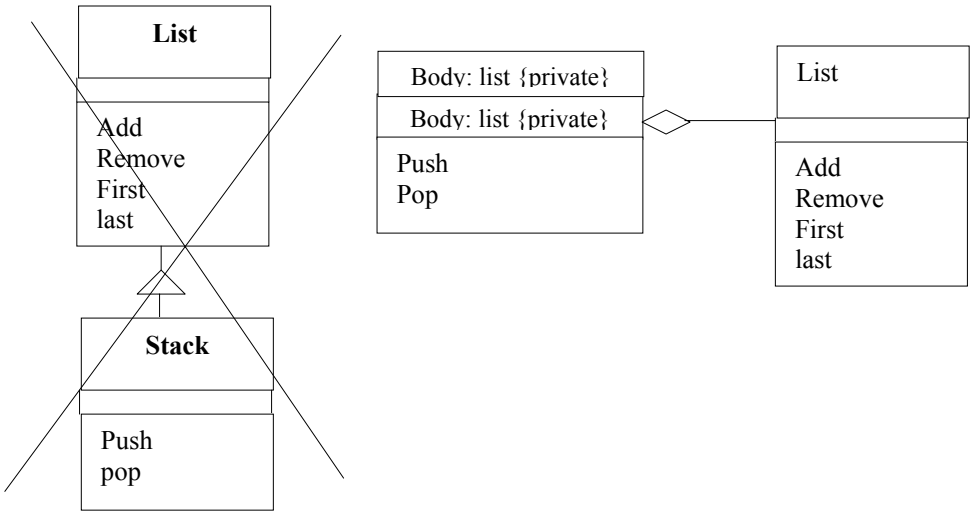


Figure 2: Alternative implementations of a Stack using inheritance (left) and delegation (right)

By *Figure 2,* it is obvious that we should discourage the use of inheritance to share the operations between two related classes. Instead, we should use delegation so that one class can selectively invoke the desired functions of another class. Now, you are aware of the concept of inheritance and its adjustment. In the next section, we will discuss association design and different types of associations.

3.4 ASSOCIATION: DESIGN

Before we define association design let us define association ***"Association is the group of links between two objects in an object model".*** It is helpful in an finding paths between objects. It is a conceptual entity, which can be used for modeling and analysis. At the final phase of advance object design, you must use strategy for applying association in the object model. Association is also defined as "*a group of links between two objects with common structure and common semantic*".

Analyzing Association Traversal

Association Traversal should be understood properly for an association design explanation. Analyzing association traversal means analyzing traversal between the objects. Associations are inherently bi-directional, which is certainly true in an abstract sense. But, if some associations in your application are only traversed in one direction, in this case implementation can be simplified.

One-way Associations

If an association is only traversed in one direction, then it is called one-way association. It is implemented as a pointer, or an attribute that contains an object reference. If the multiplicity is "one", as shown in *Figure 3*, then it is a simple pointer; otherwise, if the multiplicity is "many", then it is a set of pointers. If the "many" end is ordered, then a list can be used, instead of a set. A qualified association with multiplicity "one" can be implemented as a dictionary object.

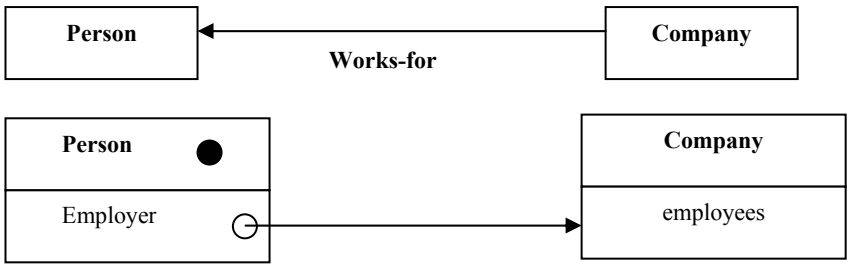


Figure 3: Implementation of one-way association using pointers

Two-way Associations

Many associations are traversed in both directions, and these are called two-way associations. You may observe that it is not essential to have some frequency of traversal from both sides. It can be implemented by using the following three methods:

- Implement as an attribute in one direction only, and perform a search when a backward traversal is required. This approach is useful only if there is a great disparity in traversal frequency in the two directions, and when minimizing both the storage cost and the update cost are important. It is observed that the rare backward traversal will be expensive.

- You should try to implement the attributes in both directions, as shown in *Figure 4.* This approach is good because it permits fast access, but if either attribute is updated then the other attribute must also be updated to keep the link consistent. This approach is useful in the case to access outnumber updates.

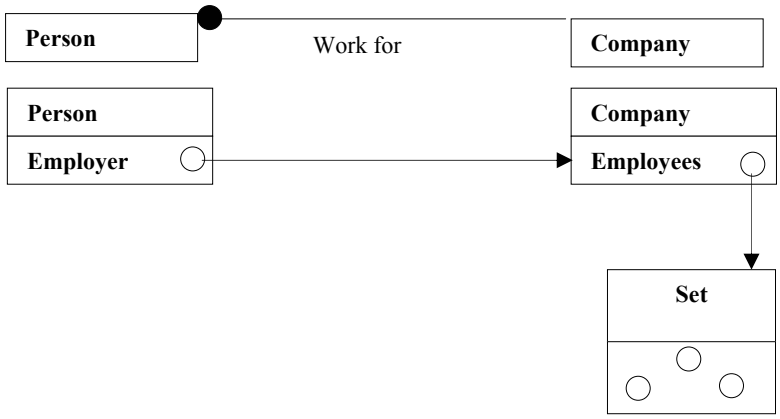


Figure 4: Implementation of two-way association using pointers

- Implement as a distinct association object, independent of either class, as shown in *Figure 4.* An association object is a set of pairs of associated objects stored in a single variable-size object. For efficiency, you can implement an association object using two dictionary objects, one for the forward direction and other for the backward direction. This idea is useful for extending predefined, the classes from a library which cannot be modified. Distinct association objects are also useful for *sparse associations*. In *sparse associations* most objects of the classes do not participate because space is used only for actual links.

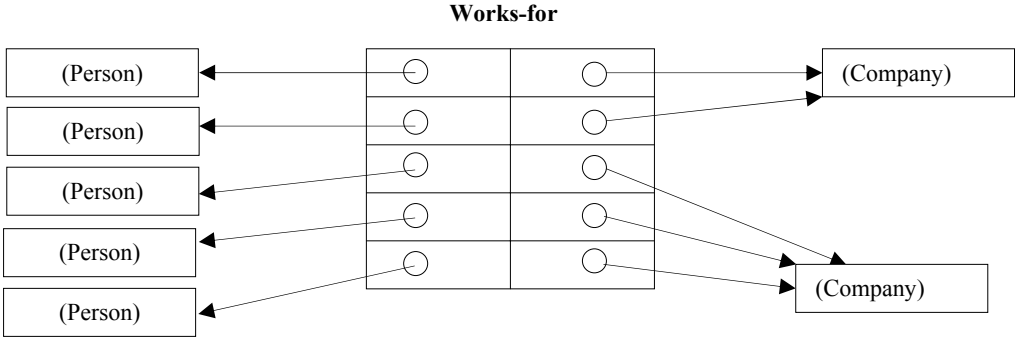


Figure 5: Implementation of association as an object

Objects are representated using certain symbols. Now we will discuss object representations.

3.5 OBJECT REPRESENTATION

The term object representation means ***"to represent object by using objects model symbols"***. Implementing objects is very simple. The object designer decides the use of primitive types or to combine groups of related objects in their representation.

We can define a class in terms of other class. The classes must be implemented in terms of built-in primitive data types, such as integers, strings, and enumerated types. For example, consider the implementation of a social security number within an employee object which is shown in *Figure 6*. The social security number attribute can be implemented as an integer or a string, or as an association to a social security number object, which itself can contain either an integer or a string. Defining a new class is more flexible, but often introduces unnecessary indirection. It is suggested that new classes should not be defined unless there is a definite need it.

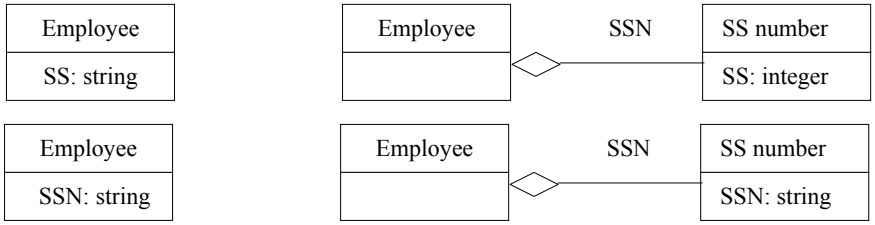


Figure 6: Alternative representations for an attribute

In a similar way, the object designer decides whether to combine groups of related objects or not.

☞ Check Your Progress 2

1) Explain inheritance with support of suitable example.

……..….

..……

…….

2) Describe the association design of an object by giving one example of it.

……..…

…….

…….

3) The definition of classes and operation can often be adjusted to increase the amount of inheritance". Justify this statement.

…….

…….

…….

Optimization is one of the areas of computing which gets great importance and considerations. Now, let us discuss the optimization possibilities of a design.

3.6 DESIGN OPTIMIZATION

In the previous Section, we have seen various ways of representing objects. This Section will cover very interesting and important aspects of design optimization. The basic design model uses the analysis model as the framework for implementation. The analysis model captures the logical information about the system. To get better result, the design model should contain details to support efficient information access. The inefficient, but semantically-correct analysis model can be optimized to make implementation more efficient, but an optimized system is more obscure, and less likely to be reusable in another context. For the design optimization, as a designer, you must strike an appropriate balance between efficiency and clarity.

During design optimization as a designer you must keep the following points in his mind:

- Add redundant associations to minimize access cost, and to maximize convenience
- Rearrange the computation for greater efficiency up to possible instant.
- Save derived attributes to avoid recomputation of complicated expressions.

Now let us discuss these issues one by one:

Adding Redundant Associations for Efficient Access

The term redundant association means using ***"duplicate association for efficient access"***. During analysis, it is not a good idea to have redundancy in the association network because redundant associations do not add any information. During design, however, you should evaluate the structure of the object model for implementation.

This can be done by asking questions:

i) Is there a specific arrangement of the network that would optimize critical aspects of the completed system?
ii) Will adding new associations that were useful during analysis restructure the network? All this sometimes may not produce the most efficient network, one that can handle complex access patterns, as well as the related frequencies of various kind of access.

To describe the analysis of access paths, consider the example of the design of a company's employee skills database. A part of the object model from the analysis phase is shown in *Figure 7.* The operation *Company:: find-skill* returns a set of persons in the company with a given skill. For example, we may ask for the data of all employees who speak Japanese.

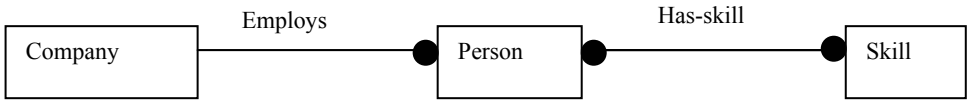


Figure 7: Chain of associations between objects

For this example, suppose that the company has 1000 employees, each of whom has 10 skills on average. A simple nested loop would traverse Employs 1000 times and ***Has-skill*** 10,000 times. If only 5 employees actually speak Japanese, then the test-to-hit ratio is 2000.

Now, let us see whether this figure can be improved or not. Actually, we can make many possible improvements in this Figure. First, *Has-skill need not be* implemented as an unordered list a hashed set. The hashing can be performed in a fixed interval of time so that the cost of testing whether a person speaks Japanese is constant, provided a unique skill object represents speaks Japanese. This rearrangement reduces the number of tests from 10,000 to 1,000, or one per employee.

For those cases where the number of hits from a query is low (since only a fraction of objects satisfy the test) we can build an index to improve access to objects that must be frequently retrieved. For example, we can add a qualified association **Speaks language** from **Company** to **Employee**, where the qualifier is the language spoken (*Figure 8*). This permits us to immediately access all employees who speak a particular language with no wasted accesses. But there is a cost to the index: "***It requires additional memory, and it must be updated whenever the base associations are updated***". The object designer must decide when it is useful to build indexes. Here, we have to consider the case where most queries return all of the objects in the search path, then an index really does not save much because the test-to-hit ratio in this case is very close to 1.

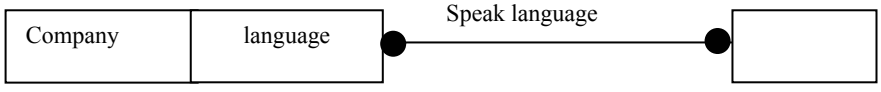


Figure 8: Index for personal skills database

From *Figure 8,* it is obvious that speaks language is a derived association, defined in terms of underlying base associations. The derived association does not add any information to the network, but permits the model information to be accessed in a more efficient manner.

You can analyse the use of paths in the association network in the following steps:

- Examine each operation and see what associations must traverse in order to obtain information. For this it is not necessarily those associations traverse in both directions.

For each operation, you should note the following points:

- How often is the operation called? How costly is the operation to perform?

- What is the "***fan-out***" along a path through the network? Estimate the average count of each "***many***" association encountered along the path. Multiply the individual ***fan-outs*** to obtain the ***fan-out*** of the entire path, which represents the number of accesses on the last class in the path. Note that "***one***" links do not increase the ***fan-out***, although they increase the cost of each operation slightly. But there is no need to worry about such small effects.

- What is the fraction of "***hits***" on the final class, (objects that meet selection criteria, if any, and are operated on? If most objects are rejected during the traversal for some reason, then a simple nested loop may be inefficient for finding target objects.

Rearrange the Execution Order for Efficiency

Rearranging the execution order for efficiency means executing such job which has less execution time. By rearranging the object in the increasing order of their execution time, we can increase the efficiency of the system.

After adjusting the structure of the object model to optimize frequent traversals, the next thing to optimize is the algorithm itself. Actually, "data structure and algorithms are directly related to each other", but we find that usually the data structure should be considered first.

The way to optimize an algorithm is *"to eliminate dead paths as early as possible"*. For example, suppose we want to find all employees who speak both Japanese and French, and suppose 5 employees speak Japanese and 100 speak French. In this case, it is better to test and find the Japanese speakers first, then test if they speak French. In general, it pays to narrow the search as soon as possible. Sometimes the execution order of a loop must be inverted from the original specification in the functional model to get efficient results.

Saving Derived Attributes to Avoid Recomputation

As we have already discussed, ***"redundancy means duplication of same data".*** *But, If multiple copies of the same data is present in a system, then it can increase availability of data, but the problem of computing overhead is also associated with it*. To overcome this problem we can "***cache"*** or store redundant data in its computed form and objects or classes may be defined to retain this information. The class that contains the cached data must be updated if any of the objects that it depends on are changed.

Figure 9 shows a use of a derived object and derived attribute in OMTool. Each class box contains an ordered list of attributes and operations, each represented as a text string (left of diagram). Given the location of the class box itself, the location of each attribute can be computed by adding up the size of all the elements in front of it. Since the location of each element is required frequently, the location of each attribute string is computed and stored. The region containing the entire attribute list is also computed and saved. In this way we can avoid the testing of input points against attribute text element. It is shown on the right side in *Figures 9* (a) and (b). If a new attribute string is added to the list, then the locations of the ones after it in the list are simply offset by the size of the new element.

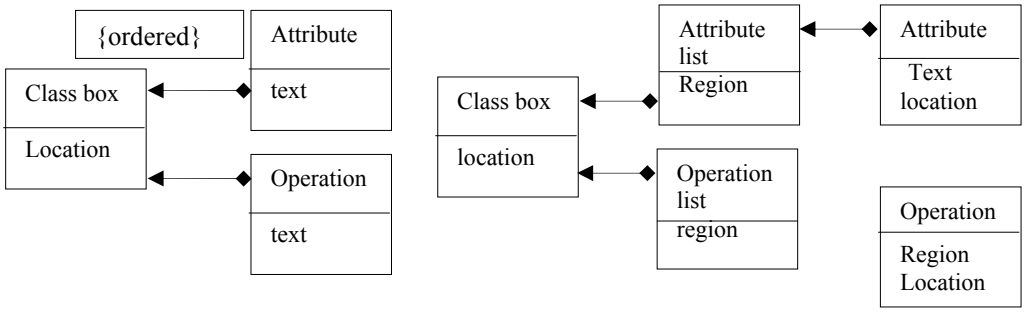


Figure 9 (a) **Figure 9 (b)**

Figure 9: Derived attribute to avoid recomputation

You can see the use of an association as a cache which is shown in *Figure 10*. In this figure a sheet contains a priority list of partially overlapping elements. If an element is moved or deleted, the elements under it must be redrawn. Scanning all elements in front of the deleted element in the priority list of the sheet, and comparing them to the deleted element can uncover overlapping elements. If the number of elements is large, this algorithm grows *linearly in the number of elements*. The Overlaps association stores those elements that overlap an object, and precede it in the list. This association must be updated when a new element is added to it, but testing for overlap using the association is more efficient.

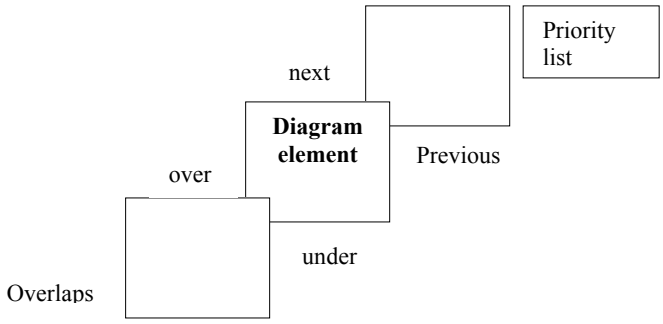


Figure 10: Association as a cache

After the base value is changed, you should update derived attributes. Now, the question is "how to recognise the need of update. There are three ways to recognize when an update is needed: by *explicit code*, *by periodic recomputation*, or by *using active values*. Now, let us three ways one by one.

- **Explicit update:** In explicit update, each derived attribute is defined in terms of one, or more fundamental base object(s). The object designer determines which derived attributes are affected by each change to a fundamental attribute, and inserts code into the update operation on the base object to explicitly update the derived attributes that depend on it.
- **Periodic recomputation:** Base values are often updated in bunches. Sometimes, it is possible to simply recompute all the derived attributes periodically without recomputing derived attributes after each base value is changed. Recomputation of *all derived attributes* can be more efficient than incremental update because some derived attributes may depend on several base attributes, and might be updated more than once by an *incremental approach*. Also, periodic recomputation is simpler than explicit updates and less prone to bugs. On the other hand, if the data set changes incrementally, a few objects at a time, periodic recomputation is not practical because too many derived attributes must be recomputed when only a few are affected.
- **Active values:** An active value is a value that has dependent values. Each dependent value registers itself with the active value, which contains a set of dependent values and update operations. An operation to update the base value triggers updates of all the dependent values, but the calling code need not explicitly invoke the updates.

Now, let us discuss design documentation in the designing of an object.

3.7 DESIGN DOCUMENTATION

The Design Document should be an extension of the Requirements Analysis Design. "**The Design Document will include a revised and much more detailed description of the Object Model**" in both graphical form (object model diagrams) and textual form (class descriptions). You can use additional notation to show implementation decisions, such as arrows showing the traversal direction of associations and pointers from attributes to other objects.

The Functional Model can also be extended during the design phase, and it must be kept current. It is a seamless process because object design uses the same notation as analysis, but with more detail and specifics. It is good idea to specify all operation interfaces by giving their arguments, results, input-output mappings, and side effects.

Despite the seamless conversion from analysis to design, it is probably a good idea to keep the Design Document distinct from the Analysis Document. **Because of the shift in viewpoint from an external user's view to an internal implementer's view, the design document includes many optimizations and implementation artifacts**. It is important to retain a clear, user-oriented description of the system for use in validation of the completed software, and also for reference during the maintenance phase of the object modeling.

☞ Check Your Progress 3

1) Improve the object diagram in *Figure 11* by generalizing the classes Ellipse and Rectangle to the class Graphics primitive, transforming the object diagram so that there is only a single one-to-one association to the object class Boundary. In effect, you have to changing the 0,1 multiplicity to exactly one multiplicity. As it stands, the class Boundary is shared between Ellipse and Rectangle. A Boundary is the smallest rectangular region that will contain the associated Ellipse or Rectangle.

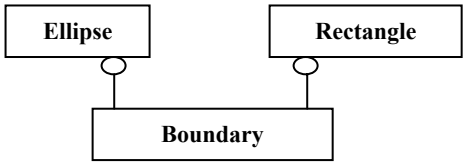


Figure 11: Portion of an object diagram with a shared class

……

……..……

2) Assign a data type to each attribute in *Figure 12.*

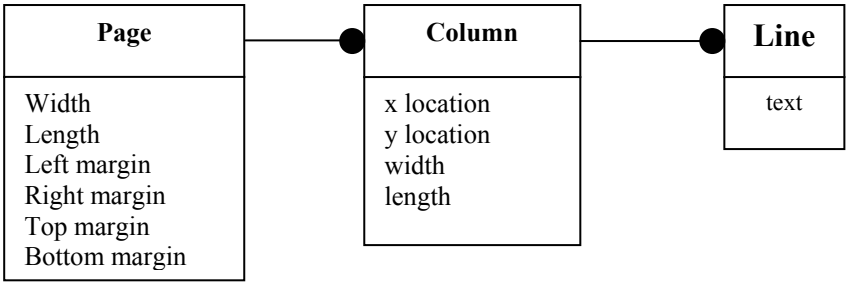


Figure 12: Portion of an object diagram of a newspaper

……

……..……

3) Improve the object diagram in *Figure 13* by transforming it, adding the class (Political party). Associate Voter with a party. Discuss why the transformation is an improvement.

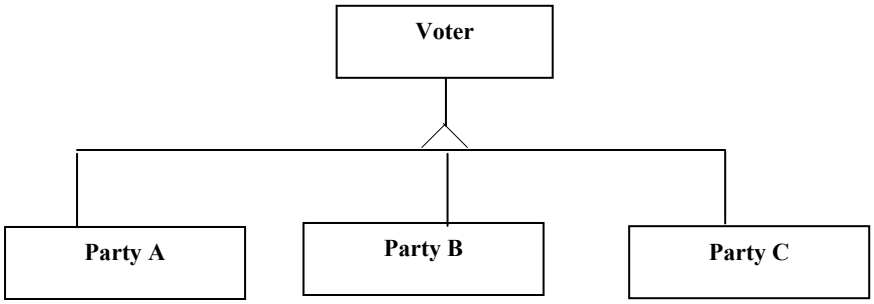


Figure 13: Object diagram representing voter membership in a political party

……

……..……

……..

……..

4) *Figure 14* is a state diagram for a garage door opener. Implement it by using state as, location within a program. You may use pseudocode, or any structured programming language.

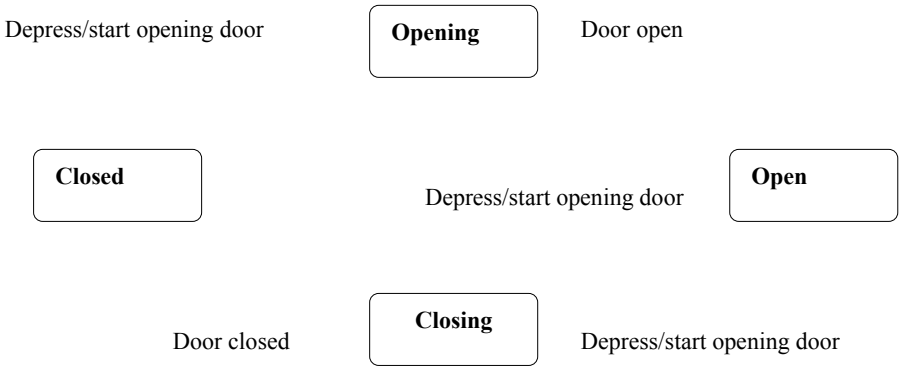


Figure 14: State diagram for a garage door opener

3.8 SUMMARY

Object design follows analysis and system design. The object design phase adds implementation details, such as restructuring classes for efficiency, internal data structures and algorithms to implement operations, implementation of control, implementation of associations, and packaging into physical modules. Object design extends the analysis model with specific implementation decisions and additional internal classes, attributes, associations, and operations.

During object design, the definitions of internal classes and operations can be adjusted to increase the amount of inheritance. These adjustments include modifying the argument list of a method, moving attributes and operations from a class into a super class, defining an abstract super class to cover the shared behavior of several classes, and splitting an operation into an inherited part and a specific part. Delegation should be used rather than inheritance when a class is similar to another class but not truly a sub class.

Associations are the “glue” of our object model, which provides access paths between objects. An association traversed in a single direction can be implemented as an attribute pointing to another object, or a set of objects, depending on the multiplicity of the association. A bi-directional association can be implemented as a pair of pointers, but operations that update the association must always modify both directions of access. Associations can also be implemented as association objects. The exact representation of objects must be chosen. At some point, user-defined objects must be implemented in terms of primitive objects, or data types supplied by the programming language. Some classes can be combined. Programs must be packaged into physical modules for editors and compilers, as well as for the convenience of programming teams. Design decisions should be documented by extending the analysis model, by adding detail to the object, dynamic, and functional models.

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) **State Diagram**

An object can receive a sequence of input instructions. The state of an object can vary depending upon the sequence of input instructions. If we draw a diagram which will represent all the processes (input) and their output (states) then that diagram is known as state diagram. Processes are represented by arrow symbol, and states by oval symbol. For example, the screen of ATM machine has many states like main screen state, request password state, process transaction state, etc.

2) **Concurrent Task**

The simultaneous occurrence of more than one event is called a concurrent task. Operating systems can handle concurrent tasks efficiently. The Air Traffic Control system (ATC) for examples, can manage concurrent tasks in fractions of a second.

3) **Event**

Happening of a process is called event. In other words, an object can receive many input instructions. The changes that occur due to these instructions are called events. For example, tossing a coin is input, but the appearance of HEAD or TAIL is an event.

State

The position of an object at any moment is called its state. An object can have many states. After receiving some input instructions, an object can change its state from one to another.

Check Your Progress 2

1) **Inheritance**

Inheritance is one of the cornerstones of object-oriented programming language because it allows a creation of hierarchical classifications. Using inheritance you can create a general class that defines trails common, to a set of related items. More specific classes can inherit this class, and each could add a certain unique thing to the resulting new class. The class that inherits from another, class, or classes is called a derived class or subclass, and the class/classes from which the derived class is made is called, a base class or a super class.

For example, racing cars. Pick up cars and saloons, etc. are all different kinds of cars in object-oriented terminology racing cars. Pick up cars, and saloons, etc. are all subclasses of the car class. Similarly, as illustrated in Figure15, below, the car class is the super class of sub classes like racing cars, saloons, sedans, convertibles, etc.

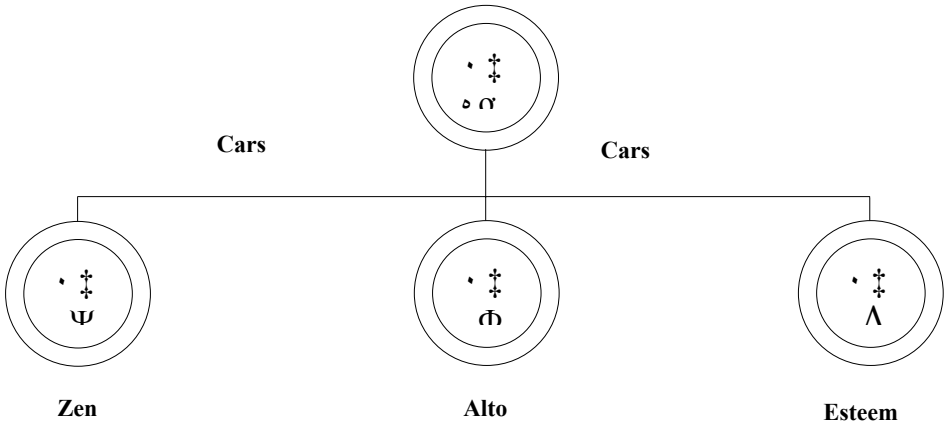


Figure 15: Hierarchy of Classes

Inheritance can be of various types, such as:

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance

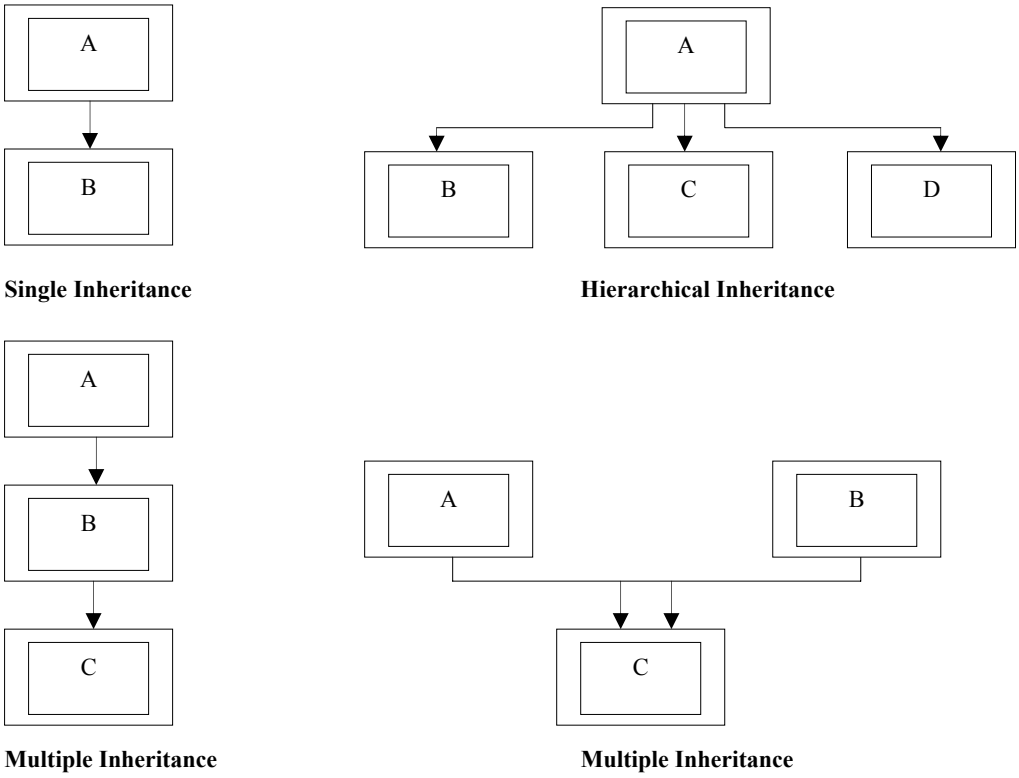


Figure 16: Forms of inheritance

2) The Design of Associations

Associations are the "glue" of advanced object oriented analysis and the design model. Association provides access paths between the objects. It is a type of conceptual entity that can be used for analysis and modeling of an object. For example:

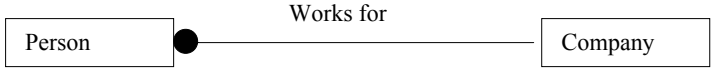


Figure 17: A simple form of Association

In the above example, there are two objects, person and company. These two objects are linked (associated) with each other by a relation called works for.

3) Adjustment of Inheritance

The definition of classes and operations can often be adjusted to increase the amount of inheritance between the objects. The object designer can rearrange and adjust classes to increase the inheritance among the different objects and classes. Sometimes the same operation is defined across several classes and can easily be inherited from a common ancestor. By slightly modifying the definitions of the operations, or the classes, the operation often can be made to match. We also can extract common behavior out of groups of classes to increase the inheritance. Similar attributes in different classes may have different names, but by giving some common name and moving them to ancestor class we can increase inheritance. An operation may be defined on several different classes in a group but be undefined on the other classes. To increase inheritance, we can define it on the common ancestor class and declare it as a no-op on the classes that do not care about it. We can also use DELEGATION instead of inheritance to share only meaningful attributes between a super class and its sub class.

Check Your Progress 3

1) The improved generalized object diagram of class Ellipse and Rectangle is given in the Figure below.

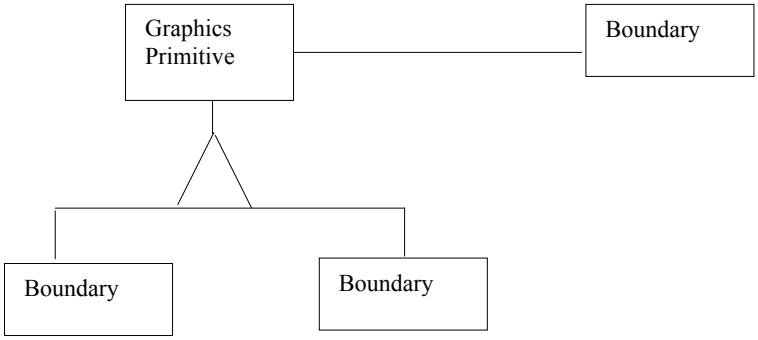


Figure: 18 Object Diagram

Here, in this diagram, class Graphics primitive is the generalized class of both Ellipse and Rectangle. The class Ellipse and Rectangle have single association with class boundary. This single association is shown by drawing a line between class boundary and generalized class Graphics primitive.

2) A derived association supports direct traversal from Page to Line. The line-page association is derived by composing the line column and column page association.

This association can be traversed from lines to pages. The Figure is shown below.

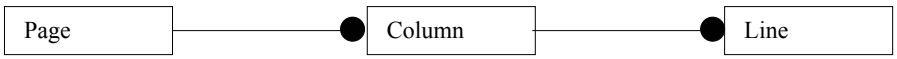


Figure 19: Association between different objects of a news paper.

The data type to the attributes of the Figure is

Attribute	**Data type**
Width	integer
Length	integer
Left margin	real
Right margin	real
Top margin	real
Bottom margin	real
X location	x: real
Y location	y: real

3) The improved object diagram of object voter and object political party is shown in the diagram below.

Figure 20: Improved object diagram for representing voter membership in a political party

Political party membership is not an inherent property of a voter but a changeable association. The revised model better represents voters with no party affiliation and permits changes in party membership. If voters belong to more than one party, then the multiplicity could easily be changed. Parties are instances of class Political Party and need not be explicitly listed in the model: new parties can be added without changing the model, and attributes can be attached to parties.

4) The Pseudo code for a garage door opener is listed below:

```
<closed>    wait for depress event
<opening>   start opening door
            wait for door opening event
<open>      wait for depress event
<closing>   start closing door
            wait for either depress or door closed event:
                  if depress event then go to opening
                  if door closed event then go to closed
```

In the above pseudo code we are using go to as a jump command to jump from one statement to other.

UNIT 1 OBJECT MODELING

Structure **Page Nos.**

1.0 Introduction 5
1.1 Objectives 5
1.2 Advanced Modeling Concepts 5
1.2.1 Aggregation
1.2.2 Abstract Class
1.3 Multiple Inheritance 9
1.4 Generalization and Specialisation 11
1.5 Meta Data and Keys 13
1.6 Integrity Constraints 14
1.7 An Object Model 17
1.8 Summary 18
1.9 Solutions/Answers 18

1.0 INTRODUCTION

In the previous Blocks of this Course we learned the differences between OOA and OOAD and how UML is used for visualizing, specifying, constructing, and documenting.

The goal of object design is to identify the object that the system contains, and the interactions between them. The system implements the specification. The goals of object-oriented design are:

(1) More closely to problem domain

(2) Incremental change easy

(3) Supports reuse. Objects during Object Oriented Analysis OOA focuses on problem or in other word you can say semantic objects. In Object Oriented Design we focuses on defining a solution. Object Oriented modeling is having three phases object modeling, dynamic modeling, and function modeling. In this Unit we will discuss the concepts of object modeling. We will also discuss aggregation, multiple inheritance, generalisation in different form and metadata.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- describe and apply the concept of generalisation;
- understand and apply the concepts abstract Class, multiple Inheritance;
- apply generalisation as an extension;
- apply generalisation as a Restriction, and
- explain the concept of Metadata and constraints.

1.2 ADVANCED MODELING CONCEPTS

You have to follow certain steps for object-oriented design.

These steps for OO Design methodology are:

1) produce the object model
2) produce the dynamic model
3) produce the functional model
4) define the algorithm for major operation
5) optimize and package.

The first step for object-oriented designing is object modeling. Before we go into details of object modeling first of all we should know "what is object modeling"? You can say that **Object** modeling identifies the objects and classes in the problem domain, and identifies the relationship between objects.

In this whole process first of all we have to identify objects, then structures, attributes, associations, and finally services.

1.2.1 Aggregation

Aggregation is a stronger form of association. It represents the **has-a** or **part-of** relationship. An aggregation association depicts a complex object that is composed of other objects. You may characterize a house in terms of its roof, floors, foundation, walls, rooms, windows, and so on. A room may, in turn be, composed of walls, ceiling, floor, windows, and doors, as represented in *Figure 1*.

In UML, a link is placed between the "whole "and "parts" classes, with a diamond head (*Figure1*) attached to the whole class to indicate that this association is an aggregation. Multiplicity can be specified at the end of the association for each of the **part-of** classes to indicate the constituent parts. The process of decomposing a complex object into its component objects can be extended until the desired level of detail is reached.

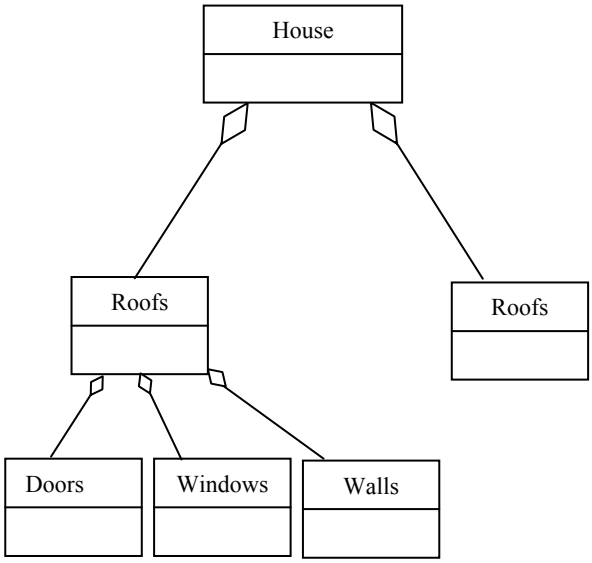


Figure 1: A house and some of its component

You can see that object aggregation helps us describe models of the **real world** that are composed of other models, as well as those that are composed of still other models. Analysts, at the time of describing a complex system of aggregates, need to describe them in enough detail for the system at hand. In the case of a customer order and services, a customer order is composed not only of header information, but also the detail lines as well. The header and detail lines may each have public customer comments and private customer service comments attached. In an order entry system, detailed technical information about a product item appearing on a customer order line may be accessible as well. This complex object-called an order can be very naturally modeled using a **series of aggregations**. An order processing system can then be constructed to model very closely the **natural aggregations** occurring in the real world.

Aggregation is a concept that is used to express "part of" types of associations between objects. An aggregate is, a conceptually, an **extended object** viewed as a Unit by some operations, but it can actually be composed of **multiple objects**. One aggregate may contain multiple **whole-part** structures, each viewable as a distinct aggregate. Components may, or may not exist in their own right and they may, or may

not appear in multiple aggregates. Also an aggregate's components may themselves have their own components.

Aggregation is a **special kind of association**, adding additional meaning in some situations. Two objects form an aggregate if they are tightly connected by a whole-part relationship. If the two objects are normally viewed as independent objects, their relationship is usually considered an association. **Grady Booch** suggests these tests to determine whether a relationship is an aggregation or not:

- Would you use the phrase "**part of**" to describe it?
- Are some operations **on the whole** automatically applied to its parts?
- Are some attribute values propagated from the **whole to all** or **some parts**?
- Is there an **intrinsic asymmetry** to the association, where one object class is **subordinate to the other**?

If your answer is yes to any of these questions, you have an aggregation. An **aggregation** is not the same as a **generalization**. Generalization relates distinct classes as a way of structuring the definition of a single object. Super class and subclass refer to properties of one object. A generalization defines objects as an instance of a super class and an instance of a subclass. It is composed of classes that describe an object (often referred to as a kind of relationship). Aggregation relates object instances: **one object that is part of another**. Aggregation hierarchies are composed of object occurrences that are each part of an **assembly** object (often called a **part of** relationship). A complex object hierarchy can consist of both **aggregations and generalizations**.

Composition: **A stronger form of aggregation** is called **composition**, which implies exclusive ownership of the part of classes by the whole class. This means that parts may be created after a composite is created, but such parts will be explicitly removed before the destruction of the composite. In UML, filled diamonds, as shown in *Figure 2*, indicate the composition relationship.

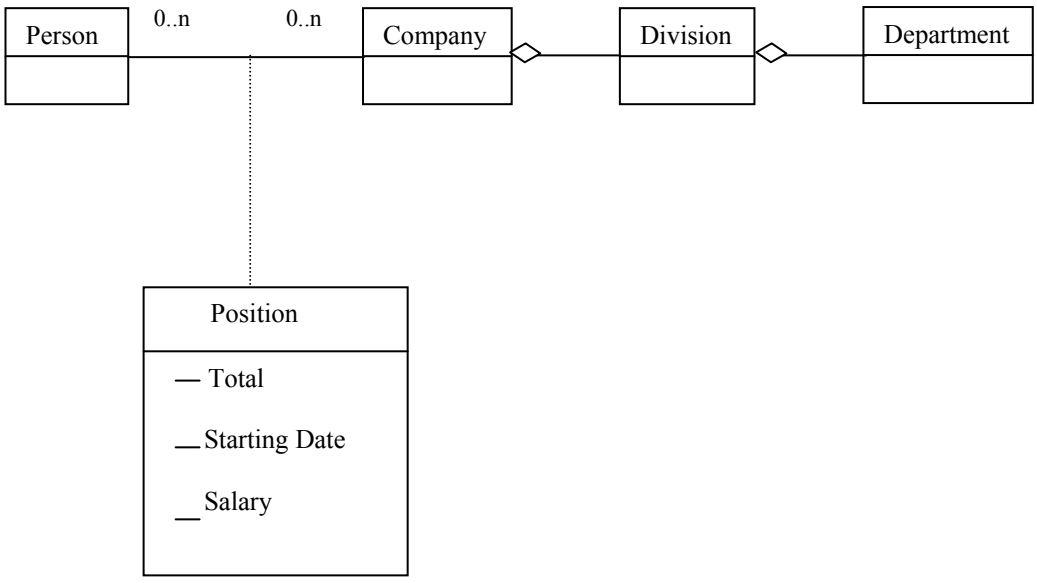


Figure 2: Example of a composition

Figure 2 shows that a person works for a company, company has many division, which are part of company and each division has many departments, which are again part of division.

1.2.2 Abstract Class

An abstract class is used to **specify the required behaviors** (operations, or method in java) of a class **without having to provide their actual implementations**. In other

words you can say that methods without the implementation (body) are part of abstract classes.

An abstract object class has no occurrences. Objects of abstract classes are not created, but have child categories that contain the actual occurrences. A "concrete" class has actual occurrences of objects. If a class has at least one abstract method, it becomes, by definition, an abstract class, because it must have a subclass to override the abstract method. An abstract class must be sub classed; you cannot instantiate it directly.

Here you may ask one question: Why, are abstract classes are created? So the answer to this question is:

- To organise many specific subclasses with a super class that has no concrete use
- An abstract class can still have methods that are called by the subclasses. You have to take is this correct point in consideration in case of abstract classes
- An abstract method must be overridden in a subclass; you cannot call it directly
- Only a concrete class can appear at the bottom of a class hierarchy.

Another valuable question to ask is: why create the abstract class method? Answer to this question is:

- To contain functionality that will apply to many specific subclasses, but will have no concrete meaning in the super class.

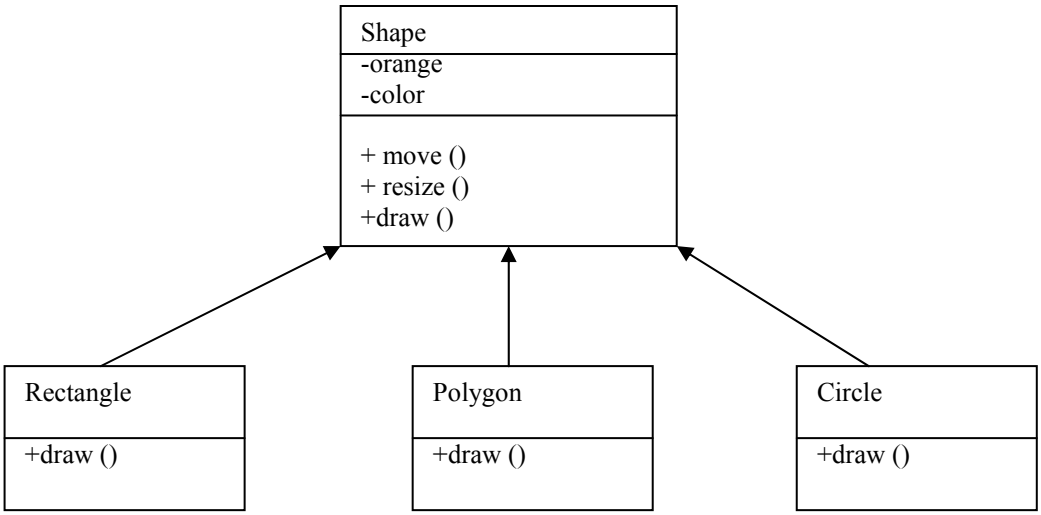


Figure 3: How abstract and concrete classes are related. Consider the example of shapes.

For example in *Figure 3*, you can see that the shape class is a natural super class for triangle, circle, etc.

Every shape requires a **draw () method**. But the method has no meaning in the Shape super class, so we make it abstract.

The subclasses provide the actual implementations of their draw methods since Rectangle, Polygon and Circle can be drawn in different ways. A subclass can override the implementation of an operation inherited from a super class by declaring another implementation. In this example, the draw method of rectangle class overrides the implementation of the draw operation inherited from the Shape class. The same applies to draw methods of Polygon and Circle.

Abstract classes can appear in the real world and can be created by modelers in order to promote **reuse of data and procedures** in systems. They are used to relate concepts that are common to multiple classes. An abstract class can be used to model an abstract super class in order to group classes that are associated with each other, or

are aggregated together. An abstract class can define methods to be inherited by subclasses, or can define the **procedures for an operation** without defining a **corresponding method**. The abstract operation defines the pattern or an operation, which each concrete subclass must define in its implementation in their own way.

☞ Check Your Progress 1

Give the right choice for the followings:

1) A class inherits its parent's….

   (a) Attribute, links
   (b) Operations
   (c) Attributes, operations, relationships
   (d) Operations, relationships, link

   ……

   ……

2) If you wanted to organise elements into reusable groups with full information hiding you would use which one of the following UML constructs?

   (a) Package
   (b) Class
   (c) Class and interface
   (d) Sub-system or component

   ……

   ……

3) Which of the following is not characteristic of an object?

   (a) Identity
   (b) Behavior
   (c) Action
   (d) State

   ……

   ……

   ……

4) Which of the following is not characteristic of an abstract class?

   (a) At least one abstract method
   (b) All the method have implementation (body)
   (c) Subclass must implement abstract method of super class
   (d) None of the above

   ……

   ……

Now, you are familiar with aggregation generalization, and abstract classes. As further extension of object oriented concepts, in the next section we will discuss multiple inheritance.

1.3 MULTIPLE INHERITANCE

Inheritance allows a class to inherit features from parent class (s). Inheritance allows you to create a new class from an existing class or existing classes.

Inheritance gives you several benefits by letting you:

- Reduce duplication by building on what you have created and debugged
- Organise your classes in ways that match the real world situations and entities.

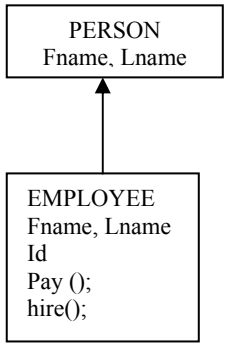


Figure 4: Example of Inheritance

For example, in *Figure 4*, you can see that EMPLOYEE class is inherited from PERSON class.

Multiple inheritance extends this concept to allow a class to have more than one parent class, and to inherit features form all parents. Thus, information may be mixed from **multiple sources**. It is a more complex kind of generalization; multiple inheritances does not restrict a class hierarchy to a tree structure (as you will find in single inheritance). Multiple inheritance provides greater modeling power for defining classes and enhances opportunities for reuse. By using multiple inheritance object models can more closely reflect the structure and function of the real world. The disadvantage of such models is that they become more complicated to understand and implement. See *Figure 5* for an example of multiple inheritance. In this example, the VAN classes has inherited properties from cargo Vehicle and Passenger vehicle.

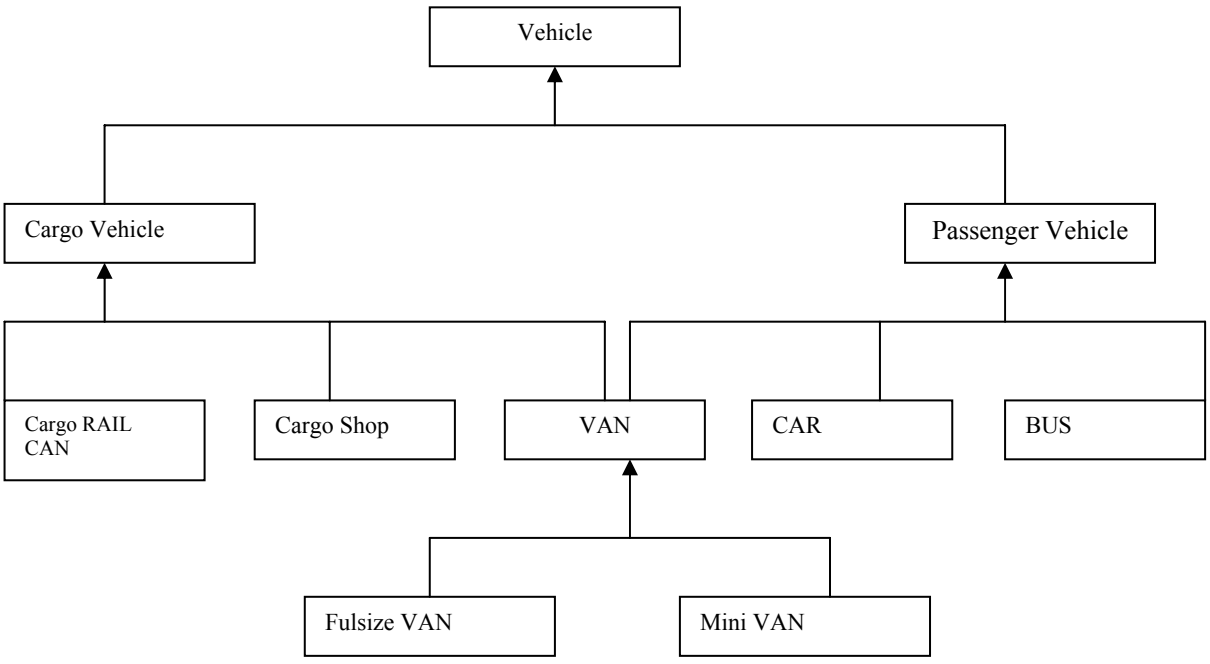


Figure 5: Example of Multiple Inheritance

The advantage of multiple inheritance is that it facilitates the re-use existing classes much better than single inheritance. If the properties of two existing classes have to be re-used and only single inheritance was available, then one of the two classes would have to be changed to become a subclass of the other class.

However, multiple inheritance should be used rather carefully. As the inheritance relationships that will be created through multiple inheritance may become rather complex and fairly difficult to understand. It is seen as a controversial aspect of object–orientation and, therefore, not implemented in some object-oriented languages, such as Smalltalk, because sometimes multiple inheritance can lead to ambiguous situations.

You will observe that:

- Working with multiple inheritance can be difficult in implementation if only single inheritance is supported, but analysis and design models can be restructured to provide a usable model. ***Rumbaugh et al***. discusses the use of delegation as an implementation mechanism by which an object can forward an operation to another object for execution. The recommended technique for restructuring-includes:

  Delegation using an aggregation of **roles a super class** with multiple independent generalizations can be recast as an aggregate in which each component replaces a generalization.

For this you have to:

- Inherit the most important class and delegate the rest. Here a join class is made a subclass of its most important super class.
- Nested generalization: factor on one generalization first, then the other, multiplying out all possible combinations.

Rumbaugh suggests issues to consider when selecting the best work around:

- If subclass has several super classes, all of equal importance, it may be best to use delegation, and preserve symmetry in the model
- If one super class clearly dominates and the others are less important, implementing multiple inheritance via single inheritance and delegation may be best
- If the number of combinations is small, consider nested generalization otherwise, avoid it
- If one super class has significantly more features than the other super classes, or one super class is clearly the performance bottleneck, preserve inheritance through this path
- If nested generalization is chosen, factor in the most important criterion first, then the next most important, etc.
- Try to avoid nested generalization if large quantities of code must be duplicated
- Consider the importance of maintaining strict identity (only nested generalization preserves this). Now, let us discuss the concept and specialization of generalization which is very important in respect of object oriented modeling.

1.4 GENERALIZATION AND SPECIALIZATION

Generalization means extracting common properties from a collection of classes, and placing them higher in the inheritance hierarchy, in a super class.

Generalization and **specialization** are the **reverse of** each other. An object type hierarchy that models generalization and specialization represents the most general concept at the top of an object type: hierarchy as the **parent** and the more specific object types as **children**.

Much care has to be taken when generalizing (as in the real world) that the property makes sense for **every single subclass** of **the super class**. If this is not the case, the property **must not** be generalized.

Specialization involves the definition of a new class which inherits all the characteristics of a **higher class** and **adds some new ones**, in a subclass.
Whether the creation of a particular class involves first, or second activity depends on the stage and state of analysis, whether initial classes suggested are **very general,** or **very particular**.

In other words, specialization is a **top-down** activity that refines the abstract class into more concrete classes, and generalization is a bottom-up activity that abstracts certain principles from existing classes, in order to find more abstract classes.

We often organise information in the real world as generalisation/***specialization*** hierarchies. You can see an example of generalization/specialization in *Figure 6*.

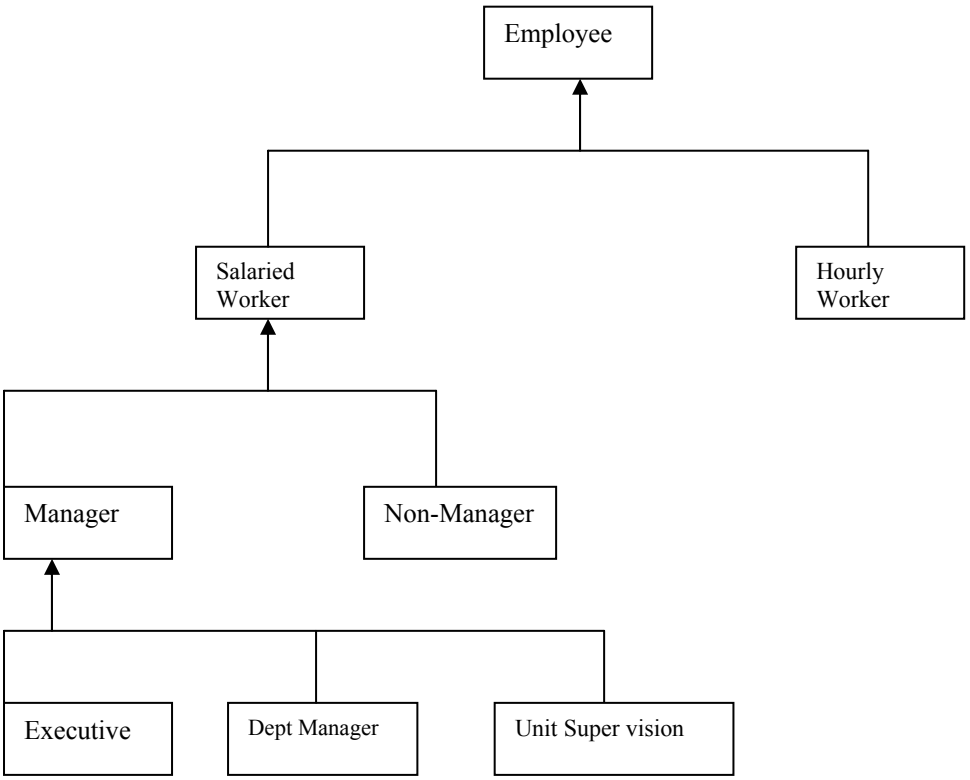


Figure 6: Generalization hierarchy of employee class

For instance, an employee may either be a salaried or an hourly worker. A salaried worker can be a manager, who in turn, can be an executive department manager, or a unit supervisor. The employee classification is **most general**, salaried worker is more **specific** and Unit supervisor is most specific. Of course, you might not model the world exactly like this for all organizations, but you get the idea. Unit Supervisor is a **subtype** of salaried worker, which is **a subtype of employee**. Employee is the highest-level super type and salaried worker is the super type of executive, department manager, and Unit supervisor. An object type could have several layers of subtypes and *subtypes of subtypes*. Generalization/specialization hierarchies help describe application systems and indicate where inheritance should be implemented in object-oriented programming language environments.

Any occurrence of a particular class is an occurrence of all ancestors of that class; so all features of a parent class automatically apply to subsclass occurrences. A child class cannot exclude, or suppress, an **attribute a parent**. Any operation on a parent must apply to all children. A child may modify an operation's implementation, but not is i**ts public** interface definition. A child class extends, parent features by adding new features. A child class may restrict the range of allowed values for inherited parent attributes.

In design and construction, operations on object data types can be over ridden, which **could substantially differ** from the original methods (rather than just refining original methods). Method overriding is performed to override **for extension, for restriction**, **for optimization,** or **for convenience**. Rumbaugh et al. proposes the following semantic rules for inheritance:

- All query operations (ones that read, but do not change, attribute values) are inherited by **all subclasses.**
- All update operations (ones that change attribute values) are inherited across **all extensions.**

- Update operations that change constrained attributes or associations are blocked across a restriction.
- Operations **may not** be overridden to make them **behave differently** in their externally visible manifestations from inherited operations. All methods that implement an operation must have the same protocol.
- Inherited operations can be refined by adding additional behavior.

Both generalization and specialization can lead to complex inheritance patterns, particularly via multiple inheritance. It is suggested that before making a final decision on generalisation/specialisation you should understand these rules very carefully and give the right choice for the following in respect of your system.

☞ Check Your Progress 2

1) Polymorphism can be described as

   (a) Hiding many different implementations behind one interface
   (b) Inheritance
   (c) Aggregation and association
   (d) Generalization

   ……

   ……..……..

   ……

   ……

2) What phrase best represents a generalization relationship?

   (a) is a part of
   (b) is a kind of
   (c) is a replica of
   (d) is composed of

   ……

   ……..……...…

   ……

   ……

3) All update operations in inheritance are updated

   (a) across all extensions
   (b) across only some of extensions
   (c) only first extension
   (d) None of the above.

   ……

   ……..……...…

   ……....

1.5 METADATA AND KEYS

Now, we will discuss the basics of Metadata and keys.

Let us first discuss metadata. You are already familiar with metadata concept in your database course. As you know, RDBMS uses metadata for storing information of database tables. Basically, metadata is such set of data which describes other data. For example, if you have to describe an object, you must have a description of the class from which it is instantiated. Here, the data used to describe class will be treated as metadata. You may observe that every real-world thing may have meta data, because every real world thing has a description for them. Let us take the example of institutes

and their directors. You can store that school A is having X as its direct, School B is having Y as its director, and so on. Now, you have concrete information to keep in metadata that is every institute is having a director.

KEY

Object instances may be identified by an attribute (or combination of attributes) called a key. A primary key is an attribute (or combination of attributes) that **uniquely identifies an object instance** and corresponds to the identifier of an actual object. For example, customer number would usually be used as the primary key for customer object instances. Two, or more, attributes in combination sometimes may be used to uniquely identify an object instance. For example, the combination of last name, first name and middle initial might be used to identify a customer or employee object instance. Here, you can say that sometimes more than one attribute gives a better chance to identify an object. For example, last name alone would not suffice because many people might have the same last name. First name would help but there is still a problem with uniqueness. All three parts of the name are better still, although a system generated customer or employee number is bests used as an identifier **if absolute uniqueness is desired**. Possible Primary Keys that are not actually selected and used as the primary keys are called **candidate keys**.

A **secondary key** is an attribute (or combination of attributes) that may **not uniquely identify an** object instance, but can describe a set of object instances that **share some common characteristic**. An attribute (customer type) might be used as a secondary key to group customers as internal to the business organisation (subsidiaries or divisions) or external to it. Many customers could be typed as internal or external at the same time, but the secondary key is useful to identify customers for pricing and customer service reasons.

1.6 INTEGRITY CONSTRAINTS

You have already studied integrity constraints in DBMS course. Here, we will review how these constraints are applied in the object oriented model.

Referential Integrity

Constraints on associations should be examined for referential integrity implications in the database models. Ask when referential integrity rules should be enforced. Immediately, or at a later time? When you are modeling object instances over time, you may need to introduce extra object classes to capture situations where attribute values can change over time. For instance, if you need to keep an audit trail of all changes to an order or invoice, you could add a date and time attribute to the order or invoice objects to allow for storage of a historical record of their instances. Each change to an instance would result in another instance of the object, stamped for data and time of instance creation.

Insert Rules

These rules determine the conditions under which a dependent class may be inserted and they deal with restrictions that the parent classes impose upon such insertions. The rules can be classified into six types.

Dependent : Permit insertion of child class instance only when the matching parent class instance already exists

Automatic : Always permit insertion of a child class instance. If the parent class instance does not exist, create one

Nullify : Always permit insertion of the child class instance.

Default : Always permit insertion of a child class in séance.

Customized : Allow child class instance insertion only if certain validity constraints are met.

No Effect : Always permit insertion of the child class instance. No matching parent class instances may or may not exist. No validity checking is performed.

Domain integrity

These integrity rules define constraints on valid values that attributes can assume. A domain is a set of valid values for a given attribute, a set of logical of conceptual values from which one or more attributes can draw their values. For example, India state codes might constitute the domain of attributes for employee state codes, customer state codes, and supplier state codes. Domain characteristics include such things as:

- Data type
- Data length
- Allowable value ranges
- Value uniqueness
- Whether a value can be null, or not.

Domain describes a valid set of values for an attribute, so that domain definitions can help you determine whether certain data manipulation operations make sense.

There are two ways to define domains.

One way to define domains and assign them to attribute is to define the domains first, and then to associate each attribute in your logical data model with a predefined domain. Another way is to assign domain characteristics to each attribute and then determine the domains by identifying certain similar groupings of domain characteristics. The first way seems better because it involves a thorough study of domain characteristics before assigning them to attributes. Domain definitions can be refined as you assign them to attributes. In practice, you may have to use the second method of defining domains due to characteristics of available repositories or CASE tools, which may not allow you to define a domain as a separate modeling construct.

Domain definitions are important because they:

- Verify that attribute values make business sense
- Determine whether two attribute occurrences of the same value really represent the same real-world value
- Determine whether various data manipulation operations make business sense.

A domain range characteristics for mortgage in a mortgage financing system could prevent a data entry clerk from entering an age of five years. Even though mortgage age and loan officer number can have the same data type, length and value, they definitely have different meanings and should not be related to each other in any data manipulation operations. The values 38 for age and 38 for officer number represent two **entirely unrelated values in the real world**, even though numerically they are exactly the same.

Table 1: Typical domain values

Data Characteristic	**Example**
Data type	character
	Integer
	Decimal
Data length	8 characters
	8 digits with 2 decimals
Allowable data values	x>=21
	0<x<100
Data value constraints	x in a set of allowable customer numbers
Uniqueness	x must be unique
Null values	x cannot be null
Default value	x can default the current date
	x can default to a dummy inventory tag number (for ordered items)

It makes little sense to match records based on values of age and loan officer number, even though it is possible. Matching customer in a customer class and customer

payment in a customer transaction class makes a great deal of sense. Typical domain characteristics that can be associated with a class attribute are shown in *Table1*.

Triggering Operation Integrity Rules

Triggering operation integrity rules govern **insert, delete, update** and **retrieval validity**. These rules involve the effects of operations on other classes, or on other attributes within class, and include domains, and insert/delete, and other attribute within a class, and include **domains** and **insert/delete** and other attributes business rules.

Triggering operation constraints involve:

- Attributes across multiple classes or instances
- Two or more attributes within a class
- One attribute or class and an external parameter.

Example triggering constraints include:

- An employee may only save up to three weeks time off
- A customer may not exceed a predetermined credit limit
- All customer invoices must include at least one line items
- Order dates must be current, or future dates.

Triggering operations have two components. These are:

- The event or condition that causes an operation to execute
- The action set in motion by the event or condition.

When you define triggering rules, you are concerned only with the logic of the operations, not execution efficiency, or the particular implementation of the rules. You will implement and tune the rule processing later when you translate the logical database model to a physical database implementation. Here, you should note that it is important to avoid defining processing solutions (like defining special attributes to serve as processing flags, such as a posted indicator for invoices) until all information requirements have been defined in the logical object model, and fully understood.

Triggering operations can be similar to referential integrity constraints, which focus on valid deletions of parent class instances and insertions of child class instances. Ask specific questions about each association, attribute, and class in order to elucidate necessary rules regarding data entry and processing constraints.

Triggering operation rules:

- Define rules for all attributes that are sources for other derived attributes
- Define rules for subtypes so that when a subtype instance is deleted, the matching super type is also deleted
- Define rules for item initiated integrity constraints.

1.7 AN OBJECT MODEL

In this last section of this unit let us examine a sales order system object model:

In this Sales Order System example, there are three methods of payment: **cash, credit Card or Check**. The attribute amount is common to all the three-payment methods, however, they have their own individual behaviors. *Figure 8* shows the object model, where the **directional association** in the diagram indicates the direction of navigation from one class to the other class.

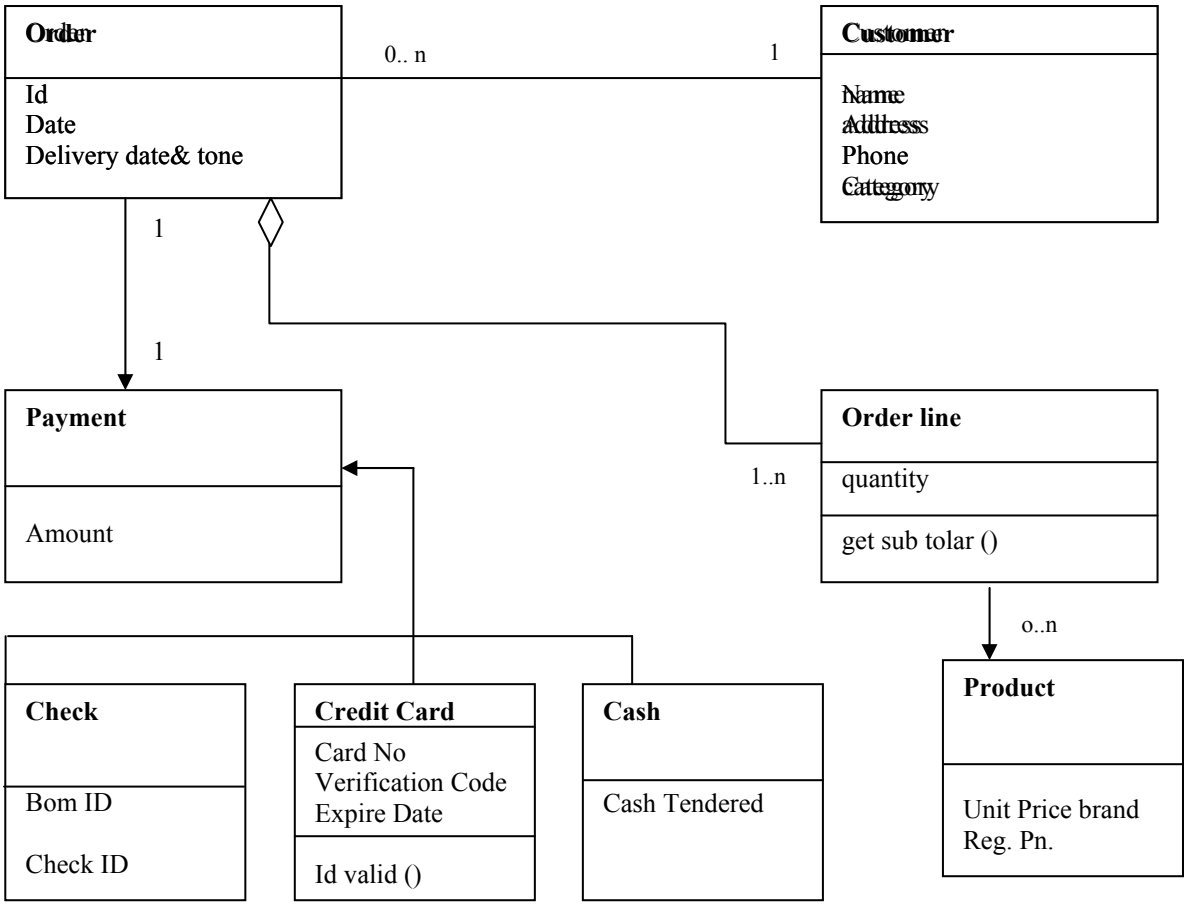


Figure 7: An object model for Sales Order System

☞ Check Your Progress 3

1) Explain in a few words whether the following UML class diagrams are correct or not

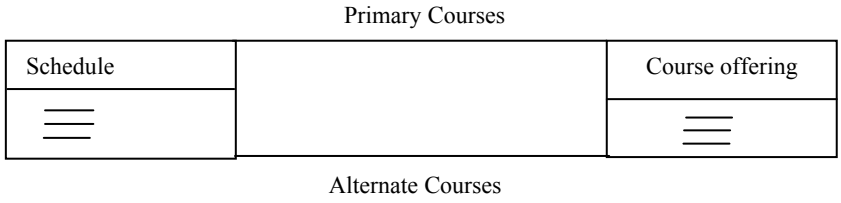


Figure 8: A Class Diagram

.....

.....

.....

2) Suppose that a computer is built out of one or more CPUs, Sound Card, and Video. Model the system with representative classes, and draw the class diagram

……

……

……

……

3) Suppose that Window WithScrollbar class is a subclass of Window and has a scrollbar. Draw the class diagram (relationship and multiplicity).

……

……

……

……

……

……

1.8 SUMMARY

This Unit over basic aspects of object modeling which includes discussion model based on objects (rather than functions) will be more stable over on a time hence the object oriented designs are more maintainable. Object do not exist in isolation from one another. In UML, there are different types of relationship (aggregation, composition, generalization). This Unit covers aggregation, and emphasizes that aggregation is the has-a or whole/part relationship. In this Unit we have also seen that generalization means extracting common properties from a collection of classes and placing them higher in the inheritance hierarchy, in a super class. We concluded the Unit with a discussion on integrity constraints, and by giving an object model for sales order system.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) C 2) D 3) C 4) D

Check Your Progress 2

1) A 2) 2 3) B

Check Your Progress 3

1) **Incorrect**: Classes are not allowed to have multiple association, unless defined by different roles

2)

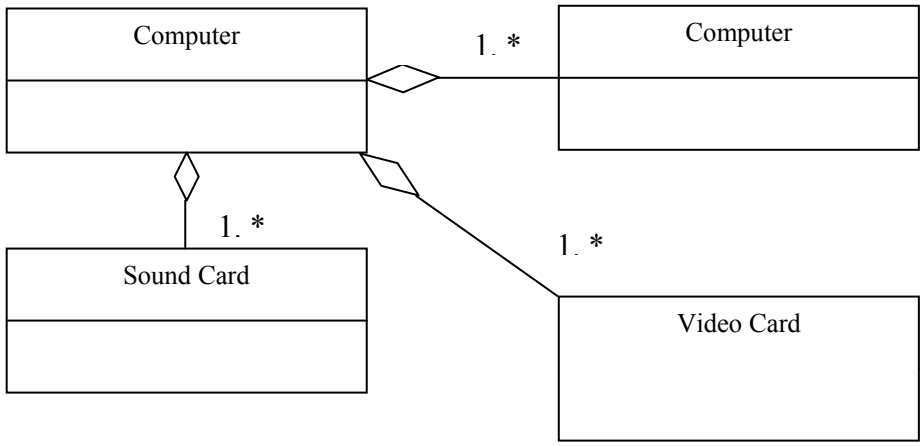


Figure 9: Class Diagram

3)

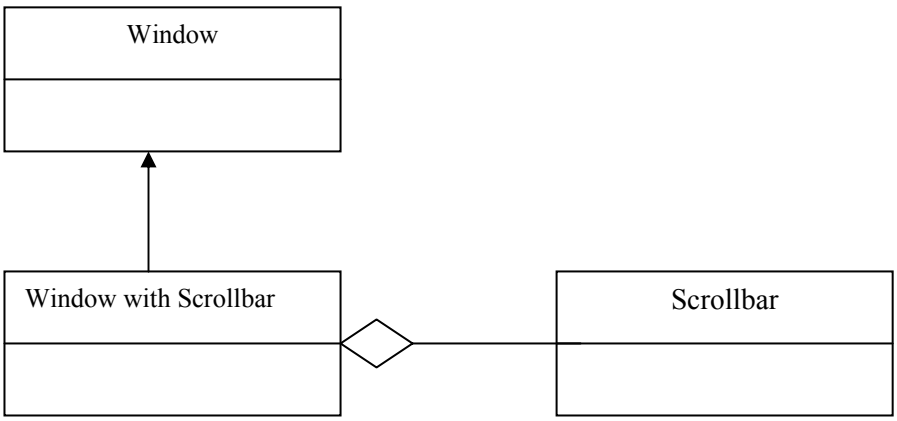


Figure 10: Class Diagram

UNIT 2 DYNAMIC MODELING

Structure **Page Nos.**

2.0	Introduction	20
2.1	Objectives	20
2.2	Events	20
2.3	State and State Diagram	21
2.4	Elements of a State Diagram	24
2.5	Advanced Concepts in Dynamic Modeling	25
2.6	Concurrency	26
2.7	A Dynamic Model	27
2.8	Summary	28
2.9	Solutions/Answers	29

2.0 INTRODUCTION

You must have observed that whenever something is done, some action is triggered. From daily life you can see that when you press a bell button, a ring tone is produced. That means some **event** has taken place. The dynamic model covers this aspect of the systems.

The dynamic model shows the **time-dependent** behavior of the system and the objects in it. Events can be defined as "something happened at a point of time". The dynamic model is important for interactive systems. The logical correctness of events depends on the sequences of interactions, or sequence of events.

You can understand a system by first looking at its static structure, the structure of its objects, and their relationships over time. Aspects of a system that are concerned with time and changes are the **dynamic models**. Control describes the sequences of operations that occur in response to external stimuli without consideration of what the **operations do**, **what they operate on**, or **how they are implemented**.

The major dynamic modeling concepts are **events**, which represent external stimuli, and **states**, which represent values of objects. The state diagram is a standard computer science concept (a graphical representation of finite state machines). Emphasis is on the use of events and states to specify control rather than as algebraic constructs. In this Unit, we will discuss the basic concepts of dynamic modeling which will cover events and states. We will also cover state diagram and concept of concurrency.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain events, and transition;
- design state diagrams;
- explain the elements of state diagram;
- use advanced concepts in dynamic modeling;
- explain, concurrency; and
- represent dynamic model of systems.

2.2 EVENTS

You can understand an event as **some action oriented result,** such as **mouse click.** Whenever you will click on a mouse, the appropriate action takes place.

You may observe that an event has no **specific time period** duration. You can click on a mouse and keep it pressed as long as you want. So, as far as events are concerned, nothing is instantaneous. An event is simply **an occurrence**. An event is actually a one way transmission of information from one object to another, but some times it may be that an event that occurs on a single object and changes the state of that object.

Two events can take place at the same time, one after the other, or both the events independently of each other and occurring simultaneously. For example, like two trains can depart at the same time for two different places, or two can depart from the same place, but one after the other. It means that the two events can be **independent as well as dependent on each other**.

Two events which are unrelated and occur at the same time are known as concurrent events. They have no effect on each other. You will not find any **particular order** between the two events, because they can occur in any order. In a distributed system, you will notice **concurrent events** and **activities**.

An **object** sending an **event to another object** may expect **a reply**, but the reply will be a **separate event** of the second object. So, you may see conversations between two objects as a combination of two or more events.

Event Classes: Every event is a **unique occurrence**; event class is a name to **indicate common structure** and **behavior**. Some events are simple signals, but most event classes have attributes indicating the information they convey. For example, events like train departs which has the attributes train number, class, city, etc. It is not necessary that all the attributes of objects contribute to attributes of events.

Here, you must note that the **time** at which the **event occurs** is an **implicit attribute** of all events.

Some events convey information in the form of **data** from one object to another. Sometime it may be that some classes of events only **signal** that something has occurred, while other classes of events **convey** data values. The data values conveyed by an event are **its attributes**; it implies that the value of data objects involved in events.

Event class name (attributes)

Sometimes event refers to event instance, or event class.

Mouse button clicked (left click, location)

Digit dialed (digit)

Phone receiver lifted.

Events include error conditions as well as normal occurrences.

Scenario and Event traces

A scenario is seen as a ***sequence of events that occurs during one particular execution of a system***. As far as the scope of a scenario is concerned, it varies. It may include **all events** in the system, or only some events from some selected objects involved in the event. Example of a scenario can be the historical records of executing a system, or a thought experiment of executing a proposed system.

Now, let us discuss states, and state diagram.

2.3 STATE AND STATE DIAGRAM

The state of an object is decided by the current values associated with the attributes of that object.

State

A state is **a condition** during the life of an **object,** or an interaction, during which, it satisfies some condition, performs some action, or waits for some event. An object remains in a state for a finite (non-instantaneous) time.

Actions are **atomic** and **non-interruptible**. A state may correspond to ongoing activity. Such activity is expressed as a **nested state machine**. Alternately, ongoing activity may be represented by a pair of actions, one that starts the activity on **entry to the state** and one that **terminates the activity on exit from the state**. So you can see that activities are the agents that are responsible for the change in state. Also, a state has its duration, and most of the time, a state is associated with some continuous activity**.**

You must see that a state should have **initial states** and **final states**. A transition to the enclosing state represents a transition to the initial state. A transition to a final state represents the completion of activity in the enclosing region. Completion of activity in all concurrent regions represents completion of activity by the enclosing state and triggers a "**completion of activity event**" on the enclosing state. Completion of the outermost state of an object corresponds to its **death**.

Notation

A state is shown as a **rectangle with rounded corners**. It may have one or more compartments. The compartments are **all optional**. They are as follows:

- Name compartment, holds the (optional) name of the state as a string. States without names are "**anonymous**" and are **all distinct**. It is undesirable to show the same named state twice in the same diagram.
- Initial state is shown by a solid circle.
- Final state is shown by a bull's eye.

Creating a State Diagram

Let us consider the scenario of travelling from station A to station B by the Bus Stand.

Following is the example of a **state diagram** of such scenario. It represents the normal flow. It does not show the **substates** for this scenario.

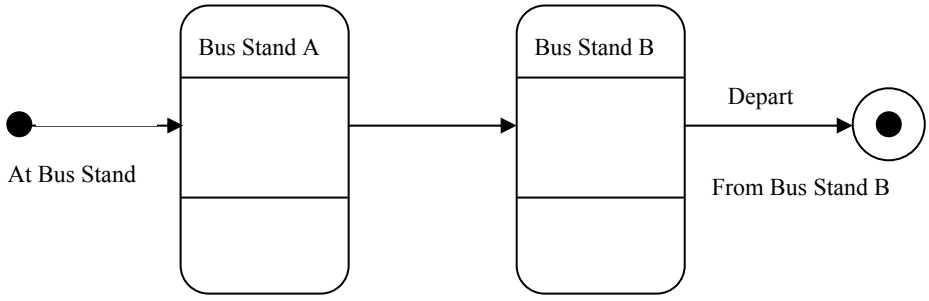


Figure 1: An example of flow in a state diagram

State chart diagrams

Objects have behaviors and state. The state of an object depends on its current activity, or condition. A **state chart diagram** shows the possible states of the object and the **transitions** that cause a change in state.

This diagram in *Figure 2* models the login part of an online banking system. Logging in consists of entering a valid social security number and personal id number, then submitting the information for validation.

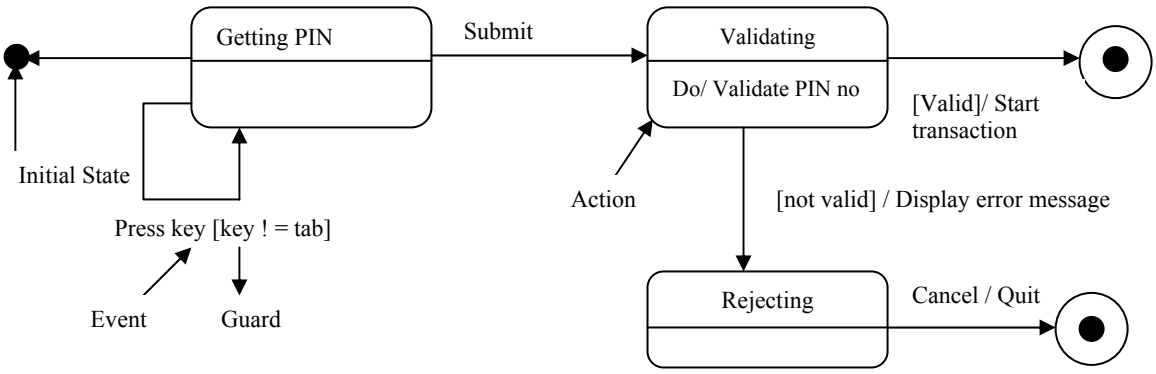


Figure 2: State chart diagram of login

Logging in can be factored into four non-overlapping states: **Getting PIN**, **Validating**, and **Rejecting**. From each state comes a complete set of **transitions** that determine the subsequent state.

States are rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows. Our diagram has self-transition, on **Getting PIN**.

The initial state (black circle) is a dummy to start the action. Final states are also dummy states that terminate the action.

The action that occurs as a result of an event or condition is expressed in the form action. While in its **Validating** state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

Now, you can see that a statechart diagram shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions. Or, in other words, you can say that:

The state machine is a graph of states and transitions that describes the response of an object of a given class to the receipt of outside stimuli. A state machine is attached to a class or a method.

A statechart diagram represents a state machine. The states are represented by state symbols, and the transitions are represented by arrows connecting the state symbols. States may also contain sub diagrams by physical containment and tiling.

☞ Check Your Progress 1

1) What is a state chart diagram?

……

……

2) What is a UML state diagram?

……

……

3) Draw a state diagram for a mobile phone.

……

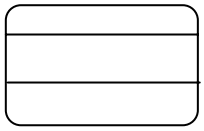
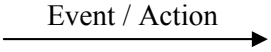
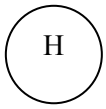
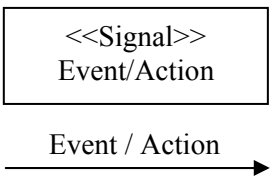
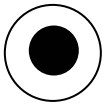
……

Now, let us discuss the basic components of a state diagram.

2.4 ELEMENTS OF A STATE DIAGRAM

We have seen different symbols used their meaning is a state diagram. Table1also explain about different State diagrams symbols.

Table 1: State Diagram Symbols

Elements and its Description	Symbol
Initial State: This shows the starting point or first activity of the flow. It is denoted by a solid circle. This is also called a **pseudo state**, where the state has no variables describing its further and no activities, to be done.	
State: Represents the state of an object at an instant of time. In a state diagram, there will be multiples of such symbols, one for each state of the object, denoted by a rectangle with rounded corners and compartments (such as a class with rounded corners to denote an object).	
Transition: An arrow indicating the object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow, separated by a slash. Transitions that occur because the state has completed an activity are called "triggerless" transitions.	
History States: A flow may require that the object go into a trance, or wait state, and on the occurrence of a certain event, go back to the state it was in when it went into a wait state — its last active state. This is shown in a State diagram with the help of a letter ***H enclosed within a circle***.	
Event and Action: A trigger that causes a transition to occur is called as an event or action. Every transition need not occur due to the occurrence of an event or action directly related to the state that transitioned from one state to another. As described above, an event/action is written above a transition that it causes.	
Signal: When an event causes a **message/trigger** to be sent to a state that causes the transition; then, that message sent by the event is called a **signal**.	
Final State: The end of the state diagram is shown by a **bull's eye** symbol, also called a final state. A final state is another example of a pseudo state because it does not have any variable or action described.	

Note: Changes in the system that occur, such as a background thread while the main process is running, are called "**substates**". Even though it affects the main state, a substate is not shown as a part of the main state. Hence, it is depicted as contained within the main state flow.

2.5 ADVANCED CONCEPTS IN DYNAMIC MODELING

Now let us look into advanced concepts in Dynamic Modeling. Entry and exit actions are part of every dynamic model. Let us see how they are performed.

Entry and Exit Actions

The following special actions have the same form, but represent reserved words that cannot be used for event names:

'Entry' '/' ***action-expression***: An atomic action performed on entry to the state.
'Exit' '/' ***action-expression:*** An atomic action performed on exit from the state.

Action expressions may use attributes and links of the owning object and parameters of incoming transitions (if they appear on all incoming transitions).

The following keyword represents the invocation of a nested state machine:
'**do**' '***/' machine-name (argument-list).***

The *machine-name* must be the name of a state machine that has an initial and final state. If the nested machine has parameters, then the argument list must match correctly. When this state is entered, after any entry action, then execution of the nested state machine begins with its initial state.

Example

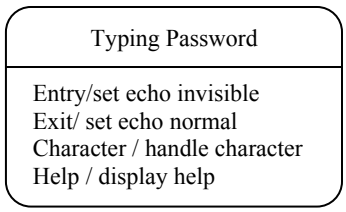


Figure 3: Entry-exit Action

The internal transition compartment holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing state.

The format to represent this is:

event-name argument-list '['guard-condition']' '/' action-expression

Each event name 'or **pseudo-event name'** may appear at most once in a single state.

You can see what happens when an event has to occur after the completion of some event or action, the event or action is called the **guard condition**. The transition takes place after the **guard condition occurs**. This guard condition/event/action is depicted by **square brackets** around the description of the event/action (in other words, in the form of a **Boolean expression**).

☞ Check Your Progress 2

1) What is a guard condition? Explain it with an example.

……

……

……

2) What are two special events?

……

……..

……

3) What is a self-transition?

……

……

……

……

……

Now, let us discuss the concept of concurrent object.

2.6 CONCURRENCY

You are already familiar with the term concurrent lines, which goes without affecting other operations. Similarly, when in a system **objects** can change state independently, they are termed **concurrent objects**.

In a dynamic model, some systems are described as a set of concurrent objects, each with its own state and state diagram.

An expansion of a state into concurrent substates is shown by **tiling the graphic region** of the state using dashed lines to divide it into **subregions**. Each subregion is a **concurrent substate**. Each subregion may have an optional name, and must contain a nested state diagram with disjoined states.

Composite States

Now you can say that **a state can be decomposed using *and*-relationships into concurrent substates or using or-relationships into mutually exclusive disjoint substates**. A given state may only be refined in one of these two ways. Its substates may be refined in the same way or the other way.

A newly-created object starts in its initial state. The event that creates the object may be used to trigger a transition from the initial state symbol. An object that transitions to its outermost final state ceases to exist.

An **expansion of a state** shows its **fine structure**. In addition to the (optional) name and internal transition compartments, the state may have an additional compartment that contains a region holding a nested diagram. For convenience and appearance, the text compartments may be shrunk horizontally within the graphic region.

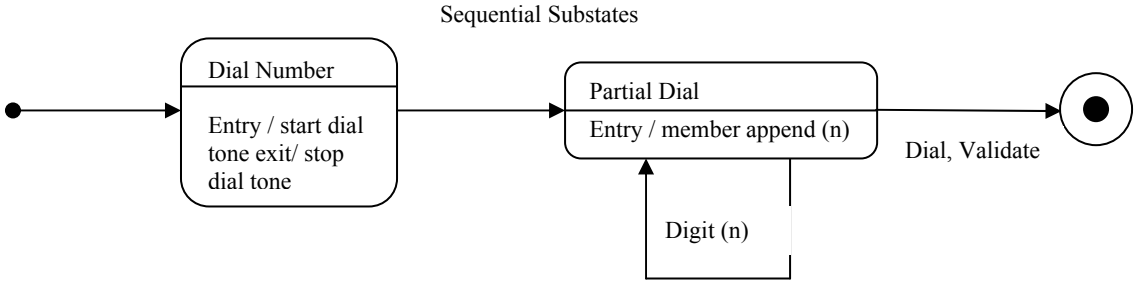


Figure 4: States Sequence

In *Figure 4*, you can see that **dial a number process** state is further divided into its sequential substrates such as, when it is **entering number state** then the state can be named as "**Partial Dial**" in which the user is still entering the number the action is "**append in digits**" then next state will **validate the number and so on**.

A state diagram for an assembly is a collection of **state diagrams**, one for each **component**. Aggregation means **concurrency**. Aggregation is the "**and-relationship**", you will see, it is the combined states of all component diagrams. For example, the state of a Car as an aggregation of component states: the Ignition, Transmission, Accelerator, and Brake. Each component state also has states. The state

of the car includes one substate from each component. Each component undergoes transitiitions in parallel with all others.

Semantics

An event is a noteworthy occurrence. For practical purposes in **state diagrams**, it is an **occurrence** that may **trigger a state transition**. Events may be of several kinds (not necessarily mutually exclusive): The event occurs whenever the value of the expression changes from false to true. **Note** that this is different from a guard condition: A guard condition is evaluated *once* whenever its **event fires**; if it is false then the transition does not occur and the event is lost. Guarded transitions for one object can depend on another object being in a given state.

2.7 A DYNAMIC MODEL

Now you are familiar with **events and their occurring time.** The *dynamic model* describes those aspects of the system concerned with the **sequencing of operations and time - events** that cause state changes, sequences of events, states that define the context for events and the organization of events and states. The dynamic model captures control information without regard **for what the operations act on** or how they are implemented.

The dynamic model is represented graphically by **state diagrams**. A state corresponds to the interval between two events received by an object, and describes the "value" of the object for that **time period**. **A state is an abstraction of an object's attribute values and links**, where sets of values are grouped together into a state according to properties that affect the general behavior of the object. Each state diagram shows the state and event sequences permitted in a system for one object class. State diagrams also refer to other models: actions correspond to functions in the functional model; events correspond to operations on objects in the object model.

The state diagram should adhere to OMT's notation and exploit the capabilities of OMT, such as **transition guards**, **actions** and **activities**, **nesting** (state and event generalization), and **concurrency**.

Here is the transition diagram for a digital watch.

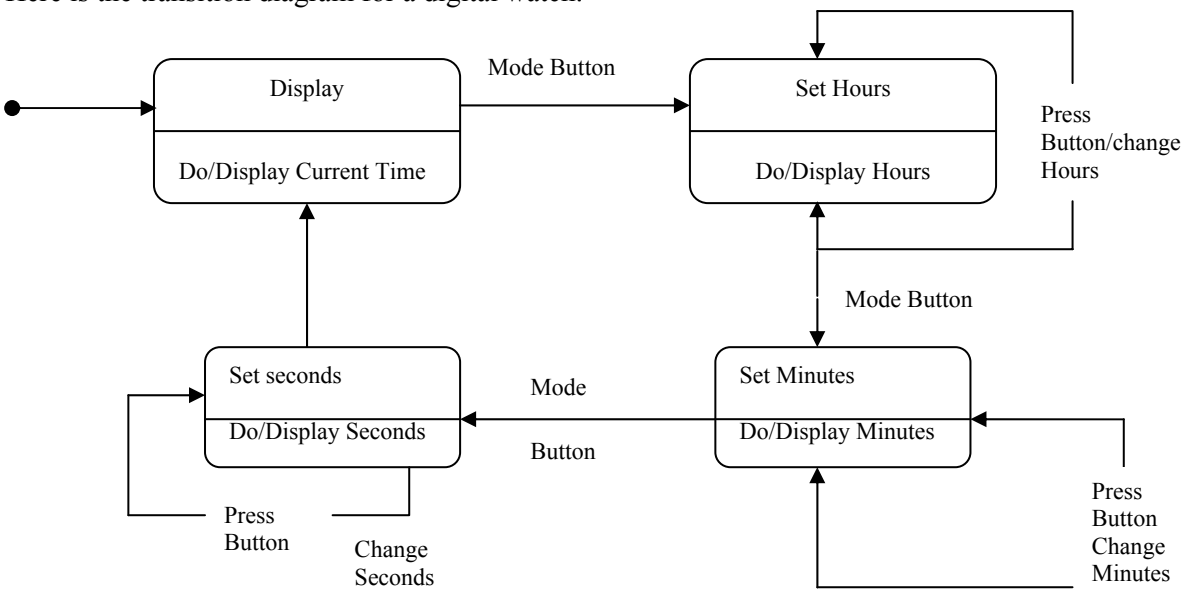


Figure 5: State Transition diagram for digital watch

In *Figure 5***,** you can see that the state diagram of a digital watch is given. Where user wants to set Hours set Minutes and followed by setting seconds.

☞ Check Your Progress 3

1) Give a Concurrent substates diagram for classroom and exam held.

……

……

……

2) Describe Dynamic Model.

……

……

3) Give a sample of a Dynamic Model.

……

……

……

……

4) Show, with an example, that relatively high-level transitions result when the system or program is stimulated by outside events.

……

……

……

……

2.8 SUMMARY

The dynamic model is the model which represents the control information: the sequence of events, states and operations that occur within a system of objects. It has scenarios to occur with meaningful constraints.

An event is triggered by an instantaneous action. One kind of action is sending an event to another object. **External event** also known as a system event is caused by something outside our system boundary. **Internal event** is caused by something inside our system boundary.

States may be "nested." A nested state usually indicates a functional decomposition within the "super" state. The term *macro-state* is often used for the super state. The macro-state may be composed of multiple *micro-states*.

The basic notion is that the system is always in **one state**, and never in more than **one state** (at any given level). The system remains in that state until a transition is triggered by an **external event**. Transitions take no time to occur. There is no time in which the system is not in one of the defined states.

State diagrams must be created for **state-dependent objects** with complex behavior like **Use Cases, Stateful session, Systems, Windows, Controllers, Transactions**, devices, and role mutators.

Actions are associated with transitions and are considered to be processes that occur quickly, and are not interrupted. The syntax for a transition label has three parts, all of which are optional: **Event [Guard] / Action**.

2.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) State diagrams (State Chart Diagram) describe all the possible states that a particular object can get into, and how the object's state changes as result of events that reach the object. It states all possible states and transitions.

2) The UML state diagram illustrates the events and states of an object and the behavior of an object in reaction to an event.

3)

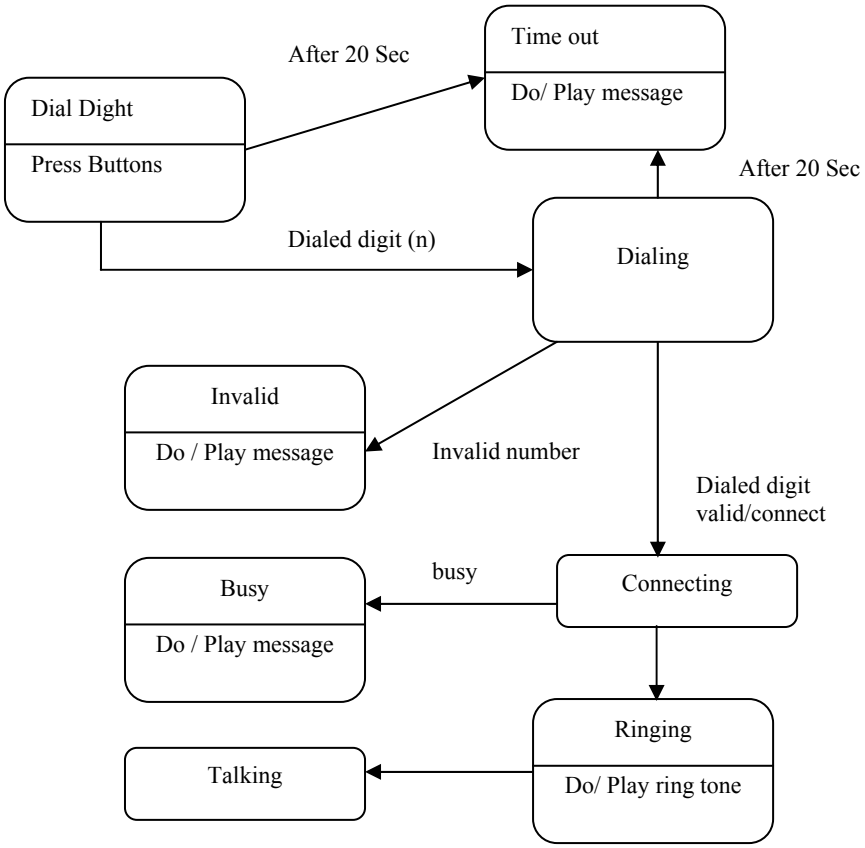


Figure 6: State Diagram for a Mobile

Check Your Progress 2

1) The *guard-condition* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the object that owns the state machine. The guard condition may also involve tests of concurrent states of the current machine (or explicitly designated states of some reachable object); for example, "**in** State1" or "**not in** State2". State names may be fully qualified by the nested states that contain them, yielding path names of the form "State1: State2::State3"; this may be used in case the same state name occurs in different composite state regions of the overall machine.

2) There are two special events "**entry"** and "**exit**". Any action that is marked as linked to the entry event is executed whenever the given state is entered via transition. The action associated with the **exit event** is executed whenever the state is left via transition.

3) If there is a transition that goes back to the same state, it is called "**self-transition**." With a trigger action the **exit action would be executed first**, then the transition's action and finally **the entry action**. If the state has an associated activity as well, that activity is executed after the **entry action**.

Check Your Progress 3

1)

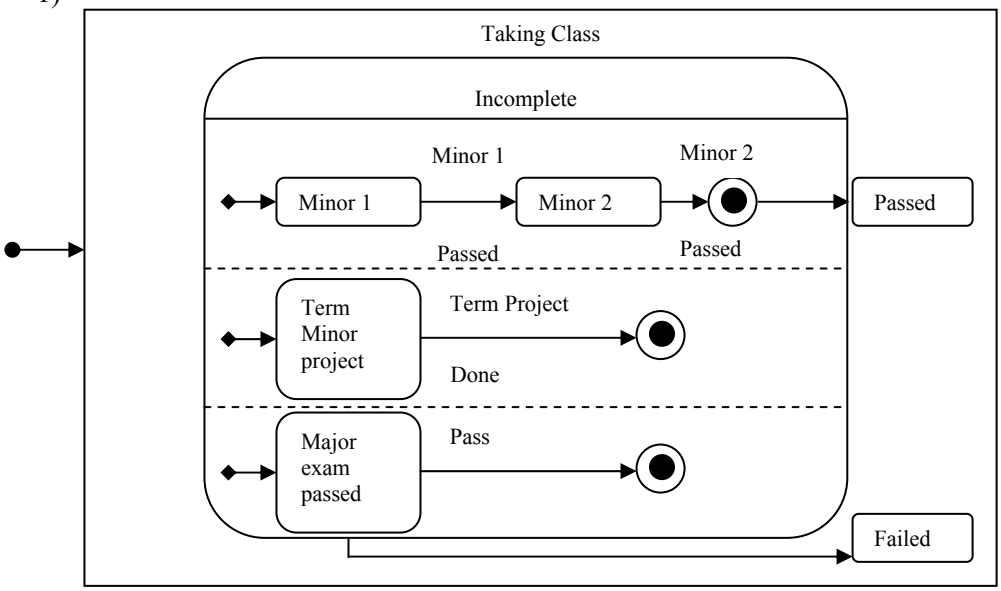


Figure 7: Concurrent Substate diagram

In *Figure 7* of concurrent substrates have been taken. After passing Minor 1 test you can give Minor 2 test. Term minor project of that semester minor should be done before Major exam of that semester.

2) The dynamic model specifies allowable sequences of changes to objects from the object model. It includes event trace diagrams describing scenarios. An event is an external stimulus from one object to another, which occurs at a particular point in time. An event is a one-way transmission of information from one object to another. A scenario is a sequence of events that occurs during one particular execution of a system. Each basic execution of the system should be represented as a scenario.

3) **Dynamic model for car:**

Accelerator and Brake	Applies Accelerator or Brake
Accelerator	Applies Accelerator
Brake	Applies Brake
off	Put off the car
on	Put on the car
off	
on	
press acc	
rel acc	
press brake	
rel brake	

4) In this diagram, you can observe if that initial state is state X if event Z occurs, then state X is terminated, and state Y is entered.

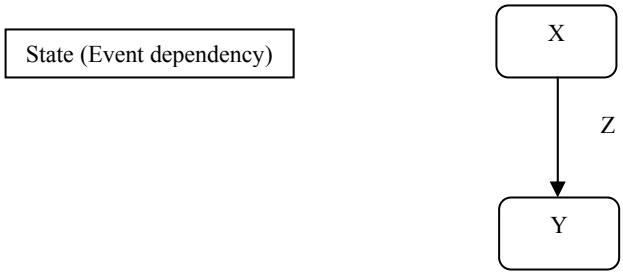


Figure 8: State event dependency

UNIT 3 FUNCTIONAL MODELING

Structure **Page Nos.**

3.0 Introduction 31
3.1 Objectives 31
3.2 Functional Models 32
3.3 Data Flow Diagrams 33
3.4 Features of a DFD 33
- 3.4.1 Processes
- 3.4.2 Data Flows
- 3.4.3 Actors
- 3.4.4 Data Stores
- 3.4.5 Constraints
- 3.4.6 Control Flows

3.5 Design Flaws in DFD 37
3.6 A Sample Functional Model 38
3.7 Relation of Functional to Object and Dynamic Model 42
3.8 Summary 44
3.9 Solutions / Answers 45

3.0 INTRODUCTION

As discussed in the previous Unit of this Course *"the dynamic model represents control information**: the sequences of events**, **states, and operations** that occur within a system of objects"*. The dynamic model is a pattern that specifies possible scenarios that may occur. An event is a signal that something has happened. A state represents the interval between events, and specifies the context in which events are interpreted. An action is an instantaneous operation in response to an event.

The functional modeling is the third, and final phase of the OMT model. The functional modeling is a complex modeling. It describes how the output values in a computation are derived from input values. *The **functional model** specifies **what happens**, the dynamic model specifies when it happens, and the object model describes what it happens, and the object model describes what happens to an object*. The functional model is **consist of multiple data flow diagrams**, which show the flow of values from input to output through operations and internal data stores. It also includes constraints among values within an object model.

In this Unit you will learn functional modeling concepts and data flow diagrams. Data flow diagrams do not show control or object structure information; these belong to the dynamic and object models. In this Unit in our discussion will follow the traditional form of the data flow diagram, with which you are familiar.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the Function Model;
- explain the concept of DFD;
- implement dataflow in the functional model;
- describe features of the data flow diagram;
- explain limitations in the design of the Data Flow Diagram, and
- relate the Object Model, Dynamic Model, and Functional Model.

3.2 FUNCTIONAL MODELS

Let us start our discussion by answering the question "what is a functional model?"

The functional model is the third leg of the OMT methodology in addition to the **Object Model** and **Dynamic Model**. *"**The functional model specifies the results of a computation specifying how or when they are computed**"*. The functional model specifies the meaning of the **operations in the object model** and the **actions in the dynamic model**, as well as any constraints in the object model. Non-interactive programs, such as compilers, have a trivial dynamic model; the purpose of a compiler is to compute a function. The functional model is the main model for such programs, although the object model is important for any problem with nontrivial data structures. Many interactive programs also have a significant functional model. By contrast, databases often have a trivial functional model, since their purpose is to store and **organize data**, not **to transform it**.

For example, **spreadsheet is** a **kind of functional model**. In many cases, the values in the spreadsheet are trivial and cannot be structured further. The only interesting object structure in the spreadsheet is the cell. The aim of the spreadsheet is to specify values in terms of other values.

Let us take the case of a compiler. A compiler is almost a **pure computation**. The input for a compiler is the text of a program in a particular language; the output is an object file that implements the program in another language often the machine language of a particular computer. Here, the **mechanics of compilation are** not **concerned** with the **functional model**. Now, we will discuss the data flow diagram. It is very helpful in visualizing the flow of data in the system, and to show the involvement of different processes at different levels.

3.3 DATA FLOW DIAGRAMS

Here, we will discuss data flow diagram and their uses in a Functional Model. As you know, the functional model consists of **multiple data flow diagrams** which specify the meanings, of operations and constraints. A data flow diagram (DFD) shows the functional relationships of the values computed by a system, including input values, output values, and internal data stores. "***A data flow diagram is a graph which shows the flow of data values from their sources in objects through processes that transform them to their destinations in other objects***". DFDs do not show control information, such as the time at which processes are executed, or decisions among alternate data paths. This type of information belongs to the dynamic model. Also, the arrangement values into object are shown by the object model, but not by the data flow diagram.

A data flow diagram contains **processes** which **transform data, data flows** which move data, **actor objects** which produce and consume data, and **data store** objects that store data passively. *Figure 1* shows a data flow diagram for the display of an icon on a windowing system. Here in this figure, the icon name and location are inputs to the diagram from an unspecified source. The icon is expanded to vectors in the application coordinate system using existing icon definitions. The vectors are clipped to the size of the window, then offset by the location of the window on the screen, to obtain vectors in the screen coordinate system. Finally, the vectors are converted to pixel operations that are sent to the screen buffer for display. The data flow diagram represents the sequence of transformations performed, as well as the **external values** and objects that affect the computation process.

Now, let us turn to the basic feature of DFD.

3.4 FEATURES OF A DATA FLOW DIAGRAM

The DFD has many features. It shows the computation of values in different states. The building blocks and features of DFD are:

3.4.1 Processes

"*The term **process** means all the computation activities that are involved from the input phase to the output phase"*. Each process contains a fixed number of input and output data arrows. These arrows carry a value of a given type. A process transforms data values. The lowest-level processes are pure functions without side effects.

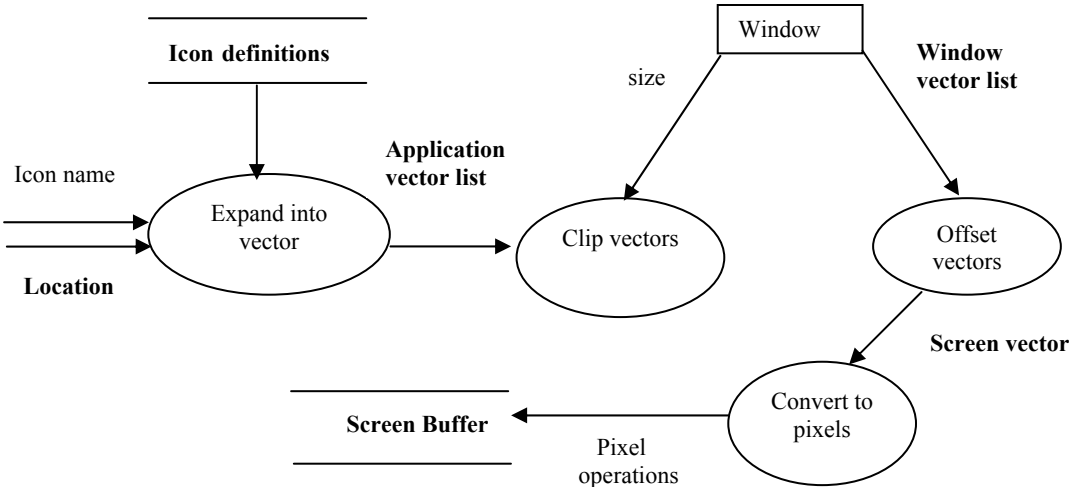


Figure 1: Data flow diagram for windowed graphics display

An entire data flow graph is a **highlevel process**. A process may have side effects if it contains non-functional components, such as **data stores** or **external objects**. The functional model does not uniquely specify the results of a process with side effects. The functional model only indicates **the possible functional paths**; it does not show which path will actually occur. The results of such a process depend on the **behavior of the system**, as specified by the dynamic model. Some of the examples of non-functional processes are **reading and writing files**, a **voice recognition algorithm** that **learns from experience**, and the display of images within a workstation windowing system.

A process is represented with the ellipse symbol and the name of process is written in it. Each process has a fixed number of input and output data arrows, each of which carries a value of a given type. The inputs and outputs can be labeled to show their role in the computation. In *Figure 2,* two processes are shown. Here, you should note that a process can have more than one output. The display icon process represents the entire data flow diagram of *Figure1* at a higher level of abstraction.

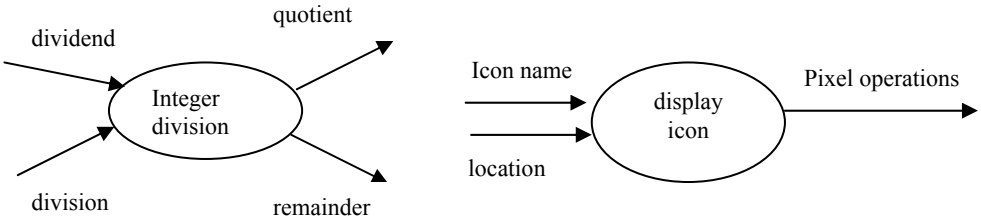


Figure 2: Processes

The diagram of processes shows only the pattern of inputs and outputs. The computation of output values from input values must also be specified. A high-level process can be expanded into an entire data flow diagram such as a subroutine which can be expanded into lower-level subroutine. Recursion processes **must be stopped** in a data flow diagram. The atomic processes must be described directly, in natural

language, mathematical equations, or by some other means. For example, *integer division* could be defined mathematically and "display icon" would be defined in terms of *Figure 1*. The atomic processes are trivial and simply access a value from an object.

3.4.2 Data Flows

The term data flow literally means flow of data. A data flow connects the output of an object or process to the input of another object or process. It represents an intermediate data value within a computation. Here, you should note that the value is not changed by the data flow.

A data flow is represented with the symbol arrow, and is used to connect the producer and the consumer of the data value. The arrow is labeled with a description of the data, usually, its name or type. The same value can be sent to several places and this is indicated by a fork with several arrows emerging from it. The output arrows are unlabeled because they represent the same value as the input. Some data flows are shown in *Figure 3* below.

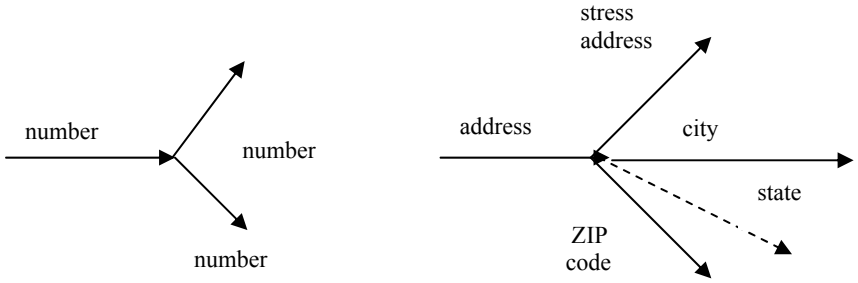


Figure 3: Data flows to copy a value and split an aggregate value

Sometimes, an aggregate data value is split into its components, each of which goes to a different process. This is shown by **a fork** in the path in which each outgoing arrow is labeled with the name of its component. The combination of several components into an aggregate value is just the opposite of it.

Each data flow represents a value at some point in the computation. The data flows internal to the diagram represent intermediate values within a computation and do not necessarily have any significance in the real world.

Flows on the boundary of a data flow diagram are its inputs and outputs. These flows may be unconnected, or they may be connected to objects. The inputs in *Figure 3* are the number and address of the location; their sources must be specified in the larger context in which the diagram is used.

3.4.3 Actors

The term actor means an object which can perform actions. It is drawn as a rectangle to show that it is an object. An actor is an active object that drives the data flow graph by producing or consuming values. Actors are attached to the inputs and outputs of a data flow graph. In a sense, the actors lie on the boundary of the data flow graph, but terminate the flow of data as sources and sinks of data, and so are sometimes called **terminators**. Examples of actors are the user of a program, a thermostat, and a motor under computer control. The actions of the actors are **outside the scope** of the data flow diagram, but should be part of the dynamic model.

3.4.4 Data Stores

The term data store literally means the place where data is stored. It is a passive object within a data flow diagram that stores data for later access. Unlike an actor, a data store does not generate any operations on its own, but merely responds to requests to store and to access data. A data store allows values to be accessed in a different order than they are generated. Aggregate data stores, such as lists and tables,

provide accesses to data by insertion order or by index keys. Some of the examples of data stores are the database of airline seat reservations, a bank account, and a list of temperature readings over the past day.

A data store is represented by a pair of parallel lines containing the name of the store. Input arrows indicate information or operations that modify the stored data. Some of the operations which we can perform are adding elements, modifying values, or deleting elements. Output arrows indicate information retrieved from the store. This includes retrieving at the values, or some part of it.

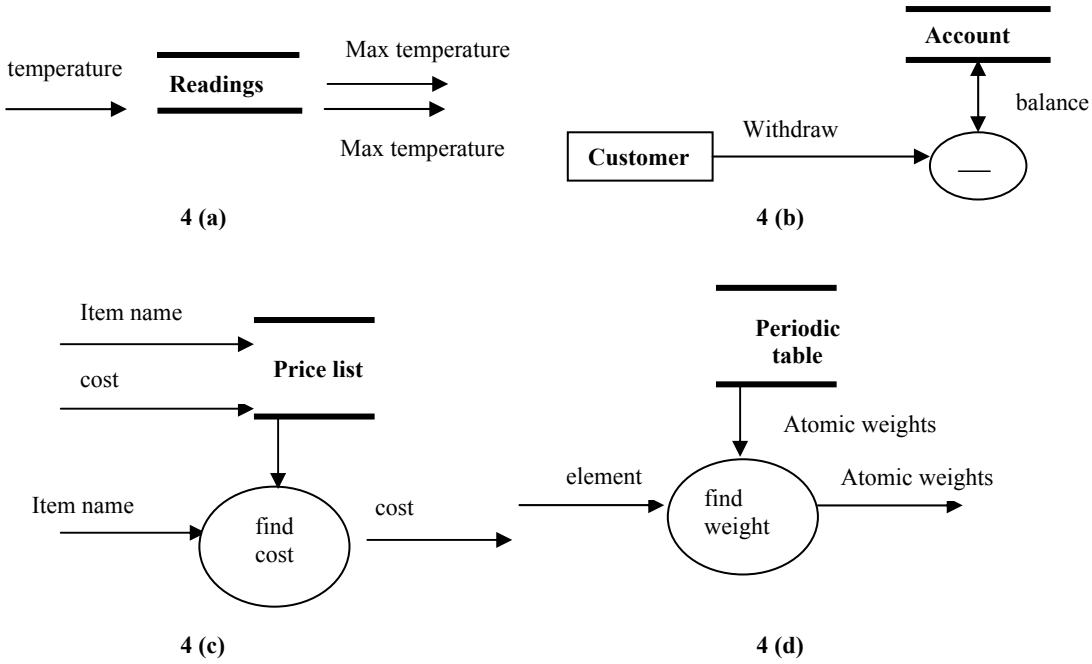


Figure 4: Data stores

Figure 4 shows a data stores for temperature readings. Every hour a new temperature reading enters the store. At the end of the day, the maximum and minimum readings are retrieved from the store. The data store permits many pieces of data to be accumulated so that the data can be used later on.

In *Figure 4b* data store for a bank account is given. The double-headed arrow indicates that a balance is both an input and an output of the subtraction operation. This can be represented with two separate arrows. Accessing and updating of the value in a data store is a common operation.

In *Figure 4c,* a price list for items is shown. Input to the store consists of pairs of item name and cost values. Later, an item is given, and the corresponding cost is found. The unlabeled arrow from the data store to the process indicates that the entire price list is an input to the selection operation.

To find the atomic weight of an element from a periodic table we can use a data flow diagram. This data flow diagram is represented in *Figure 4d*. Obviously, the properties of chemical elements are constant and not a variable of the program. It is convenient to represent the operation as a simple access of a constant data store object. Such a data store has no inputs.

Take the case that both actors and data stores can be represented as objects. We distinguish them because their behavior and usage is generally different, although in an object-oriented language they might both be implemented as objects. On the other hand, a data store might be implemented **as a file** and an actor as an **external device**. Some data flows are also objects, although in many cases they are pure values, such as integers, which lack individual identity.

An object as **a single value** and **as a data store containing many values** have different views. In *Figure 5*, the customer name selects an account from the bank. The result of this operation is the account object itself, which is then used as a data store in the update operation. A data flow that generates an object used as the target of another operation is represented by a hollow triangle at the end of the data flow. In contrast, the update operation modifies the balance in the account object, as shown by the small arrowhead. The hollow triangle represents a data flow value that subsequently is treated as an object.

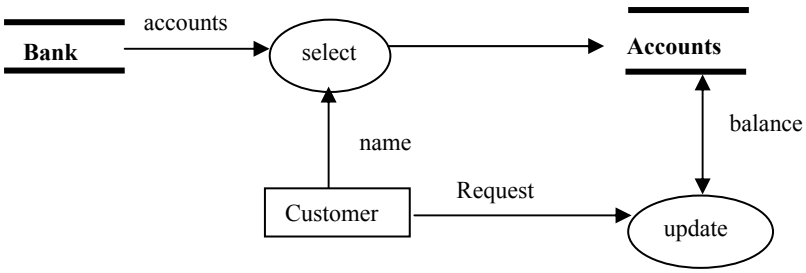


Figure 5: Selection with an object as result

The data flow diagram given in *Figure 6* represents the creation of a new account in a bank. The result of the create account process is a new account, which is stored in the bank. The customer's name and deposit are stored in the account. The account number from the new account is given to the customer. In this example, the account object is viewed both as a **data value** and as a **data store**.

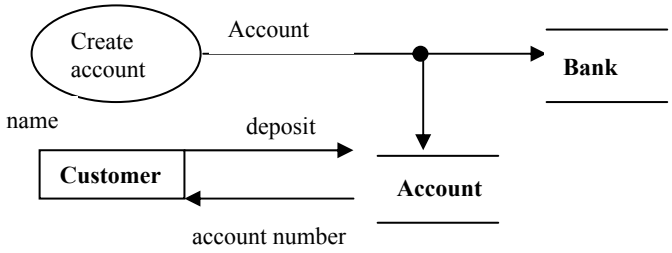


Figure 6: Creation of a new object

3.4.5 Constraints

A constraint shows the relationship between two objects at the same time, or between different values of the some object at different times. For example, in coordinate geometry, the location of a point can be obtained by finding the x and y coordinates, and the scale of these two abscissas can be the same or different.

Constraints can appear in each kind of model. **Object constraints** describe where some objects depend entirely or partially on other objects. **Dynamic constraints** represent relationships among the states or events of different objects. Similarly, the **functional constraints** shows the restrictions on operations, such as the scaling transformation.

A constraint between values of an object over time is often called an invariant. For example, conservation laws in physics are invariants: the total energy, or charge, or angular momentum of a system remains constant. Invariants are useful in **specifying the behavior of operations**.

3.4.6 Control Flows

A control flow is a Boolean value that affects whether a process is evaluated. The control flow is not an input value to the process itself. It is shown by a dotted line from a process producing a Boolean value to the process being controlled. The use of control flow is shown in the *Figure 7*. This figure shows the withdrawal of money from a bank account. The customer enters a password and an account. The withdrawal

was made after successfully verifying the password. The update process also can be explained by using a similar control flow to guard against overdrafts.

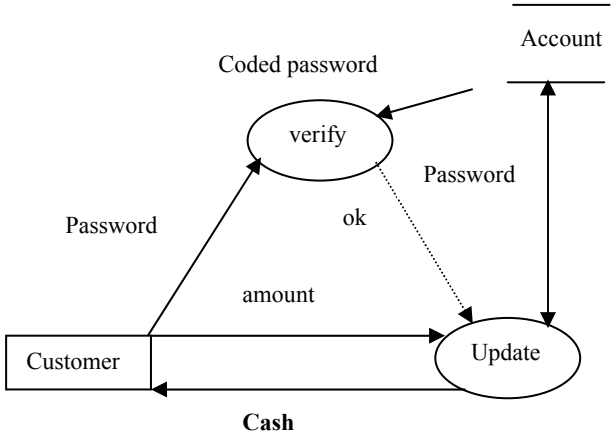


Figure 7: Control flow

☞ Check Your Progress 1

1) Explain Functional Mode with the help of an example.

……..

……..

……..

2) i) What is a Data Flow Diagram?
ii) What is the State Diagram?

……

……

……

……

3) Explain the following.

i) Process
ii) Data Flows
iii) Actor
iv) Data Stores

……..

……

……

……..

Now we will discuss some limit actions of DFD.

Although the data flow diagram is very helpful in describing the functional model of an object. But similar to other models such as **object model** and **dynamic model** it also have some limitations.

3.5 DESIGN FLAWS IN DFD

Some of the common limitations of the data flow diagram are as follow:

- The Functional Model does not specify when values are computed.

- The Functional Model does not specify how often values of an object are computed.
- The Functional Model does not specify why the values of an object are changes.
- A Data Flow Diagram contain different symbols to represent different objects and actions, and is very difficult to prepare.
- The Data Flow Diagram for some lengthy problems becomes very complex to prepare as well as to remember.

After a detailed discussion of the various components of functions model, now, you are able to understand the complete sample functional model. In the next section, we describe the functional model for a flight simulator.

3.6 A SAMPLE FUNCTIONAL MODEL

The flight simulator is responsible for handling the pilot input controls, computing the motion of the airplane, computing and displaying the outside view from the cockpit window, and displaying the cockpit gauges. The simulator is intended to be an accurate, but simplified model of flying an airplane, ignoring some of the smaller effects and making some simplifying assumptions. For example, we ignore the rudder, under the assumption that it is held so as to keep the plane pointing in the direction of motion. In *Figure 6,* the **top-level data flow diagram** for the flight simulator is shown. There are two input actors; the **Pilot**, who operates the airplane controls, and the **weather**, which varies according to some specified pattern. There is one output actor the **Screen**, which displays the pilot's view.

There are two read-only data stores: The **Terrain database**, which specifies the geometry of the surrounding terrain as a set of colored polygonal surfaces, and the **Cockpit database**, which specifies the shape and location of the cockpit view port and the locations of the various gauges. There are three internal data stores which are Spatial parameters, which holds the 3-D position, velocity, orientation, and rotation of the plane; **Fuel**, which holds the amount of fuel remaining; and **Weight**, which holds the total weight of the plane. The initialization of the internal data stores is necessary but is not shown on the data flow diagram.

We can divide the processes given in DFD into three kinds, which are **handling controls, motion computation**, and **display generation**. The control handling processes are adjust controls, which transforms the position of the pilot's controls (such as joysticks) into positions of the airplane control surfaces and engine speed; consume fuel, which computes fuel consumption as a function of engine speed; and compute weight, which computes the weight of the airplane as the sum of the base weight and the weight of the remaining fuel. Process adjust controls is expanded in *Figure 7* where it can be seen as comprising three distinct controls; the elevator, the ailerons, and the throttle. There is no need to expand these processes further, as they can be described by input-output functions easily.

The motion computation processes are compute forces, which computes the various forces and torques on the plane and sums them to determine the net acceleration and the rotational torques, and integrate motion, which integrates the differential equations of motion. Process compute forces incorporates both geometrical and aeronautical computations. It is expanded in *Figure 8*. Net force is computed as the vector sum of drag, lift, thrust, and weight.

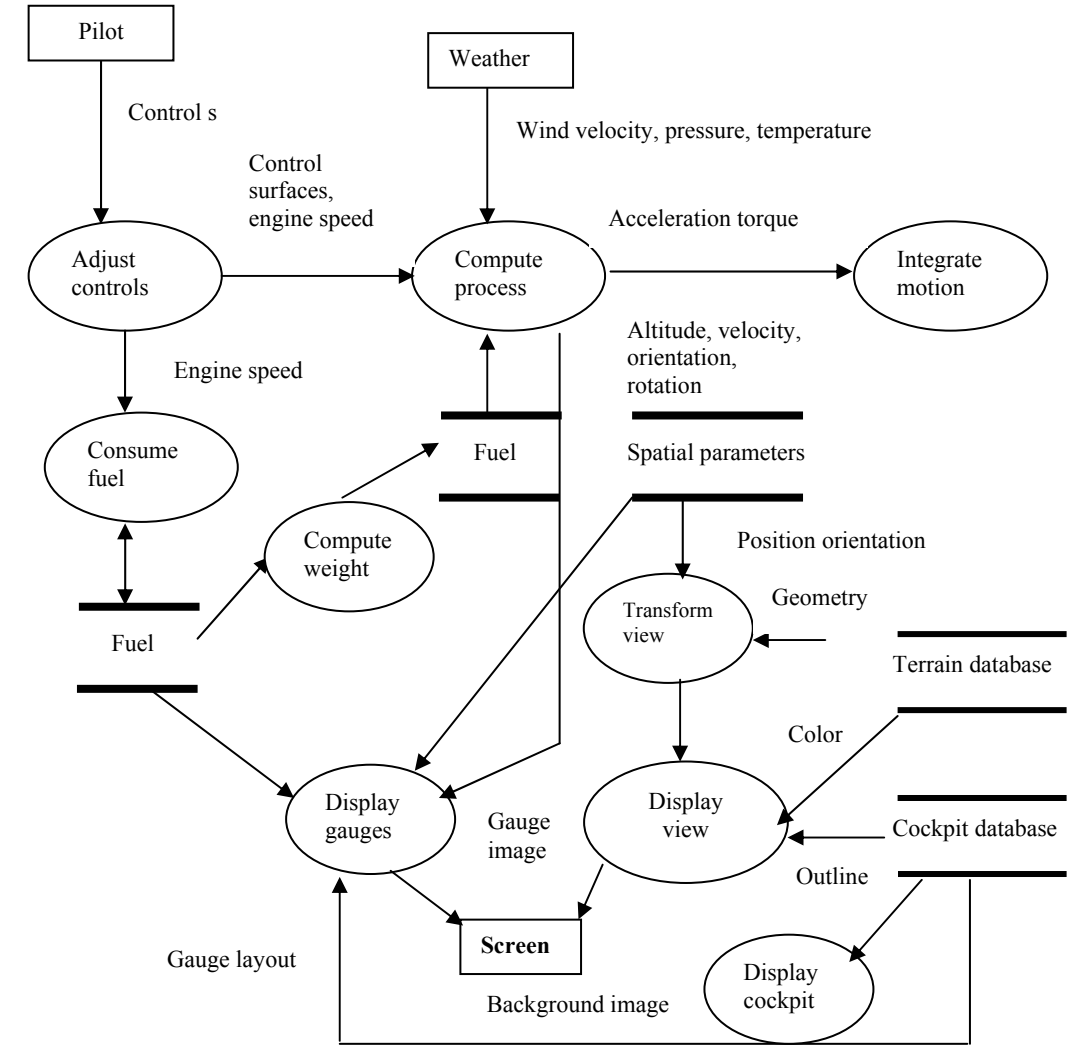


Figure 8: Functional model of flight simulator

These forces in turn depend on intermediate parameters, such as airspeed, angle of attack, and air density. The aerodynamic calculations must be made relative to the air mass, so the wind velocity is subtracted from the plane's velocity to give the airspeed relative to the air mass.

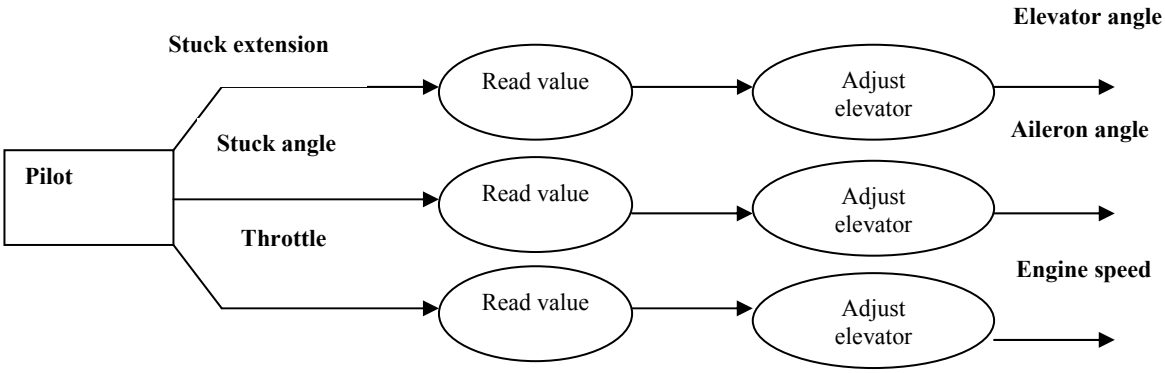


Figure 9: Expansion of adjust controls process

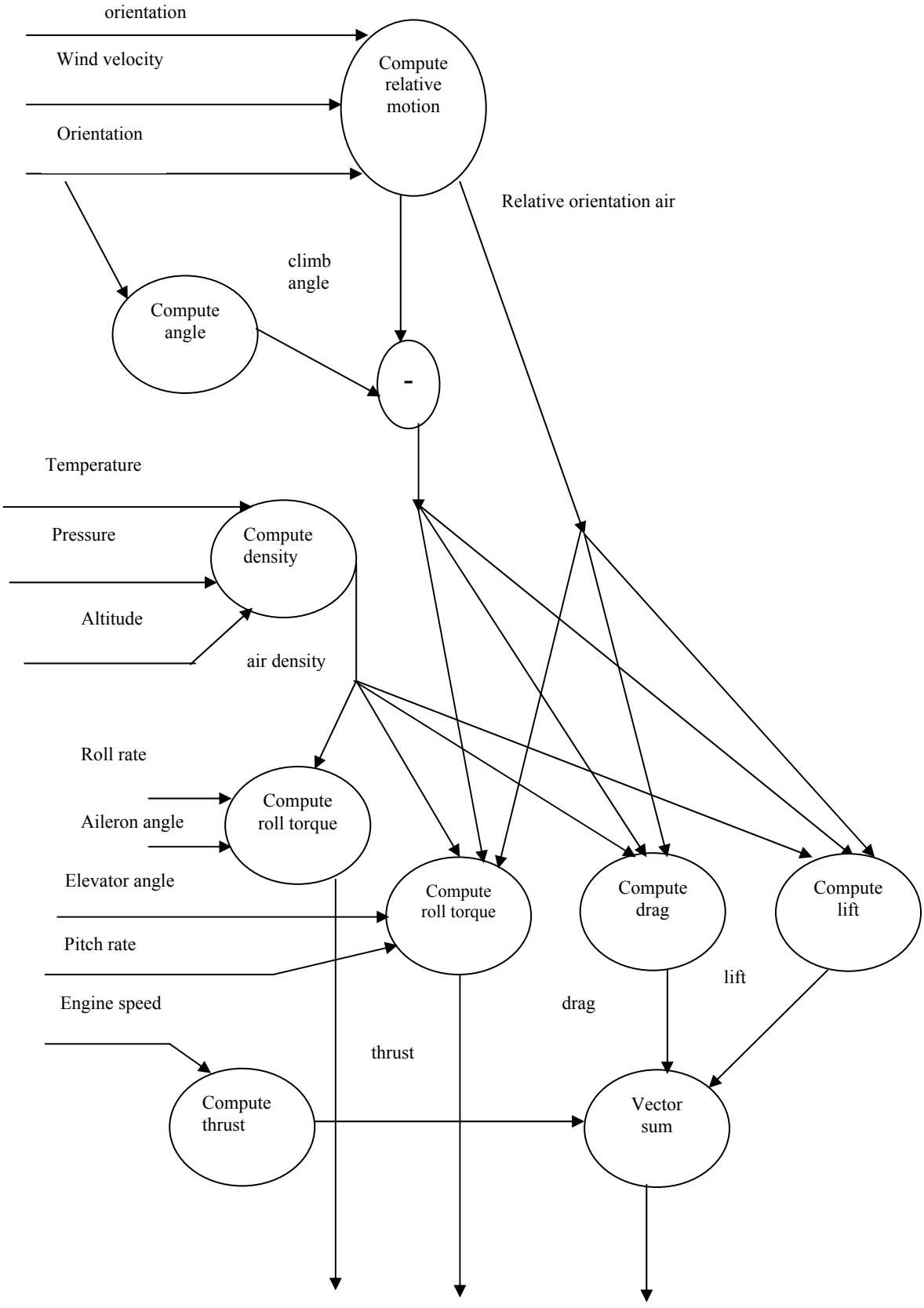
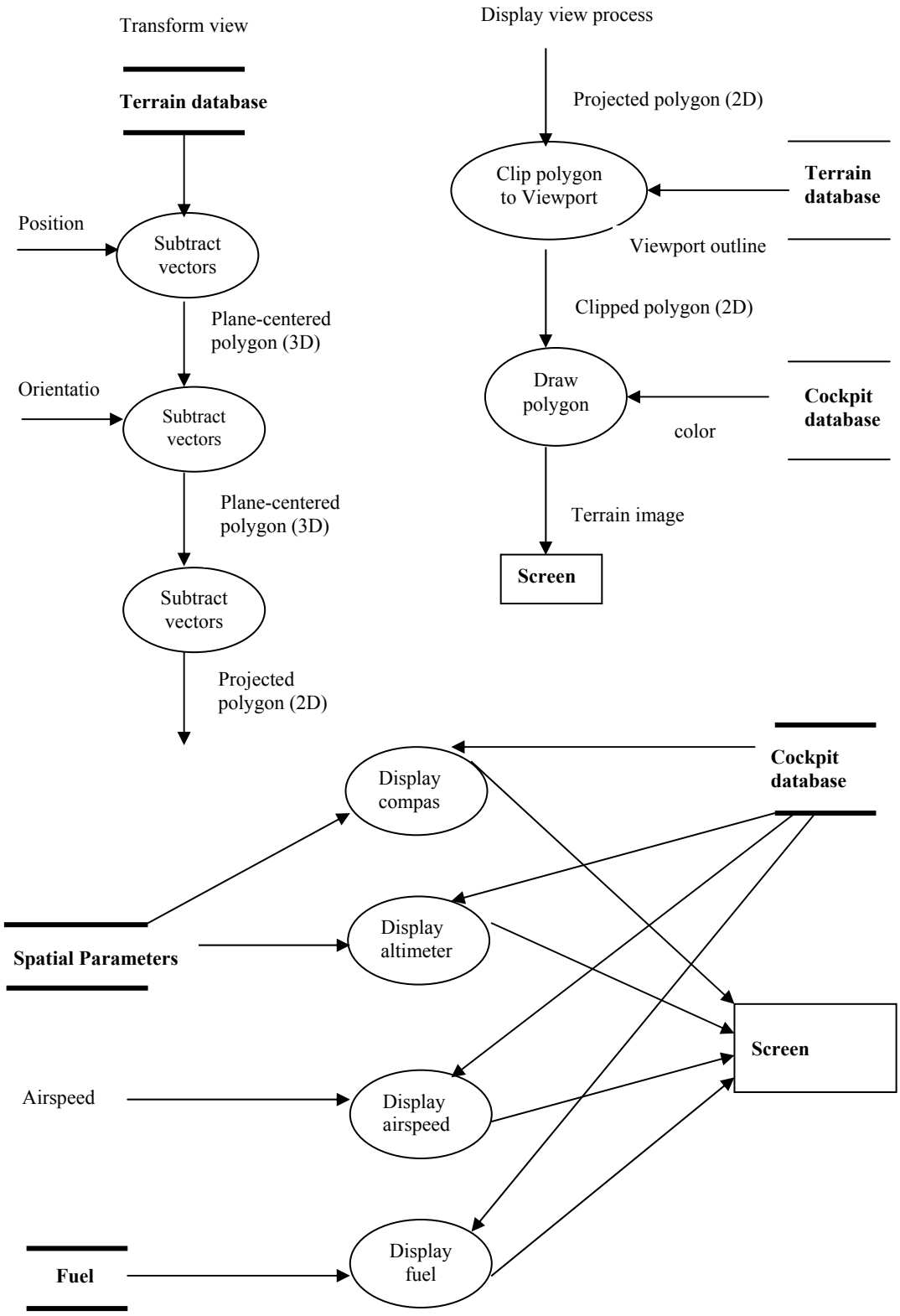


Figure 10: Expansion of compute forces processes

Air density is also computed and used in subsequent processes. The intermediate parameters are computed in terms of data store parameters, such as airplane velocity, orientation, rotation rates, roll rate, pitch rate, and altitude, obtained from spatial parameters; **wind velocity**, **temperature**, and **pressure**, obtained from **Weather**; weight, obtained from **Weight**; and in terms of output data flows from other processes, such as elevator angle, aileron angle, and engine speed, obtained



Display gauges process

Figure 11: Expansion of display processes

from process adjust controls. The internal processes, such as compute drag, compute lift and compute density, would be specified by aeronautical formulas and look-up tables for the specific airplane. For example, computer lift is specified by the equation **L=C (a) SpV^2/2**, where L is lift, a is the angle of attack, S is the wing area, P is the air density, V is the airspeed, and C is the coefficient of lift as a function of angle of attack, specified by a table for the particular kind of wing. Process integrate motion is the solution to the differential equations of motion.

The display processes are transform view, display view, display gauges, and display cockpit. These processes convert the airplane parameters and terrain into a simulated view on the screen. They are expanded on *Figure 9*. Process transform view transforms the coordinates of a set of polygons in the Terrain database into the pilot's coordinate system, by first offsetting them by the plane's position, rotating the polygons by the plane's orientation, and then transforming the polygons' perspective onto the viewing plane to produce a 2-D image in the pilot's eye view. The position and orientation of the plane are input parameters. Process display view clips the image of the transformed polygons to remain within the output of the cockpit view port, whose shape is specified in a Cockpit database. The clipped 2-D polygons are drawn on the screen using the colors specified in the Terrain database. Process display gauges displays various airplane parameters as gauges, using locations specified in the cockpit database. Process display cockpit, displays a fixed image of the stationary parts of the cockpit, and need not be expanded. You must have observed in this example, that the functional model does not specify when, why, and how often values are computed.

☞ Check Your Progress 2

1) Explain the use of constraints in functional model with suitable example.

……

……

2) Take any object from your surrounding environment and describe, it by making a complete functional model of it, using data flow diagrams.

……

……

3) Prepare a data flow diagram for computing the volume and surface area of a cylinder. Inputs are the height and radius of the cylinder. Outputs are the volume and surface area of the cylinder. Discuss several ways of implementing the data flow diagram.

……

……

3.7 RELATION OF FUNCTIONAL TO OBJECT AND DYNAMIC MODEL

Let us now discuss the relationship between the **Object Model**, **Dynamic Model,** and **Functional Model**. The functional model shows what **has to be done** by a system. The leaf processes are the operations on objects. The object model shows the "**doers**" of the objects. Each process is implemented by performing a method on some object. The dynamic model shows **the sequences** in which the operations are performed. Each sequence is implemented as a sequence, loop, or **alternation** of statements within some method. The three models come together in the implementation of methods. The functional model is a guide to the methods.

The processes in the functional model **correspond** to operations in the object model. Often, there is a direct correspondence between each level. A top level process corresponds to an operation on a complex object, and lower level processes correspond to operations on more basic objects that are part of the complex object or that implement it. Sometimes, one process corresponds to several operations, and one operation corresponds to several processes.

Processes in the functional model show objects that are related by function. Often, one of the inputs to a process can be identified as the target object, with the rest being

parameters to the operations. The target object is a client of the other objects because it uses them in performing the operations. The target knows about the clients, but the clients do not necessarily know about the target. The target object class is dependent on the argument classes for its operations. The client-supplier relationship establishes implementation dependencies among classes; the clients are implemented in terms of suppliers class, and are therefore dependent on the supplier classes.

A process is usually implemented as a method. If the same class of object is an input and an output, then the object is usually the target, and the other inputs are arguments. If the output of the process is a data store, the data store is the target. If an input of the process is a data store, the data store is the target. Frequently, a process with an input **from** or output **to** a data store corresponds to two methods, one of them being an **implicit selection** or **update** of the data store. If an input or output is **an actor**, then it is the target. If an input is an object and an output is **a part** of the object or a neighbor of the object in the object model, then the object is the target. If an output object is created out of input parts, then the process represents a class method. If none of these rules apply, then the target is often implicit and is not one of the inputs or outputs. Often the target of a process is the target of the entire subdiagram. For example, in *Figure10* the target of compute forces is actually the airplane itself. Data stores weight and spatial parameters are simply components of the airplane that are accessed during the process.

Actors are explicit objects in the object model. Data flows to or from actors represent operations on or by the objects. The data flow values are the arguments or results of the operations. Because actors are self-motivated objects, the functional model is not sufficient to indicate when they act. The dynamic model for an actor object specifies when it acts.

Data stores are also objects in the object model, or at least fragments of objects, such as attributes. Each flow into a data store is an update operation. Each flow out of a data store is a query operation, with no side effects on the data store object. Data stores are passive objects that respond to queries and updates, so the dynamic model of the data store **is irrelevant to its behavior**. A dynamic model of the actors in a diagram is necessary to determine the order of operations.

Data flows are values in the object model. Many data flows are simply pure values, such as numbers, strings, or lists of pure values. Pure values can be **modeled as classes** and implemented as objects in most languages. A pure value is not a container whose value can change, but just the value itself. A pure value, therefore, has no state and no dynamic model. Operations on pure values yield other pure values and have no side effects. Arithmetic operations are examples of such operations.

Relative to the Functional Model: The **object model** shows the structure of the actors, data stores, and flows in the functional model. The **dynamic model** shows the sequence in which processes are performed.

Relative to the Object Model: The **functional model** shows the operations on the classes, and the arguments of each operation as well. The **dynamic model shows** the status of each object and the operations that are performed as it receives events and changes state.

Relative to the Dynamic Model: The **functional model** shows the definitions of the leaf actions and activities that are undefined with the **dynamic mode**. The object model shows changes of state during the operation.

Operations can be specified by a variety of means, including mathematical equations, tables, and constraints between the inputs and outputs. An operation can be specified by pseudopodia, but a specification does not imply a particular implementation; it may be implemented by a different algorithm that yields equivalent results. Operations have signatures that specify their external interface and transformations that specify their effects. Queries are operations without side effects; they can be implemented as pure functions. Actions are operations with side effects and duration; they must be

implemented as tasks. Operations can be attached to classes within the object model and implemented as methods. Constraints specify additional relationships that must be maintained between values in the object model.

☞ Check Your Progress 3

1) Describe the meaning of the data flow diagram in *Figure 12*.

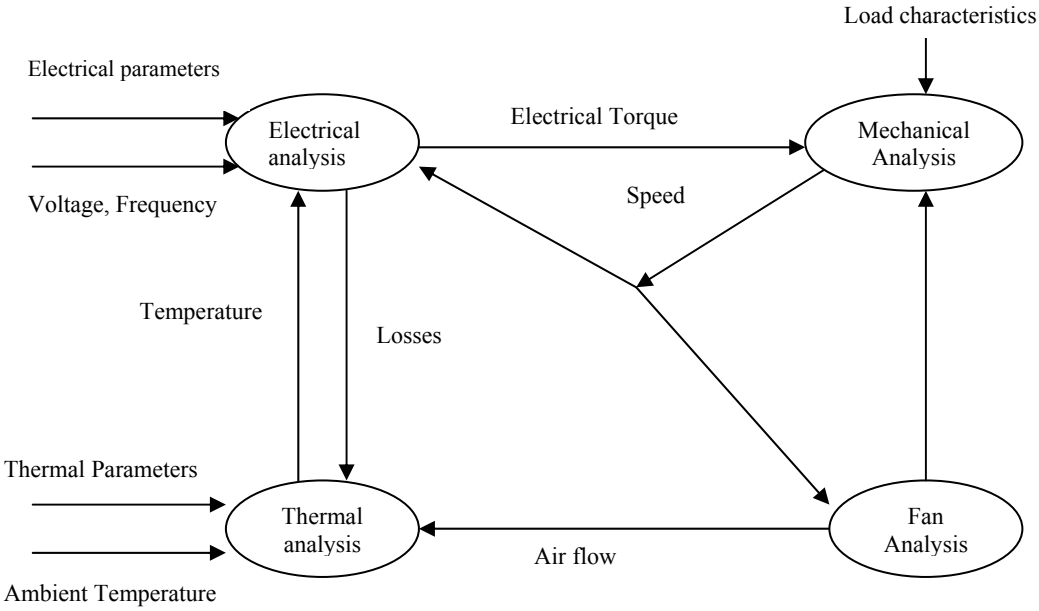


Figure 12: Data flow diagram of motor analysis

……..

……..

2) Prepare a data flow diagram for computing the mean of a sequence of input values. A separate control input is provided to reset the computation. Each time a new value is input, the mean of all values input since the last reset command should be output. Since you have no way of knowing how many values will be processed between resets, the amount of data storage that you use should not depend on the number of input values. Detail your diagram down to the level of multiplications, divisions, and additions.

……..

……..……

3) Using the quadratic formula as a starting point, prepare a data flow diagram for computing the roots of the quadratic equation $ax^2 + bx + c = 0$. Real numbers a, b and c are inputs. Outputs are values of X= R1 and X = R2, which satisfy the equation. Remember. R1 and R2 may be real or complex, depending on the values of a, b, and c. The quadratic formula for R1 and R2 is $(-b \pm SQRT(b^2 \pm 4ac)) / (2a)$.

……..……..….

…….

3.8 SUMMARY

The functional model shows a computation and the functional derivation of the data values in it without indicating how, when, or why the values are computed. The dynamic model controls which operations are performed and the order in which they are applied. The object model defines the structure of values that the operations

operate on. For batch-like computations, such as compilers or numerical computations, the functional model is the primary model, but in large systems all three models are important.

Data flow diagrams show the relationship between values in a computation. A data flow diagram is a graph of process, data flows, data stores, and actors. Processes transform data values. Low-level processes are simple operations on single objects, but higher-level processes can contain internal data stores subject to side effects. A data flow diagram is a process. Data flows relate values on processes, data stores, and actors. Actors are independent objects that produce and consume values. Data stores are passive objects that break the flow of control by introducing delays between the creation and the use of data.

The object model, dynamic model, and functional model all involve the same concepts, namely, data, sequencing, and operations, but each model focuses on a particular aspect and leaves the other aspects uninterrupted. All three models are necessary for a full understanding of a problem, although the balance of importance among the models varies according to the kind of application.

3.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) **Functional Model**

The functional model shows a computation and the functional derivation of the data values in it without indicating how, when, or why the values are compounded. For example a spreadsheet is a type of functional model. The values in a spreadsheet can be calculated by using some formula, but it can not be structured further.

2 i) **Data Flow Diagram**: A data flow diagram is a graph which shows the flow of data values from their sources in objects through processes that transform them to their destinations in other objects. It does not show how the values are controlled during computation. The Data Flow Diagram shows the functional relationship of the values computed by a system. DFD contains *processes* that transform data, *data flows* that move data*, actor* objects that produce and consume data, and *data store* objects that store data passively.

ii) **State Diagram:** An object can receive a sequence of input instructions. The state of an object can vary depending upon the sequence of input instructions. If we draw a diagram which will represent all the processes (input) and their output (states) then that diagram is known as state diagram. Processes are represented by the arrow symbol and states by an oval symbol. For example, the screen of an ATM machine has many states like main screen state, request password state, process transaction state.

Short Note

3i) **Process:** A process transforms data values. It is represented as an ellipse containing a description of the transformation. Each process has a fixed number of input and output data arrows, each of which carries a value of a given type.

ii) **Data Flows:** A data flow connects the output of an object or process to the input of another object or process. It represents an intermediate data value within a computation. Data flow specifies direction of flow of data from source objects to the destination object.

iii) **Actors:** An actor is an active object that drives the data flow graph by producing or consuming values. Actors are attached to the inputs and outputs of a data flow graph.

iv) **Data Stores:** A data store is a passive object within a data flow diagram that stores data for later access. A data store allows values to be accessed in a different order than they are generated.

Check Your Progress 2

1) A constraint shows the relationship between two objects at the same time, or between different values of the same object at different times. A constraint may be expressed as a total function or as a partial function. For example, a coordinate transformation might specify that the scale factor for the x-coordinate and the y-coordinate will be equal; this constraint totally defines one value in terms of the other.

2) Give a functional model for your example system with the help of section 9.6 of this Unit.

3) The data flow diagram for computing the volume and surface area of a cylinder is given in *Figure13* below.

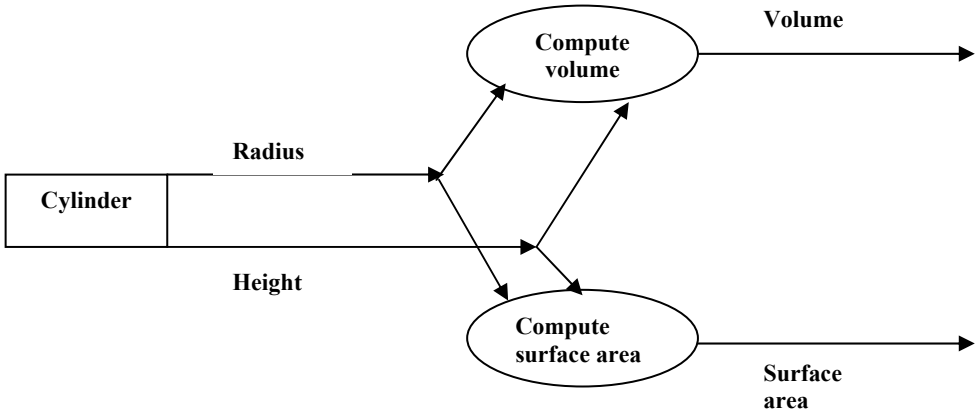


Figure 13: Data flow diagram for computing volume and surface area of a cylinder.

In this figure, the object cylinder is represented by a rectangle. The processes compute volume and surface are by ellipse, and the data flow by the arrow. The formula for volume and surface area of represented cylinder can be taken respectively.

Check Your Progress 3

1) The data flow diagram shows the relationship between values in a computation. The flow of electrical parameters is shown by the arrow symbol. In this DFD, there are **four processes**. These are **electrical analysis**, **mechanical analysis**, **fan analysis**, and **thermal analysis**. These four processes compute the various electrical parameters that are required for the safe running of an electrical motor.

The input electrical parameters are checked by the electrical analysis process. After proper verification, the mechanical movement (electrical torque) is measured by the mechanical analysis process. The characteristics of fan are computed by fan analysis process. The temperature of the motor is computed by the thermal analysis process. The flow of data from source to destination for this DFD is given by **Electrical analysis – mechanical analysis – fan analysis–thermal analysis**, and *vice versa*.

2) *Figure 14* (a) or (b) below shows the data flow diagram for computing the mean for a sequence of input values.

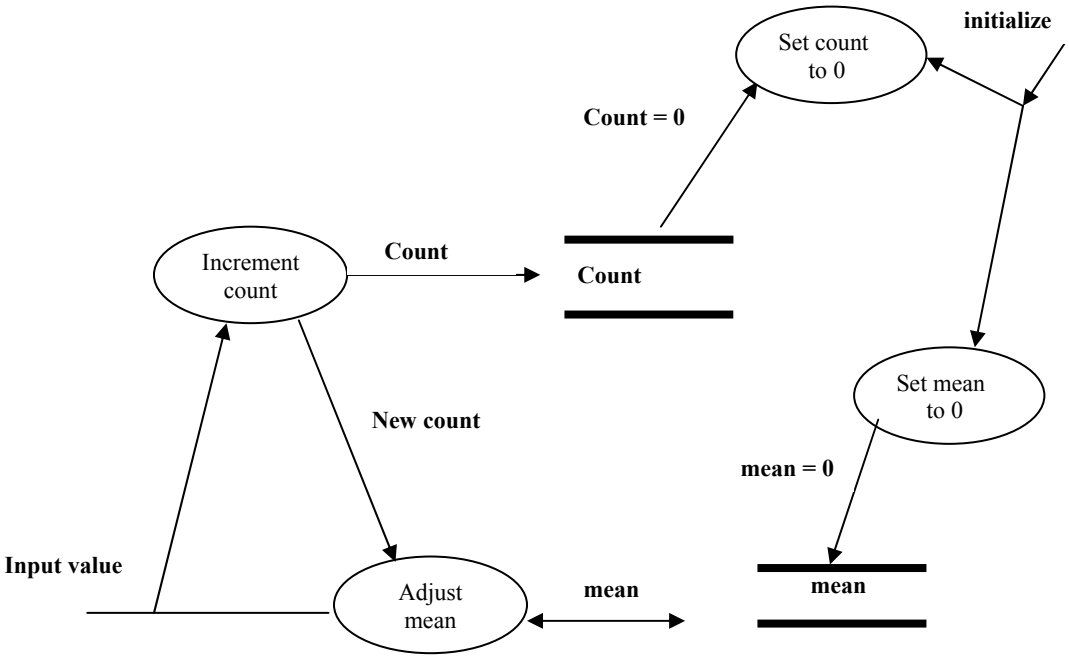
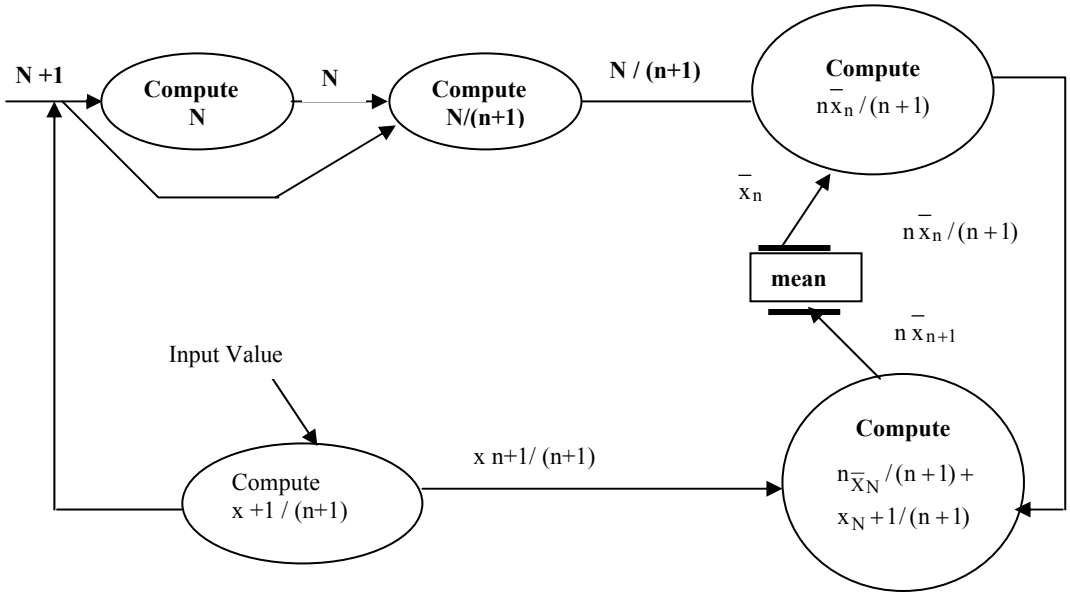


Figure 14 (a): DFD for computing mean



(Note: n+1 = new count, x_n = n^{th} value, $\bar{x}_n$ = average values)

Figure 14 (b): Data flow diagram for computing mean of a sequence of values

The above data flow diagram computes the mean of a sequence of input values. This DFD is the required solution of the problem.

3) The Data Flow Diagram for computing the roots of the quadratic equation is given by the following diagram.

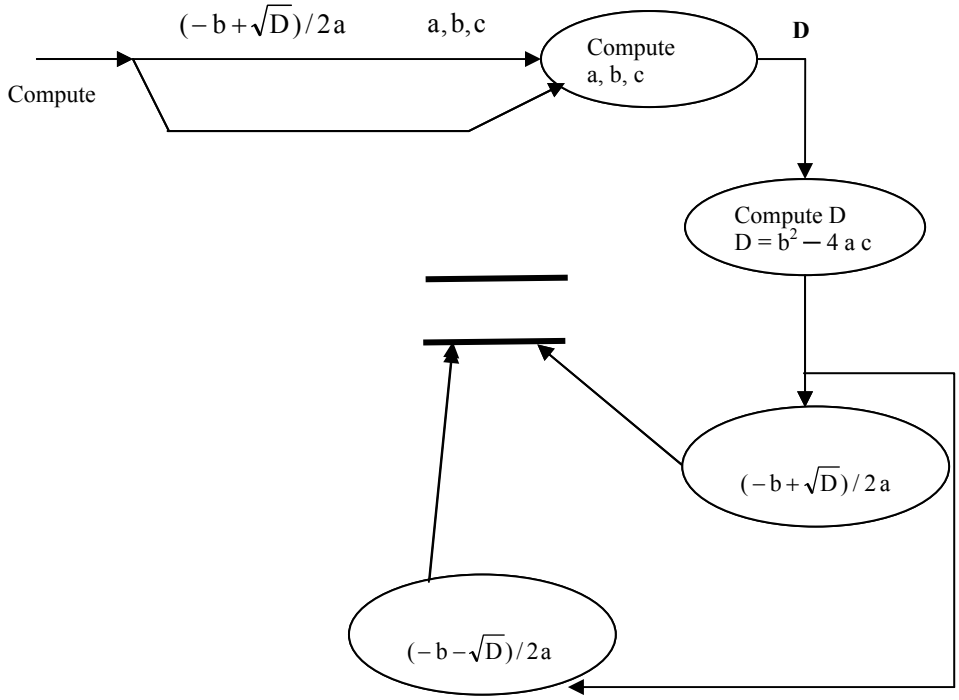


Figure 15: Data Flow Diagram for computing the roots of quadratic equation

UNIT 1 IMPLEMENTATION STRATEGIES

Structure **Page Nos.**

1.0 Introduction 5
1.1 Objectives 5
1.2 Implementation Associations 6
1.3 Unidirectional Implementations 6
1.3.1 Optional Associations
1.3.2 One-to-One Associations
1.3.3 Associations with Multiplicity 'Many'
1.4 Bi-directional Implementations 10
1.4.1 One-to-One and Optional Associations
1.4.2 One-to-Many Associations
1.4.3 Immutable Associations
1.5 Implementing Associations as Classes 14
1.6 Implementing Constraints 15
1.7 Implementing State Charts 16
1.8 Persistency 18
1.9 Summary 20
1.10 Solutions/Answers 20

1.0 INTRODUCTION

The transformation, from design models to code, is an easy and simple feature of object-oriented design, but there are some features of the design models which do not directly map into programming language structures. This unit considers some of the most noticeable features, and discusses the various strategies that have to be adopted for their implementation.

The most significant feature of class diagrams is **association** that is not directly present in the programming languages structures. In this unit, we will describes the different ways of implementation of complex types of association, such as qualified associations and association classes.

The information contained in the dynamic models of an application is reflected in the code that **implements individual operations** for that application. Object interaction diagrams are used to describe the order in which messages are communicated in the execution of an operation, and this information is used as guidance for the implementation of individual operations.

State charts are used to describe constraints that must apply across all the operations of a class, and which can affect the implementation of all a class's operations. A **consistent strategy** should be adopted to check that all these **constraints are correctly** reflected in the implementation of the member functions of the class.

Basically, this unit will cover different aspects of implementation, which covers implementing, associations, implementing constraints, and implementing statecharts.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- explain how the design models are implemented in coding;
- explain how the unidirectional associations are implemented;
- explain how the bi-directional associations are implemented;
- describe how the associations are implemented as classes;
- show how the generalizations are mapped to the tables;
- explain how the constraints are implemented.

- explain how the statecharts are implemented, and
- use the concept of persistence in the implementation.

1.2 IMPLEMENTATION ASSOCIATIONS

Associations describe the properties of the **links** that exist between **different objects** when a system is **running**. A link from one object to another informs each object of the identity, or the location of the other object. Association enables the objects to send messages to each other using the **link as a kind of communication channel**. When the implementation of links is done then these features of links should also to be supported. You can implement a simple association by using **references to linked objects**.

The basic difference between links and references is that links are symmetrical whereas references refer only in one direction. If two objects are linked, a single link serves as a **channel for sending messages in either direction**. By using a reference, however, one object can send messages to another, but the other object is **not aware of** the object that is referring to it. So, it has no way of sending messages back to the first object. You can say that if a link has to support message passing in both directions, it will require **a pair of references** for the implementation for each direction.

But, the use of two references adds a considerable overhead for the implementation. While implementing both references it has to be checked that inverse references are consistently maintained. The implementation of associations is required to be implemented only in one direction because that particular link only needs to be traversed in one direction. The link used association for one direction can be implemented by a single reference, pointing in the direction of traversal.

But, the implementation of an association only in one direction involves a tradeoff between the present implementation strategy and the future modifications that may be incorporated in the design which may also affect the association.

You can take unidirectional and bi-directional associations in this way:

i) unidirectional implementation associations are those in which the decision has been taken to maintain the association in only one direction.

ii) in bi-directional implementations, association must be maintained in both directions.

In general, there are two distinct aspects to the implementation of associations.

i) It is necessary to define the data declarations that will enable the details of actual links to be stored. It will consist of defining data members in one class that can store references to objects of the associated class.
ii) It is necessary to consider the means by which these pointers will be manipulated by the rest of the application. The details of the underlying implementation of the association should be hidden from client code.

Now, let us see how unidirection implementation takes place.

1.3 UNIDIRECTIONAL IMPLEMENTATIONS

In this section we will discuss cases by taking in to consideration that an association will only be supported in one direction. This design decision can be represented on a class diagram by writing an arrow **(→)** on the association to show the required direction of traversal.

We will discuss the cases where the multiplicity of the type is:

- Optional
- One, and/or
- Many.

1.3.1 Optional Associations

The *Figure1* shows an association that is implemented in **one direction**. Every bank account can have a debit card issued to the account holder for use. But it is not necessary that **all the account holders** will take the debit card from the bank. It depends upon the bank as well as on the account holder to have a debit card.

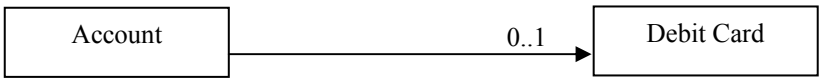


Figure 1: An optional association

This type of association you can implement using a simple reference variable, as shown below. This allows an account object to hold a reference to, at most, one debit card object. Cases where an account is not linked to a card are modeled by allowing the reference variable that will point to the **null**.

```
public class Account
{
public DebitCard getCard()
{
return theCard;
}
public void setCard(DebitCard card){
thecard = card;
}
public void removeCard()
{
theCard = null;
}
private DebitCard theCard
}
```

The above code assumes that a card may be supplied when an account is created. In addition, operations are provided to change the card linked to an account, or to remove a link to a card altogether.

You can see that in this implementation, different cards are linked to an account at different times during its lifetime. Associations with this type of property are called **mutable associations**. On the other hand, immutable associations are those that require that a link to one object cannot be replaced by a link to a separate object, i.e., the link cannot be changed. In this case, we will take the assumption that only one card has been issued to a particular bank account.

If the association between accounts and debit cards cannot be changed, then the following declaration of the account class should be used. This provides an operation to add a card to an account, and only allows a card to be added if **no card is already held**. Once allocated, a card cannot be changed or even removed, and the relevant operations have been removed from the interface.

```
public class Account
{
public Debitcard getCard( )
{
return theCard ;
}
public void setCard(DebitCard card)
```

```
{
if (theCard != null)
{
// throw ImmutableAssociationError
}
theCard = card ;
}
private DebitCard the Card ;
}
```

1.3.2 One-to-One Associations

Some of the properties of associations can be implemented **directly** by **providing suitable declarations** of data members in the relevant classes. Other **semantic features** of an association can be enforced by **providing only a limited range of operations** in the class's **interface**, or by including code in the implementation of member functions that ensures that the necessary constraints are maintained.

Now, let us consider the association shown below in *Figure 2*. This association describes a situation where **bank accounts** must have a **guarantor** who will pay any debts incurred by the account holder in exceptional conditions. It may frequently be needed to find out the guarantor of an account, but it is not necessary to find the details of the account of the guarantor for which s/he is responsible. So, the implementation of the association will be only in the direction **from account to guarantor**.

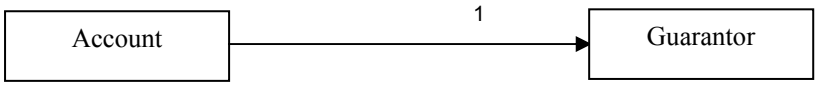


Figure 2: A one-to-one association

You may implement the account class in the following manner if the above assumptions are taken into consideration. The constructor will throw an exception if a null reference to a guarantor object is provided, and no operation is provided in the class's interface to update the reference held.

```
public class Account
{
public Account (Guarantor g)
{
if ( g == null) {
// throw Null Link Error
}
The Guarantor = g;
}
public Guarantor get Guarantor()
{
return the Guarantor;
}
private Guarantor the Guarantor;
}
```

The above code implements the association between account and guarantor objects as an **immutable association**. If the association is mutable, in that case the guarantor of an account could be changed, and a suitable function could be added to this class provided that feature. It should also check that the new guarantor reference **is –null** or not.

1.3.3 Associations with Multiplicity 'Many'

Figure 3 shows an association with multiplicity '**many'** for which a **unidirectional** implementation is required. In a bank, **each manager is responsible** for looking after a **number of accounts**, but it is not necessary to find out the manager of **a particular account**. In this association, a manager object could be linked not just to one, but potentially to many account objects.

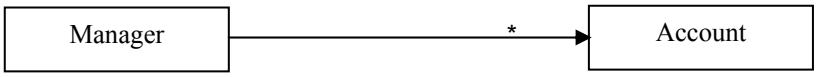


Figure 3: An association with multiplicity 'many'

In order to implement this association, the manager object **must maintain multiple pointers**, **one** to **each linked account**. A data structure is required to store all the pointers. Apart from this, it may be the case that the interface of the manager class will provide operations to maintain the collection of these pointers.

This type of association can be implemented by using of a suitable container class from a class library. The class declares a **vector of accounts** as a private data member, and the functions to add and remove accounts from this collection simply call the corresponding functions defined in the interface of the vector class.

```
public class Manager
{
public void addAccount(Account acc)
{
theAccounts.addElement(acc) ;
}
public void removeAccount(Account acc){
theAccounts.removeElement(acc) ;
}
private Vector theAccounts
}
```

In implementation, there can only be, at most, one link between **a manager** and **any particular account**. In this implementation, however, there is no reason why many pointers to the same account object could not be stored in the vector held by the manager. A correct implementation of the "***addAccount***" function should check whether the account that is added is linked to the other manager object or not.

☞ Check Your Progress 1

1) What is the most significant feature that does not directly map into programming language structures? Why?

……

……

……

2) What are object interaction diagrams?

……

……

……

3) What is the difference between link and reference?

……

……

……

4) What are the two distinct aspects to the implementation of associations?

……

……

5) Discuss how the one-to-one associations have to be implemented.

……

……

1.4 BI-DIRECTIONAL IMPLEMENTATIONS

In a bi-directional implementation of an association, a **pair of references** should **implement each link**. The declarations and code required to support such implementations are essentially the same as discussed in the unidirectional implementation. The only difference is that **suitable fields** have to be declared in **both classes** participating in the association.

On problem with bi-directional implementation is that the extra complexity in its implementations arises due to the need to keep the two pointers that implement a link consistent at the run-time.

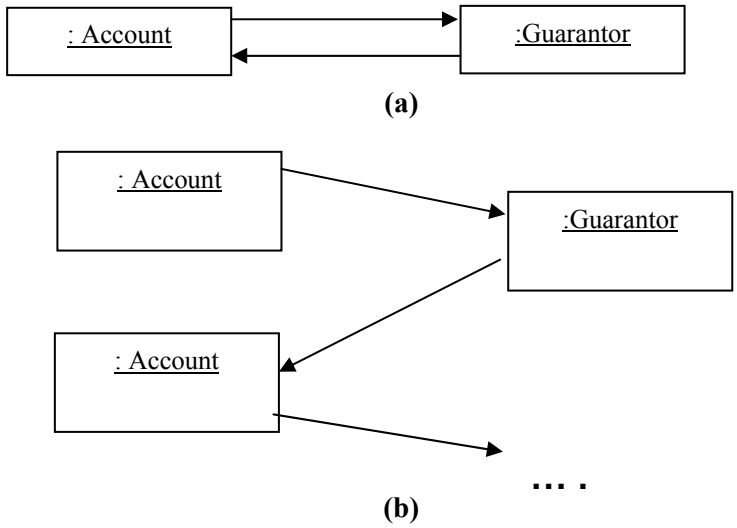


Figure 4: Referential integrity and its violation

It is necessary that the guarantor of an account must guarantee the same account type of the property of an association should hold between the accounts and guarantors as shown in *Figure 4(a). Figure 4(b)* violates this property, the top account **object holds a reference** to a guarantor object which in turn **holds a reference** to **entirely a different** account. These two references cannot be understood as being an implementation of a single link. This is a case of not being in referential integrity.

It should be clear from the above example that **referential integrity** cannot be ensured by simply giving appropriate definitions of data members in the relevant classes.

The unidirectional implementations will not support every association needs for all possible forms of manipulation of links. The behaviour that is required to be supported will depend on the **details of individual applications,** and will be defined by the **active operational interfaces** of the classes participating in the association.

1.4.1 One-to-One and Optional Associations

Assume that we are now required to provide a bi-directional implementation of the association discussed in previous section of this unit. Let us also assume that the association is immutable in the debit card to account direction, i.e., **once a debit card is linked to an account**, it must stay linked to that account until the end of its

lifetime. An account, on the other hand, can have different cards associated with it at different times, to cater for situations where the account holder loses a card.

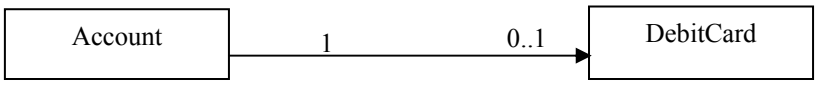


Figure 5: A bi-directional one-to-one association

We may think of this association as a combination of a mutable and optional association in the left-to-right direction with an immutable association in the other. A simple approach to its implementation would be to simply combine the implementations of those given in unidirectional implementation as shown below:

```
public class Account
{
public DebitCard getCard() {...}
public void setCard(DebitCard card) {...}
public void removeCard( ) {...}
private DebitCard theCard ;
}
public class DebitCard
{
public DebitCard(Account a) { ...}
public Account getAccount() { ...}
private Account theAccount;
}
```

The above implementation provides the data members to store the bi-directional links. But the operations available **maintain the two directions** of the link **independently,** e.g., the pointer from the account to the card is set up in the account constructor and the pointer from card to account in the card constructor.

For example, to create a link between a new debit card and an account, two separate operations are required, first to create the card and second to link it to the account. The link from card to account is created when the card itself is created. Code to implement can be as follows:

```
Account accl = new Account() ;
Debitcard card1 = new Debitcard(acc1);
acc1.setCard(card1) ;
```

It is necessary to ensure that these two operations are always performed together. However, as two separate statements are needed, there is a possibility that one might be omitted, or an erroneous parameter supplied, leading to an inconsistent data structure. In the following example, the debit card is **initialized** with the **wrong** account.

```
Account acc1= new Account (), acc2 = new Account ();
DebitCard card1 = new DebitCard(acc2) ;
accl.setCard(card1);
```

A better solution is to give only one of the classes the responsibility of maintaining the association. A link between two objects could then be created by means of a single function call, and encapsulation could be used to ensure that only trusted functions have the ability to directly manipulate links.

The choice of proper class in this regard is very important.

The choice of which class to give the maintenance responsibility depends upon the other aspects of the overall design. In this case, it is likely that there would be an operation on the account class to create a new debit card for the account. Also, it would provide a strong argument for making, the account class responsible for maintaining the association. If this is the case, then the classes could be redefined as follows.

Implementation

```
public class Account
{
public DebitCard getCard()
{
return theCard;
}
public void addCard()
{
theCard = new DebitCard(this) ;
}
private DebitCard theCard ;
}
public class DebitCard
{
DebitCard(Account a) {  theAccount = a ; }
public Account getAccount( ) {
return theAccount;
}
private Account theAccount ;
}
```

You can create debit cards by the 'addCard' operation in the account class. The implementation of this operation dynamically creates a new debit card object, passing the address of the current account as an initializer. The constructor in the debit card class uses this initializer to set up the pointer back to the account object creating the card. A single call to the add card operation is now guaranteed to set up a bi-directional link correctly.

In the above example, implementation is simple because the association declared is immutable in one direction. If, both directions of an association are mutable then a lot of situations will arise in which links can be altered. Then a correct implementation will be required to ensure that all the above things are correctly handled.

Take the assumption that a customer could hold many accounts but s/he has only one debit card. S/he had the flexibility to select which account has to be debited when the card is used. The association between accounts and debit cards would now be mutable in both directions. It would be reasonable for the card class to provide an operation to change the account with which card was associated.

There are many manipulations involved in above discussed case. First, the existing link between the **card and its account must be broken**. Second, a new link must be established between the new account and the card. Finally, this should only happen if the new account is not already linked to a card.

The card class must call functions in the account class to update pointers, as shown in the implementation of the '**changeAccount**' operation given below.

```
public class Account
{
public DebitCard getCard() { ...}
public void addCard(DebitCard c) { ... }
public void removeCard()
{
theCard = null ;
}
private DebitCard theCard ;
}
public class DebitCard
{
```

```
public DebitCard(Account a) {…}
public Account getAccount() {…}
public void changeAccount(Account newacc)
{
if (newacc.getCard() != null) {
// throw AccountAlreadyHasACard
}
theAccount.removeCard() ;
newacc.addCard(this) ;
}
private Account theAccount;
}
```

1.4.2 One-to-Many Associations

The bi-directional implementation of associations involving multiplicities of **many** raises the same problems as discussed in last sub-section. For example, *Figure 6* shows an association specifying that customers can hold many accounts, each of which is held by a single customer.

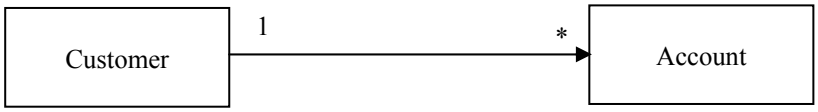


Figure 6: Customers holding accounts

The customer class could contain a data member to store a collection of pointers to accounts, and additionally each account should store a single pointer to a customer. The responsibility of maintaining the links of this should be given to the customer class.

1.4.3 Immutable Associations

Take the assumption that the association between accounts and guarantors was intended to be immutable and are traversed in both the directions. As *Figure 7* shows, the relevant class diagram which preserves the restriction that each guarantor can only guarantee one account.

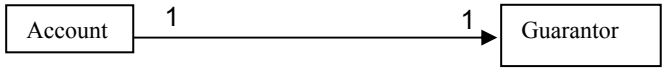


Figure 7: An immutable one-to-one association

Each class will define a data member **holding a reference to an object of the other class**. The class declarations might be as follows:

```
public class Account
{
public Account(Guarantor g)
{
theGuarantor = g;
}
public Guarantor getGuarantor()
{
return theGuaranter;
}
private Guarantor the Guarantor ;
}
public class Guarantor
{
public Guarantor(Account a)
{
theAccount = a;
```

```
}
public Account getAccount()
{
return theAccount;
}
private Account theAccount ;
}
```

You can see, in above declarations, that they introduce a circularity. When an account is created, it must be supplied with an already existing guarantor object, and likewise when a guarantor is created it must be supplied with an already existing account. It might be thought that this could be achieved by creating the two objects simultaneously, as shown in the following code:

```
Account a = new Account (new Guarantor (a));
Guarantor g = a.getGuarantor();
```

Associations are decided at the time of describing the requirements of the systems. Associations can be implemented in association classes and in the next section, we will discuss the implementation of association as classes.

1.5 IMPLEMENTING ASSOCIATIONS AS CLASSES

It is not possible to handle association classes with a simple implementation of associations based on references. For example, consider the diagram shown below which shows that many students can be registered as taking modules, and that a mark is associated with each such registration.

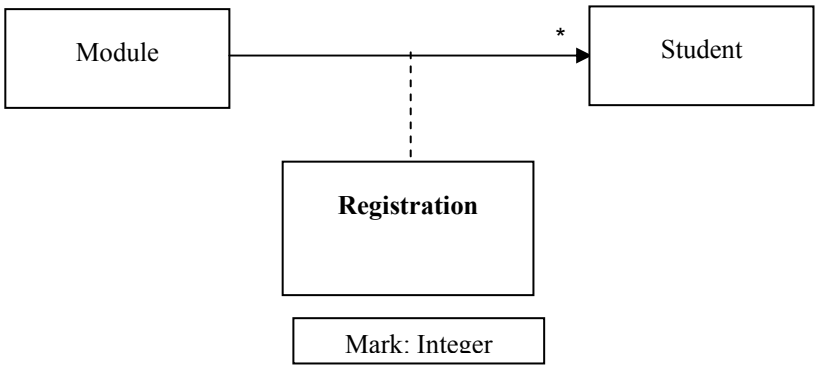


Figure 8: An association class

The association class shown in *Figure 8* needs to be implemented **as a class**, to provide a place to store the attribute values representing mark. The links corresponding to the association cannot then be implemented as references between the module and student classes.

A common approach, in this case, is to transform the association class into a simple class linked to the two original classes with two new associations, as shown in *Figure 9* below. In this diagram, the fact that many students can take a module is modeled by stating that the module can be linked to many objects of the registration class, each of which is further linked to a unique student, namely, the student for whom the registration applies.

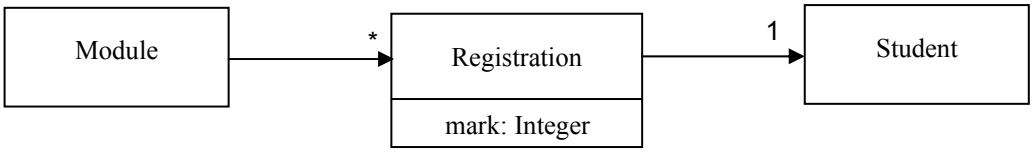


Figure 9: Transforming the association into a class

As a result of this transformation, neither of the associations in *Figure 9* have link attributes and so, they can be given straightforward implementations using references.

The original association shown in *Figure 8* would naturally have been maintained by the module class, which might provide operations to add a link to a student, and to record a mark stored for a student. Despite the fact that the association is now being implemented **as a class**, the interface presented to client code should **remain unchanged**. This implies that the module class must maintain both the registration class in *Figure 8* and *Figure 9*. Also, you can see that there are two new associations. Therefore, the operation to add a student to a module must create a new object and two new links.

The implementation of the registration class is very simple. It must store a reference to the linked student object and the mark obtained by that student. No need to provide any exclusive operational interface.

```
class Registration
{
Registration (Student st)
{student = st; mark= 0}
private Student student ;
private int mark;
}
```

The relevant parts of the definition of the module class are written below. Here, you can see that the implementation given in this example does not perform any of the validation discussed above.

```
public class Module
{
public void enrol(Student st)
{ registrations.addElement(new Registration(st));
}
private Vector registrations;
}
```

Associations, are implemented as a class whenever a class diagram contains l**ink attributes,** or **associations modeled as classes. Many-to-many association as a class** can be implemented in many situations. A simple pointer-based implementation will not be able to handle many- to-many association efficiently, whereas if the association had been implemented as a class, it would then be almost insignificant to add link attributes.

Bi-directional implementations of associations that are implemented, as a class do **not add new problems** to the above-mentioned situation. This make it easily implementable.

☞ **Check Your Progress 2**

1) Discuss how the bi-directional Implementations are made.

……

……

2) Explain how the associations are implemented as classes.

……

……

1.6 IMPLEMENTING CONSTRAINTS

Class constraints are used to describe relationships that must be between the attribute values of an instance of the class; and preconditions and post-conditions specify what must be true before and after an operation are called. Once these are implemented by

including code in the class which checks these conditions at the appropriate times; then the applications become more reliable and robust.

It is necessary that all the preconditions that are specified for an operation should be explicitly checked in an implementation. Preconditions state properties of an operation's parameters that must be satisfied if the operation is to be able to run to completion successfully. It is the responsibility of the caller of the operation to ensure that the precondition is satisfied when an operation is called.

If an operation does not check its parameter values, then, there is a possibility that wrong or meaningless values will go undetected, and this results in unpredictable **run-time errors**. A better strategy is for an operation to check its precondition and to raise an exception if a violation of the precondition is detected. The following example provides a possible implementation of the withdraw operation of the savings account class.

```
public class SavingsAccount
{
public void withdraw(double amt)
{
if (amt >= balance)
{
// throw PreconditionUnsatisfied
}
balance = balance - amt ;
}
private double balance ;
}
```

In general**, any constraint can be checked at run-time** by writing code that will validate the current state of the model. But such checks increase overhead. That is why, except for the case of precondition checking, constraints are rarely implemented explicitly.

Statecharts are diagrams that specify an object's responses to the events it might detect during its lifetime. We will discuss statechart in next section.

1.7 IMPLEMENTING STATE CHARTS

It is very important to apply proper strategy in final implementation of the systems. Here we will see some strategies for implementations.

A Basic Implementation Strategy

This approach models the different states in the statechart explicitly by means of an enumeration in the class to which the statechart applies. The current state that an object is in, is recorded by a special data member of the class that can take on values from this enumeration.

Member functions that can have different effects depending on the state of the object are implemented as **switch statements**, each case of which represents one possible state of the object. The implementation of each case corresponds to a single transition on the statechart. It should check any applicable conditions, perform any actions, and if necessary change the state of the object by assigning a new value to the data member which records the current state.

This implementation of this approach is simple and can be generally applied but it has some disadvantages also. It does not provide much flexibility in the case where a new state is added to the statechart. The implementation of every member function of the class would have to be updated in such a case, **even if they were unaffected by the change**. Also, the strategy assumes that most functions have some effect in the

majority of the states of the object. If this is not the case, the switch statements will be full of '**empty cases**' and the implementation will contain a lot of redundant code.

An Alternative Approach

An alternative implementation of statecharts can avoid these problems at the cost of **representation of individual states**. Rather than representing the current state of an object by the value of a data member in the object itself, this approach represents states as objects. Each instance of the class described by the statechart maintains a pointer to its current state, which is an instance of one of the state classes. The state classes are arranged in a generalization hierarchy so that different states can be referred, by the same pointer.

Now, let us see a class diagram illustrating the structure of this implementation. You can see in *Figure10* below, which shows the classes that would be declared to implement the creation tool class from the diagram editor.

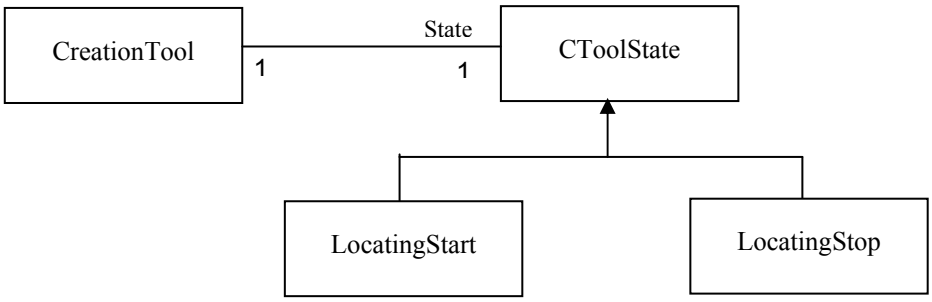


Figure 10: Representing states with classes

A field in the creation tool class will hold a reference to an object of type 'CtoolState'. This is an abstract class, so at run-time the object referred to will be an instance of one of the subclasses 'LocatingStart' or 'LocatingStop'. In this way, a creation tool always holds a reference to an object representing its current state.

The classes representing states provide implementations of the operations declared in the creation tool interface. When a creation tool receives a message, it simply passes it on to the object representing its current state, which contains a suitable implementation. A partial definition of the creation tool class could be given as follows.

```
public class CreationTool
{
public void press()
{
state.press() ;
}
private CToolState state;
}
```

The interface of the 'CtoolState' class must include all the messages that will be passed on from tools. Default implementations can be provided in 'CtoolState', however, so that individual states need only define those operations which evoke some interesting behaviour in that state. The following code gives a partial declaration of the 'CtoolState' class.

```
public abstract class CToolState
{
public abstract void press();
}
```

Subclasses that represent individual states must now redefine the operations that interest them. In the case of 'press', the press event can be detected in the 'LocatingStart' state, and in response the tool should change state. A possible

definition of the function is given below. For reasons that are discussed below, only pseudocode implementations of the functions are given.

```
public class LocatingStart extends CToolState
{
public void press()
{
set start position to current;
draw faint image of shape;
set current state to 'LocatingStop' ;
}
}
```

As the press event cannot be received in the 'LocatingStop' state, as the mouse button is already depressed, no definition of this function is required in the class 'LocatingStop'. The default implementation of the function inherited from 'CTooIState' is quite adequate.

It is important that any implementation must be sensitive to the **fact that an object can be in different states at different times**, and that the effect of operations is dependent on the **current state**. The simple implementation makes this all explicit, and the programmer must write switch statements to detect the current state of the object. In the more sophisticated implementation outlined above, detection of the current state is performed implicitly by the dynamic binding performed when a virtual function from the general 'CtoolState' class is called.

The sophisticated implementation has several advantages over the simple one. For example, the classes that represent individual states only need to define the operations that are relevant to them, and can inherit default implementations of the others from the general state class. This can considerably simplify the implementation of a statechart, especially in the case where many operations are only applicable in a small subset of the object's states.

The sophisticated approach is also more maintainable than the simple one. For example, if the statechart is extended to include extra states, these can simply be added as new subclasses of the general state class, and existing code which is not relevant to the change will be unaffected. This contrasts with the simple implementation, where adding a state requires the implementation of every member function to be updated.

There are costs associated with the sophisticated approach, however, which mostly stop from the fact that the implementation of the member functions of the state classes often needs to update the state of the object itself.

1.8 PERSISTENCY

Persistent data is data which has a longer lifetime than the program created it. In the context of an object oriented program, this means that it must be possible to save the objects created in one run of a program and to reload them at a later date. The user should not have to create all the data used by a program from scratch every time the program is run. The usefulness of the program will be rather limited if it is not possible to save diagrams to disk and to continue working on them at a later date.

Enabling data to be **stored on a permanent storage medium** provides persistency. The most common techniques used are to store data in files, or to make use of a back-end database system.

Identifying Persistent Data

The basic problem is that it may not be always clear from a model exactly what data needs to be persistent. Models in UML are not restricted to describing permanent data,

or database schemas, and as a result a single model can combine persistent and transient data.

The only notation that UML provides for persistency is a tagged value ‘persistence’. This has two values ‘persistent’ and ‘transient’ and can be applied to classes, associations and attributes.

For example, the diagram and element classes in the diagram editor need to be persistent. This is the data that the user would expect to be able to save and reload at a later date. The tool class, on the other hand, does not need to be persistent. Tools represent transient features of the user’s interaction with the editor, and it would not be felt to be a major shortcoming if the tool that was being last used was not available when the program was restarted.

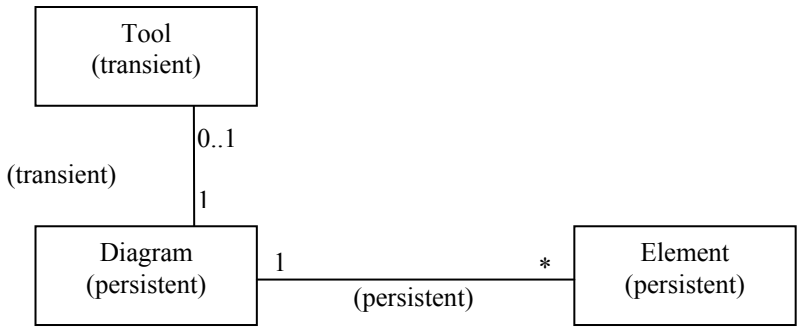


Figure 11: Persistent and transient classes

The primary unit of persistency is the class. Associations between two persistent classes are normally persistent, so that information about the links between persistent objects is stored. An **association between a persistent and a transient class will be transient**, as the instances at one end of the association are not being stored. Attributes normally have the same persistence value as their enclosing class, though it is sometimes natural to have a transient attribute within a persistent class.

Dealing with Object Identities

References are transient, and the persistency of an object model cannot be obtained simply by copying the values of these references to disk. The problem is that references are normally implemented using the address of the object referred to in memory, and therefore an object’s identity depends on where free memory happened to be available when it was created.

A more common approach, however, is to adopt some form of encoding whereby references and object identities are stored in such a way that they can be consistently recreated.

Serialization

Serialization is a generic term used for mechanisms that enable objects and object structures to be converted into a portable form, removing the volatility created by object addresses. As we have already studied in MCS-024, serialization is provided in Java by means of an interface called ‘Serializable’. This interface defines no methods, so in order to make a class serializable it is sufficient to declare that it implements this interface. Once this has been done, the methods ‘writeObject’ and ‘readObject’ can be used to transfer objects to and from streams, and persistence can then easily be implemented. Serialization therefore provides a convenient and straightforward way of making data persistent. It is most appropriate when the amount of data involved is relatively small. If larger amounts of data are to be stored, serialization may no longer be appropriate.

☞ Check Your Progress 3

1) What do you mean by persistence? How you will make your data persistent?

……

……

……

2) What is serialization? Where it is used and why?

……

……

……

3) What are the different strategies of implementation of the Statecharts?

……

……

……

1.9 SUMMARY

A simple strategy for implementing associations uses references to model links. Implementations can be either unidirectional or bi-directional, depending on how the association needs to be navigated. Bi-directional implementations of associations need to maintain the referential integrity of the references implementing a link. A strategy for robustness is to assign the responsibility of maintaining references to only one of the classes involved. Also, we have seen that Association classes should be implemented as classes. This involves introducing additional associations connecting the association class to the original classes. Implementing an association as a class is a general strategy for the implementation of associations that can increase the ability of a system to withstand future modifications at the expense of a more complex implementation.

We saw that Constraints in a model can be explicitly checked in code, but often it is only worth doing this for the preconditions of operations. A sophisticated technique for implementing statecharts was described, which represented each state by a class. This offers considerable benefits, at the cost of a significantly more complex implementation. Persistent data is data that must be preserved after the program which created it has finished running. Providing persistence for object structures is not easy, because of the use of references to implement links between objects. Small amounts of data can be saved using the technique of serialization, provided by many object-oriented programming libraries.

1.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Association, because there are complex types of association, such as qualified associations and association classes which are very difficult to implement.

2) Object interaction diagrams are used to **describe the order** in which messages are communicated in the execution of an operation, and this information is used as guidance for the implementation of individual operations.

3) The basic difference between links and references is that **links are symmetrical** whereas references **refer only in one direction**. If two objects are linked, a single link serves **as a channel** for sending messages in either direction. By using a reference, however, one object can send messages to another, **but the other object is not aware of the object that is referring to it**.

4 i) It is necessary to define the data declarations that will enable the details of actual links to be stored. It will consist of defining data members in one class that can store references to objects of the associated class.

ii) It is necessary to consider the means by which these pointers will be manipulated by the rest of the application. The details of the underlying implementation of the association should be hidden from client code.

5) Some of the properties of associations can be implemented directly by providing suitable declarations of data members in the relevant classes. Other semantic features of an association can be enforced by providing only a limited range of operations in the class's interface, or by including code in the implementation of member functions that ensures that the necessary constraints are maintained.

Check Your Progress 2

1) In a bi-directional implementation of an association, **a pair of references** should implement **each link**. The declarations and code required to support such implementations are same as needed in the unidirectional implementation. The only difference is that suitable fields have to be declared in both classes participating in the association.

2) A common approach in this case is to **transform** the **association class** into a simple class linked to the two original classes with two new associations, as shown in *Figure 9*.

Check Your Progress 3

1) Persistent data is data, which has a longer lifetime than the program that created it. Enabling data to be stored on a permanent storage medium provides persistency. The most common techniques used are to store data is in the form of files or to make use of a back-end database system.

2) *Serialization* is a generic term used for mechanisms that enable objects and object structures to be converted into a portable form, removing the volatility created by object addresses.

   Serialization provides a convenient and straightforward way of making data persistent. It is most appropriate when the amount of data involved is relatively small. If larger amounts of data are to be stored, serialization may no longer be appropriate.

3) There are two strategies of implementation for state charts:

   i) **Basic Implementation Strategy**

   This approach models the different states in the statechart explicitly by means of an enumeration in the class to which the statechart applies. It represents the current state of an object by the value of a data member in the object itself.

   ii) **An Alternative Approach**

   This approach represents states as objects. Each instance of the class described by the statechart maintains a pointer to its current state, which is an instance of one of the state classes. The state classes are arranged in a generalization hierarchy so that the same pointer can refer different states.

Implementation

UNIT 2 OBJECT MAPPING WITH DATABASE

Structure **Page Nos.**

2.0 Introduction 22
2.1 Objectives 22
2.2 Relational Database Schema for Object Modes 22
  2.2.1 General DBMS Concepts
  2.2.2 Relational DBMS Concepts
  2.2.3 RDBMS Logical Data Structure
2.3 Object Classes to Database Tables 27
  2.3.1 Extended Three Schema Architecture for Object Models
  2.3.2 The use of Object IDs
  2.3.3 Mapping Object Classes to Tables
2.4 Mapping Associations to Tables 29
  2.4.1 Mapping Binary Associations to Tables
  2.4.2 Mapping Many-to-Many Association to Tables
  2.4.3 Mapping Ternary Associations to Tables
2.5 Mapping Generalizations to Tables 33
2.6 Interfacing to Databases 36
2.7 Summary 37
2.8 Solutions/Answers 38

2.0 INTRODUCTION

Object oriented designs are **efficient, coherent** and **less prone to the update problems** that are present in many other database design techniques presently. Relational databases are mostly widely used and employed in the organisations. It is not wrong to say that relational DBMS are increasing their advantage in functionality and flexibility and are also improving their performance. In spite of having many advantages in comparison to other databases, object oriented DBMS have not able to reach the commercial mainstream. Even so, object oriented DBMS is in the development stage. Many DBMS companies have started to support the concept of object oriented Design in their new products.

In this unit, we will discuss data base concept related object orientation which includes discussion on relational database schema for object models, object classes and database tables, and mapping of associations and generalizations to tables.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain object modes for relational database schema;
- explain how the object classes are mapped to the database tables;
- explain how the associations are mapped to the tables;
- explain how the generalizations are mapped to the tables;
- describe the interfacing to database, and
- describe the object mapping with databases.

2.2 RELATIONAL DATABASE SCHEMA FOR OBJECT MODES

You have already studied the basics of DBMS in MCS-023. Let us very quickly refresh those concepts.

2.2.1 General DBMS Concepts

The **database management system** (DBMS) is a software system that enables users to define, create, maintain, and control access to the database. Physically, this database is stored in one, or more files.

There are many reasons for using a DBMS, some of which are:

1. Control of data redundancy.
2. Data consistency: by eliminating or controlling redundancy, the inconsistency in the data of database is reduced.
3. Sharing between users: multiple users can access the database at the same time.
4. Sharing between applications: multiple application programs can read and write data to the same database.
5. Improved data integrity: database integrity refers to the validity and consistency of stored data. A DBMS can control the quality of its data over and above facilities that may be provided by application programs.
6. Improved security: database security is the protection of the database from unauthorized users.
7. Improved backup and recovery services: the database is protected from hardware crashes, and disk media failures.
8. Extensibility: data may be added to the database without changing the existing programs.
9. Data distribution: the database may be partitioned across various sites, organizations, and hardware platforms.
10. Economy of scale: combining all the organization's operational data into one database, and creating a set of applications that work on this one source of data can result in cost savings.

The lifecycle of the database applications includes the following steps:

1. Requirement analysis of the application.
2. Design of the application.
3. Devise a specific architecture for coupling the application to a database.
4. Selection of the desired DBMS depending upon the feasibility analysis.
5. Design of the database.
6. Write the application programs which provides a user interface, validate data, and perform computations.
7. Populate the database with the required data.
8. Execute the application and then query, insert, and update the database as needed.

There are two approaches to database design, **attribute driven** and **entity driven**.

- **Attribute driven**: Compile a list of attributes relevant to the application and normalize the groups of attributes that preserve functional dependencies.
- **Entity driven**: Discover entities that are meaningful to the application and describe them.

In a typical database design, there are few entities as compared to the attributes. Entity design is much more easily to manage. Object modeling is a form of entity design.

The three schema architecture on DBMS that was originally proposed by ANSI/SPARC committee is shown in *Figure1*. The database design consists of three layers, which are:

- External schema,
- Conceptual schema, and
- Internal schema.

External schema: an external view is an abstract representation of some portion of the total database. An external schema is a definition of an external view. Each external schema is a database design from the perspective of a single application. The external schema isolates applications from most changes in the conceptual schema.

- **Conceptual schema**: The conceptual view is a logical representation of the database in its entirety. The conceptual schema is a definition of that conceptual view. It integrates related applications and hides the details of the implementation of the underlying DBMS.
- **Internal schema**: The internal schema is the database as it is physically stored. The internal schema is the definition of that internal view. The internal schema level consists of actual DBMS code required for the implementation of the conceptual schema.

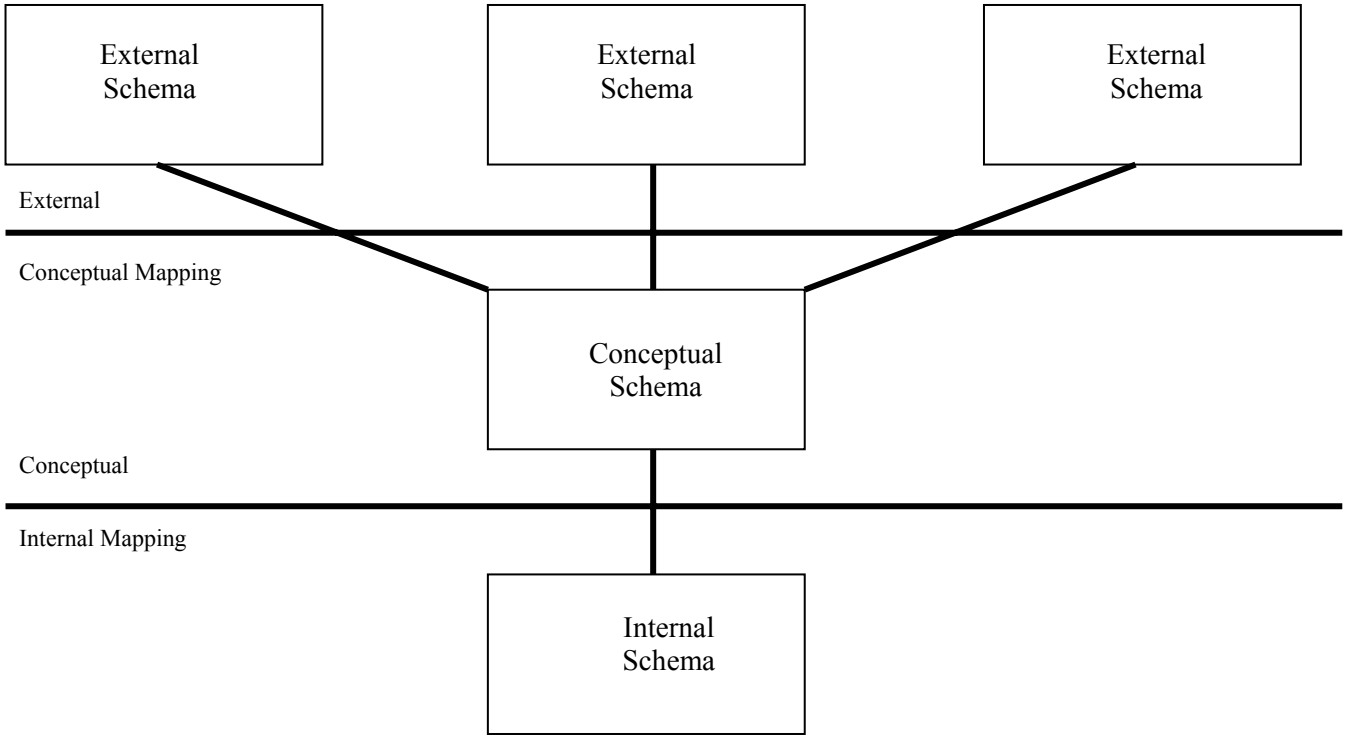


Figure 1: ANSI-SPARC Three level Architecture

Object modeling is useful for designing both the **external** and **conceptual** schema. For this, one should construct **one object model for each external schema, and another object model for the conceptual schema**.

2.2.2 Relational DBMS Concepts

RDBMS, as defined by Codd, has three major parts:

1. Data that is presented as tables
2. Operators for manipulating tables
3. Integrity rules on tables.

Now, let us discuss the concepts covering these three parts.

2.2.3 RDBMS Logical Data Structure

A relational database logically appears **simply as a collection of tables**. Tables have a **specific number** of columns and an **arbitrary number** of rows. The columns of tables are called attributes and directly correspond to attributes in object models. The rows are called **tuples** and correspond to **object instances and links**. A value is stored at each table row and column intersection.

Each value in a table **must belong to the domain of its attribute** or **can be null**. Null means that an attribute value is unknown or indeterminate.

The RDBMS decides if tuning is required for processing a query, and if so, automatically uses it. The RDBMS automatically updates tuning information

whenever the corresponding tables are updated. Indexing, hashing, and sorting are common tuning techniques.

RDBMS Operators

SQL is the most popular language for RDBMS. It follows ANSI as well as ISO standards. SQL provides operators for manipulating tables. The SQL 'SELECT' statement queries tables. The syntax of the select command looks something like:
SELECT attribute-list {as Alias}
FROM table-1, table-2, . . .
WHERE predicate-is-true
Logically Table-1, Table-2, and any others are combined into one temporary table. The attribute list specifies which columns should be retained in the temporary table. The predicate expression specifies which rows should be retained.
SQL can be classified into two categories:
SQL DDL commands: for creating the tables.
SQL DML commands : for inserting,updating, deletion operations.
SQL commands create Tables, insert rows into tables, delete rows from tables, and perform other functions.

RDBMS Integrity

A primary key is a combination of one, or more attributes whose unique value locates each row in a table. The primary key is always a candidate key. In the table shown below, **emp_no** is the primary key of the employee table; **dept_no** is the primary key of the department table.

Table 1: Employee Table

emp_no	Emp_name	Address	dept_no
1	Ram Kumar	20/780, Hauz Khas, New Delhi	02
2	Shyam Kumar	2/462, Sector 62,Noida	01

Table 2: Department Table

dept_no	Department Name	Address
01	Sales	2/2 Sector 2, Noida
02	Finance	1/2 Hauz Khas, New Delhi

Referential integrity requires that the RDBMS keep each foreign key consistent with its corresponding primary key. A foreign key is a primary key of one table that is references in another table. Referential integrity is useful when we are mapping object models to tables.

Normal Forms

Normal forms are rules developed to avoid logical inconsistencies from table update /insert operations. Each normal form avoids a form of redundancy in table organization that could yield ambiguity results if one table is updated independently of other tables. There are many levels of normal form, i.e., 1NF to 5NF and BCNF. Each higher level of normal form adds a constraint to the normal below it. As the database designer satisfies higher normal form, tables tend to become fragmented, higher normal forms improve database consistency, but at the cost of added navigation and slower query execution.

A table is in first normal form when each attribute will have **atomic value**. A table is in second normal form when **it satisfies first normal form and the partial dependency(ies) in the table are removed**.

A table is in third normal form **when it satisfies second normal form and transitive dependency is removed if any.**

Views

View is a virtual table that is dynamically computed as needed. A **view does not physically exist in the database** but is derived from one or more underlying tables. In

theory, views are the **means for deriving external schema from conceptual schema** for the ANSI three schema architecture. Commercial, RDBMS **usually support reading** through views, but **rarely support writing** through views.

Student Table

Table 3: Different views from Student Table

Name_id	Course_No	Grade	Phoneno	Major_no	Major-elective	prof
N1	C1	A	232456	M1	M1E1	SANJAY
N2	C2	B	256665	M1	M1E1	RAM
N3	C2	D	267677	M2	M2E1	RAM

View 1

Course_No	Prof.
C1	Sanjay
C2	ram

View 2

Name_id	Course_No	grade
N1	C1	A
N2	C2	B
N3	C2	D

View 3

Major_no	Major-Elective
M1	M1E1
M2	M2E1

View 4

Name_id	Phoneno	Major_no
N1	232456	M1
N2	256665	M1
N3	267677	M2

☞ Check Your Progress 1

1) What are the advantages of object oriented databases in comparison with others? Why it is still not widely used?

……

……

……

2) What are three components of the schema architecture proposed by ANSI/SPARC?

……

……

……

3) What are the different approaches for database design?

……

……

……

4) What are the different integrity constraints in RDBMS?

……

……

2.3 OBJECT CLASSES TO DATABASE TABLES

Here, we will consider relational database design as RDBMS technology is gaining more acceptance among organisation and companies and it dominates the marketplace.

2.3.1 Extended Three Schema Architecture for Object Models

First, we should formulate object models for the external, and conceptual schema. Then we should translate each object model to ideal tables, that is, the table model. Views and interface programs connect external tables to conceptual tables. Conceptual tables convert these to the internal schema.

The object model focuses on logical data structures. Each object model consists of many classes, associations, generalizations, and attributes. Object models are effective for communicating with application experts and reaching a consensus about the important aspects of a problem. Object models help developers achieve a consistent, understandable, efficient, and correct database design.

Each table model consists of many ideal tables. These ideal tables are generic and DBMS independent. Ideal tables abstract the common characteristics of RDBMS implementations. The table model decouples DBMS from object model to table model mapping rules.

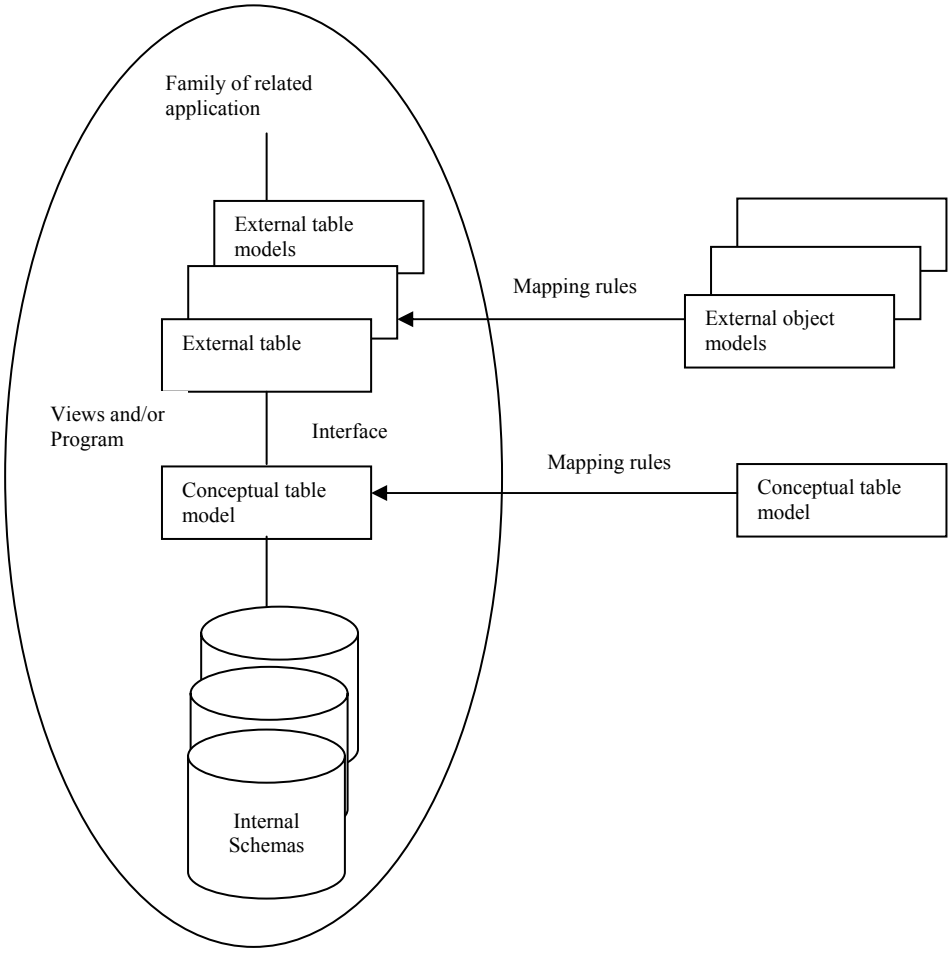


Figure 2: Object modeling and three schema architecture for relational DBMS.

In order to translate from an object model to ideal tables, we must choose from different mapping alternatives. For example, there are **two ways to map an association to tables,** and **four ways to map a generalization**. We must also supply details that are missing from the object model, such as the primary key and candidate keys for each table and whether each attribute can be null, or not null. Attributes in

candidate keys usually should not be null. You must assign a domain to each attribute and list groups of attributes that are frequently accessed.

The internal schema of the three-schema architecture consists of SQL commands that create the tables, attributes, and performance tuning structures.

2.3.2 The use of Object IDs

Each class-derived table has an ID for the primary key, one or more object IDs form the primary key for association derived tables. An ID is the equivalent database construct. There are benefits of using the IDs. IDs are never changing and completely independent of changes in data value and physical location. The stability of IDs is particularly important for associations since they refer to objects. A changing in a name requires update of many associations. IDs provide a uniform mechanism for referencing objects.

On the other hand, IDs have some disadvantages also. There is no support by the RDBMS for generating the IDs. For example, it is difficult to track previously allocated IDs, and, to reclaim deleted IDs for reuse.

Object oriented programming uses IDs of many bits, so that reuse of IDs is avoided. We can define IDs as attributes, and adopt a mechanism for handling them.
IDs are not used in applications where users directly access the database. A user thinks in terms of descriptive properties such as names, and not in terms of numbers. The advantage of IDs may prevail when database access is restricted via programs. Restricted access often occurs because application software is needed to compensate for DBMS deficiencies, to enforce integrity, and to provide a user interface.

2.3.3 Mapping Object Classes to Tables

Each **object class** maps to **one,** or **more tables** in the database. The objects in a class may be **partitioned horizontally** and/or **vertically**. For instance, if a class has many instances from which a few are often referenced, then horizontal partitioning may improve efficiency by placing the frequently accessed objects in a table, and the remaining objects in another tables. Similarly, if a class has attributes with different access patterns, it may help to partition the objects vertically.

Employee Table

Table 4: Horizontal Partition

emp_no	Emp_name	Address
1	Ram Kumar	20/780, Hauz Khas, New Delhi
2	Shyam Kumar	2/462, Sector 62, Noida

emp_no	Emp_name	Address
3	Avinash Kumar	F32, Madangir, New Delhi

Table 5: Vertical Partition Horizontal and Vertical Partitioning of Tables

emp_no	Emp_name
1	Ram Kumar
2	Shyam Kumar

emp_no	Address
1	20/780, Hauz Khas, New Delhi
2	2/462, Sector 62, Noida

An object class is converted to one table. Class employee has attributes emp_name and address. The table model lists these attributes, and adds the implicit object ID. As part of formulating the table model, we also add details. We specify that emp_ ID, ID cannot be null since it is a candidate key. Emp_name cannot be null_name must be

entered for every employee. Attribute address can be null. We assign a domain to each attribute, and specify the primary key for each table. Finally, the SQL DDL statements creates the employee table.

Table Model

Attribute Name	Nulls?	Domain
Emp_ID	N	ID
Emp_name	N	Name
Address	Y	address

Employee Table
Candidate Key: (Emp_ ID)
Primary key : (Emp_ ID)
Frequently accessed: (Emp_ ID)(emp_name)

Create TABLE Employee

(Emp_ID) ID not null,
emp_name char(20) not null,
address char(20)
PRIMARY KEY (Emp_ ID));
CREATE SECONDRY INDEX employee-index-name

ON Employee (emp_name);

MAPPING A CLASS TO A TABLE

2.4 MAPPING ASSOCIATIONS TO TABLES

In this section, we will discuss the mapping of different kinds of associations into database tables.

2.4.1 Mapping Binary Associations to Tables

In general, an association may, or may **not map to a** table. It depends on the type and multiplicity of the association, and the database designer's preferences in terms of extensibility, number of tables, and performance tradeoffs. Let us see one example of mapping a binary association into tables.

Object Model

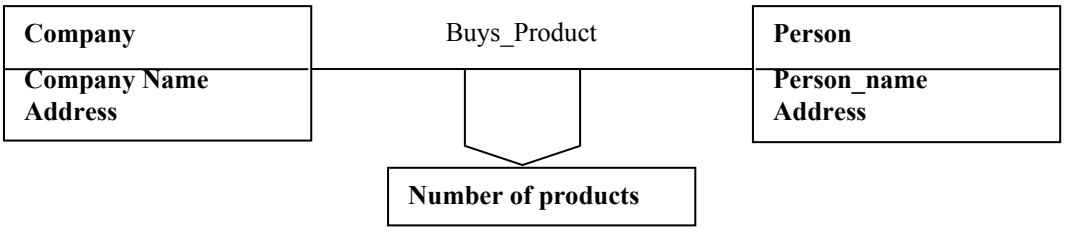


Table Model

Attribute Name	Nulls?	Domain
person_ ID	N	ID
person_name	N	Name
Address	Y	address

Person Table

Candidate Key: (person_ ID)
Primary key : (person_ ID)
Frequently accessed: (person_ ID)(person_name)

Attribute Name	Nulls?	Domain
company_ ID	N	ID
company_name	N	Name
Address	Y	address

Company Table

Candidate Key: (company_ ID)
Primary key: (company_ ID)
Frequently accessed: (company_ ID)
(company_name)

Attribute Name	Nulls?	Domain
company_ ID	N	ID
person_ ID	N	ID
Number_of _products	Y	product-count

Buys_products Table

Candidate Key: (company_ID, person_ ID)
Primary key : (company_ID, person_ ID)
Frequently accessed: (company_ID, person_ ID)

SQL Code

```
Create TABLE Person
(person_ID          ID          not null,
person_name         char(20)    not null,
address             char(20)
PRIMARY KEY (person_ ID));

CREATE SECONDARY INDEX person-index-name
ON Person (person_name);

Similarly, table and indexes can be created for company also
Create TABLE Buys_product
```

SQL Code

```
(company_ID         ID          not null,
person_ID           ID          not null,
number_of_products integer      ,
PRIMARY KEY ((company_ID, person_ ID)),
FOREIGN KEY (company_ID) references Company,
FOREIGN KEY (person_ ID)) references Person);,

CREATE SECONDARY INDEX buys_product-index-
company
 ON buys_product (company_ID);

CREATE SECONDARY INDEX buys_product-index-person
ON buys_product (person_ID);
```

2.4.2 Mapping Many-to-Many Association to Tables

A many to many association always maps to a distinct table. The primary keys for both related classes and any link attributes become attributes of the association table. Attributes company ID, and person ID combine to form the only candidate key for the Buys_product table. In general, an association may be traversed starting from either

class so that both company ID and Person ID could be frequently accessed. The foreign key clauses to the SQL code indicates that each Buys_product table tuples must reference a company and person that had been defined in their respective tables.

An association table always sets the foreign keys from the related objects to not null. If a given pair of objects does not have a link, we omit an entry in the association table.

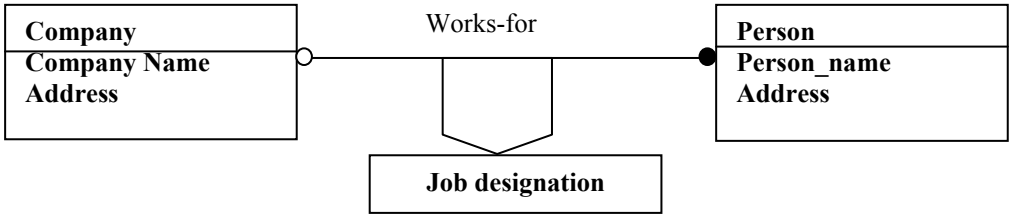


Figure 3: Object Model for one-to-many Association

Company Table (similar to the Company table in the last example)
Person Table (similar to Person table in the last example)

Attribute Name	Nulls?	Domain
company_ ID	N	ID
person_ ID	N	ID
Job_designation	Y	Designation

Works_for Table

Candidate Key: (person_ ID)
Primary key : (person_ ID)
Frequently accessed: (company_ID, person_ ID)

Table Model for one-to-many Associations-distinct association table
Company Table (similar to the Company table in the last example)

Attribute Name	Nulls?	Domain
person_ ID	N	ID
person_name	N	Name
Address	Y	Address
company_ ID	Y	ID
Job_designation	Y	Designation

Person Table

Candidate Key: (person_ ID)
Primary key: (person_ ID)
Frequently accessed: (person_ ID) (person_name)(company_ID)

Table Model for one-to-many Associations-added association table

The *Figure 3* shows two options for mapping a one-to-many association to the tables. We can create a distinct table for the association, or add a foreign key in the table for many class. The advantages of merging an association into a class are:

1. Fewer tables
2. Faster performance due to fewer tables to navigate.

The disadvantage of merging an association into a class are:

1. **Less design firmness:** associations are between independent objects of equal weight. In general, it seems inappropriate to mix objects with knowledge of other objects.
2. **Reduced Extensibility**: it is difficult to get multiplicity right on the first few design passes. One-to-one and one-to-many associations may be externalised. Many-to-many associations must be externalised.

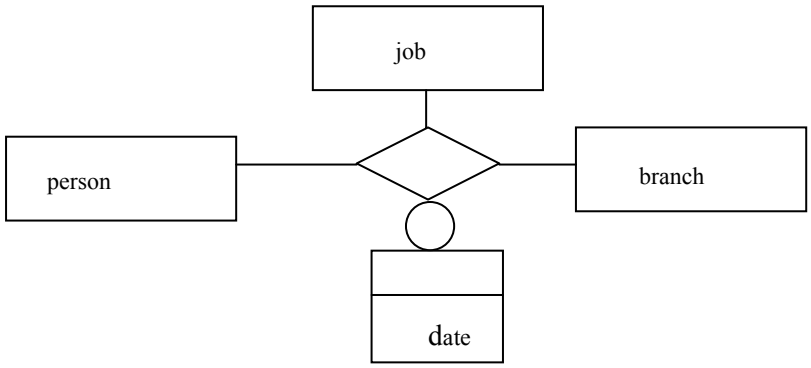
3. **Increased complexity**: an assymetrical representation of the association complicates search and update.

The final decision on whether to merge an association into a related class depends on the application. For a one-to-many association you may also merge both classes and the association all into one table.

2.4.3 Mapping Ternary Associations to Tables

Whenever a ternary relationship is there between the different classes then each class is mapped to a table with the inclusion of object ID. Apart from this a new ternary table is also created which has attributes from the different classes involved in the relationship. The attributes will the object ID of all the classes involved in the relationship and the attributes of the relation. Let us look at one example of mapping a ternary association to the tables.

Object model



Candidate key: (person_ID, branch_ID, job_ID)

Ternary Table:

Attribute Name	Nulls?	Domain
person_ID	N	ID
job_ID	N	ID
branch_ID	N	ID
hours	Y	Time

Table Model Candidate key: (person_ID, branch_ID, job_ID)
Primary key: (person_ID, branch_ID, job_ID)
Frequently accessed: (person_ID, branch_ID, job_ID)

Person Table:

Attribute Name	**Nulls?**	**Domain**
person_ ID	N	ID
person_name	N	Name
Address	Y	Address

Candidate Key: (person_ ID)
Primary key: (person_ ID)
Frequently accessed: (person_ ID)(person_name)

Similarly, the job and branch tables will have the required attributes.

SQL Code

```
Create TABLE Person
(person_ID          ID          not null,
person_name         char(20)    not null,
address             char(20)
PRIMARY KEY (person_ ID));
```

Similarly, create the other table for job and branch

```
Create TABLE Person-job-branch-ternary
(person_ID      ID      not null,
job_ID          ID      not null,
branch_ID       ID      not null,
date            date            ,
                PRIMARY KEY (person_ID, branch_ID, job_ID),
                Foreign key (person_ID) references person,
                Foreign key (job_ID) references job,
                Foreign key (branch_ID) references branch);
```

Also, create the indexes in the same manner

2.5 MAPPING GENERALIZATIONS TO TABLES

There are different approaches for mapping generalizations to table.

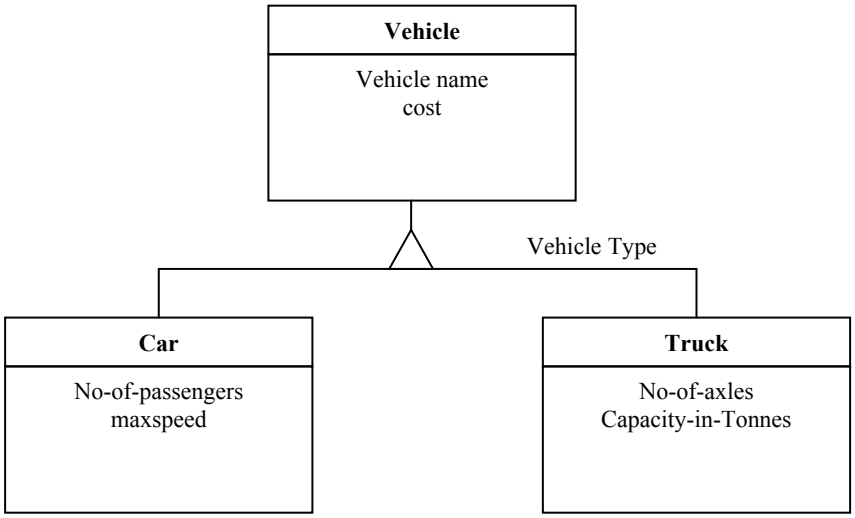


Figure 4: Object Model for Generalization

The first approach is shown in the *Figure 4*. The superclass and the subclass each map to a table. The identity of an object across a generalization is preserved through the use of a shared ID. Thus, vehicle car may have one row in the vehicle table with ID 1001, and another row in the Car table also with ID 1001.This approach is simple and extensible. However, it involves many tables, and super class to subclass navigation may be slow.

The navigation of the tables is as follows:

1. The user gives a vehicle name.
2. Find the Vehicle row that corresponds to vehicle name.
3. Retrieve the vehicle ID and vehicle type for this row.
4. Go to the subclass table indicated by vehicle type, and find the subclass row with the same ID as the Vehicle row.

Table: Model for Generalization Superclass and Subclass Tables:

Vehicle table:

Attribute name	Nulls	Domain
Vehicle-ID	N	ID
Vehicle-name	N	Name
Cost	Y	Money
Vehicle-type	N	Vehicle type

Candidate Key: (Vehicle-ID, Vehicle-name)
Primary key: (Vehicle-ID)
Frequently accessed: (Vehicle-ID, Vehicle-name)

Car table:

Attribute name	Nulls	Domain
Vehicle-ID	N	ID
No-of-passengers	Y	Person
Maxspeed	Y	speed

Candidate Key: (Vehicle-ID)
Primary key : (Vehicle-ID)
Frequently accessed: (Vehicle-ID)

Truck table:

Attribute name	Nulls	Domain
Vehicle-ID	N	ID
No-of-axles	Y	count-axles
Capacity-in-Tonnes	Y	capacity

Candidate Key: (Vehicle-ID)
Primary key: (Vehicle-ID)
Frequently accessed: (Vehicle-ID)

The user may specify vehicle type name “vehicle car.” The application looks in the vehicle table and finds that vehicle car has ID 1001 and vehicle type car. The application then searches the car table and retrieves additional data for ID 1001. The SQL code for the generalization of the super-class and sub-class as mentioned above is as follows:

Create table vehicle

```
(Vehicle-ID          ID          not null,
Vehicle-name         char (40)   not null,
cost                 number,
Vehicle-type         char (8)    not null,
primary key (Vehicle-ID));

Create Secondary index vehicle-index-name on vehicle (vehicle-name);

Create table car
(Vehicle-ID          ID          not null,
no-of-passengers     number,
maxspeed             real,
primary key (Vehicle-ID) ,
foreign key (Vehicle-ID) references vehicle);

Create table truck
(Vehicle-ID          ID          not null,
no-of-axles          number,
Capacity-in-Tonnes   real,
primary key (Vehicle-ID),
foreign key (Vehicle-ID) references Vehicle);
```

In the other approach, the navigation from superclass to subclass is eliminated and thus speed performance is achieved. The improved performance incurs a price by inserting the attributes of Vehicle table in both the subclasses.

Table Model for Generalisation Many Subclass Tables:

Car table:

Attribute name	Nulls	Domain
Vehicle-ID	N	ID
Vehicle-name	N	Name
Cost	Y	Money
No-of-passengers	Y	Person
Maxspeed	Y	Speed

Candidate Keys: (Vehicle-ID, Vehicle-name)
Primary key: (Vehicle-ID)
Frequently accessed: (Vehicle-ID, Vehicle-name)

Truck table:

Attribute name	Nulls	Domain
Vehicle-ID	N	ID
Vehicle-name	N	Name
Cost	Y	Money
No-of-axles	Y	count-axles
Capacity-in-Tonnes	Y	Capacity

Candidate Keys: (Vehicle-ID, Vehicle-name)
Primary Key: (Vehicle-ID)
Frequently accessed: (Vehicle-ID, Vehicle-name)

The above Table (Car Table, Truck Table) illustrates various subclass approaches. This approach eliminates the superclass table and replicates all the superclass attributes in each subclass table. We may use this approach if a subclass has many attributes, the superclass has few attributes, and the application knows which subclass to search.

Table Model for Generalization one Super Class Table

Vehicle table:

Attribute name	Nulls	Domain
Vehicle-ID	N	ID
Vehicle-name	N	Name
Cost	Y	Money
No-of-passengers	Y	Person
Maxspeed	Y	Speed
No-of-axles	Y	count-axles
Capacity-in-Tonnes	Y	Capacity

Candidate Keys: (Vehicle-ID, Vehicle-name)
Primary Key: (Vehicle-ID)
Frequently accessed: (Vehicle-ID, Vehicle-name)

The one superclass table approach shown above brings all subclass attributes up to the superclass level. Each record in the superclass table uses attributes present in one subclass, the other attribute values are null. The table in the above table (Vehicle table) violates third normal form. Vehicle-ID or vehicle-name is the primary key, but attributes values also depend on equipment type. This may be a useful approach if there are only two or three subclasses with few attributes.

The best way to handle generalization relationships that exhibits multiple inheritance from disjoint classes is to use **one table per super-class**, **one table per subclass**. The best way to handle multiple inheritance from overlapping classes is to use one table for the superclass, one table for each subclass, and one table for the generalization relationship.

☞ Check Your Progress 2

1) What is the main advantage of the object model? What are its different components?

……

……

2) What are advantages of object models?

……

……

3) What is an object ID?

……

……

4) What are the advantages of object ID?

……

……

5) How the object classes are mapped to tables?

……

……

6) What are the advantages and disadvantages of merging an association into a class?

……

……

2.6 INTERFACING TO DATABASES

For many applications, the only adequate solution to providing persistency of large amounts of data is to make use of an existing database system. As well as being able to handle effectively unlimited amounts of data, databases provide a number of valuable services, such as support for multiple users, which cannot realistically be implemented again and again, for every applications.

Object oriented databases provide seamless database support for applications designed using object oriented methods. However, affordable object oriented databases are not easily writable for object oriented applications. Object oriented applications may be written using backend relational database. However, this approach can create significant problems because the relational model of data is in some ways quite different from the object model.

Suppose that we have a class diagram describing the data model of an application, containing a number of persistent classes related by associations and generalization relationships. To provide database support for this application, we will have to create a relational database schema enabling the same information to be stored. This activity

involves **translating one notation into another**, and is similar in principle to that of implementing a model in a programming language: we need to find a way of expressing each UML construct using the concepts and notations of the target environment.

Representing associations

Associations can be translated into relational schemas in a number of ways. In the simplest case, a unidirectional link from object X to object Y can be implemented by storing the key value of Y in the row of the table corresponding to X. This is the relational equivalent of one object holding a reference to another; the embedded reference is known as a foreign key in relational database terminology.

Representing generalization

Implementing generalizations in relational databases is slightly problematic, as there is no single feature or technique which provides the required semantics. The most straightforward approach is to represent both the superclass and the subclass in the generalization relation as tables, with attributes of each class defined in the corresponding table.

Interfacing to databases

Once an object oriented data model has been implemented as a relational database, it is necessary to write the code that provides the functionality of the system. This code must be able to read and write data from the database, interpreting it wherever necessary, in terms of the model used by the program.

To achieve this, programming environments typically support an interface that allows programmers to abstract away from the details of individual databases, and enables an application to work with a variety of databases, or data sources. A typical example of such an interface is the Java Database Connectivity (JDBC) API that enables Java programmers to write programs, which interface to relational databases.

Essentially, an API like JDBC enables programmers to manipulate a database by constructing commands in the database query language SQL, executing them on the database, and then dealing with the data that is returned as a result.

☞ Check Your Progress 3

1) Explain how the object classes are mapped to tables.

……

……

2) Explain how the ternary associations are mapped to the tables.

……

……

3) Explain how the generalizations are mapped to the tables.

……

……

2.7 SUMMARY

In this unit, we have discussed mapping object classes to tables, i.e., each class maps to one, or more tables. Mapping associations to tables, each many-to-many association maps to a distinct table.

We have also seen that each one-to-many association maps to a distinct table, or may be buried as a foreign key in the table for the many class.

- Each one-to-one association maps to a distinct table, or may be buried as a foreign key in the table for either class.
- For one-to-many, and one-to-one associations, if there are no cycles, you have the additional option of storing the association, and both related objects, all in one table. Be aware that this may introduce redundancy, and violate normal forms.
- Role names are incorporated as a part of the foreign key attribute name.
- N-ary (n>2) associations map to a distinct table. Sometimes, it helps to promote an n-ary association to a class.
- A qualified association maps to a distinct table with at least three attributes, the primary key of each related class, and the qualifier.
- Aggregation follows the same rules as association.

Further, we have discussed mapping single inheritance generalizations to tables which includes:

- The superclass and each subclass map to a table.
- No superclass table, superclass attributes are replicated for each subclass.
- No subclass tables, bring all subclass attributes up to the superclass level.

We have also seen that in mapping disjointed multiple inheritance to tables, the super-class and each subclass map to a table. In the mapping overlapping multiple inheritance to tables the superclass and each subclass map to a table, the generalization relationship also maps to a table.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Object oriented designs are efficient, coherent, and less prone to the update problems that are not present in many other database design techniques presently.

   It is not used widely because only few database vendors have supported it. It is still not in the commercial stream. It is in the development stage.

2) The three component schema architecture proposed by ANSI/SPARC is as follows:

   External schema: an external view is an abstract representation of some portion of the total database.
   Conceptual schema: the conceptual view is an logical representation of the database in its entirety.
   Internal schema: the internal schema is the database as it is physically stored.

3) There are two basic approaches for database design which are as follows:

   Attribute driven: compile a list of attributes relevant to the application, and normalize the groups of attributes that preserve functional dependencies.

   Entity driven: discover entities that are meaningful to the application, and describe them.

4) The different integrity constraints in RDBMS are as follows:

Primary Key: a primary key is a combination of one or more attributes whose unique value locates each row in a table.

Referential integrity (A foreign key): foreign key is an attribute of one table that references to attribute in another table.

Check Your Progress 2

1) The object model focuses on logical data structures. Each object model consists of many classes, associations, generalizations, and attributes.

2) Object models are effective for communicating with application experts and reaching a consensus about the important aspects of a problem. Object models help developers achieve a consistent, understandable, efficient, and correct database design.

3) Each class-derived table has an ID for the primary key, one or more object IDs form the primary key for association derived tables. An object ID is the equivalent database construct.

4) The advantages of Object IDs are as follows:

   i) IDs are never changing
   ii) IDs are completely independent of changes in data value and physical location
   iii) IDs provide a uniform mechanism for referencing objects.

5) Each object class maps to one or more tables in the database. The objects in a class may be partitioned horizontally and/or vertically.

6) Advantages:

   i) Fewer tables
   ii) Faster performance due to fewer tables to navigate.

Disadvantages:

   i) Less design rigor
   ii) Reduced Extensibility
   iii) Increased complexity

Check Your Progress 3

1) Each object class maps to one, or more tables in the database. The objects in a class may be partitioned horizontally and/or vertically. For instance, if a class has many instances of which, a few are often referenced. In this case, horizontal partitioning may improve efficiency by placing the frequently accessed objects in a table, and the remaining objects in another tables. Similarly, if a class has attributes with different access patterns, it may help to partition the objects vertically.

2) Whenever a ternary relationship is there between the different classes, then each class is mapped to a table with the inclusion of object ID. Apart from this, a new ternary table is also created which has attributes from the different classes involved in the relationship. The attributes will include the object ID of all the classes involved in the relationship, and the attributes of the relation.

3) There are two approaches in which generalizations are mapped to the tables:

   The **superclass** and the **subclass** each map to a table. The identity of an object across a generalization is preserved through the use of a **shared ID.**
   The one superclass table will have all the subclasses attributes up to the super-class level. Each record in the superclass table uses attributes present in one subclass, the other attribute values are null.

UNIT 3 CASE STUDY: INVENTORY CONTROL SYSTEM

Structure **Page Nos.**

3.0	Introduction	40
3.1	Objectives	40
3.2	Class Diagram	40
3.3	Object Diagram	41
3.4	Generalization and Association Diagram	42
3.5	Collaboration Diagram	44
3.6	Activity Diagram and Events	44
3.7	Use Case Diagram	48
3.8	Deployment Diagram	49
3.9	Summary	51

3.0 INTRODUCTION

Inventory control systems are used for managing the stocks of companies and big distribution organisations. In this unit we will discuss about OOM for invention control systems. We will cover class diagram design, object diagram different kind of relationships, which include generalization, association and collaboration. We will also discuss use case diagrams activities and events.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- explain class diagram and object diagram of Inventory control System;
- describe generalization and Specialization of the system;
- describe collaboration diagram of the system;
- explain different activities and events of the systems, and
- explain deployment Diagram.

3.2 CLASS DIAGRAM

You know that a class is represented in a box like figure. Here we are taking the case study of Inventory Control System.

Let us first set an idea of the Inventory Control System. It is the system in which you can manage the stock of the products that a company sells. Basically, this system is stock oriented where it makes sure that the quantity-in-stock does not reach the danger level (Qty-ordered>Qty-in-stock).

In any system when we reach this level, we place a new order. To avoid this situation, when in our system Qty-in-stock reaches a minimum level called the Reorder-level then a new order is placed. Here, in this case study, you will see various diagrams.

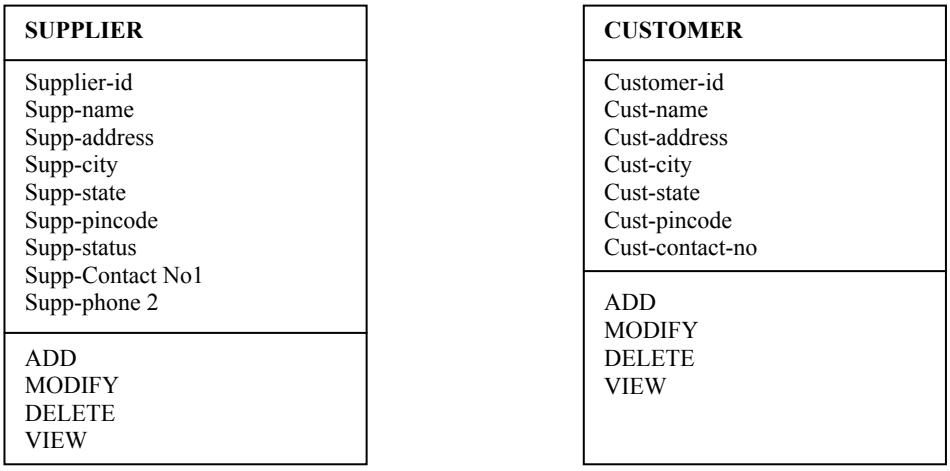


Figure 1: Class diagram represents the static structure of a System

You know that a class is represented in a box like Figure which can have at the most three regions.

- Class Name
- List of Attributes
- List of Operations/Methods

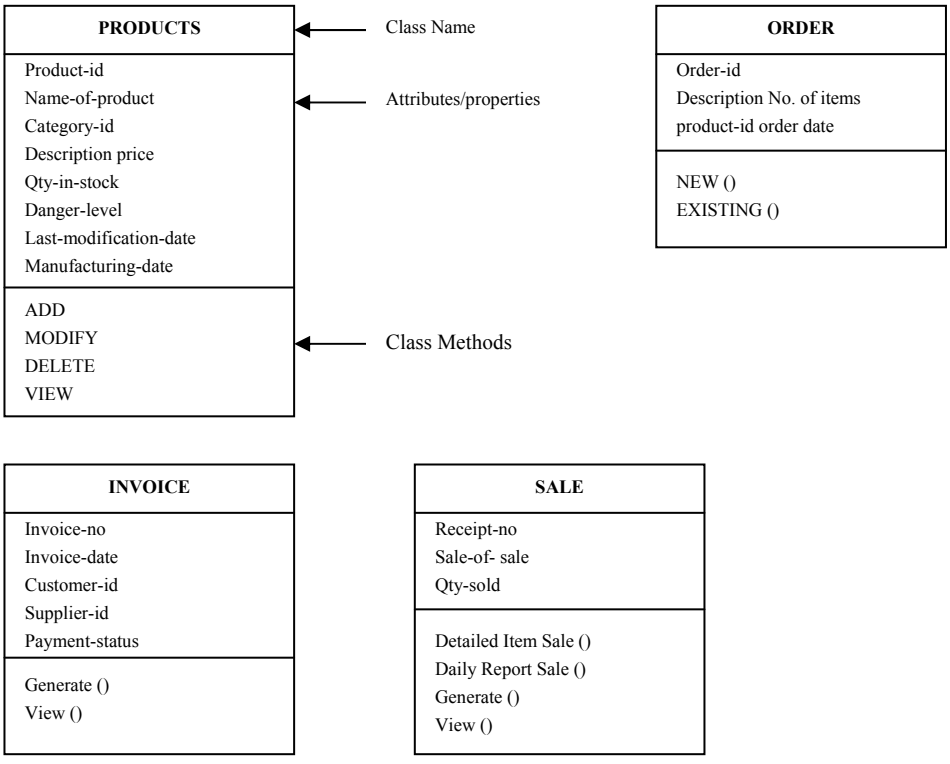


Figure 2: Class diagram

3.3 OBJECT DIAGRAM

Object Diagram is an instance of a class. It describes the static structure of a system at a particular time and are used to test the accuracy of classes.

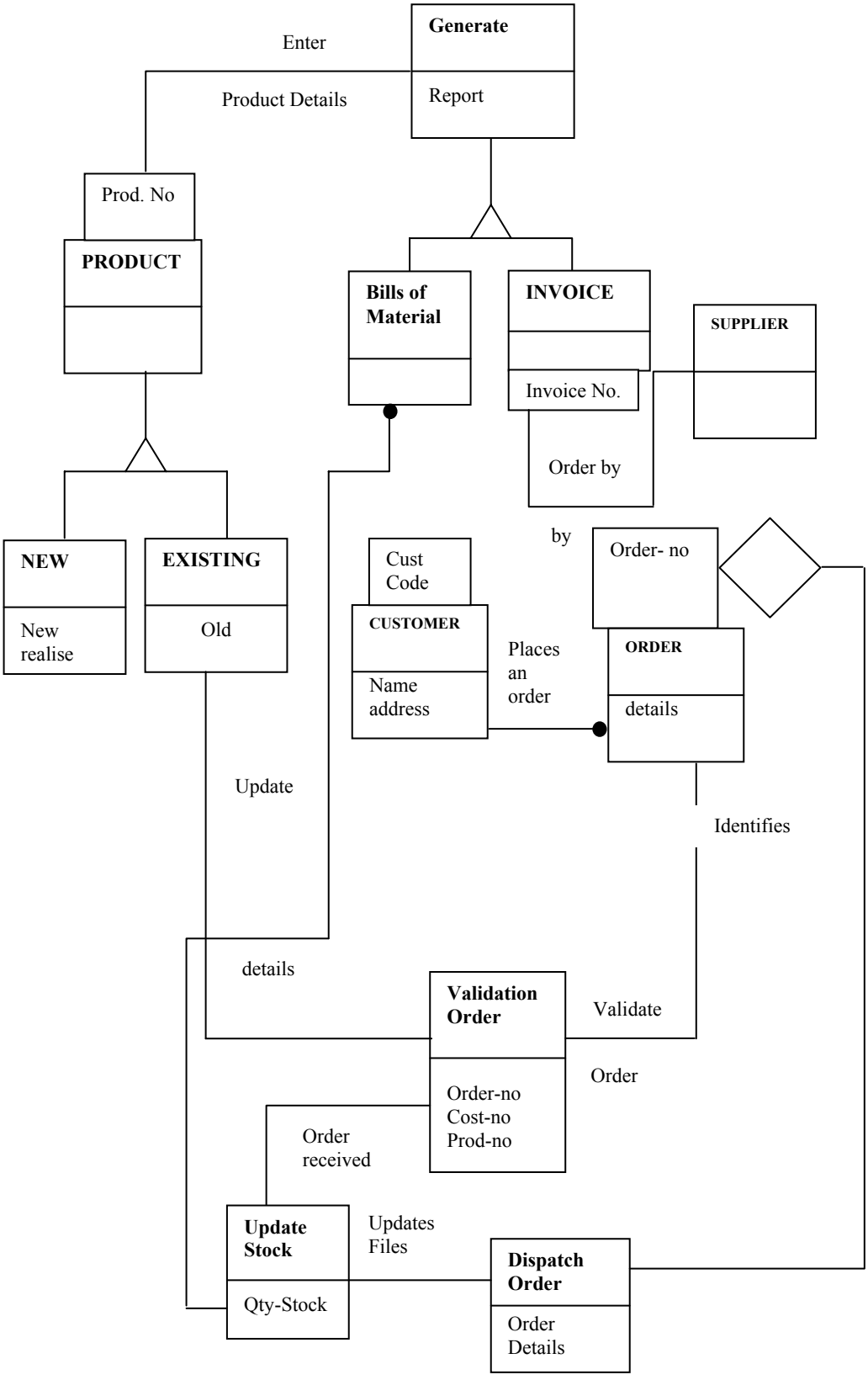


Figure 3: Object Diagram

3.4 GENERALIZATION AND ASSOCIATION DIAGRAM

Generalisation: This is another name for inheritance, or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another.

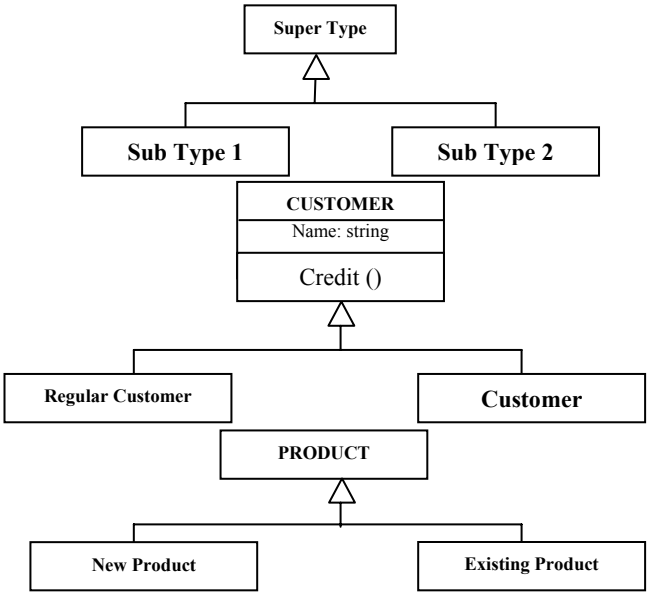


Figure 4: Generalisation of customer and product

Association: This represents static relationship between classes.
Roles represent the way the two classes see each other.

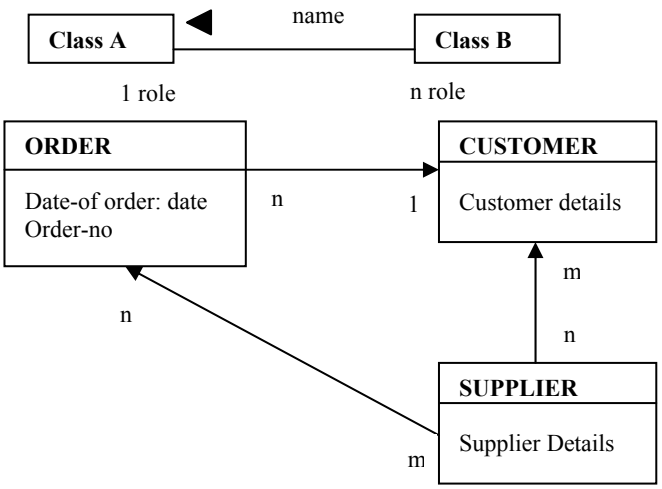


Figure 5: Association of Order and customer

Aggregation: This denotes a strong ownership between class A, the whole, and class B, and its part.

Hollow Diamond Simple Aggregation

Filled Diamond Strong Aggregation

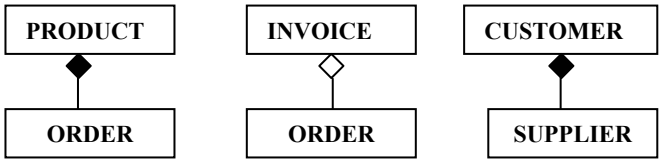


Figure 6: Aggregation

Ternary Association for Customer Supplier

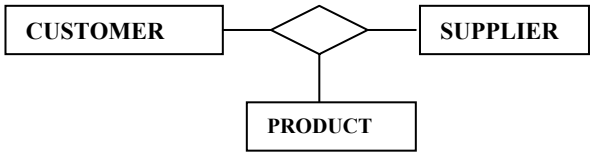


Figure 7: Ternary association

3.5 COLLABORATION DIAGRAM

This represents the interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behaviour of a system.

Representation

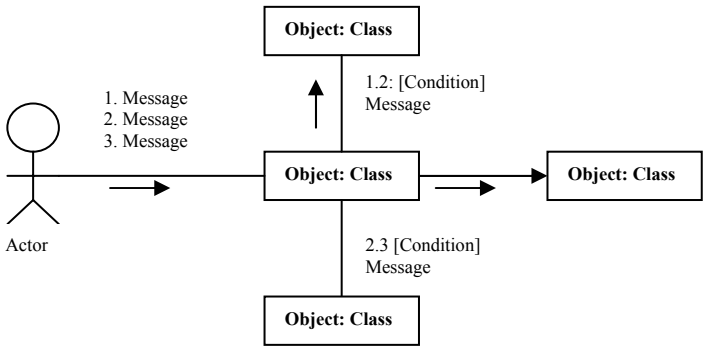


Figure 8 a: Collaboration diagram for inventory control system

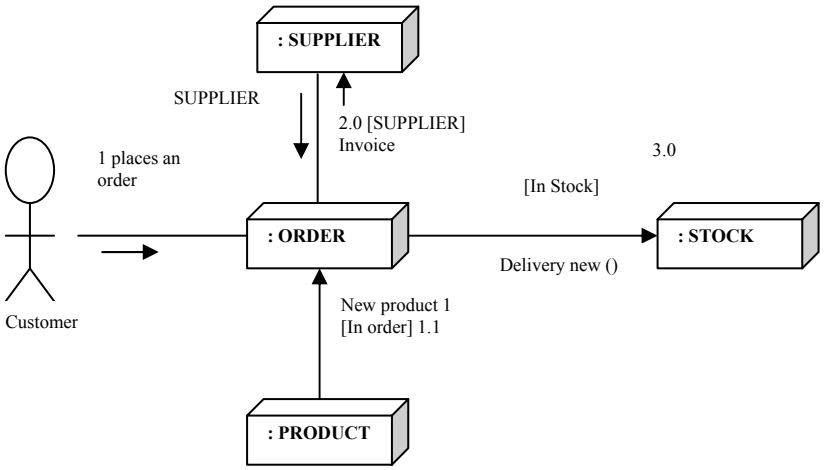


Figure 8 b: Collaboration diagram for the inventory control system

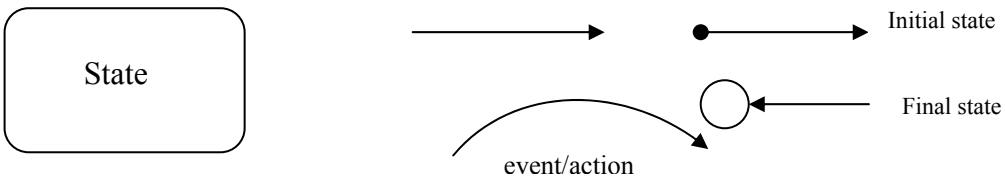
3.6 ACTIVITY DIAGRAM AND EVENTS

STATECHART DIAGRAM

This describes the dynamic behaviour of a system in response to external stimuli

- Basically, states are triggered by specific events.

Representation:



Activity Diagram

This illustrates the dynamic nature of a system by modeling the flow of control from activity to activity, or you can say operation on some class that results in a change in the state of the system.

- Basically, this shows the workflow model, or business process and the internal operation

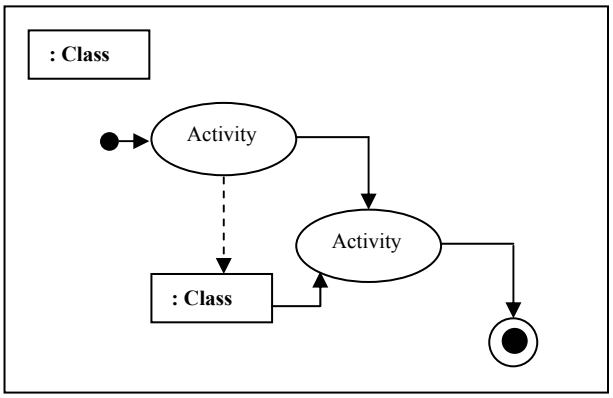


Figure 9: Activity flow

Synchronization and Splitting of Control

- A short heavy bar with two transitions entering it represents a synchronization of Control.
- Splitting of Control that creates multiple states.

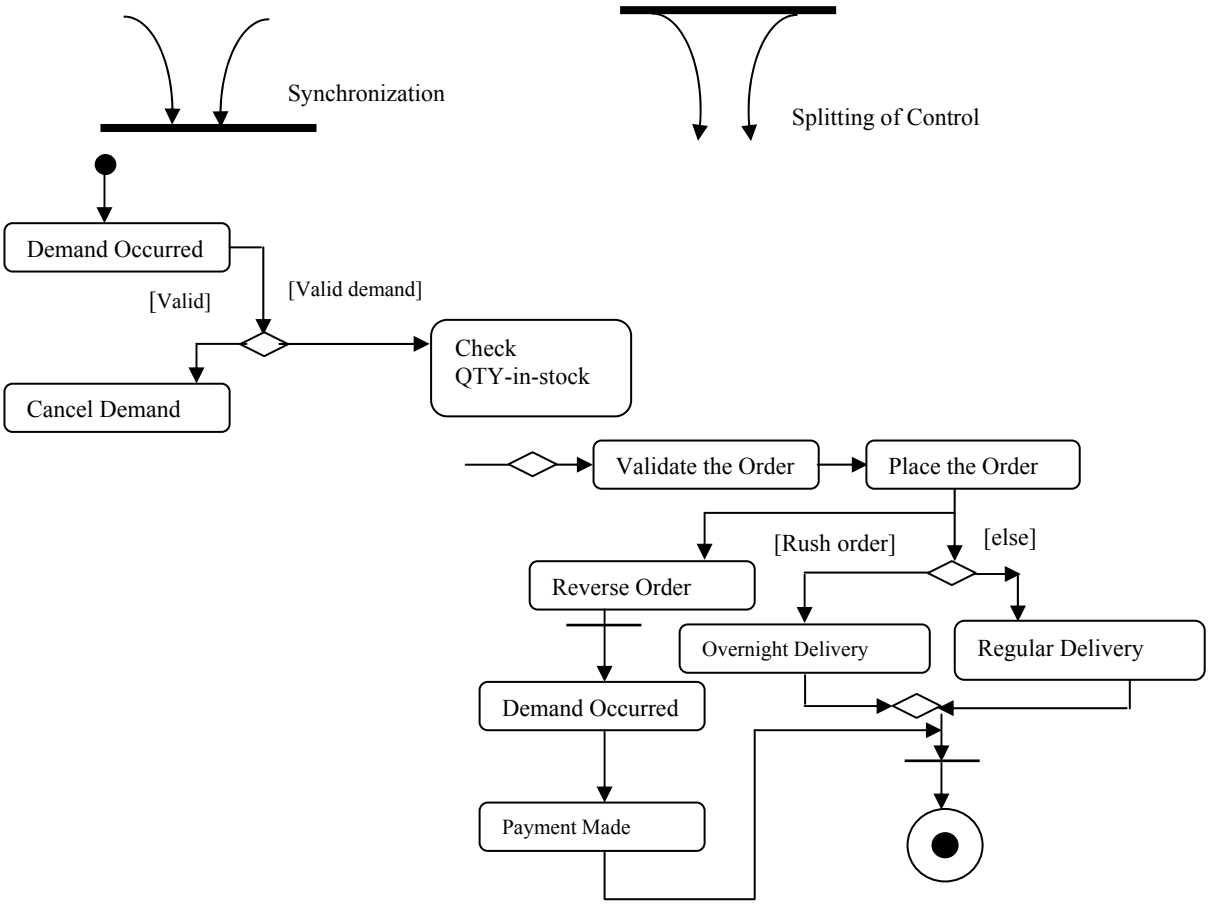


Figure 10: Activity diagram

Events

Actions taken in Inventory Control System:

1) Order is placed by the CUSTOMER
2) Order is received by the SUPPLIER
3) Checking of Quantity-in-Stock and Reorder-level
4) Checking of Inventory Status
5) Generating Bills of Material

6) Generating the INVOICE VOUCHER
7) Updating Inventory Status File

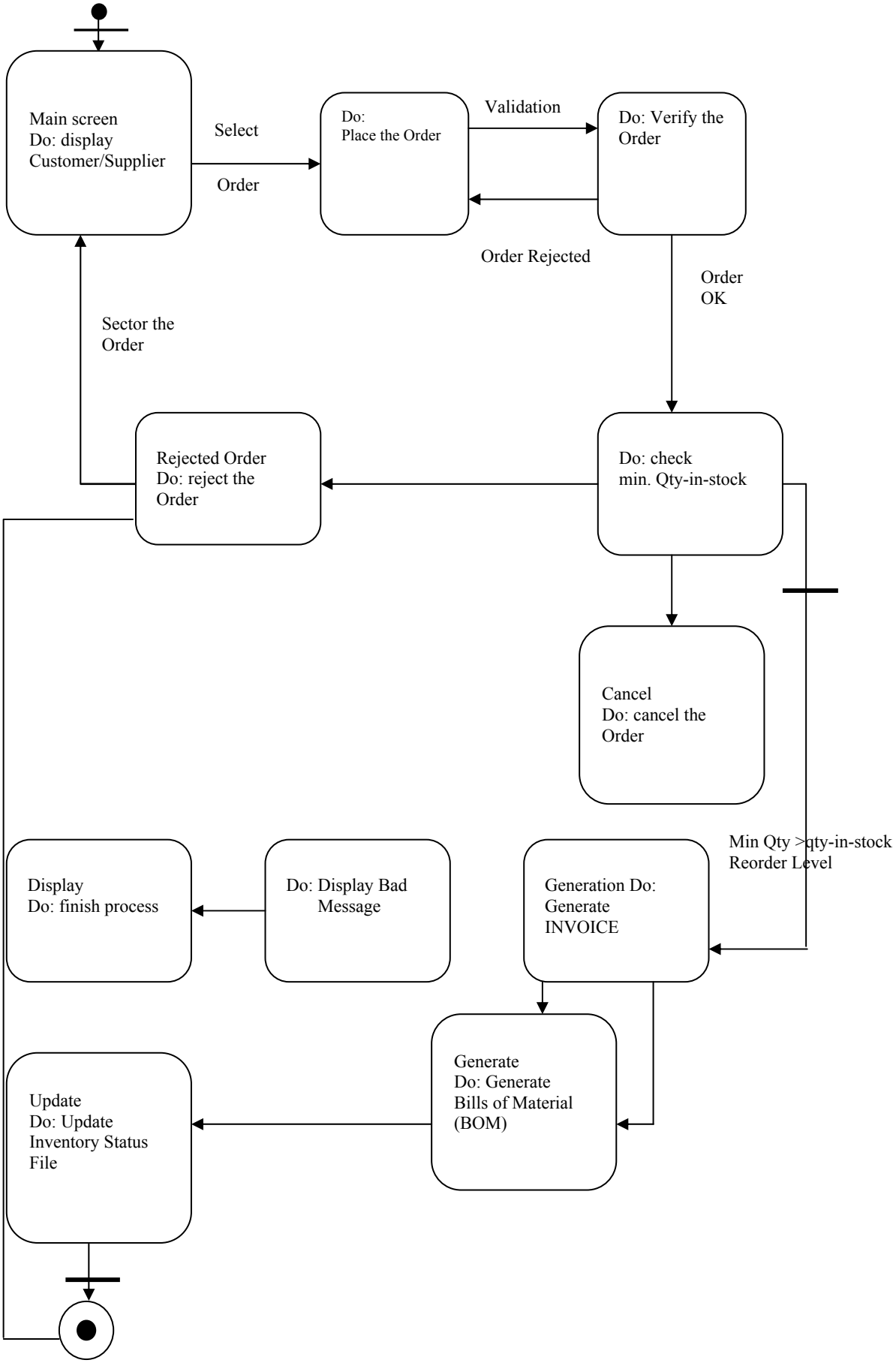


Figure 11: State diagram

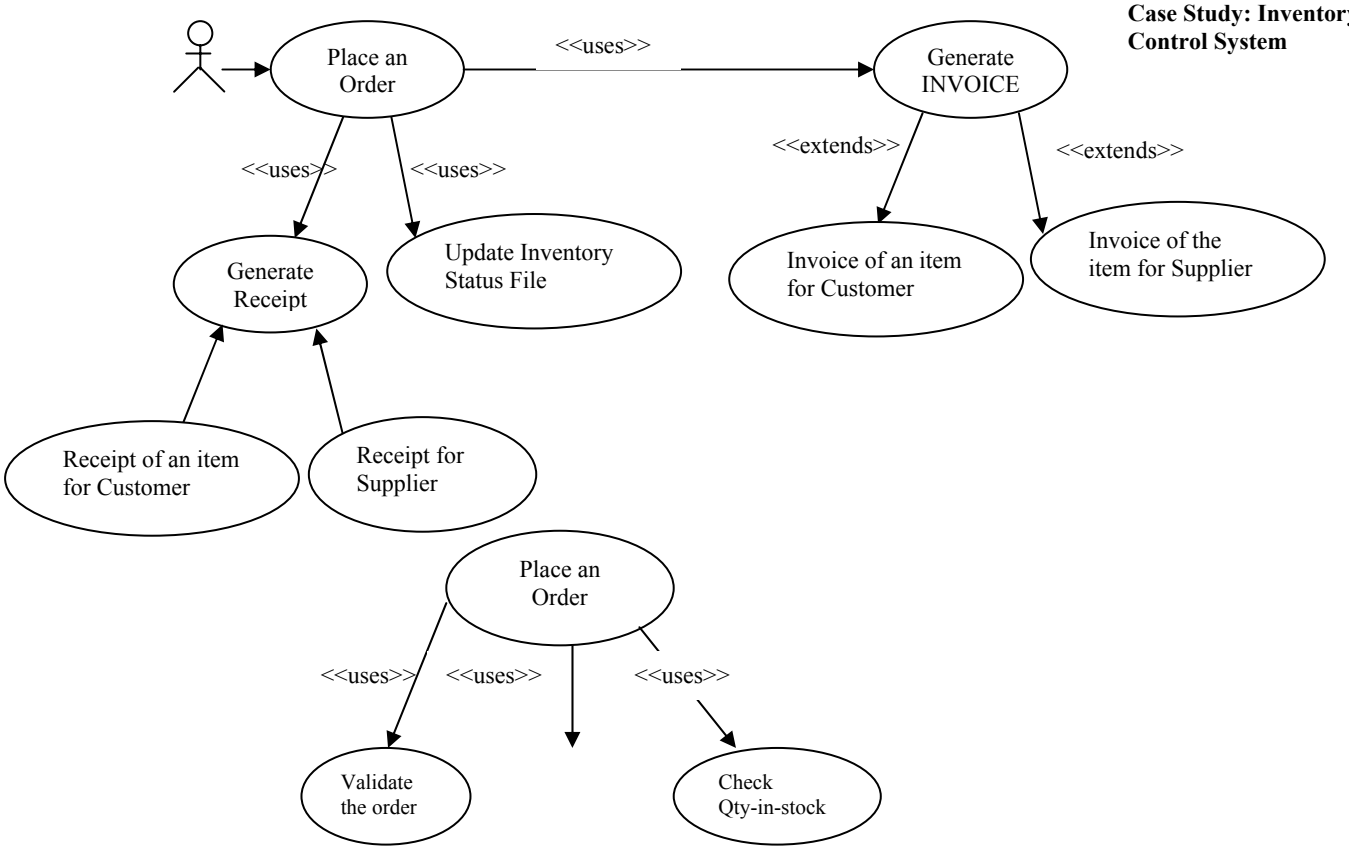


Figure 12: Adding details

Data Flow Diagram for System

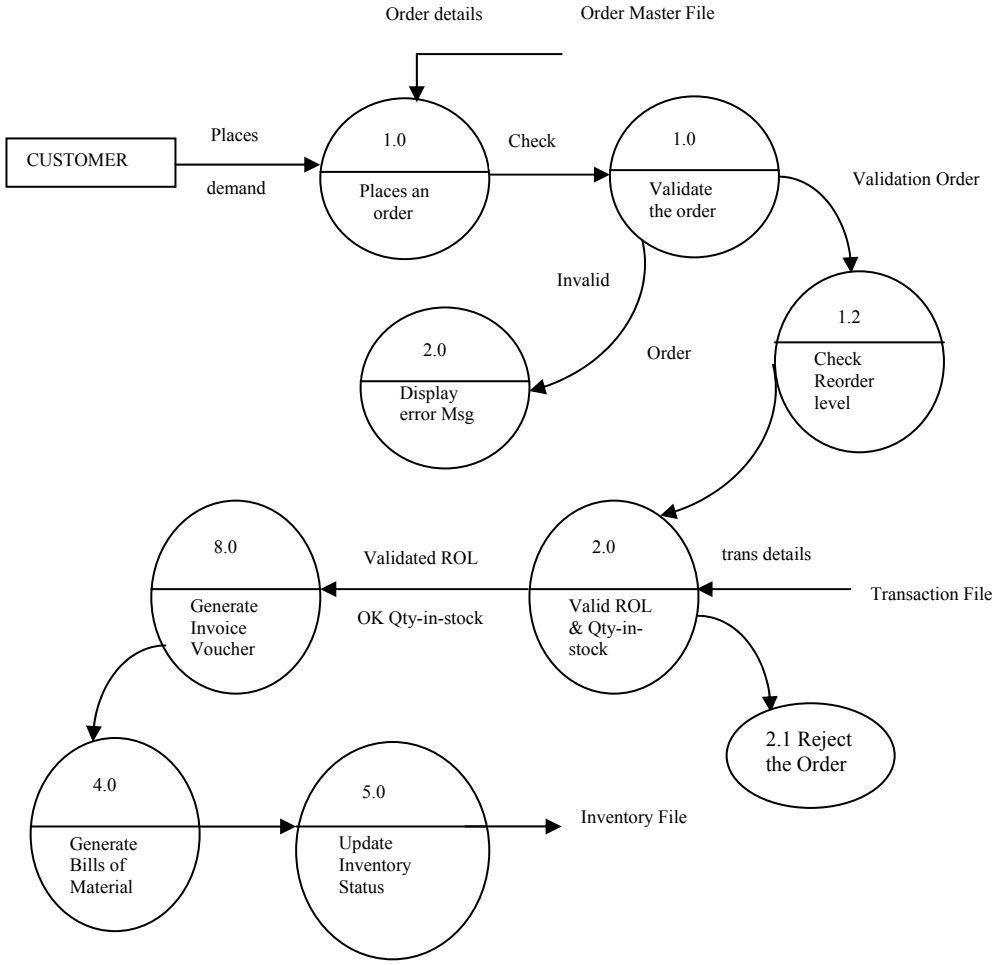


Figure 13: Data flow diagram

3.7 USE CASE DIAGRAM

A “uses” relationship indicates that the use case is needed by another in order to perform a task

- An “extends” relationship indicates alternative options under a certain use case.
- Use case diagrams model the functionality of a system using actors and use cases.
- Use cases are services or functions provided by the system to its users.

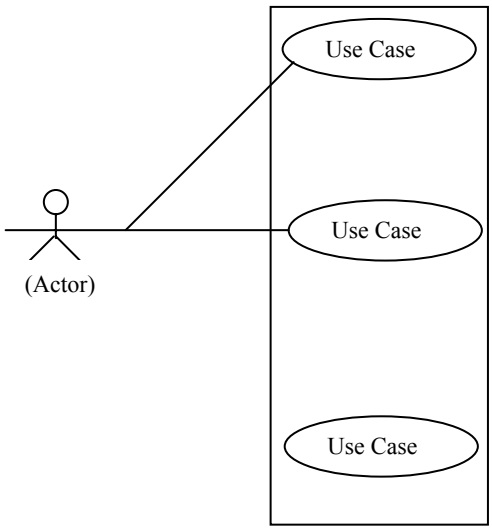


Figure 14: Use Case diagram

For Inventory Control System the initial design is:

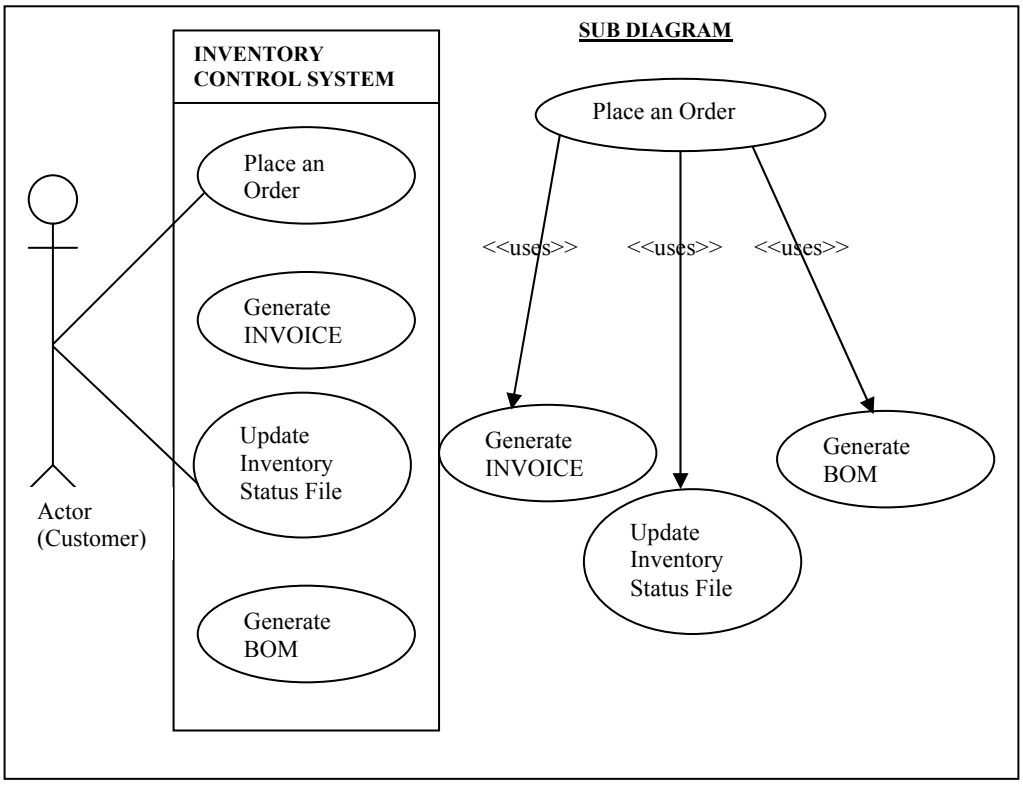
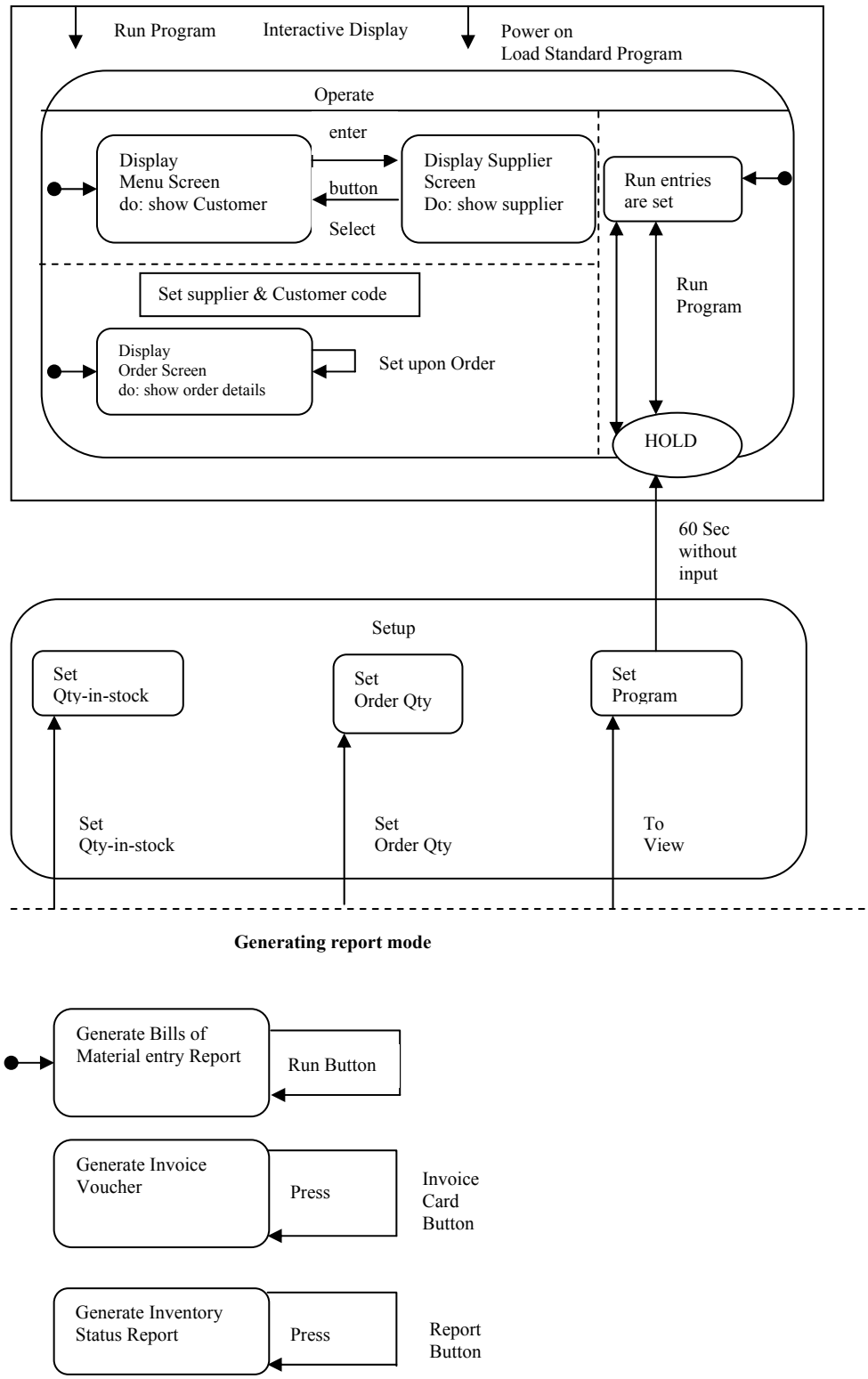


Figure 15: Initial design of the Inventory Control system



Generating report mode

Figure 16: User interface

3.8 DEPLOYMENT DIAGRAM

Deployment diagram: This shows the hardware for your system, the software that is installed on that hardware, and the middleware that is used to connect the machines to one another.

Implementation

- Deployment diagrams depicts the physical resources in a system including nodes, components and connections, where a node is a physical resource that executes code components.

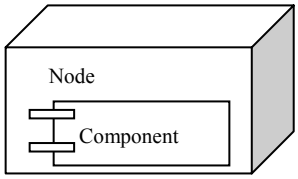
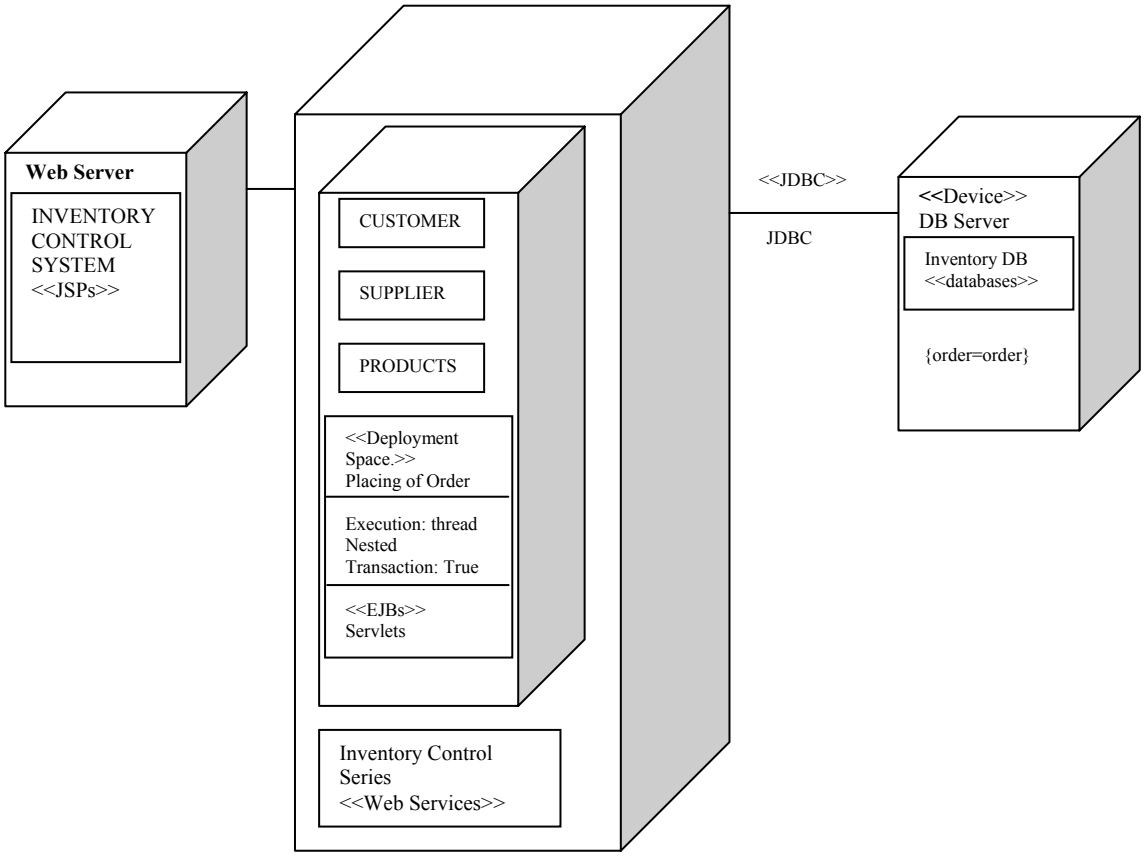


Figure 17: Deployment



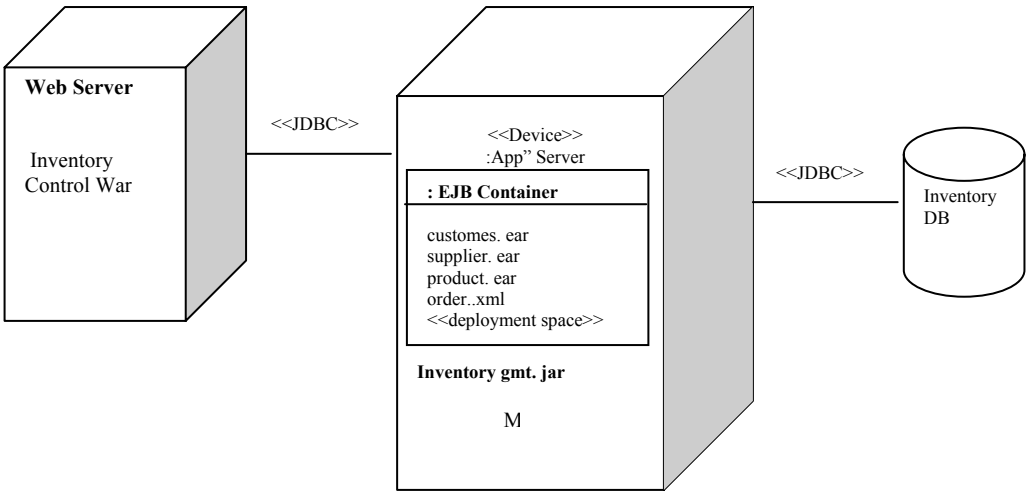


Figure 18: Deployment diagram

3.9 SUMMARY

In this unit different diagram are design to represent OOM of inventory control system. These diagrams are:

- Class diagram
- Object diagram
- Generalization and association
- Collaboration diagram
- Activity diagram
- State diagram
- Dataflow diagram
- Use case diagram
- Use Interface and Deployment Diagram.