Writer: Dr. Rajender Nath

Vetter: Dr. Dharminder Kumar

Chapter 1

Introduction to Object-Oriented Methodology

Structure

1.0 Introduction

Most of the methods used in software development houses are based on a functional and/or data-driven decomposition of the systems. These approaches differ in many ways from the approaches taken by object-oriented methods where data and functions are highly integrated. Object-oriented systems development is a way to develop software by building self-contained modules or objects that can be easily replaced, modified and reused. It depicts the view of real world as a system of cooperative and collaborative objects. In this, software is a collection of discrete objects that encapsulates data and operations performed on that data to model real world objects. A class describes a group of objects having similar structures and similar operations.

Object Oriented Philosophy is very much similar to real world and hence is gaining popularity as the systems here are seen as a set of interacting objects as in the real world. To implement this concept, the process-based structural programming is not used; instead objects are created using data structures. Just as every programming language provides various data types and various variables of that type can be created, similarly, in case of objects certain data types are predefined.

There are several object-oriented development methods around. In this chapter, Object Modeling Technique (OMT) will be discussed. The OMT method given by Rumbaugh et. al. is based on entity/relationship modeling with extensions to modeling classes, inheritance and behavior.

1.1 Objectives

The main objective of this chapter is to introduce the concept of modeling in general and to give an overview of object-oriented methodologies. This chapter introduces OMT methodology and its three basic models viz object

model, dynamic model and functional model. Concepts of objects and classes are discussed in detail along with the notations of OMT methodology. Links and associations are defined and illustrated with lucid examples.

1.2 Introduction to Modeling

1.2.1 What is modeling?

A model is an abstraction of something for the purpose of understanding it before building it. Because, real systems that we want to study are generally very complex. In order to understand the real system, we have to simplify the system. So a model is an abstraction that hides the non-essential characteristics of a system and highlights those characteristics, which are pertinent to understand it. Efraim Turban describes a model as a simplified representation of reality. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus modeling enables us to cope with the complexity of a system.

Most modeling techniques used for analysis and design involve graphic languages. These graphic languages are made up of sets of symbols. As you know one small line is worth thousand words. So, the symbols are used according to certain rules of methodology for communicating the complex relationships of information more clearly than descriptive text.

Modeling is used frequently, during many of the phases of the software life cycle such as analysis, design and implementation. Modeling like any other object-oriented development, is an iterative process. As the model progresses from analysis to implementation, more detail is added to it.

1.2.2 Why do we model?

Before constructing anything, a designer first build a model. The main reasons for constructing models include:

- To test a physical entity before actually building it.
- To set the stage for communication between customers and developers.
- For visualization i.e. for finding alternative representations.
- For reduction of complexity in order to understand it.

1.3 Object Oriented Methodologies

We live in a world of objects. These objects exist in nature, in man-made entities, in business, and in the products that we use. They can be categorized, described, organized, combined, manipulated and created. Therefore, an object-oriented view has come into picture for creation of computer software. An object-oriented approach to the development of software was proposed in late 1960s.

Object Oriented Methodology (OOM) is a new system development approach encouraging and facilitating reuse of software components. With this methodology, a computer system can be developed on a component basis, which enables the effective reuse of existing components and facilitates the sharing of its components by other systems. By using OOM, higher productivity, lower maintenance cost and better quality can be achieved.

OOM requires that object-oriented techniques be used during the analysis, design and implementation of the system. This methodology makes the analyst to determine what the objects of the system are, how they behave over time or in response to events, and what responsibilities and relationships an object has to other objects. Object-oriented analysis has the analyst look at all the objects in a system, their commonalties, difference, and how the system needs to manipulate the objects.

During design, overall architecture of the system is described. During implementation phase, the class objects and the interrelationships of these classes are translated and actually coded using the programming language. The databases are created and the complete system is made operational.

1.3.1 Object Oriented Process

The OOM for building systems takes the objects as the basis. For this, first the system to be developed is observed and analyzed and the requirements are defined. Once this is done, the objects in the required system are identified. For example, in case of a Banking System, a customer is an object, a ledger is an object, passbook is an object and even an account is an object.

OOM is somewhat similar to the traditional approach of system designing, in that it also follows a sequential process of system designing but with a different approach. The basic steps of system designing using OOM may be listed as:

- System Analysis
- System Design
- Object Design
- Implementation

1.3.1.1 System Analysis

As in any other system development model, system analysis is the first phase of OOM too. In this phase, the developer interacts with the user of the system to find out the user requirements and analyses the system to understand the functioning of it.

Based on this system study, the analyst prepares a model of the desired system. This model is purely based on what the system is required to do. At this stage the implementation details are not taken care of. Only the model of the system is prepared based on the idea that the system is made up of a set of interacting objects. The important elements of the system are emphasized.

1.3.1.2 System Design

System Design is the next development stage in OOM where the overall architecture of the desired system is decided. The system is organized as a set of sub systems interacting with each other. While designing the system as a set of interacting subsystems, the analyst takes care of specifications as observed in system analysis as well as what is required out of the new system by the end user.

The system analysis is to perceive the system as a set of interacting objects. A bigger system may also be seen as a set of interacting smaller subsystems that in turn are composed of a set of interacting objects. While designing the system, the stress lies on the objects comprising the system and not on the processes being carried out in the system.

1.3.1.3 Object Design

In this phase, the details of the system analysis and system design are implemented. The Objects identified in the system design phase are designed. Here the implementation of these objects is decided in the form of data structures required and the interrelationships between the objects. For example, we can define a data type called customer and then create and use several objects of this data type. This concept is known as creating a class.

In this phase of the development process, the designer also decides about the classes in the system based on these concepts. He decides on whether the classes need to be created from scratch or any existing classes can be used as it is or new classes can be inherited from them.

1.3.1.4 Implementation

During this phase, the class objects and the interrelationships of these classes are translated and actually coded by using an object-oriented programming language. The required databases are created and the complete system is transformed into operational one.

1.3.2 Advantages of Object Oriented Methodology

- As compared to the conventional system development techniques, OOM provides many benefits.
- The systems designed using OOM are closer to the real world as the real world functioning of the system is directly mapped into the system designed using OOM. Because of this, it becomes easier to produce and understand designs.
- The objects in the system are immune to requirement changes because of data hiding and encapsulation features of object-orientation. Here, encapsulation we mean a technique that allows the programmer to hide the internal functioning of the objects from the users of the objects. Encapsulation separates the internal functioning of the object from the external functioning thus providing the user flexibility to change the external behavior of the object making the programmer code safe against the changes made by the user.
- OOM designs encourage more reusability. The classes once defined can easily be used by other applications. This is achieved by defining classes and putting them into a library of classes where all the classes are maintained for future use. Whenever a new class is needed the programmer first looks into the library of classes and if it is available, it can be used as it is or with some modification. This reduces the development cost & time and increases quality.
- Another way of reusability is provided by the inheritance feature of the object-orientation. The concept of inheritance allows the programmer to use the existing classes in new applications i.e. by making small additions to the existing classes can quickly create new classes. This provides all the benefits of reusability discussed in the previous point.
- As the programmer has to spend less time and effort so he can utilize saved time (due to the reusability feature of the OOM) in concentrating on other aspects of the system.

- OOM approach is more natural as it deals with the real world objects. So, it provides nice structures for thinking and abstracting and leads to modular design.

Many OOMs have been developed since its inception. Some of the popular object oriented methodologies are listed below:

- Booch Methodology [1994]: He developed the Object Oriented Analysis and Object Oriented Design (OOA/OOD) concepts.
- RDD Methodology [1990]: Wirfs-Brock, Wilkerson, and Wiener developed Responsibility Driven Design (RDD) methodology.
- OMT methodology [1991]: James Rumbaugh led a team at research labs of General Electric to develop the Object Modeling Technique (OMT).
- OOSE [1994]: Ivar Jacobson developed the Object Oriented Software Engineering (OOSE).

Other Object-Oriented methodologies that have been around in the world are:

- Berand [Berand 93]
- BON [Nerson 92]
- Coad/Yourdon [Coad 1991]
- Embley [Embley 1993]
- EVB [Jurik 1992]
- ROOM [Selic 1994]
- FUSION [Coleman 1994]
- HOOD [HOOD 89]
- Shlaer and Mellor (Shlaer 1992]

Discussion of these methodologies is beyond the scope of this course material as these are not the part of your course curriculum. Object Modeling Technique (OMT) methodology [1991] developed by James Rumbaugh et. al.

is the part of your syllabus and will be discussed in detail in rest of the chapters.

1.4 OMT Methodology

OMT is an object-oriented software development methodology given by James Rumbaugh et.al. This methodology describes a method for analysis, design and implementation of a system using object-oriented technique. It is a fast, intuitive approach for identifying and modeling all the objects making a system. The OMT consists of three related but different viewpoints each capturing important aspects of the system i.e. the static, dynamic and functional behaviors of the system. These are described by object model, dynamic model and functional model of the OMT.

The object model describes the static, structural and data aspects of a system. The dynamic model describes the temporal, behavioral and control aspects of a system. The functional model describes the transformational and functional aspects of a system. So every system has these three aspects. Each model describes one aspect of the system but contains references to the other models.

The entire OMT software development process has four phases: analysis, system design, object design, and implementation of the software. Most of the modeling is performed in the analysis phase. In this phase, three basic models - Object Model, Dynamic Model and Functional Model are developed. While the Object Model is most important of all as it describes the basic element of the system, the objects, all the three models together describe the complete functional system.

Object Model describes the objects in a system and their interrelationships. This model observes all the objects as static and does not pay any attention to their dynamic nature. Dynamic Model depicts the dynamic aspects of the system. It portrays the changes occurring in the states of various objects with the events that might occur in the system. Functional Model basically

describes the data transformations of the system. This describes the flow of data and the changes that occur to the data throughout the system.

In this chapter, we will discuss the object model in detail. Remaining models will be described in the ensuing chapters.

1.4.1 Object Model

The object model describes the structure of the objects in the system - their identity, their relationships to other objects, their attributes, and their operations. The object model depicts the primary view of how the real world in which the system interacts is divided and the overall decomposition of the system. The object model provides the framework into which the other models are placed.

The object model is represented graphically with an object diagram. The object diagram contains classes interconnected by association lines. Each class represents a set of individual objects. The association lines establish relationships among classes. Each association line represents a set of links from the object of one class to the object of another class.

Now we discuss the concepts of object, class and relationships between objects and/or classes. We also introduce the notations and symbols of OMT used for object, class and relationships.

1.4.1.1 Object and Class

A class describes a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined. A class defines the basic attributes and the operations of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created, instances of the class are to be created as per the requirement of the case.

Classes are built on the basis of abstraction, where a set of similar objects is observed and their common characteristics are listed. Of all these, the characteristics of concern to the system under observation are taken and the class definition is made. The attributes of no concern to the system are left out. This is known as abstraction. So, the abstraction is the process of hiding superfluous details and highlighting pertinent details in respect to the system under development.

It should be noted that the abstraction of an object varies according to its application. For instance, while defining a pen class for a stationery shop, the attributes of concern might be the pen color, ink color, pen type etc., whereas a pen class for a manufacturing firm would be containing the other dimensions of the pen like its diameter, its shape and size etc.

Each application-domain concept from the real world that is important to the application should be modeled as an object class. Classes are arranged into hierarchies sharing common structure and behavior and are associated with other classes. This gives rise to the concept of inheritance.

Through inheritance, a new type of class can be defined using a similar existing class with a few new features. For instance, a class vehicle can be defined with the basic functionality of any vehicle and a new class called car can be derived out of it with a few modifications. This would save the developers time and effort as the classes already existing are reused without much change.

In OMT, classes are represented by a rectangular box which may be divided into three parts as shown in Figure 1.1. The top part contains the name of the class written in bold, middle part contains a list of attributes and bottom part contains a list of operations.

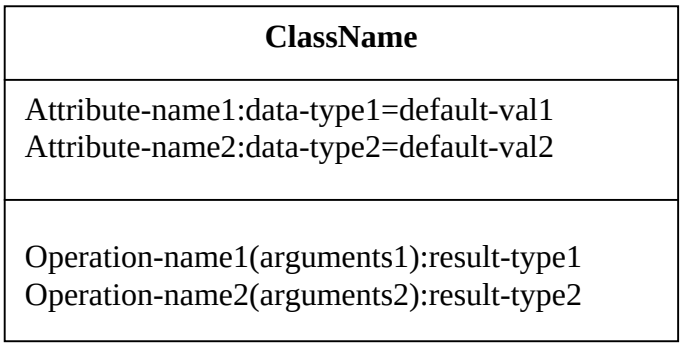


Figure 1.1

An attribute is a data value held by objects in a class. Each attribute has a value for each object instance. This value should be a pure data value, not an object. Attributes are listed in the second part of the class box. Attributes may or may not be shown; it depends on the level of detail desired. Each attribute name may be followed by the optional details such as type and default value. An object model should generally distinguish independent base attributes from dependent derived attributes. A derived attribute is that which is derived from other attributes. For example, age is a derived attribute, as it can be derived from date-of-birth and current-date attributes.

An operation is a function or transformation that may be applied to or by objects in a class. Operations are listed in the third part of the class box. Operations may or may not be shown; it depends on the level of detail desired. Each operation may be followed by optional details such as argument list and result type. The name and type of each argument may be given. An empty argument list in parentheses shows explicitly that there are no arguments. All objects in a class share the same operations. Each operation has a target object as an implicit argument. An operation may have arguments in addition to its target object, which parameterize the operation. The behavior of the operation depends on the class of its target.

An operation may be polymorphic in nature. A polymorphic operation means that the same operation takes on different forms in different/same classes. Overloading of operators, overloading of functions and overriding of functions

provided by object-oriented programming languages are all examples of polymorphic operations. A method is the implementation of an operation for a class. The method depends only on the class of the target object.

Now, we present a number of examples of classes to clarify what we have discussed above. Figure 1.2 shows the class Book. Attributes of Book are title, author, publisher along with their data types. Operations on Book are open(), close(), read().

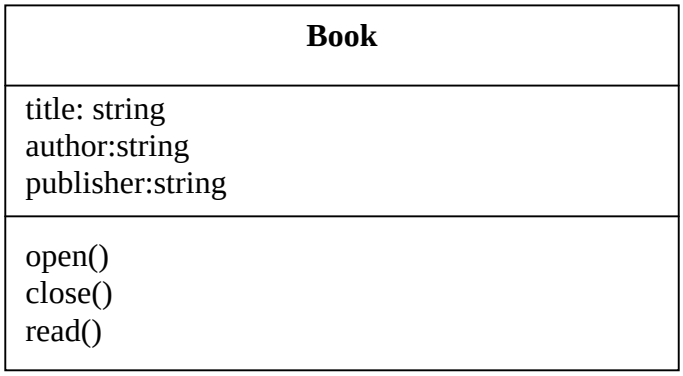


Figure 1.2

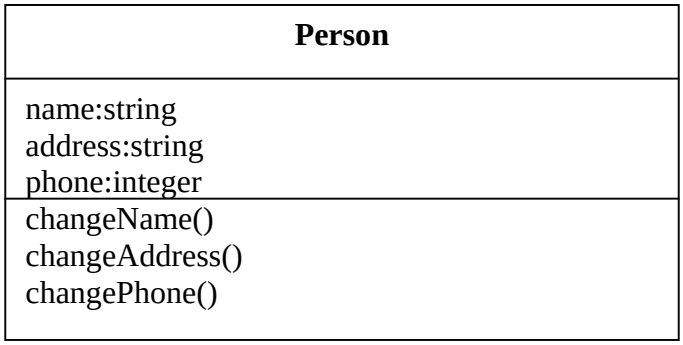


Figure 1.3

Figure 1.3 shows the class Person with name, address & phone along with their data types as attributes and changeName(), changeAddress() & changePhone() as operations.

Figure 1.4 shows the class Window with xMin, yMin, xMax, yMax along with their data types as attributes and draw(), cut(), erase() & move() as operations.

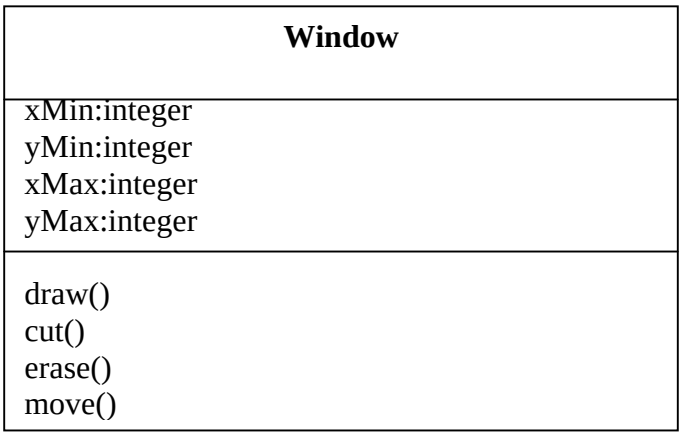


Figure 1.4

Now, let us formally define an object. An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. An object has the following four main characteristics:

- Unique identification
- Set of attributes
- Set of states
- Set of operations (behavior)

Unique identification, we mean every object has a unique name by which it is identified in the system. Set of attributes, we mean every object has a set of properties in which we are interested in. Set of states we mean values of attributes of an object constitute the state of the object. Every object will have a number of states but at a given time it can be in one of those states. Set of operations we mean externally visible actions an object can perform. When an operation is performed, the state of the object may change.

In other words, an object is an instance of an object class. Figure 1.2 (rounded box) represents an object instance in OMT. Object instance is a particular object from an object class. The box may/may not be divided in particular regions. Object instances can be used in instance diagrams, which are useful for documenting test cases and discussing examples.

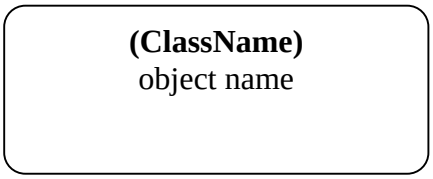


Figure 1.5

Now we give three examples of objects of class Book shown in Figures 1.2, which are shown in Figure 1.6 (a), (b) & (c) respectively.

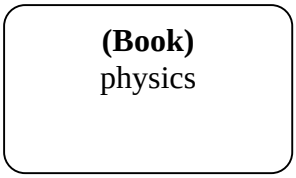


Figure 1.6(a)

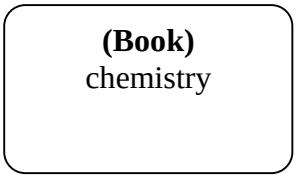


Figure 1.6(b)

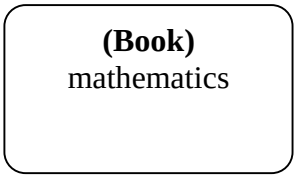


Figure 1.6(c)

Here we give three examples of objects of class Person shown in Figures 1.3, which are shown in Figure 1.7 (a), (b) & (c) respectively.

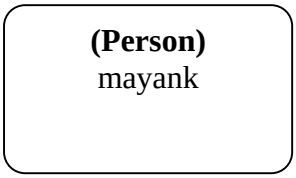


Figure 1.7(a)

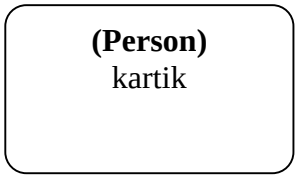


Figure 1.7(b)

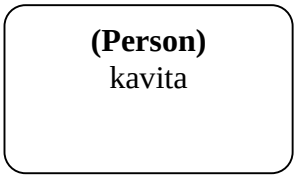
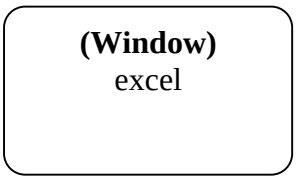
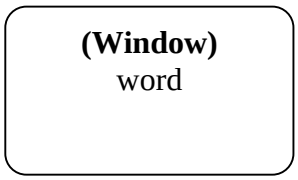


Figure 1.7(c)

Next we give three examples of objects of class Window shown in Figures 1.4, which are shown in Figure 1.8 (a), (b) & (c) respectively.





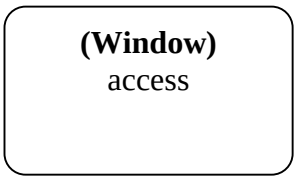


Figure 1.8(a) **Figure 1.8(b)** **Figure 1.8(c)**

Let us introduce the concept of derived object. It is defined as a function of one or more objects. It is completely determined by the other objects. Derived object is redundant but can be included in the object model.

After having discussed the concepts of objects and their classes, let us discuss relationships between objects and between classes.

1.4.1.2 Links and Associations

A link is a physical or conceptual connection between object instances. In OMT, link is represented by a line labeled with its name as shown in Figure 1.9.

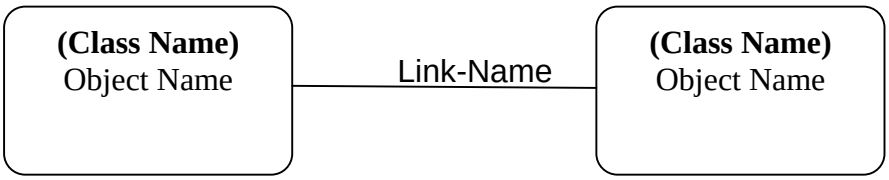


Figure 1.9

For example Mayank flies Jaguar. So ‘flies’ is a link between object instance Mayank of class Person and object instance Jaguar of class Airplane as shown in Figure 1.10. Kartik reads Physics. Here reads is a link between object instance Kartik of class Person and object instance Physics of class Book as shown in Figure 1.11.

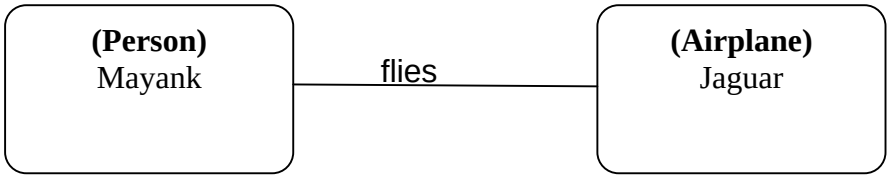


Figure 1.10

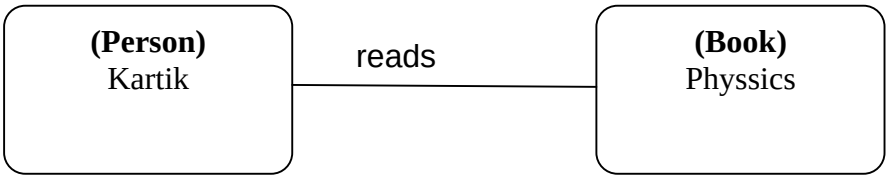


Figure 1.11

An association describes a group of links with common structure and common semantics between two or more classes. Association is represented by a line labeled with the association name in italics as shown in Figure 1.12.

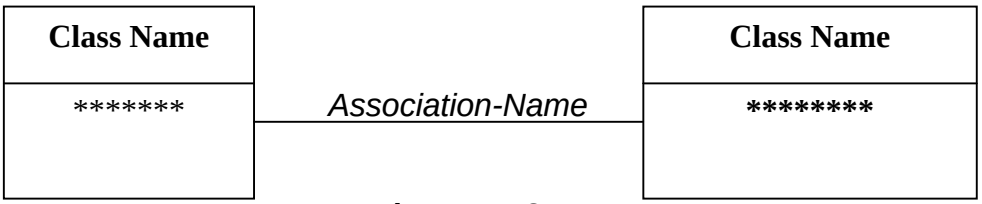


Figure 1.12

Association names are optional. If the association is given a name, it should be written above the line. Association names are italized. In case of a binary association, the name reads in a particular direction (i.e. left to right), but the binary association can be traversed in either direction. For example, a pilot flies an airplane or an airplane is flown by a pilot. All the links in an association connect objects from the same classes. Associations are bi-directional in nature.

Multiplicity: It specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects.

There are special line terminators to indicate certain common multiplicity values. A solid ball is the symbol for "many", meaning zero, one or more. A hollow ball indicates "optional", meaning zero or one. The multiplicity is indicated with special symbols at the ends of association lines. In the most general case, multiplicity can be specified with a number or set of intervals. If no multiplicity symbol is specified that means a one-to-one association. The rules of multiplicity are summarized below:

- Line without any ball indicates one-to-one association.
- Hollow ball indicates zero or one.
- Solid ball indicates zero, one or more.
- Numbers written on solid ball such as 1,2,6 indicates 1 or 2 or 6.
- Numbers written on solid ball such as 1+ indicates 1 or more, 2+ indicates 2 or more etc.

An association can be unary, binary, ternary or n-ary. Unary association is between the same class as shown in Figure 1.13.

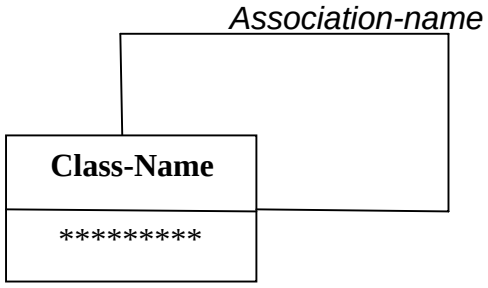


Figure 1.13

Example of unary association is Person teaches Person as shown in Figure 1.14. Other examples of unary association can be Person marries a Person, Person manages Person etc.

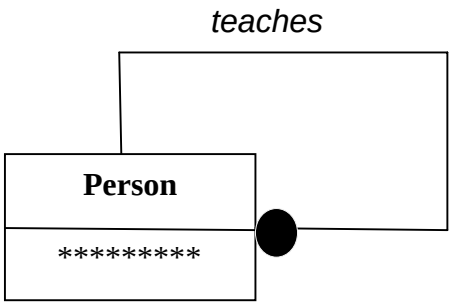


Figure 1.14

A binary association is an association between two classes as shown in Figure 1.15.

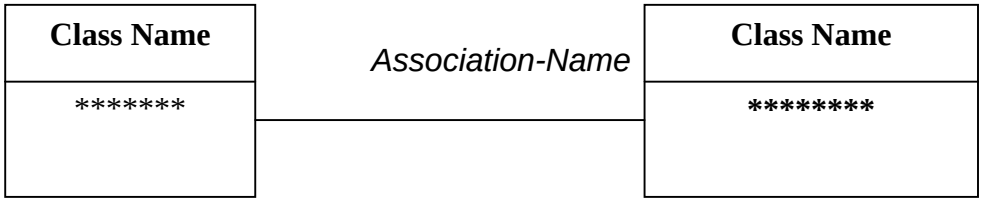


Figure 1.15

Example of a binary association is “Person sits on a Chair”. One person can sit at one chair. So multiplicity of this association is one-to-one as shown in Figure 1.16.

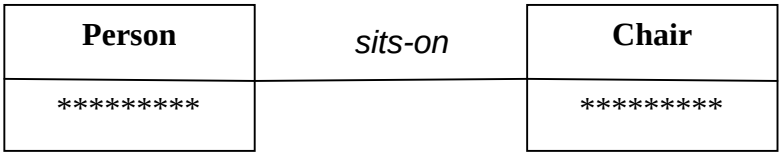


Figure 1.16

Example of a binary association is “Supplier supplies Parts”. One supplier can supply many parts or one part can be supplied by many suppliers. So multiplicity of this association is many-to-many as shown in Figure 1.17.

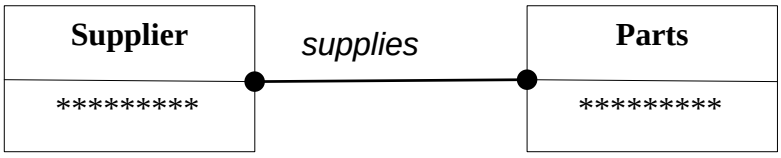


Figure 1.17

Another example of binary association is “Person possesses a Passport”. Either a person can have one passport or no passport but one passport can be with one person. So multiplicity of this association is one-to-optional as shown in Figure 1.18.

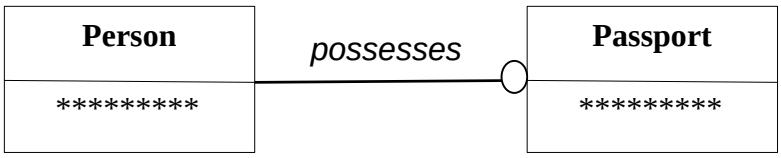


Figure 1.18

Another example of binary association is “Company employs Person”. One company can employ zero, one or more persons but one person can be employed in one company only (assume). So multiplicity of this association is one-to-many as shown in Figure 1.19.

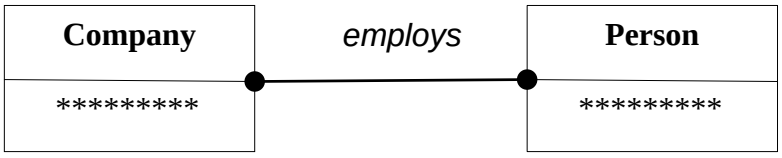


Figure 1.19

Ternary association is an association among three classes. On the same line, n-ary association is an association among n classes. The OMT symbol for ternary and n-ary associations is a diamond with lines connecting to related classes as shown in Figure 1.20. A name for the association is optional and is

written next to the diamond. An n-ary associations cannot be subdivided into binary associations without losing information.

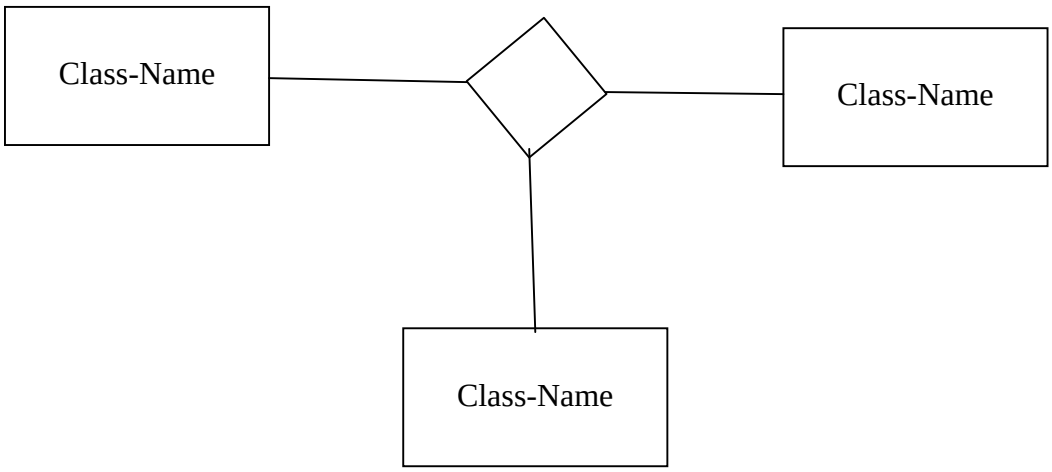


Figure 1.20

Now let us consider the example of a ternary association. Programmers develop Projects in (programming) Languages. One programmer can be engaged in zero, one or more projects and can know zero, one or languages. Similarly, one project can be developed by one or more programmers and in one or more languages. So this association along with its multiplicity is shown in Figure 1.21.

Other examples of ternary and higher order associations are “Teacher teaches Students in a Classroom”, “Doctor diagnoses Patient in Room at a given Schedule” etc.

There are many other concepts related to associations such as link attribute, link class, role names, qualifiers etc. which will be discussed in the next chapter.

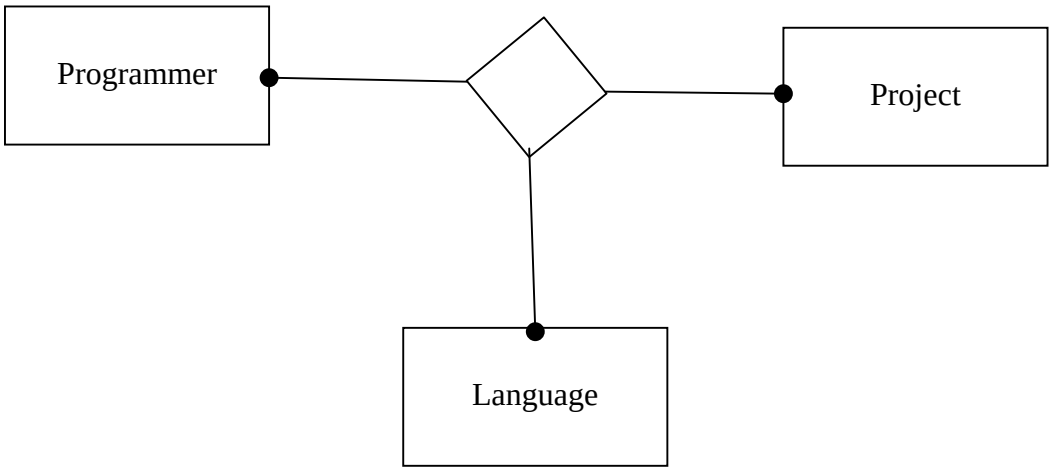


Figure 1.21

1.5 Summary

In this chapter you have learnt the concepts of model, object modeling, OMT methodology, object model, dynamic model, functional model, class, object, link and association, which are summarized below:

- A model is a simplified representation of reality. It provides a means for conceptualization and communication of ideas in a precise and unambiguous form.
- Object Oriented Methodology (OOM) is a new system development approach encouraging and facilitating reuse of software components. By using OOM, higher productivity, lower maintenance cost and better quality can be achieved.
- The basic steps of system designing using OOM include analysis, system design, object design and implementation.
- The object model describes the static, structural and data aspects of a system.
- The dynamic model describes the temporal, behavioral and control aspects of a system.

- The functional model describes the transformational and functional aspects of a system.
- Every system has all the three models. Each model describes one aspect of the system but at the same time contains references to the other models.
- An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand.
- An object has the following four main characteristics - unique identification, set of attributes, set of states, and set of operations (behavior).
- Class is a template where certain basic characteristics of a set of objects are defined. A class defines the basic attributes and the operations of the objects of that type.
- Defining a class does not define any object, but it only creates a template. For objects to be actually created, instances of the class are to be created as per the requirement of the case.
- A link is a physical or conceptual connection between object instances. In OMT, link is represented by a line labeled with its name.
- An association describes a group of links with common structure and common semantics between two or more classes. Association is represented by a line labeled with the association name in italics.
- An association may be unary, binary, ternary or n-ary.
- Multiplicity specifies how many instances of one class may relate to a single instance of an associated class.

1.6 Suggested Readings/Reference Materials

1. Object-Oriented Modeling and Design with UML, M. Blaha, J. Rumbaugh, Pearson Education-2007
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson-2007
3. Object Oriented Analysis & Design, Grady Booch, Addison Wesley-1994

4. Timothy C. Lethbridge, Robert Laganiere, Object Oriented Software Engineering, TMH, 2004

1.7 Self-Assessment Questions

1. What is model? Why do we model?
2. What is object oriented methodology? What are the advantages of object oriented methodology?
3. What is object oriented process? Discuss the steps of object oriented process.
4. What is Object Modeling Technique (OMT)? What are the phases of OMT? Discuss each in brief.
5. What are the three models involved in OMT? Define each one of them.
6. What is object? Discuss the main characteristics of the object with examples from the real world.
7. List 20 objects from the real world around you. Write their attributes and operations.
8. What is class? What is OMT notation for a class Discuss the relationships between class and object.
9. List 10 classes from the real world and define them.
10. Define link and give five different examples of links and represent them using OMT notations.
11. Define association. Discuss the characteristics and OMT notations of the association.
12. Give five examples of each unary, binary and ternary associations from the real world.
13. What do you mean by multiplicity of the association? Discuss different types of multiplicity by giving suitable examples.

Writer: Dr. Rajender Nath

Vetter: Dr. Dharminder Kumar

Chapter 2

Advanced Object Modeling

Structure

2.4 Suggested Readings/Reference Materials

2.5 Self-Assessment Questions

2.0 Introduction

In this chapter, we will present the advanced features of object modeling. In the last chapter, we discussed association, which is a relationship between classes. There can be some attributes, which cannot be associated with either of the two classes related by an association. Such attributes are called as link attributes. A binary association can have two roles, which may be written at the ends of the association.

Other relationships between classes are aggregation and inheritance. Aggregation specifies one object may be composed of other objects. It is a part-whole relationship. Inheritance is a way to form new classes using classes that have already been defined. Inheritance is intended to help reuse existing code with little or no modification.

2.1 Objectives

In the last chapter, a foundation of object model was created. The objective of this chapter is to introduce advanced concepts of object modeling such as link attributes, link class, role names, ordering qualifiers etc. you will also learn other relationships between classes such as aggregation and inheritance. At the end of this chapter, you will be able to answer what is aggregation? What is recursive aggregation? What are different types of inheritance? You will also learn the concepts of metadata, candidate keys and grouping constructs.

2.2 Presentation of Contents

2.2.1 Link attributes

Some times, an attribute(s) cannot be associated with either of the two classes associated by the association. In such cases, the attribute(s) is associated with the association and is called as link attribute. It is a property of the links in an association.

The OMT notation for a link attribute is a box attached to the association by a loop, see Figure 2.1.

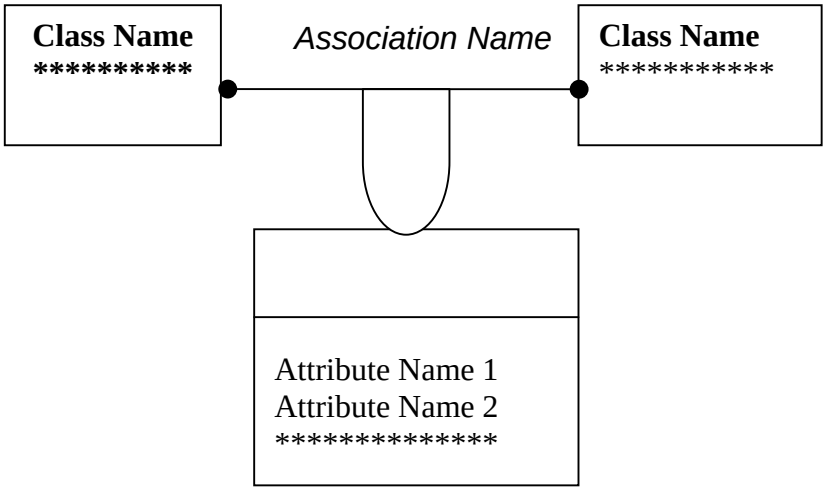


Figure 2.1

One or more link attributes may appear in the second region of the box. Sometime it is possible, for one-to-one and one-to-may associations, to fold link attributes into the class opposite to the "one" side. But as a rule, link attributes should not be folded into a class because future flexibility is reduced if the multiplicity of the association changes.

Consider an example as shown in Figure 2.2. File is accessed by a User. So, the classes File and User are related by association the association named “AccessedBy”. Many users can access one file and one user can access many files. So, multiplicity of the association is many-to-many. Now, the attribute “access-permission” cannot be associated with either File class or with User class. This attribute can be associated with the link as shown in Figure 2.2. Hence, access-permission is link attribute.

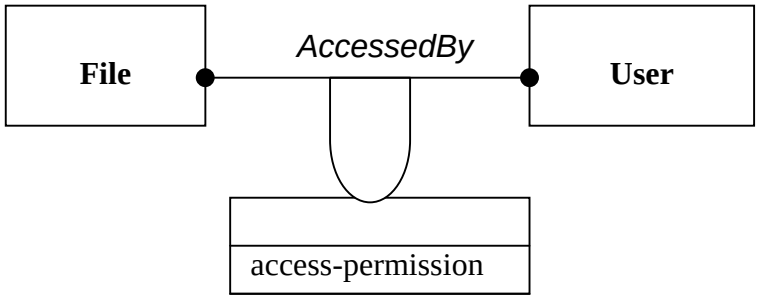


Figure 2.2

It is also possible to model an association as a class such class is called as link class as shown in Figure 2.3. Each link becomes one instance of the class. The notation for this kind of association is the same as for a link attribute and has a name and (optional) operations added to it.

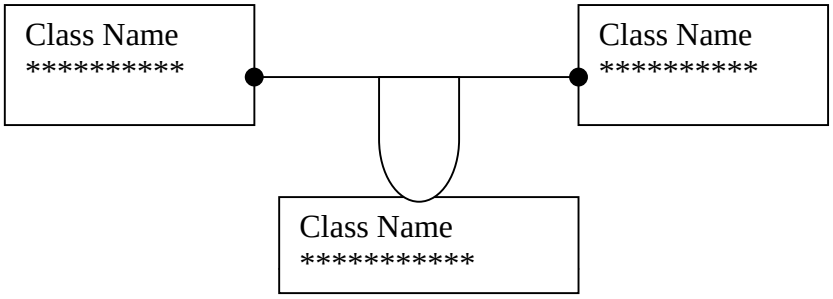


Figure 2.3

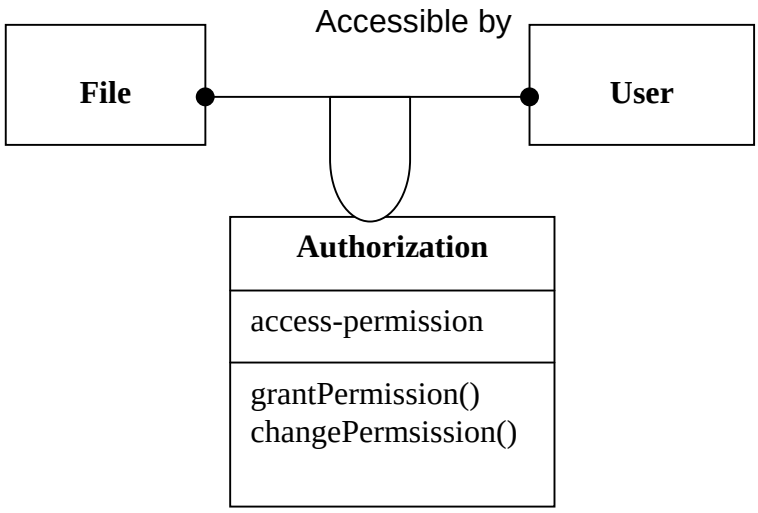


Figure 2.4

Now, consider the example shown in Figure 2.4, where whole class is associated with the link. In this example, the class Authorization is a link class. It has one attribute “access-permission” and two methods grantPermission() and changePermission().

2.2.2 Role Names

A role is one end of an association. A binary association can have two roles, each of which may have a role name. A role name is a name that uniquely identifies one end of an association. Roles provide a way of viewing a binary association as a traversal from one object to a set of associated objects. Each role on a binary association identifies an object or set of objects associated with an object at the other end. Figure 2.5 shows how to represent roles in OMT methodology.

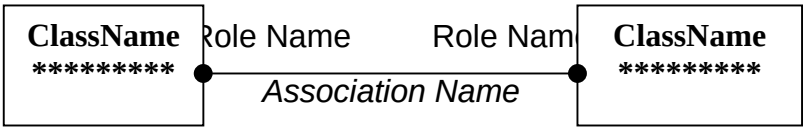


Figure 2.5

The use of role names is optional, but is often easier and less confusing to assign role names instead of, or in addition to, association names. Role names are necessary for associations between two objects of the same class. They are also useful to distinguish between two associations between the same pair of classes. We can follow these two guidelines: All role names on the far end of associations attached to a class must be unique. No role name should be the same as an attribute name of the source class. It is also possible to use role names for n-ary associations.

The role name is a derived attribute whose value is a set of related objects.
Use of role names provides a way of traversing associations from an object at one end, without explicitly mentioning the association.

For example, consider the association ‘a person works for a company’, in this employee and employer are role names for the classes Person and Company respectively as shown in Figure 2.6.

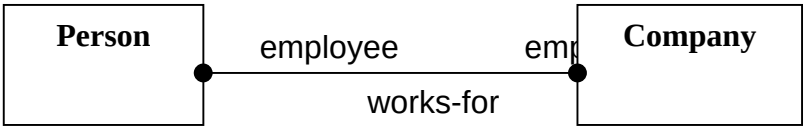


Figure 2.6

2.2.3 Ordering

Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Some times the objects on the many side of an association have order. Writing {ordered} next to the multiplicity dot as shown in Figure 2.7 indicates an ordered set of objects of an association.

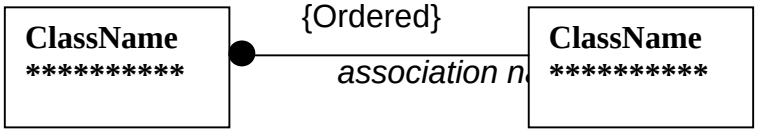


Figure 2.7

Consider the example of association between Window class and Screen class. A screen can contain a number of windows. Windows are explicitly ordered. Only topmost window is visible on the screen at any time. Figure 2.8 shows this example.

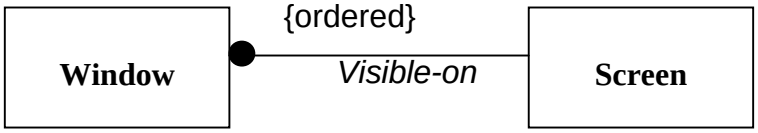


Figure 2.8

2.2.4 Qualification

A qualifier is an association attribute. A qualified association relates two object classes and a qualifier. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified. Figure 2.9 shows how to represent a qualification.

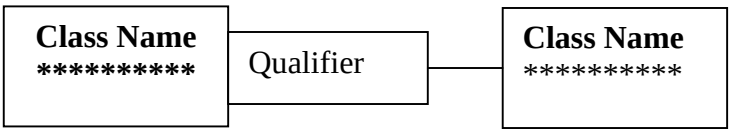


Figure 2.9

The qualifier is drawn as a small box on the end of the association line near the class it qualifies. The qualifier rectangle is part of the association, not of class. The qualifier distinguishes among the set of objects at the "many" end of an association. A qualified association can also be considered a form of ternary association. The advantage of the qualification is that it improves semantic accuracy and increases the visibility of navigation paths.

For example, a person object may be associated to a Bank object as shown in Figure 2.10. An attribute of this association is the accoutNo. The accountNo is the qualifier of this association.

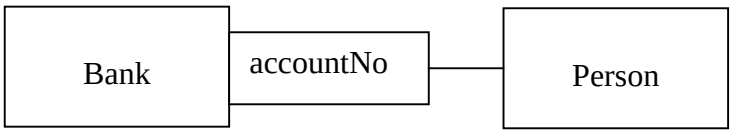


Figure 2.10

2.2.5 Aggregation

Aggregation is another relationship between classes. It is a tightly coupled form of association with some extra semantics. It is the “part-whole” or “a-part-

of" relationship in which objects representing the component of something are associated with an object representing the entire assembly. Aggregations are drawn like associations, except a small hollow diamond indicating the assembly end of the relationship as shown in Figure 2.11. The class opposite to the diamond side is part of the class on the diamond side.

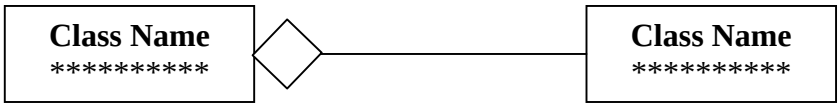


Figure 2.11

For example, a team is aggregation of players. This can be modeled as shown in Figure 2.12.

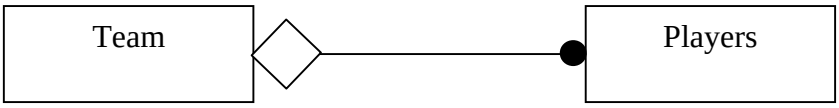


Figure 2.12

Aggregation can be fixed, variable or recursive.

- In a fixed aggregation number and subtypes are fixed i.e. predefined.
- In a variable aggregation number of parts may vary but number of levels is finite.
- A recursive aggregate contains, directly or indirectly, an instance of the same aggregate. The number of levels is unlimited. For example, as shown in Figure 2.13, a computer program is an aggregation of blocks, with optionally recursive compound statements. The recursion terminates with simple statement. Blocks can be nested to arbitrary depth.

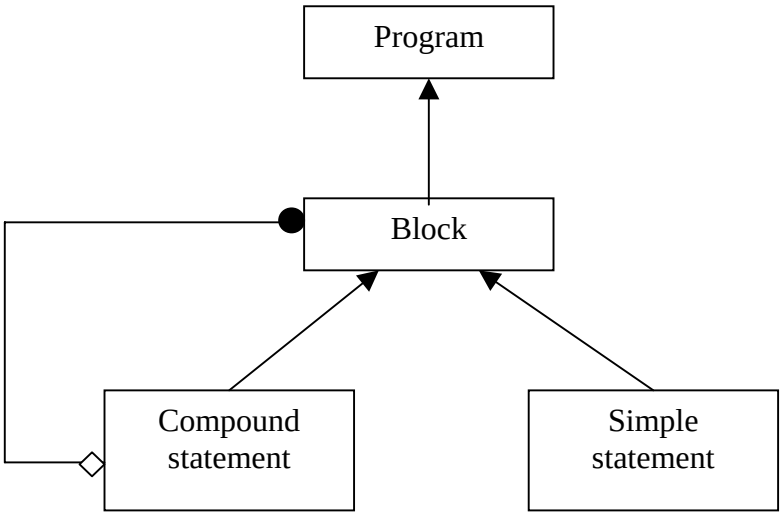
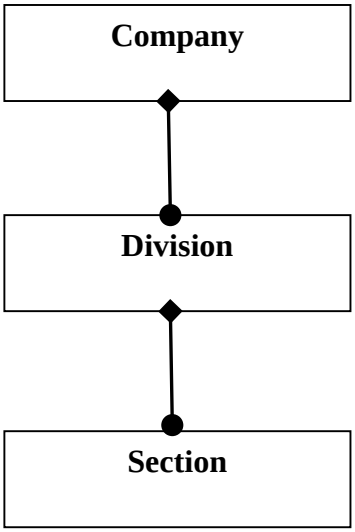


Figure 2.13

One more example of aggregation is shown in Figure 2.14. A company is composed of zero, one or more divisions. A division is composed of zero, one or more sections.

Figure 2.15



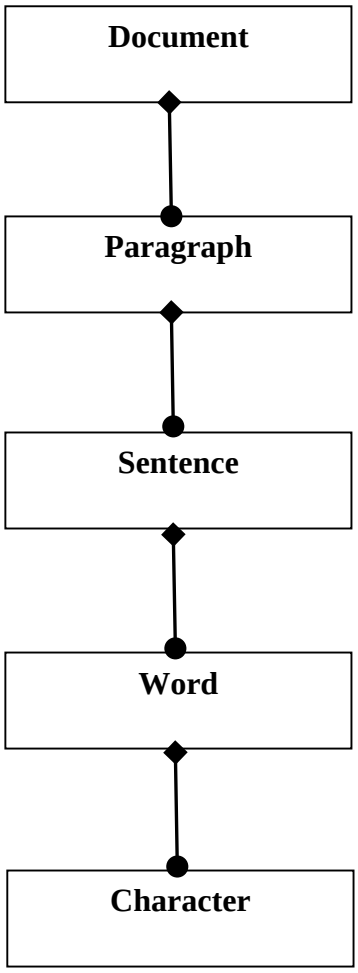


Figure 2.15

Another example of aggregation is shown in Figure 2.15. A document is composed of zero, one or more paragraphs. A paragraph is composed of zero, one or more sentences. A sentence is composed of one or more words. A word is composed of one or more characters.

2.2.6 Inheritance

The inheritance concept was invented in 1967 for Simula. Inheritance is a way to form new classes using classes that have already been defined. Inheritance is intended to help reuse existing code with little or no modification. The new classes, known as derived classes (or child classes or sub classes), inherit attributes and behavior of the pre-existing classes, which

are referred to as base classes (or parent classes or super classes) as shown in Figure 2.16. The inheritance relationship of sub- and super classes gives rise to a hierarchy.

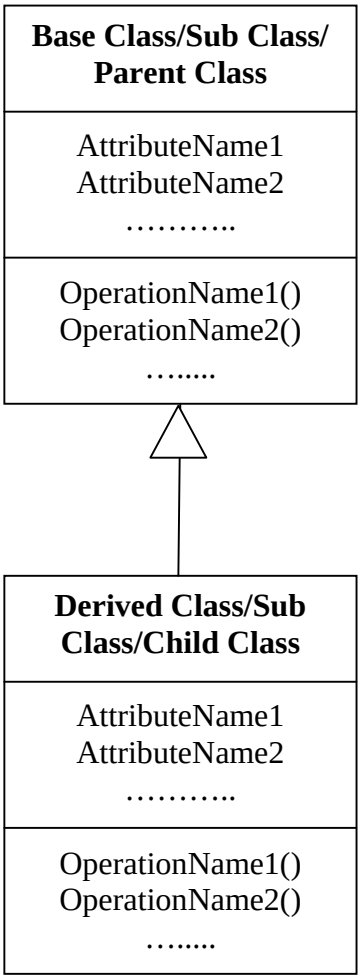


Figure 2.16

Inheritance is a “is-a” relationship between two classes. For example, Student is a Person; Chair is Furniture; Parrot is a Bird etc. in all these examples, first class (i.e. Student, Chair, Parrot) inherits properties from the second class (i.e. Person, Furniture, Bird).

Example of an inheritance: Manager is an Employee. Manager class inherits features from Employee class as shown in Figure 2.17.

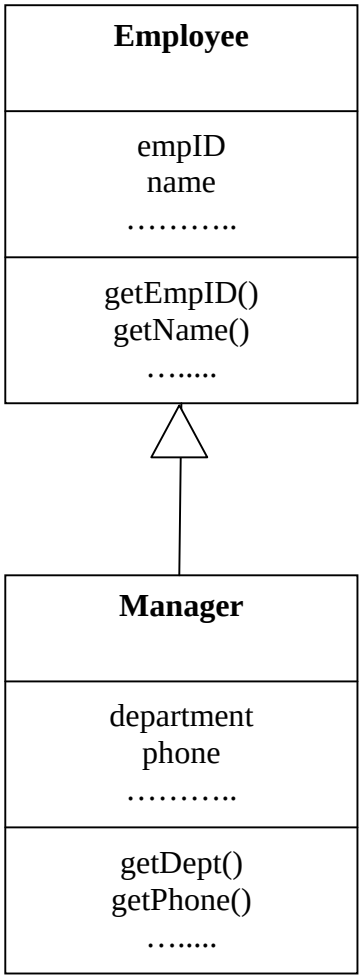


Figure 2.17

There are several reasons to use inheritance as enumerated below:

- Inheritance for specialization
- Inheritance for generalization
- Inheritance for extension
- Inheritance for restriction
- Inheritance for overriding

2.2.6.1 Inheritance for Specialization

One common reason to use inheritance is to create specializations of existing classes. In specialization, the derived class has data or behavior aspects that are not part of the base class. For example, Square is a Rectangle. Square class is specialization of Rectangle class. Similarly, Circle is an Ellipse. Here

also, Circle class is specialization of Ellipse class. Another example, a BankAccount class might have data members such as accountNumber, customerName and balance. An InterestBearingAccount class might inherit BankAccount and then add data member interestRate and interestAccrued along with behavior for calculating interest earned.

Another form of specialization occurs when a base class specifies that it has a particular behavior but does not actually implement the behavior. Each non-abstract, concrete class which inherits from that abstract class must provide an implementation of that behavior. This providing of actual behavior by a subclass is sometimes known as implementation or reification.

For example, there is a class Shape having operation area(). The operation area() cannot be implemented unless we have concrete class. So, Shape class is abstract class. Rectangle is a Shape. Now, Rectangle is a concrete class, which can implement the operation area().

2.2.6.2 Inheritance for Generalization

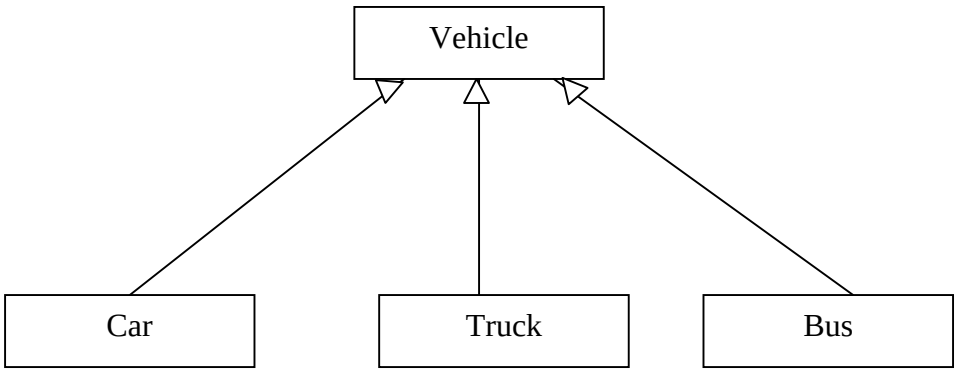


Figure 2.18

Generalization is reverse of specialization. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc. Conversely, since apples are fruit (i.e., an apple is-a fruit), apples may naturally inherit all the properties common to all fruit, such as being a fleshy container for the seed of a plant.

Another example: Vehicle is a generalization of Car, Truck, Bus etc. as shown in Figure 2.18. Car, Truck, Bus etc. share some properties such as “number of wheels”, speed, capacity etc. these common properties are abstracted out and put into another class say Vehicle, which comes higher in the hierarchy.

2.2.6.3 Inheritance for Extension

In this case, inheritance extends the existing class functionalities by adding new operations in the derived class. It can be distinguished from generalization that the later must override at least one method from the base and the functionality is tied to that of the base class. Extension simply adds new methods to those of the base class and functionality is less strongly tied to the existing methods of the base class.

For example, StringSet class inherits from Set class, which specializes for holding string values. Such a class might provide additional methods for string related operations – for instance - search by prefix, which returns a subset of all the elements of the set that begin with a certain string value. These operations are meaningful to the derived class but are not particularly relevant to the base class.

2.2.6.4 Inheritance for Restriction

In this case, the derived class does not implement the functionality, which a base class has. In other words, inheritance for restriction occurs when the behavior of the derived class is smaller or more restrictive than the behavior of the base class.

For example, an existing class library provides a double-ended queue (deque). Elements can be added or removed from either end of the deque, but the programmer wishes to write a stack class, enforcing the property that elements can be added or removed from only one end of the stack. Here, the programmer can make the Stack class a derived class of the existing Deque

class and can modify or override the undesired methods so that they produce an error message if used.

2.2.6.5 Inheritance for Overriding

When a class replaces the implementation of a method that it has inherited is called overriding. Overriding introduces a complication: which version of the method does an instance of the inherited class use the one that is part of its own class, or the one from the parent (base) class. The answer varies between programming languages, and some languages provide the ability to indicate that a particular behavior is not to be overridden.

2.2.6.6. Constraints of inheritance-based design

When using inheritance extensively in designing a program, one should be aware of certain constraints that it imposes. For example, consider a class Person that contains a person's name, address, phone number, age, gender, and race. We can define a subclass of Person called Student that contains the person's grade point average and classes taken, and another subclass of Person called Employee that contains the person's job title, employer, and salary.

In defining this inheritance hierarchy we have already defined certain restrictions, not all of which are desirable:

- Singleness: Using single inheritance, a subclass can inherit from only one superclass. Continuing the example given above, Person can be either a Student or an Employee, but not both. Using multiple inheritance partially solves this problem, as a StudentEmployee class can be defined that inherits from both Student and Employee. However, it can still inherit from each superclass only once; this scheme does not support cases in which a student has two jobs or attends two institutions.
- Static: the inheritance hierarchy of an object is fixed at instantiation when the object's type is selected and does not change with time. For example, the inheritance graph does not allow a Student object to become a

Employee object while retaining the state of its Person superclass. (Although similar behavior can be achieved with the decorator pattern.) Some have criticized inheritance, contending that it locks developers into their original design standards.

- Visibility: whenever client code has access to an object, it generally has access to all the object's superclass data. Even if the superclass has not been declared public, the client can still cast the object to its superclass type. For example, there is no way to give a function a pointer to a Student's grade point average and transcript without also giving that function access to all of the personal data stored in the student's Person superclass.

2.2.6.7 Roles and inheritance:

One consequence of separation of roles and superclasses is that compile-time and run-time aspects of the object system are cleanly separated. Inheritance is then clearly a compile-time construct. Inheritance does influence the structure of many objects at run-time, but the different kinds of structure that can be used are already fixed at compile-time.

To model the example of Person as an employee with this method, the modeling ensures that a Person class can only contain operations or data that are common to every Person instance regardless of where they are used. This would prevent use of a Job member in a Person class, because every person does not have a job, or at least it is not known that the Person class is only used to model Person instances that have a job. Instead, object-oriented design would consider some subset of all person objects to be in an "employee" role. The job information would be associated only to objects that have the employee role. Object-oriented design would also model the "job" as a role, since a job can be restricted in time, and therefore is not a stable basis for modeling a class. The corresponding stable concept is either "WorkPlace" or just "Work" depending on which concept is meant. Thus, from object-oriented design point of view, there would be a "Person" class and a "WorkPlace" class, which are related by a many-to-many association "works-

in", such that an instance of a Person is in employee role, when he works-in a job, where a job is a role of his work place in the situation when the employee works in it.

Note that in this approach, all classes that are produced by this design process are part of the same domain, that is, they describe things clearly using just one terminology. This is often not true for other approaches.

The difference between roles and classes is especially difficult to understand if referential transparency is assumed, because roles are types of references and classes are types of the referred-to objects.

In this example, a Student is a type of Person. Likewise, a Employee is a type of Person. Both Student and Employee inherit all the attributes and methods of Person. Student has a locally defined student ID attribute. Employee has a locally defined employee ID attribute.

So, if you would look at a Student object, you would see attributes of name, date of birth, parents, children, and student ID as shown in Figure 2.19.

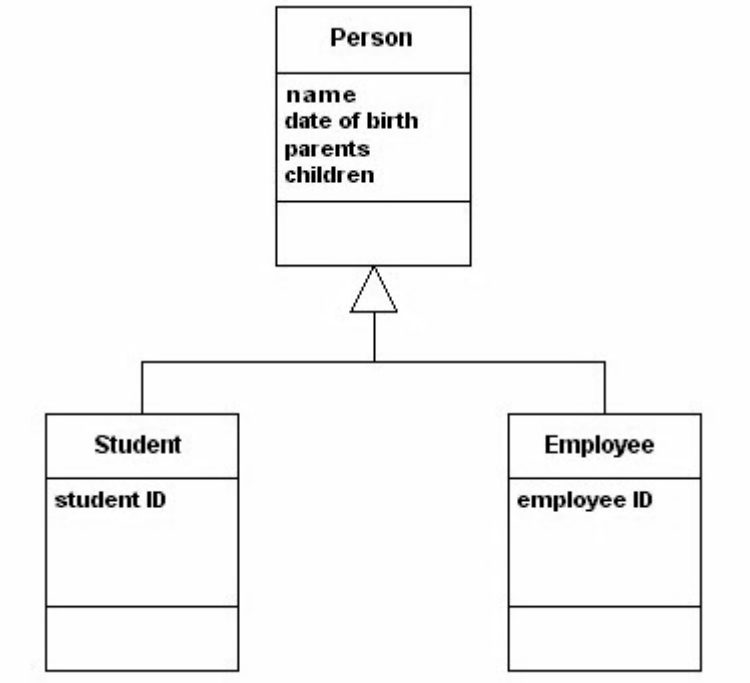


Figure 2.19

2.2.7 Types of Inheritance

There are many ways a derived class inherits properties from the base class. Following are the types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multipath Inheritance
- Hybrid Inheritance

2.2.7.1 Single Inheritance

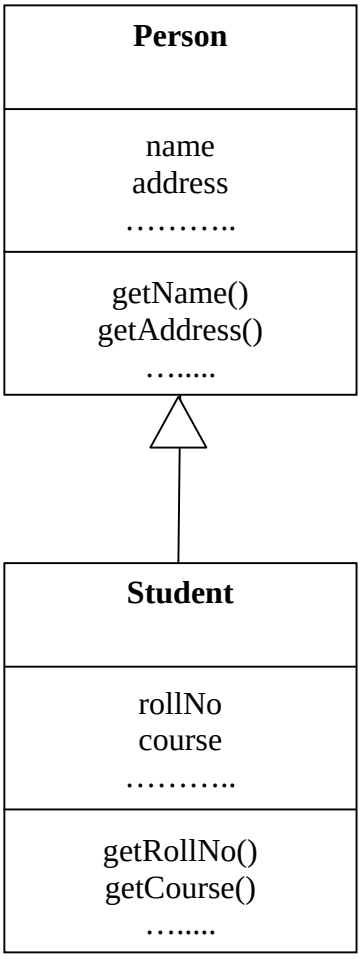


Figure 2.20

When a (derived) class inherits properties (data and operations) from a single base class, it is called as single inheritance. For example, Student class inherits properties from Person class as shown in Figure 2.20.

2.2.7.2 Multiple Inheritance

When a (derived) class inherits properties (data and operations) from more than one base class, it is called as multiple inheritance. For example, BoatHouse class inherits properties from both Boat class and House class as shown in Figure 2.21.

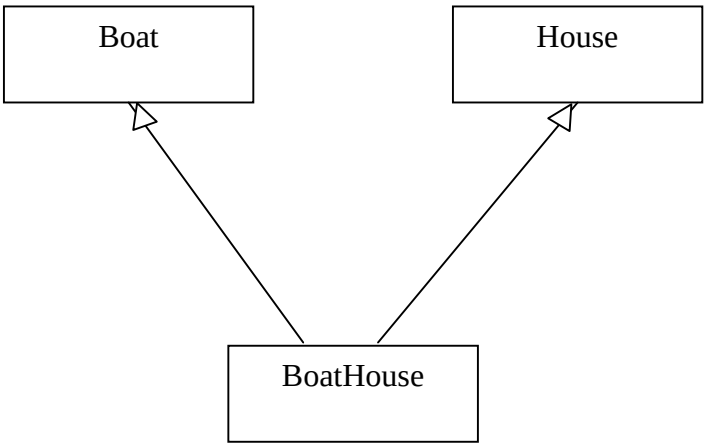


Figure 2.21

2.2.7.3 Multilevel Inheritance

When a (derived) class inherits properties (data and operations) from another derived class, it is called as multilevel inheritance. For example, Rectangle class inherits properties from Shape class and Square inherits properties from Rectangle class as shown in Figure 2.22.

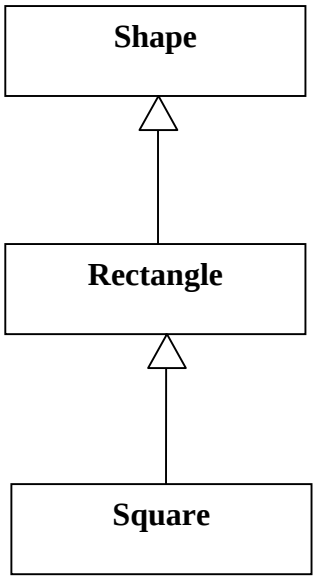


Figure 2.22

2.2.7.4 Hierarchical Inheritance

When more than one (derived) class inherits properties (data and operations) from a single base class, it is called as hierarchical inheritance. For example, Chair class, Table class and Bed class all inherit properties from Furniture class as shown in Figure 2.23.

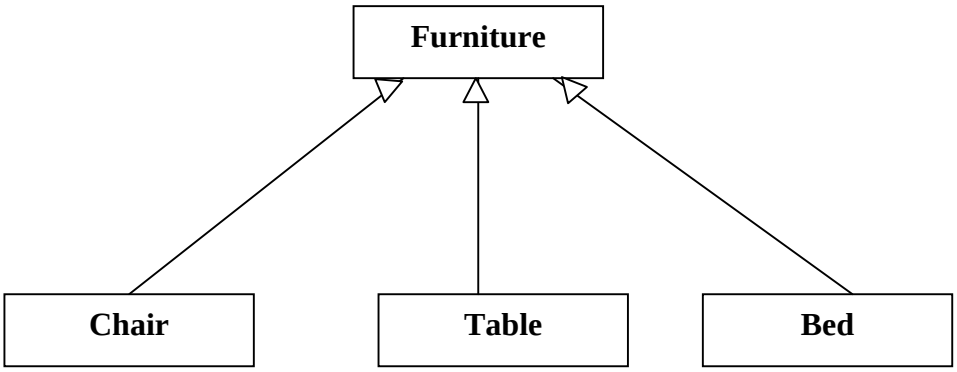


Figure 2.23

2.2.7.5 Multipath Inheritance

When more than one inheritance paths are available between two classes in the inheritance hierarchy, it is called as multipath inheritance. For example, Carnivorous and Herbivorous class inherit properties from Animal class. Omnivorous class inherits properties from Carnivorous and Herbivorous classes. So, there are two alternative paths available from Animal class to Omnivorous class as shown in Figure 2.24.

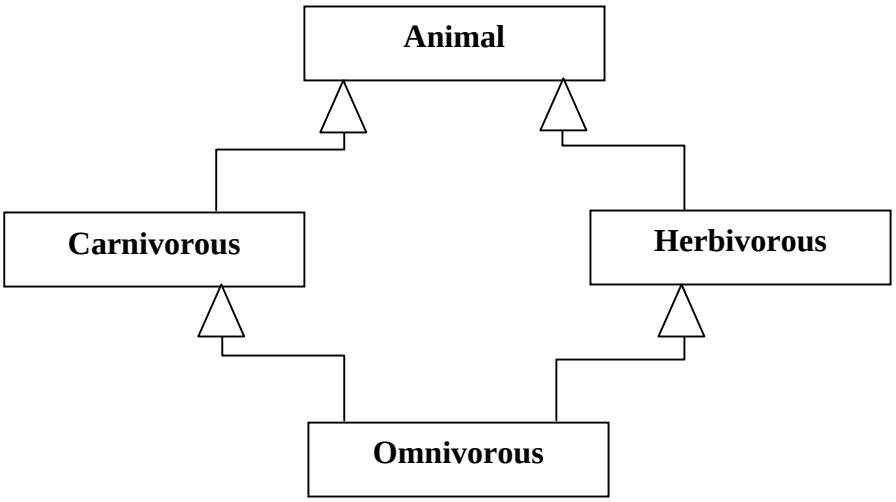


Figure 2.24

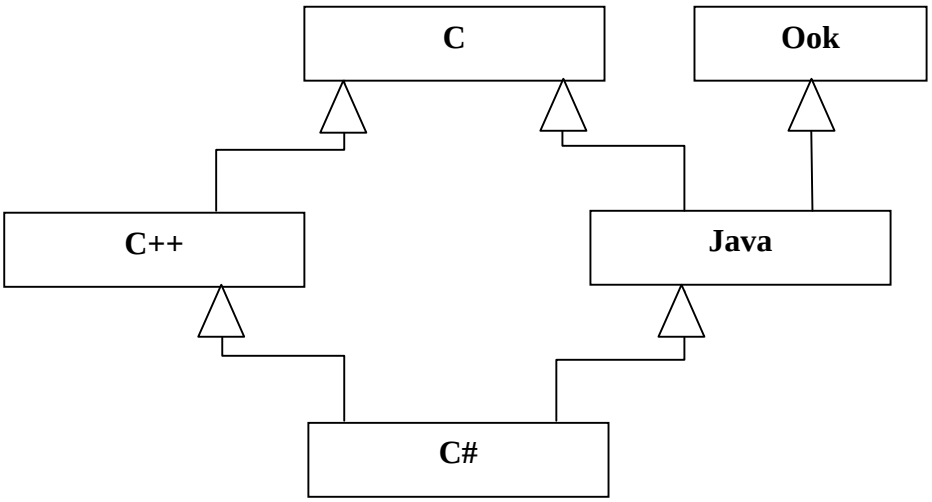


Figure 2.25

2.2.7.6 Hybrid Inheritance

Mixture of single, multiple, hierarchical and multilevel inheritance forms hybrid inheritance as shown in Figure 2.25.

2.2.8 Grouping Constructs

There are two grouping constructs: module and sheet.

Module is logical construct for grouping classes, associations and generalizations. An object model consists of one or more modules. The module name is usually listed at the top of each sheet.

A sheet is a single printed page. Sheet is the mechanism for breaking a large object model into a series of pages. Each module is contained in one or more sheets. Sheet numbers or sheet names inside circle contiguous to a class box indicate other sheets that refer to a class.

2.3 Summary

- Some times, an attribute(s) cannot be associated with either of the two classes associated by the association. In such cases, the attribute(s) is associated with the association and is called as link attribute. It is a property of the links in an association.

- A role is one end of an association. A binary association can have two roles, each of which may have a role name. A role name is a name that uniquely identifies one end of an association.

- A qualifier is an association attribute. A qualified association relates two object classes and a qualifier. The qualifier is a special attribute that reduces the effective multiplicity of an association.

- A qualifier is an association attribute. A qualified association relates two object classes and a qualifier. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified.

- Inheritance is a “is-a” relationship between two classes. For example, Student is a Person; Chair is Furniture; Parrot is a Bird etc. in all these examples, first class (i.e. Student, Chair, Parrot) inherits properties from the second class (i.e. Person, Furniture, Bird).

2.4 Suggested Readings/Reference Materials

1. Object-Oriented Modeling and Design with UML, M. Blaha, J. Rumbaugh, Pearson Education-2007
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson-2007
3. Object Oriented Analysis & Design, Grady Booch, Addison Wesley-1994
4. Timothy C. Lethbridge, Robert Laganiere, Object Oriented Software Engineering, TMH, 2004

2.5 Self-Assessment Questions

1. Explain the following properties of an association by giving at least two examples of each: link attribute, role name, ordering and qualification.
2. What is aggregation? How is it different from association?
3. What are different types of aggregation? Explain each with suitable examples.
4. What is recursive aggregation? Give suitable example of recursive aggregation.
5. What is inheritance? What are different uses of an inheritance? Explain.
6. Distinguish between the following:

a. Generalization and Specialization
b. Inheritance and Aggregation

7. Comment on “Inheritance for extension or restriction”.
8. What are different types of inheritance? Explain them with two examples of each.
9. Discuss different types of grouping constructs.

Writer: Dr. Rajender Nath

Vetter: Dr. Dharminder Kumar

Chapter 3

Dynamic Modeling

Structure

3.0 Introduction

The complete OOM revolves around the objects identified in the system. When observed closely, every object exhibits some characteristics and behavior. The objects recognize and respond to certain events. For example, considering a Window on the screen as an object, the size of the window gets changed when resize button of the window is clicked. Here the clicking of the button is an event to which the window responds by changing its state from the old size to the new size. While developing systems based on this approach, the analyst makes use of certain models to analyze and depict these objects.

The dynamic model represents a state/transition view on the model. Main concepts are states, transitions between states, and events to trigger transitions. Actions can be modeled as occurring within states. Generalization and aggregation (concurrency) are predefined relationships. The outcomes of a dynamic model are scenarios, event-trace diagrams and sate diagrams.

3.1 Objectives

In this chapter you will learn what is dynamic model and concepts related to dynamic modeling such as event, state, state transition, action and activity. You will also learn how to model dynamic behavior of a system through scenario, event-trace diagram and state diagram.

3.2 Presentation of Contents

3.2.1 Dynamic Modeling

Dynamic model describes those aspects of the system that changes with the time. It is used to specify and implement control aspects of the system. It depicts states, transitions, events and actions. The dynamic model 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.

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.

The outcomes of dynamic modeling are scenario, event-trace diagram and state diagram. These are discussed one by one in detail in the following sections.

3.2.1.1 Scenario

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. The scope of scenario may vary. It may include all events in the system or it may include only those events generated by certain objects. A scenario can be written as a list of text statements.

Figure 3.1 shows a scenario to use ATM for withdrawing money. Each event transmits information from one object to another. For example, the event “the ATM asks the user to insert a card” transmits a signal from the ATM to the User. The next event is “the user inserts a cash card”. The next event is “the ATM accepts the card and reads its serial no.” and so on.

Normal ATM scenario	
1.	The ATM asks the user to insert a card The user inserts a cash card
2.	The ATM accepts the card and reads its serial number.
3.	The ATM requests the password The user enters 1234
4.	The ATM verifies the serial number and password with the consortium The consortium checks it with bank ABC and notifies the ATM of acceptance
5.	The ATM requests amount The user enters 15000
6.	The ATM processes the request and dispenses the required amount of money.

Figure 3.1

3.2.1.2 Event-Trace Diagram

The limitation of scenario is that it is not clear from scenario, how many objects are involved and which object generates an event and which object receives an event. To overcome this limitation, an event-trace diagram is introduced. In the event-trace diagram, the sequence of events and the objects exchanging events both can be shown. The diagram shows each object as a vertical line and each event as a horizontal arrow from the sender object to the receiver object. Time increases from top to bottom. Spacing between horizontal arrows carries no information. Figure 3.2 below shows the event-trace diagram for interaction with the ATM.

In this diagram, four objects – User, ATM, Consortium and Bank - are involved, which are shown with four vertical lines. User generates an event “insert card” which is shown as horizontal arrow from user to ATM. That means source of the event is

User object and destination of the event is ATM. In response to this event, the ATM generates the “request password” event to the User. Spacing between these two arrows is insignificant but the event “insert card” occurs before the event “request password” and so on.

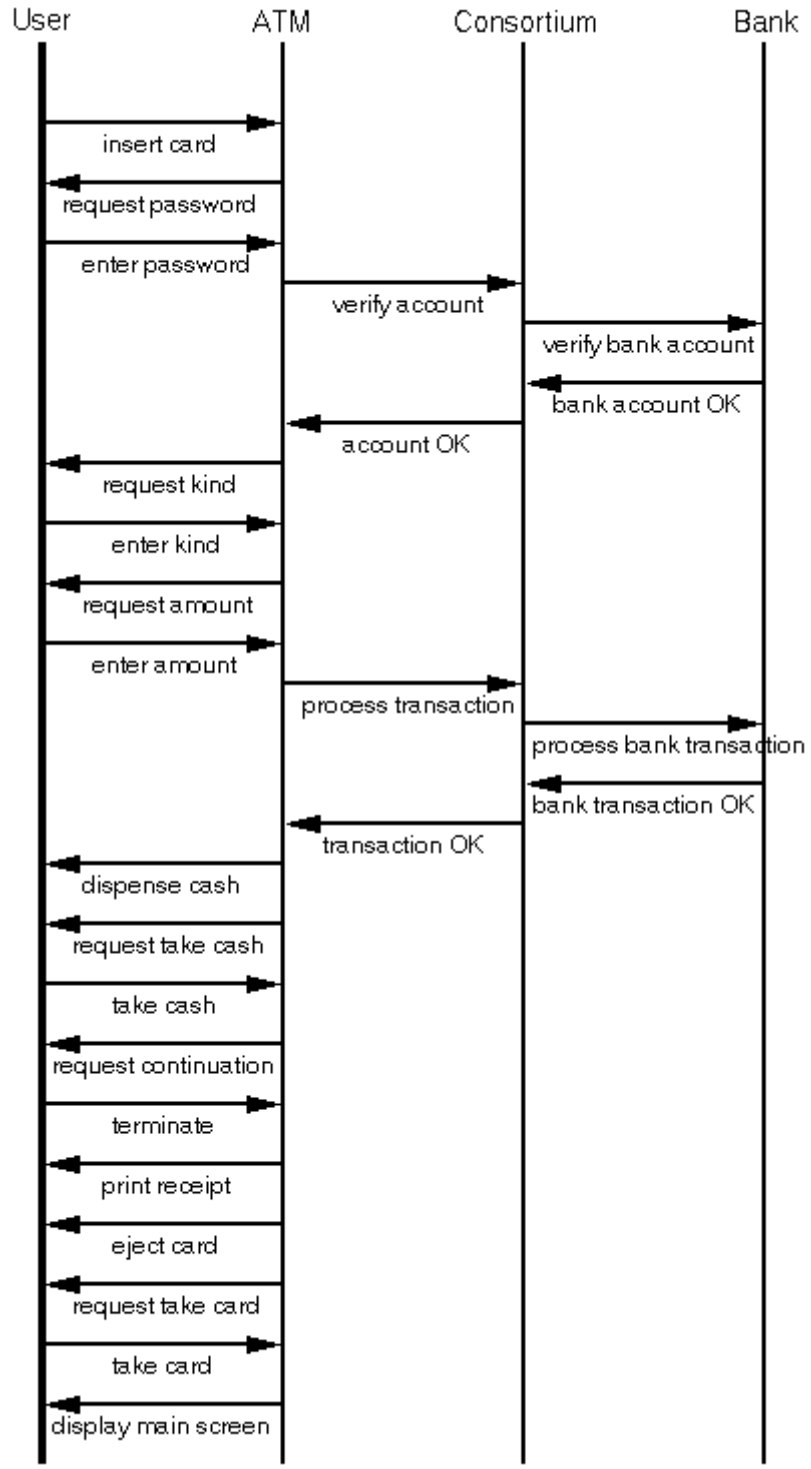


Figure 3.2

3.2.1.3 State Machine

A state machine is a behavior which specifies the sequence of states an object visits during its lifetime in response to events, together with its responses to those events. Now, we describe the various concepts related to state machine in the following subsections.

3.2.1.3.1 State

A state is a condition during the life of an object during which it satisfies some condition, performs some activity, or waits for some external event. 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. For instance, stack is empty or stack is full are different states of the object stack. As state corresponds to interval between two events received by an object so it has duration.

A substate is a state that is nested in another state. A state that has substates is called a composite state. A state that has no substates is called a simple state. Substates may be nested to any level.

3.2.1.3.2 Event

An event is the specification of a significant occurrence. For a state machine, an event is the occurrence of a stimulus that can trigger a state transition. In other words, we can say an event is something that happens at a point in time. An event does not have duration. An individual stimulus from one object to another is an event. Press a button on mouse, airplane departs from an airport are examples of events.

3.2.1.3.3 Transition

A transition is a relationship between two states indicating that an object in the first state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the second state. Transition can be self-transition. It is a transition whose source and target states are the same.

If a transition is to a composite state, the nested state machine must have an initial state. If a transition is to a substate, the substate is entered after any entry action for the enclosing composite state is executed followed by any entry action for the substate.

If a transition is from a substate within the composite state, any exit action for the substate is executed followed by any exit action for the enclosing composite state. A transition from the composite state may occur from any of the substates and takes precedence over any of the transitions for the current substate.

3.2.1.3.4 Action

An action is an executable, atomic (with reference to the state machine) computation. Actions may include operations, the creation or destruction of other objects, or the sending of signals to other objects (events). An action is an instantaneous operation. An action represents an operation whose duration is insignificant compared to the resolution of the state diagram. For instance, disconnect phone line might be an action in response to an on-hook eventfor the phone line. An action is associated with an event.

3.2.1.3.5 Activity

Activity is an operation that takes time to complete. An activity is associated with a state. Activity includes continuous operations such as displaying a picture on a television screen as well as sequential operations that terminate by themselves after an interval of time such as closing a valve or performing a computation. A state may

control a continuous activity such as ringing a telephone bell that persists until an event terminates it causing the transition of the state. Activity starts on entry to the state and stops on exit. A state may also control a sequential activity such as a robot moving a part that progresses until it completes or until it is interrupted by an event that terminates prematurely.

3.2.1.4 State Diagram

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and tracks the different states of its objects through the system.

It relates events and states. A change of state caused by an event is called a transition. Transition is drawn as an arrow from the receiving state to the target state. A state diagram is graph whose nodes are states and whose directed arcs are transitions labeled by event names. State diagram specifies the state sequence caused by an event sequence.

3.2.1.4.1 When to use state diagrams

Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case. State diagrams are combined with other diagrams such as interaction diagrams and activity diagrams.

3.2.1.4.2 How to draw state diagrams

State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state as shown Figure 3.3 below.

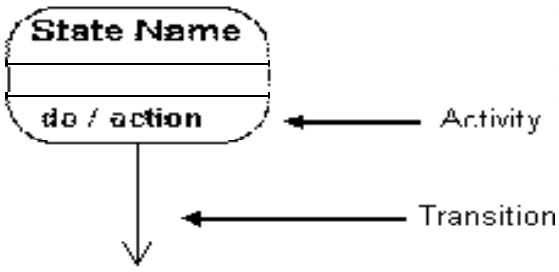


Figure 3.3

Initial and Final States: All state diagrams being with an initial state of the object as shown in Figure 3.4. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

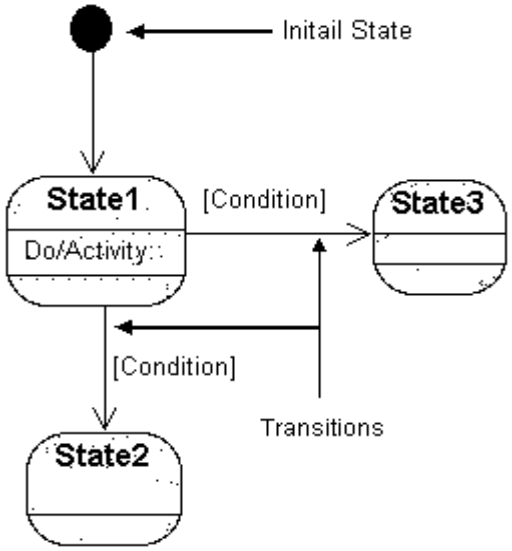


Figure 3.4

The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name as shown in Figure 3.5.

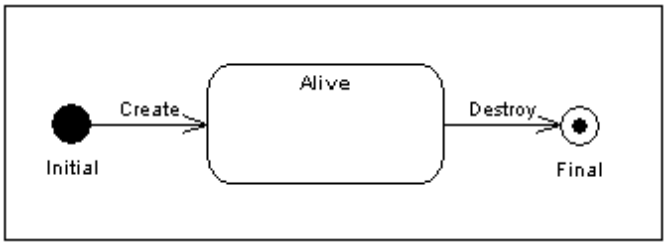


Figure 3.5

Transitions: Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as shown in Figure 3.6 below.

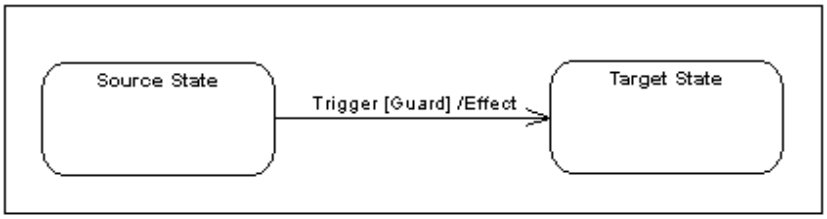


Figure 3.6

"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Actions: In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram in Figure 3.7 below shows a state with an entry action and an exit action. It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.

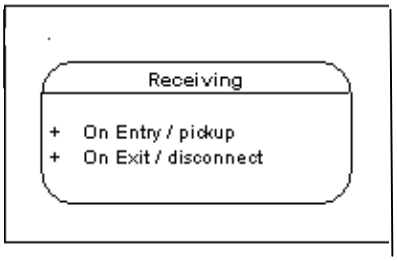


Figure 3.7

Self-Transitions: A state can have a transition that returns to itself, as shown in the Figure 3.8. This is the most useful when an effect is associated with the transition.

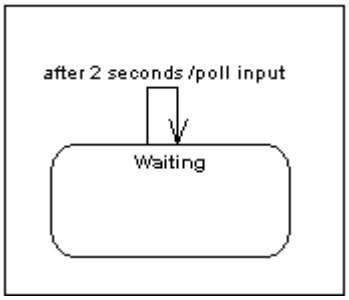


Figure 3.8

Compound States: A substate is a state that is nested in another state. A state that has substates is called a composite state. A state that has no substates is called a simple state. Substates may be nested to any level.

A state machine diagram may include sub-machine diagrams, as in Figure 3.8 below. The alternative way to show the same information is as shown in Figure 3.9.

The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

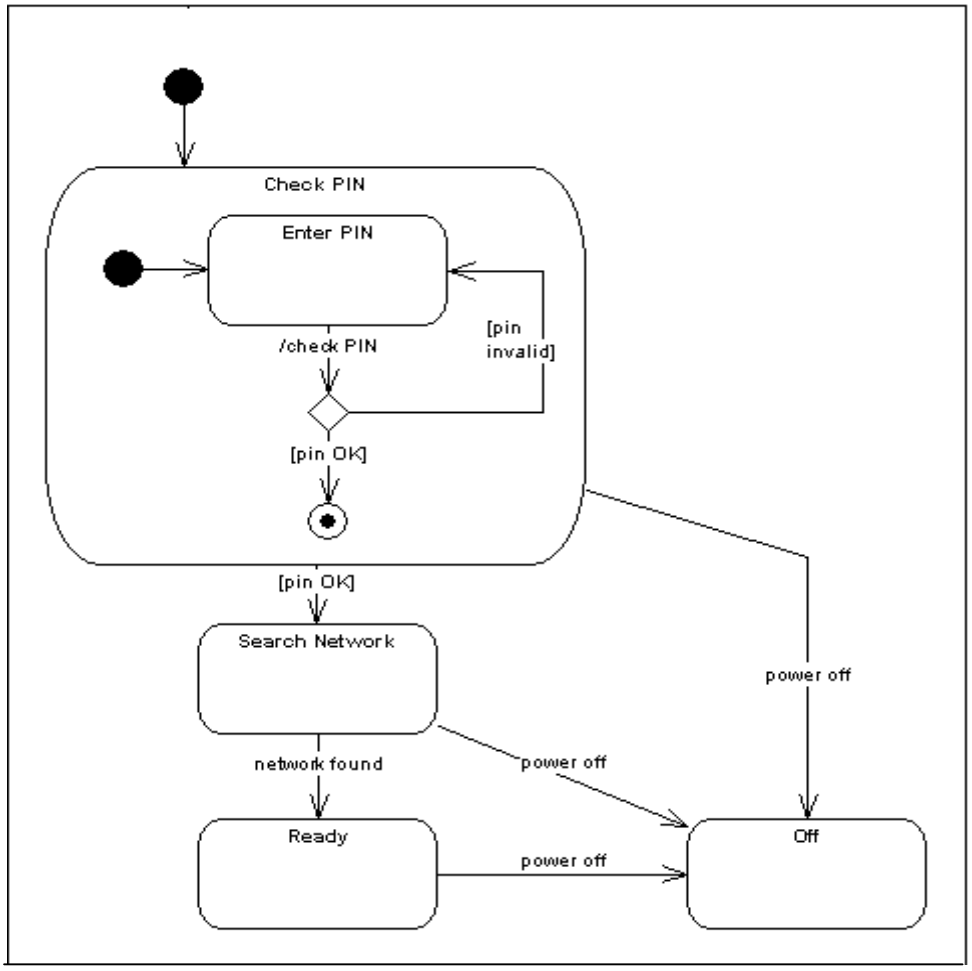


Figure 3.8

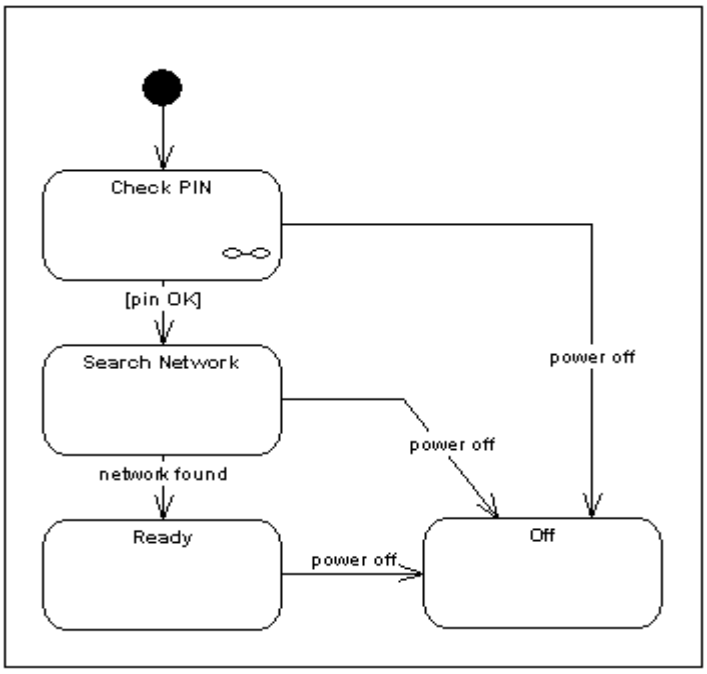


Figure 3.9

Entry Point: Sometimes you won't want to enter a sub-machine at the normal initial state. For example, in the sub-machine shown in Figure 3.10, it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.

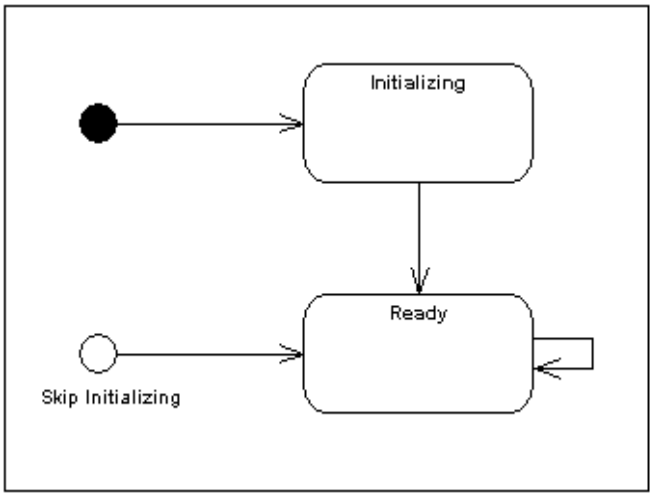


Figure 3.10

Exit Point: In a similar manner to entry points, it is possible to have named alternative exit points. The Figure 3.11 gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.

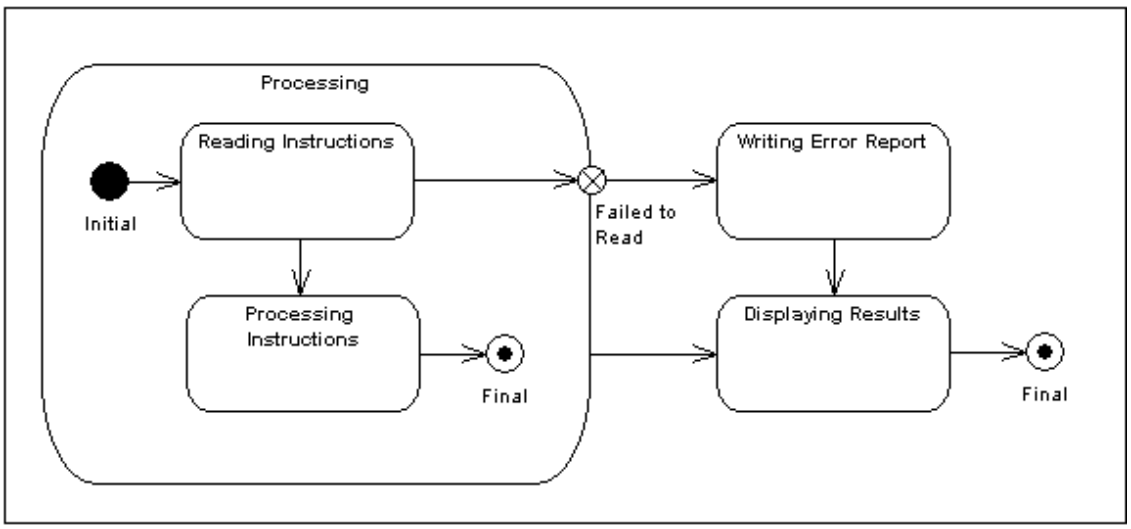


Figure 3.11

Choice Pseudo-State: A choice pseudo-state is shown as a diamond with one transition arrives and two or more transitions leaving. The Figure 3.12 shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.

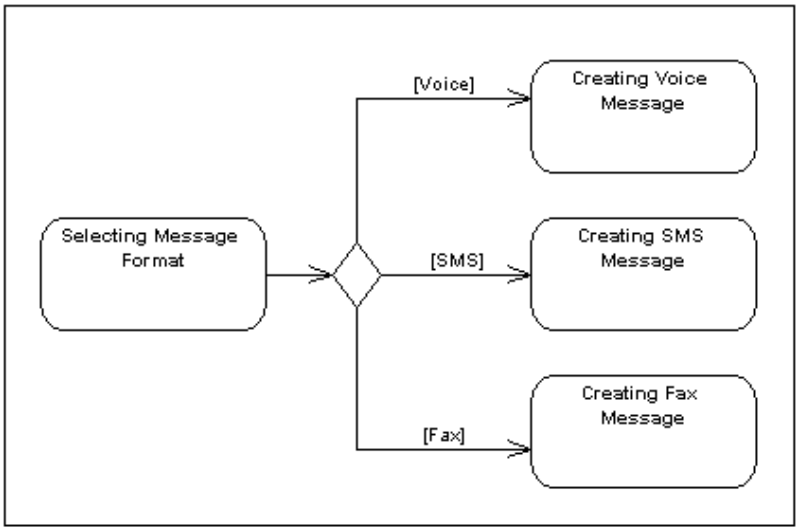


Figure 3.12

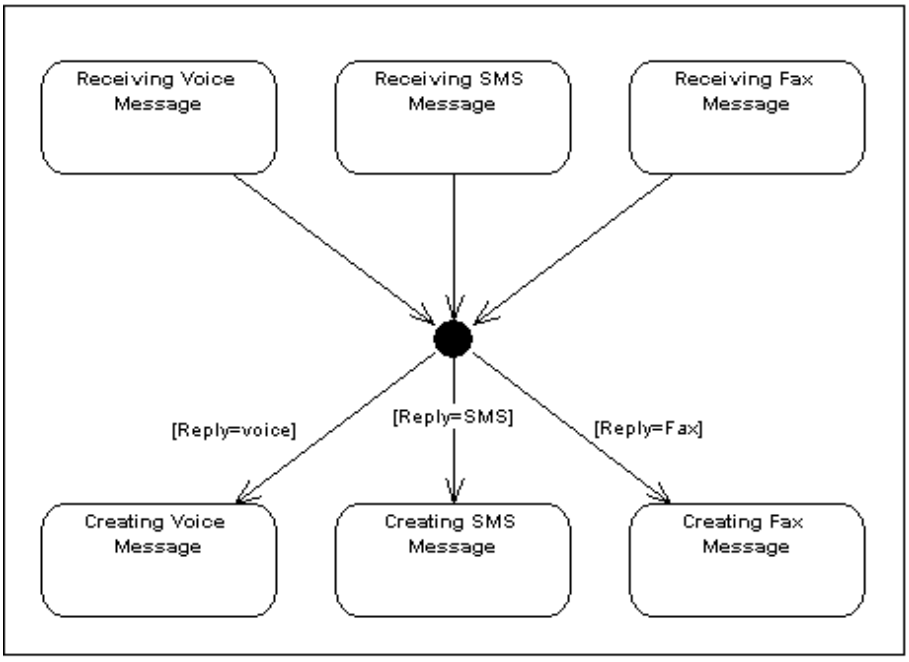


Figure 3.13

Junction Pseudo-State: Junction pseudo-states as shown in Figure 3.13 are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free. A junction, which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.

Terminate Pseudo-State: Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is denoted as a cross as shown in Figure 3.14.

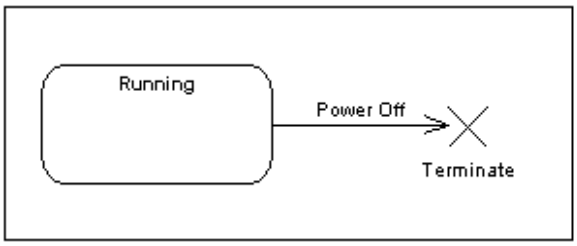


Figure 3.14

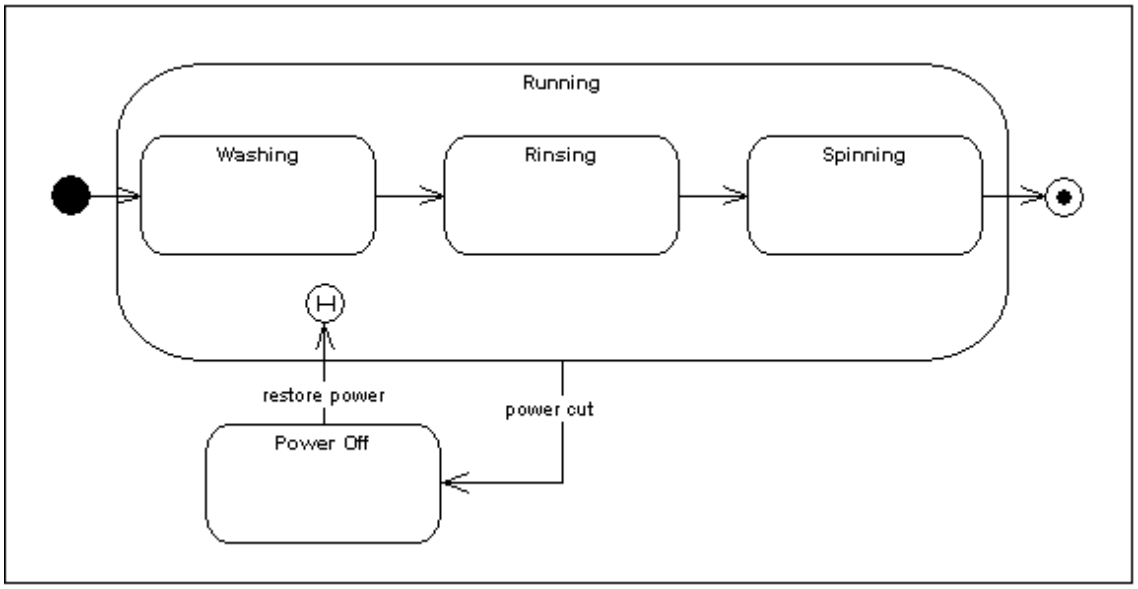


Figure 3.15

History States: A history state is used to remember the previous state of a state machine when it was interrupted. The diagram shown in Figure 3.15 illustrates the use of history states. The example is a state machine belonging to a washing machine.

In this state machine, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning". If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

Concurrent Regions: A state may be divided into regions containing sub-states that exist and execute concurrently. Figure 3.16 shows that within the state “Applying Brakes”, the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

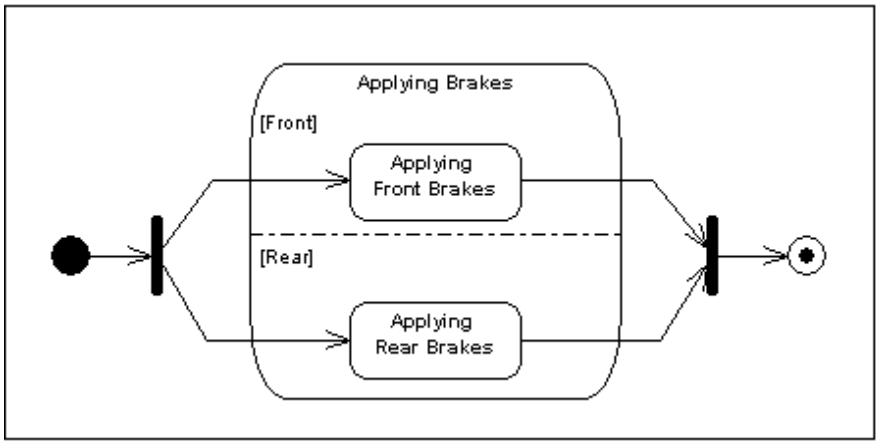


Figure 3.16

A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

As an example, the following state machine in Figure 3.17 shows the states that a door goes through during its lifetime.

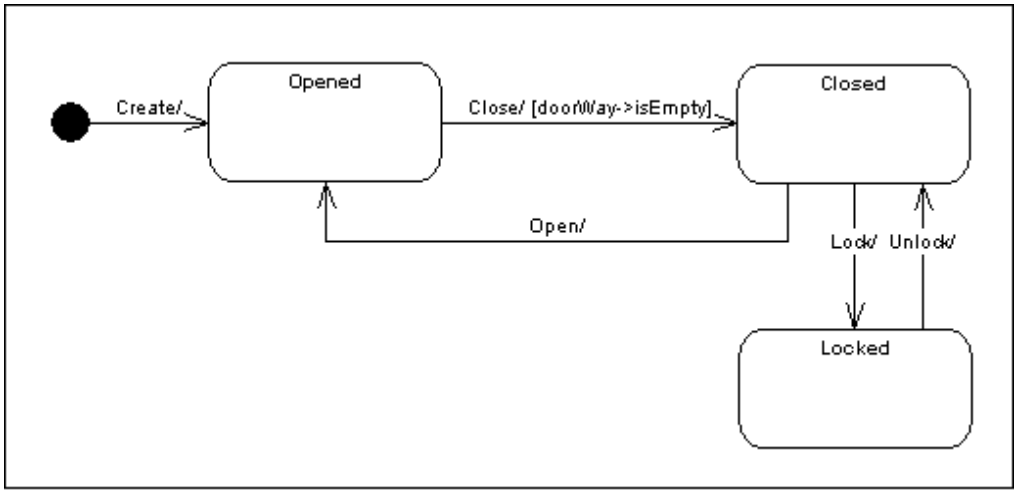


Figure 3.17

The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock. Notice that not all events are valid in all states; for example, if a door is opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition door Way->isEmpty is fulfilled.

Let us take another example of the state diagram for an Order object as shown in Figure 3.18. When the object enters the Checking state it performs the activity “check items.” After the activity is completed the object transitions to the next state based on the conditions [all items available] or [an item is not available]. If an item is not available the order is canceled. If all items are available then the order is dispatched. When the object transitions to the Dispatching state the activity “initiate delivery” is performed. After this activity is complete the object transitions again to the Delivered state.

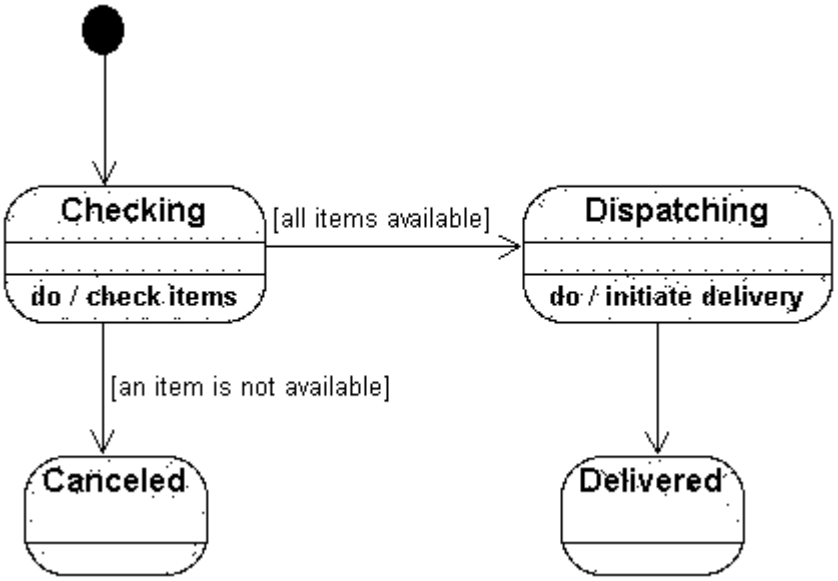


Figure 3.18

State diagrams can also show a super-state for the object. A super-state is used when many transitions lead to a certain state. Instead of showing all of the transitions from each state to the redundant state a super-state can be used to show that all of the states inside of the super-state can transition to the redundant state. This helps make the state diagram easier to read.

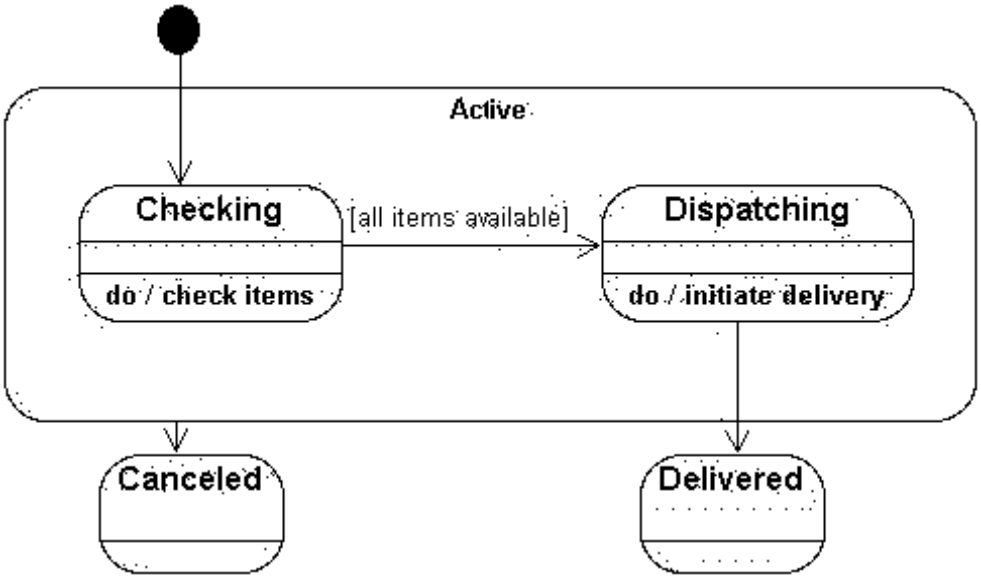


Figure 3.19

Figure 3.19 shows a super-state. Both the Checking and Dispatching states can transition into the Canceled state, so a transition is shown from a super-state named Active to the state Cancel. By contrast, the state Dispatching can only transition to the Delivered state, so we show an arrow only from the Dispatching state to the Delivered state.

Another illustrative example of state diagram is shown in Figure 3.20 below. It shows the bank transaction and verification of bank account. Activities are shown in the state such as do: update account, do: verify card number and do: verify password. Diagram also shows initial and final states.

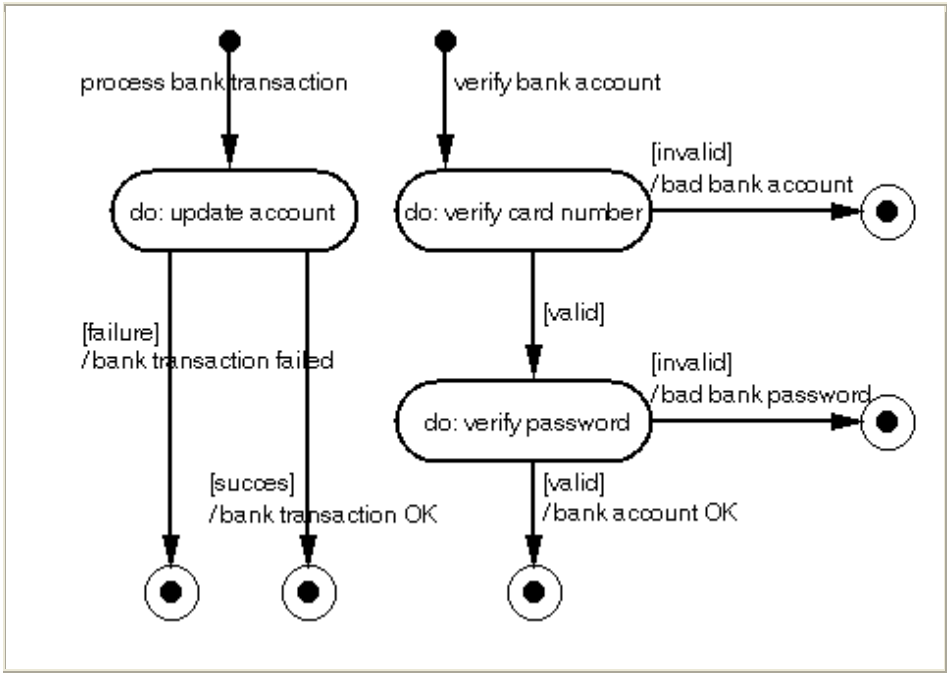


Figure 3.20

Now, we consider one more example of state diagram for phone line is shown Figure 3.21. A phone can have many states such as idle, dial tone, dialing, connecting ringing etc. various states and events are shown in figure below.

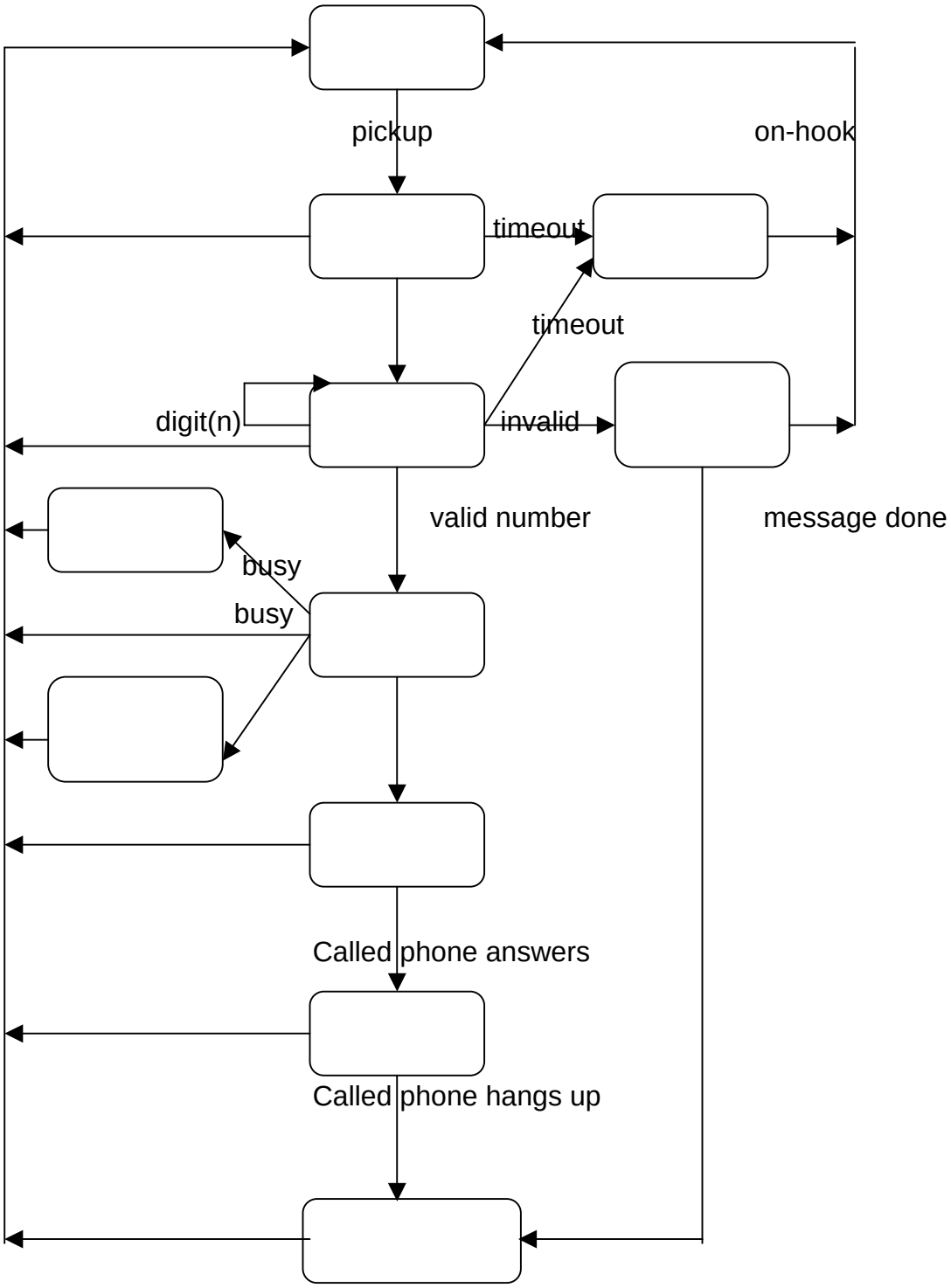

pickup
on-hook
timeout
timeout
digit(n)
invalid
valid number
message done
busy
busy
Called phone answers
Called phone hangs up

Figure 3.21

3.3 Summary

- The dynamic model represents a state/transition view on the model. Main concepts are states, transitions between states, and events to trigger transitions. Actions can be modeled as occurring within states.
- The dynamic model is represented graphically by state diagrams as well as scenarios and event trace 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 can be simple or compound.

- 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 transition is a relationship between two states indicating that an object in the first state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the second state. Transition can be self-transition. It is a transition whose source and target states are the same.

- An action is an executable, atomic (with reference to the state machine) computation. Actions may include operations, the creation or destruction of other objects, or the sending of signals to other objects (events). An action is an instantaneous operation.

- An activity is associated with a state. Activity includes continuous operations such as displaying a picture on a television screen as well as sequential operations that terminate by themselves after an interval of time such as closing a valve or performing a computation.

- 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.

- 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.

3.4 Suggested Readings/Reference Materials

1. Object-Oriented Modeling and Design with UML, M. Blaha, J. Rumbaugh, Pearson Education-2007
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson-2007
3. Object Oriented Analysis & Design, Grady Booch, Addison Wesley-1994
4. Timothy C. Lethbridge, Robert Laganiere, Object Oriented Software Engineering, TMH, 2004

3.5 Self-Assessment Questions

1. What is dynamic model? How is it represented?
2. Define the following concepts with examples and their notations:
   State, Event, Transition, Action, and Activity
3. What is scenario? Write a scenario for making a call on a telephone.
4. Write a scenario to withdraw money from ATM.
5. What are the limitations of a scenario? How are these limitations overcome?
6. What is event-trace diagram? Draw an event-trace diagram for a call on a telephone.
7. Draw an event-trace diagram for withdrawing money from ATM.
8. What is a state diagram? Draw a state diagram for an answering machine.

9. Distinguish between the following:
    a. State and Event
    b. Simple and Compound States
    c. Action and Activity
    d. Initial and Final states
    e. Entry and Exit Point
10. Moving a bag of corn, a goose, and a fox across the river in a boat. Only one thing may be carried in the boat at a time. If the goose is left alone with the corn, the corn will be eaten. If the goose is left alone with the fox, the goose will be eaten. Prepare two scenarios one in which something gets eaten and one in which everything is safely moved across the river.
11. Prepare a scenario to log on to a computer.
12. Prepare a state diagram for selecting and dragging items with the diagram editor.
13. Explain choice pseudo states, junction pseudo states and history states with suitable examples.
14. What are concurrent regions? How are they depicted in the state diagrams? Explain with an example.

Writer: Dr. Rajender Nath

Vetter: Dr. Dharminder Kumar

Chapter 4

Functional Modeling

Structure

4.0 Introduction

The functional model describes computations and specifies those aspects of the system concerned with transformations of values - functions, mappings, constraints, and functional dependencies. The functional model captures what the system does, without regard to how or when it is done.

The functional model uses a hierarchy of data flow diagrams (DFDs) in a similar fashion to the Yourdon method. However, the DFDs are not very well integrated with the object and dynamic models and it is difficult to imagine many projects making extensive use of the functional model.

Some aspects of the functional model are, however, quite useful. For example, the context diagram is vital for defining the scope of the system. Also, it may be possible to strengthen the real-time aspects of the method by introducing the concept of a processor and task model using DFDs.

Data Flow Diagrams are commonly used during problem analysis. They are quite general and are not limited to problem analysis for software requirements specification. DFDs are very useful in understanding a system and can be effectively used during analysis.

4.1 Objectives

In this chapter, you will learn what is functional model and DFDs. The chapter discusses how to draw a DFD, what is data dictionary and metadata. It also introduces concepts of candidate keys.

4.2 Presentation of Contents

4.2.1 Functional Modeling

The functional model describes computations and specifies those aspects of the system concerned with transformations of values - functions, mappings, constraints, and functional dependencies. The functional model captures what the system does, without regard to how or when it is done.

The functional model is represented graphically with multiple data flow diagrams, which show the flow of values from external inputs, through operations and internal data stores, to external outputs. Data flow diagrams show the dependencies between values and the computation of output values from input values and functions. Functions are invoked as actions in the dynamic model and are shown as operations on objects in the object model. The data flow diagram should adhere to OMT's notation and exploit the capabilities of OMT, such as nesting, control flows, and constraints.

4.2.1.1 Data Flow Diagram (DFD)

Data Flow Diagrams are commonly used during problem analysis. They are quite general and are not limited to problem analysis for software requirements specification. DFDs are very useful in understanding a system and can be effectively used during analysis.

A DFD shows the flow of data through a system. It views a system as a function that transforms the inputs into desired outputs. Any complex system will not perform this transformation in a "single step", and a data will typically undergo a series of transformations before it becomes the output. The DFD aims to capture the transformations that take place within a system to the input data so that eventually the output data is produced. The agent that performs the transformation of data from one state to another is called a process. So a DFD shows the movement of data through the different transformation or process in the system.

DFDs are basically of 2 types: Physical and logical ones. Physical DFDs are used in the analysis phase to study the functioning of the current system. Logical DFDs are used in the design phase for depicting the flow of data in a proposed system.

In a nutshell, a data flow diagram is a graph showing the flow of data values from their sources in objects through processes that transform them to their destinations in other objects.

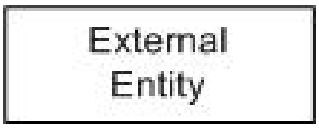
Data Flow Diagrams are composed of the four basic symbols – external entities, processes, data stores and data flow - as discussed below.

- The External Entity symbol represents sources of data to the system or destinations of data from the system.
- The Process symbol represents an activity that transforms or manipulates the data (combines, reorders, converts, etc.).
- The Data Store symbol represents data that is not moving (delayed data at rest).
- The Data Flow symbol represents movement of data.

Any system can be represented at any level of detail by these four symbols. Now, these four elements of DFD are discussed in detail.

4.2.1.1.1 External Entity

The External Entity symbol represents sources of data to the system or destinations of data from the system. They determine the system boundary. They are external to the system being studied. They are often beyond the area of influence of the developer. They can represent another system or subsystem. These go on margins/edges of data flow diagram. They are represented by a rectangle symbol and are named with appropriate name as shown in Fig below.



Some authors call them actors as they are active objects that drive the data flow diagram by producing or consuming values. Actors are attached to the inputs and outputs of a data flow diagram. Actors are also called as terminators as they act as source and sink for data.

4.2.1.1.2 Process

Processes are work or actions performed on incoming data flows to produce outgoing data flows. These show data transformation or change. Data coming into a

process must be "worked on" or transformed in some way. Thus, all processes must have inputs and outputs. In some cases, data inputs or outputs will only be shown at more detailed levels of the diagrams. Each process is always "running" and ready to accept data. Major functions of processes are computations and making decisions. Each process may have dramatically different timing: yearly, weekly, daily etc.

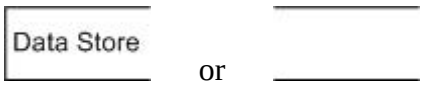
A process is depicted by a circle as shown below:

Every process is named. Processes are named with one carefully chosen verb and an object of the verb. There is no subject. Name is not to include the word "process". Each process should represent one function or action. If there is an "and" in the name, you likely have more than one function (and process). For example, get invoice, update customer and create Order. Processes are numbered within the diagram as convenient. Levels of detail are shown by decimal notation. For example, top level process would be Process 4, next level of detail Processes 4.1, and so on. Processes should generally move from top to bottom and left to right.

4.2.1.1.3 Data Store

Data stores are repository for data that are temporarily or permanently recorded within the system. It is an "inventory" of data. These are common link between data and process models. Only processes may connect with data stores.

There can be two or more systems that share a data store. This can occur in the case of one system updating the data store, while the other system only accesses the data. Data stores are represented by open rectangle or two parallel lines as shown below.



Data stores are named with an appropriate name, not to include the word "file", Names should consist of plural nouns describing the collection of data. Like customers, orders, and products. These may be duplicated.

They store data for later use. They do not generate any operation on its own but can respond to request. That is why they are passive objects in a data flow diagram.

4.2.1.1.4 Data Flow

Data flow represents the input (or output) of data to (or from) a process, data store or an actor. Data flow only data, not control. Represent the minimum essential data the process needs. Using only the minimum essential data reduces the dependence between processes. Data flows must begin and/or end at a process.

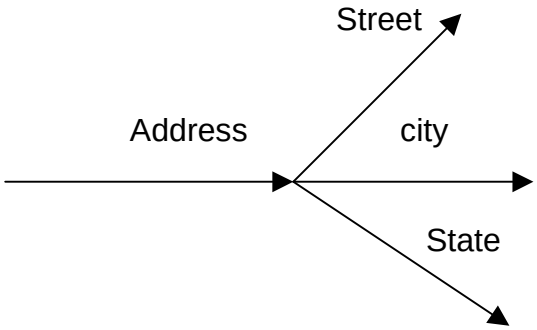
Data flows are always named. Name is not to include the word "data". It should be given unique names. Names should be some identifying noun. For example, marks, order, payment, complaint, registration no.

A data flow is represented by an arrow as shown in Fig below.

An arrow between the producer and the consumer of the data value represents a data flow. Arrow is labeled with description of data. Data can be elementary or aggregate. Input arrow indicates storing data in the data store and output arrow indicates accessing of data from data store.

Elementary data can not be decomposed into its meaningful constituents. For example, roll no, pin code, and quantity. Aggregate data can be decomposed into its meaningful constituents. For example, name can be decomposed into first name, middle name and last name. Sometimes an aggregate value is split into its constituents, each of which goes to a different process. A fork in the path as shown below is used to do this. Reverse can also be done. That is elementary data coming

from different sources can be aggregated. This is done by reversing the arrows in the diagram below.



4.2.1.2 Control Flow

Control flow is a Boolean value in the DFD that affects whether a process is evaluated or not. The control flow is not an input value to the process. It is represented by a dotted line from a process originating the Boolean value to the process being controlled as shown in fig below. This DFD is for a withdrawal from a bank account. The customer supplies a password and an amount. The update (withdrawal) can occur only when password is OK, which is shown as control flow in the diagram.

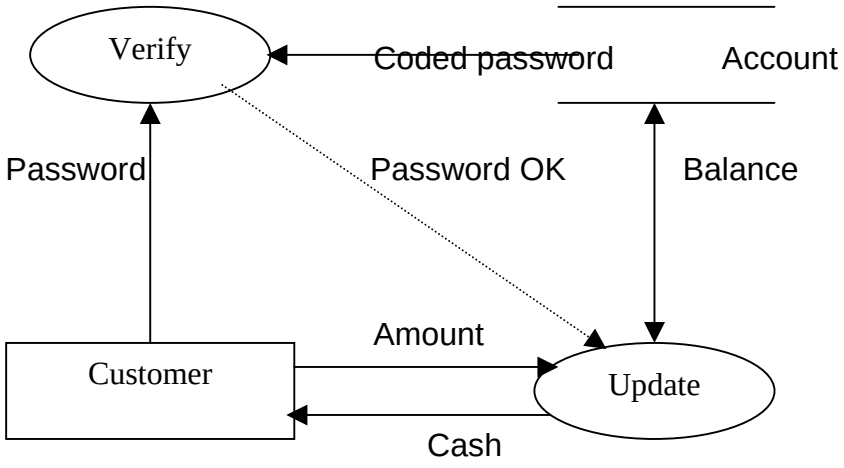


Figure: Control flow

4.2.3 Examples of DFDs

Example1: An example of a Data Flow Diagram - DFD for a system that pays workers is shown in the figure below. In this DFD there is one basic input data flow, the weekly time sheet, which originates from the source worker. The basic output is the pay check, the sink for which is also the worker. In this system, first the employee's record is retrieved, using the employee ID, which is contained in the time sheet. From the employee record, the rate of payment and overtime are obtained.

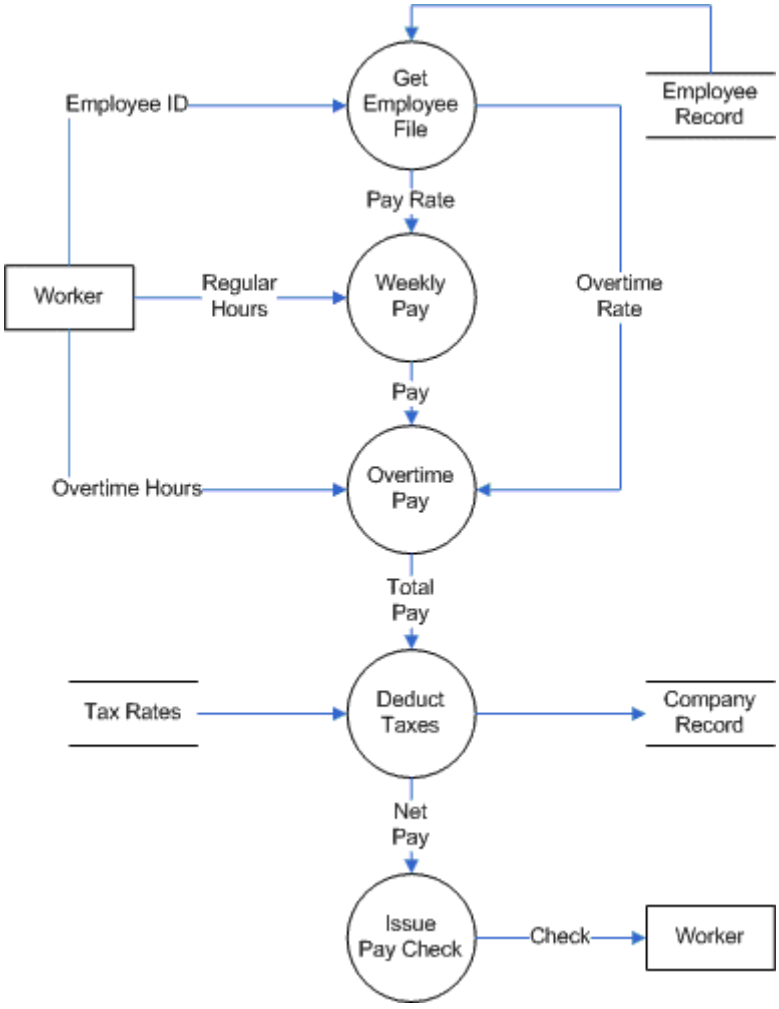
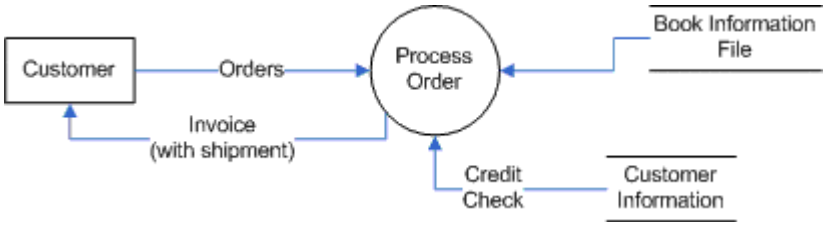


Figure: DFD of a system that pays workers.

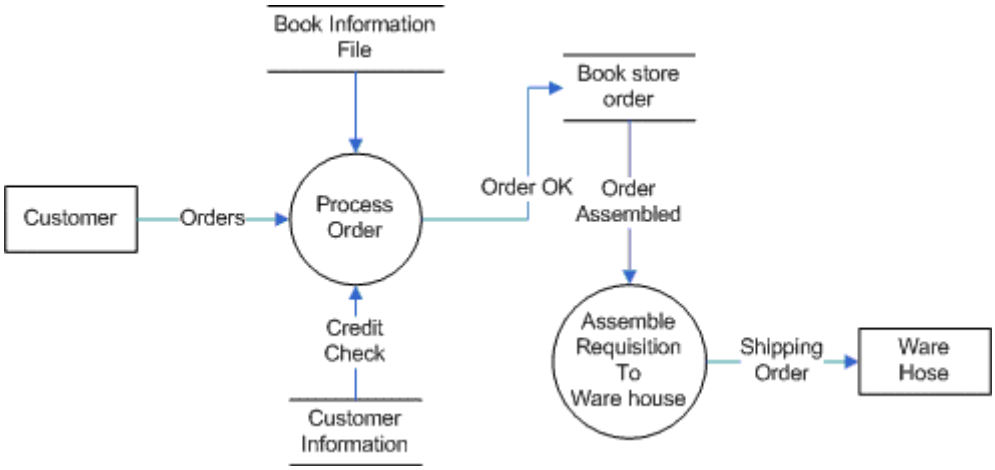
These rates and the regular and overtime hours (from the time sheet) are used to complete the payment. After total payment is determined, taxes are deducted. To computer the tax deduction, information from the tax rate file is used. The amount of tax deducted is recorded in the employee and company records. Finally, the

paycheck is issued for the net pay. The amount paid is also recorded in company records.

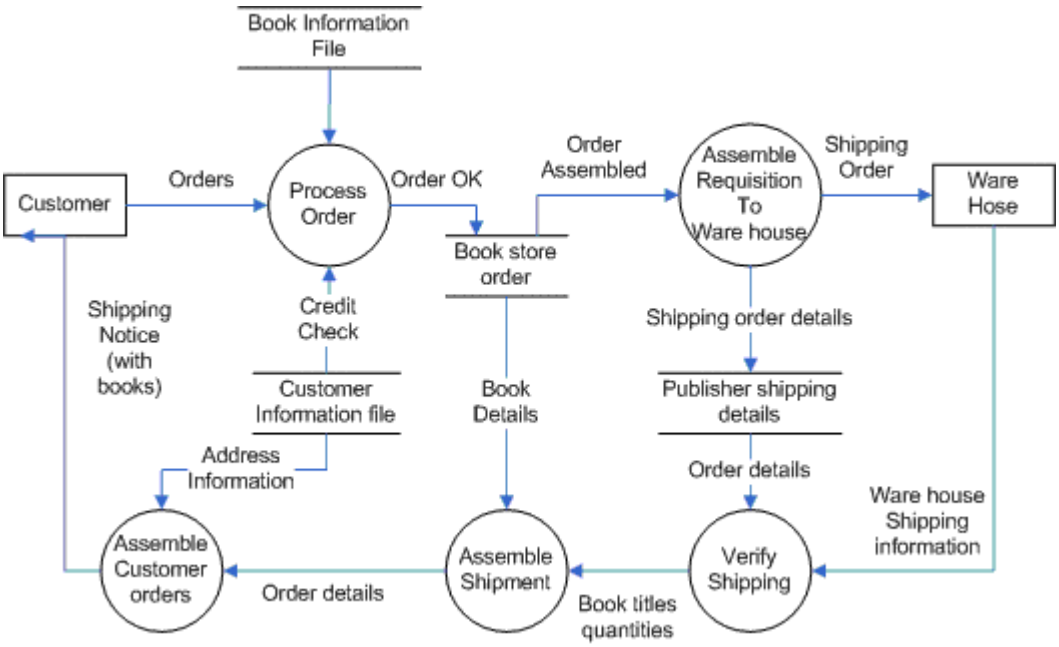
Following are the set of DFDs drawn for the General model of publisher's present ordering system.



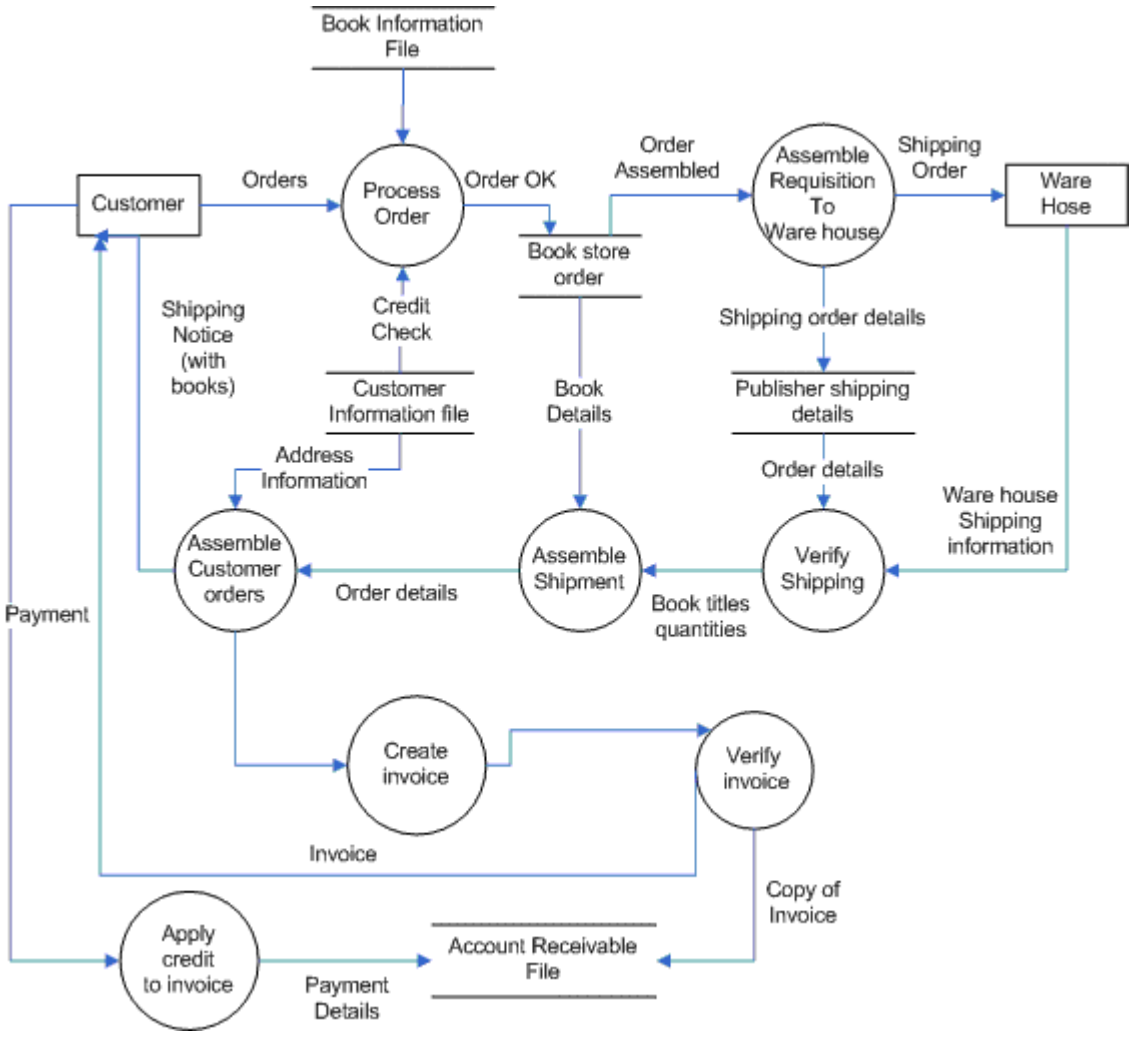
First Level DFD



Second Level DFD - Showing Order Verification & credit check



Third Level DFD - Elaborating an order processing & shipping



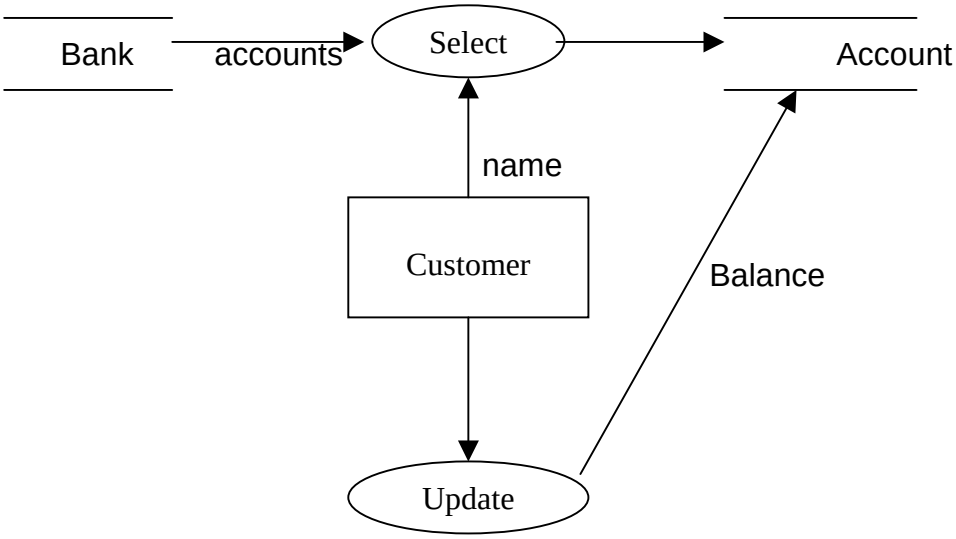
Fourth level DFD: Completed DFD, Showing Account Receivable Routine.

From the level one it shows the publisher's present ordering system. Let's expand process order to elaborate on the logical functions of the system. First, incoming orders are checked for correct book titles, author's names, and other information and then batched into other book orders from the same bookstore to determine how may copies can be shipped through the ware house. Also, the credit status of each book stores is checked before shipment is authorized. Each shipment has a shipping notice detailing the kind and numbers of booked shipped. This is compared to the original order received (by mail or phone) to ascertain its accuracy. The details of the order are normally available in a special file or data store, called "Bookstore Orders". It is shown in the second level DFD diagram.

Following the order verification and credit check, a clerk batches the order by assembling all the book titles ordered by the bookstore. The batched order is sent to the warehouse with authorization to pack and ship the books to the customer. It is shown in the third level DFD diagram.

Further expansion of the DFD focuses on the steps in billing the bookstore shown in the fourth level DFD, additional functions related to accounts receivable.

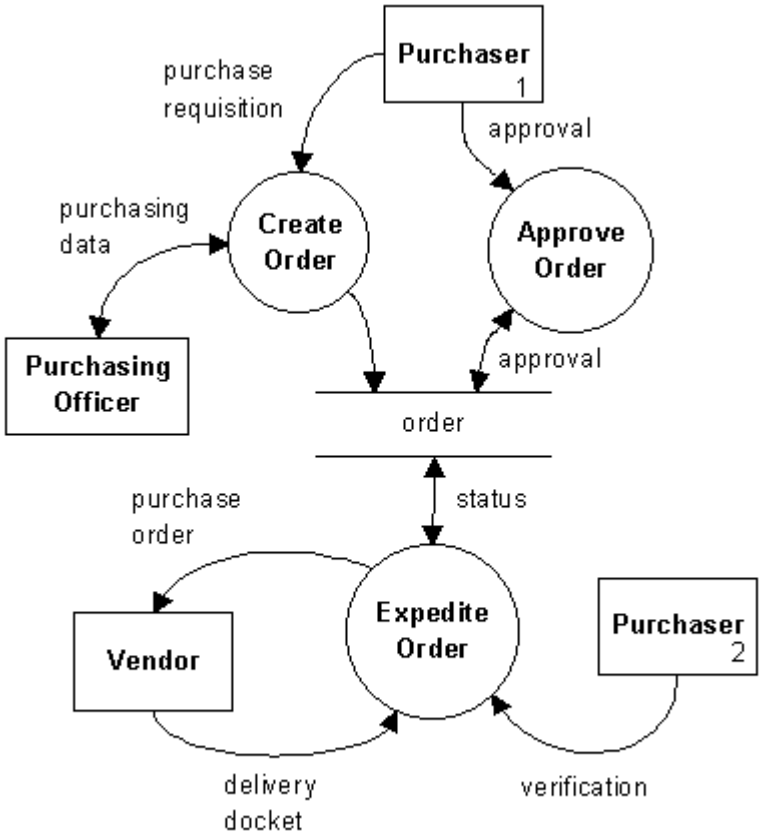
Example 3: DFD below shows updation of a bank account in a banking system.



Example 4: DFD below shows the purchase order processing system. There are four actors involved in this system – Purchasing Officer, Vendor, Purchaser 1 and Purchaser 2. Purchasing Officer creates purchase orders on receiving the purchase acquisition requests from the perspective purchasers. Purchasers approve orders and they can verify the status of the orders. All order details are kept in a database.

This functional model represents:

Functions. For example, create order and approve order.
Data flows. For example, purchasing data flows from the purchasing officer to the create order function.
External entities. For example, a vendor is an entity external to the system.
Data stores. For example, orders are kept in an order data store



4.2.2 Data Dictionary and Meta Data

In the data flow diagrams, we have given names to data flows, processes and data stores. Although the names are descriptive of the data, they do not give details. So following the DFD, the interest is to build some structures place to keep details of the contents of data flows, processes and data stores. Here comes the concept of data dictionary.

A data dictionary is a structured repository of data about data. It is a set of rigorous definitions of all DFD data elements and data structures. To define the data structure, different notations are used. These are similar to the notations for regular expression. Essentially, besides sequence or composition (represented by +) selection and iteration are included. Selection (represented by vertical bar "|") means one or the other, and repetition (represented by "*") means one or more occurrences.

The data dictionary for the DFD of system that pays to workers given above is created as shown below:

Weekly timesheet = Employee_Name + Employee_ID + {Regular_hours + overtime_hours}

Pay_rate = {Horly | Daily | Weekly} + Rupees_amount

Employee_Name = Last + First + Middle_Initial

Employee_ID = digit + digit + digit + digit

Most of the data flows in the DFD are specified here. Some of the most obvious ones are not shown here. The data dictionary entry for weekly timesheet specifies that this data flow is composed of three basic data entities - the employee name, employee ID and many occurrences of the two - tuple consisting of regular hours and overtime hours. The last entity represents the daily working hours of the worker. The data dictionary also contains entries for specifying the different elements of a data flow.

Once we have constructed a DFD and its associated data dictionary, we have to somehow verify that they are "correct". There can be no formal verification of a DFD, because what the DFD is modeling is not formally specify anywhere against which verification can be done. Human processes and rule of thumb must be used for verification. In addition to the walkthrough with the client, the analyst should look for common errors. Some common errors are

- Unlabeled data flows.
- Missing data flows: Information required by a process is not available.
- Extraneous data flows: Some information is not being used in the process
- Consistency not maintained during refinement
- Missing processes
- Contains some control information

The DFDs should be carefully scrutinized to make sure that all the processes in the physical environment are shown in the DFD. It should also be ensured that none of the data flows is actually carrying control information.

Meta Data: It is loosely defined as data about data. Metadata is a concept that applies mainly to electronically archived or presented data and is used to describe the: a) definition, b) structure and c) administration of data files with all contents in context to ease the use of the captured and archived data for further use. For example, a web page may include metadata specifying what language it's written in, what tools were used to create it, where to go for more on the subject and so on.

Metadata is defined as data providing information about one or more other pieces of data, such as:

- means of creation
- purpose of the data
- time and date of creation
- creator or author of data
- placement on a network (electronic form) where the data was created,
- What standards used etc.

For example: A digital image may include metadata that describes how large the picture is, the color depth, the image resolution, when the image was created, and other data. A text document's metadata may contain information about how long the document is, who the author is, when the document was written, and a short summary of the document.

Metadata is data. As such, metadata can be stored and managed in a database, often called a registry or repository. However, it is impossible to identify metadata just by looking at it. We don't know when data is metadata or just data.

Metadata is structured data which describes the characteristics of a resource. It shares many similar characteristics to the cataloguing that takes place in libraries, museums and archives. The term "meta" derives from the Greek word denoting a nature of a higher order or more fundamental kind. A metadata record consists of a number of pre-defined elements representing specific attributes of a resource, and each element can have one or more values. Below is an example of a simple metadata record:

Element name	Value
Title	Web catalogue
Creator	Rajender Nath
Publisher	G.J.U. Hisar
Identifier	www.gju.ac.in.metadata.html
Format	Text/html
Relation	Distance Education Web site

Each metadata schema will usually have the following characteristics:

- a limited number of elements
- the name of each element
- the meaning of each element

Typically, the semantics is descriptive of the contents, location, physical attributes, type (e.g. text or image, map or model) and form (e.g. print copy, electronic file). Key metadata elements supporting access to published documents include the originator of a work, its title, when and where it was published and the subject areas it covers. Where the information is issued in analog form, such as print material, additional metadata is provided to assist in the location of the information, e.g. call numbers used in libraries. The resource community may also define some logical grouping of the elements or leave it to the encoding scheme. For example, Dublin Core may provide the core to which extensions may be added.

In a nutshell, metadata is structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. Metadata is often called data about data or information about information.

There are three main types of metadata:

Descriptive metadata: It describes a resource for purposes such as discovery and identification. It can include elements such as title, abstract, author, and keywords.
Structural metadata: It indicates how compound objects are put together, for example, how pages are ordered to form chapters.
Administrative metadata: It provides information to help manage a resource, such as when and how it was created, file type and other technical information, and who can access it.

4.2.3 Steps to Produce a DFD

- Identify and list external entities providing inputs/receiving outputs from system
- Identify and list inputs from/outputs to external entities
- Draw a context DFD

Defines the scope and boundary for the system and project

- Think of the system as a container (black box)
- Ignore the inner workings of the container
- Ask end-users for the events the system must respond to
- For each event, ask end-users what responses must be produced by the system
- Identify any external data stores
- Draw the context diagram

    i. Use only one process

    ii. Only show those data flows that represent the main objective or most common inputs/outputs

- identify the business functions included within the system boundary
- identify the data connections between business functions
- confirm through personal contact sent data is received and vice-versa
- trace and record what happens to each of the data flows entering the system (data movement, data storage, data transformation/processing)
- Draw an overview DFD

- Shows the major subsystems and how they interact with one another
- Exploding processes should add detail while retaining the essence of the details from the more general diagram

- Consolidate all data stores into a composite data store

- Draw middle-level DFDs

  - Explode the composite processes

- Draw primitive-level DFDs

  - Detail the primitive processes

  - Must show all appropriate primitive data stores and data flows

- verify all data flows have a source and destination;
- verify data coming out of a data store goes in;
- review with "informed";
- Explode and repeat above steps as needed.

Balancing DFDs

- Balancing: child diagrams must maintain a balance in data content with their parent processes
- Can be achieved by either:
- exactly the same data flows of the parent process enter and leave the child diagram, or
- the same net contents from the parent process serve as the initial inputs and final outputs for the child diagram or
- the data in the parent diagram is split in the child diagram

Rules for Drawing DFDs

- A process must have at least one input and one output data flow

- A process begins to perform its tasks as soon as it receives the necessary input data flows
- A primitive process performs a single well-defined function
- Never label a process with an IF-THEN statement
- Never show time dependency directly on a DFD
- Be sure that data stores, data flows, data processes have descriptive titles. Processes should use imperative verbs to project action.
- All processes receive and generate at least one data flow.
- Begin/end data flows with a bubble.

Rules for Data Flows

- A data store must always be connected to a process
- Data flows must be named
- Data flows are named using nouns Customer ID, Student information
- Data that travel together should be one data flow
- Data should be sent only to the processes that need the data

Use the following additional guidelines when drawing DFDs

- Identify the key processing steps in a system. A processing step is an activity that transforms one piece of data into another form.
- Process bubbles should be arranged from top left to bottom right of page.
- Number each process (1.0, 2.0, etc). Also name the process with a verb that describes the information processing activity.
- Name each data flow with a noun that describes the information going into and out of a process. What goes in should be different from what comes out.
- Data stores, sources and destinations are also named with nouns.
- Realize that the highest level DFD is the context diagram. It summarizes the entire system as one bubble and shows the inputs and outputs to a system
- Each lower level DFD must balance with its higher level DFD. This means that no inputs and outputs are changed.

- Think of data flow not control flow. Data flows are pathways for data. Think about what data is needed to perform a process or update a data store. A data flow diagram is not a flowchart and should not have loops or transfer of control. Think about the data flows, data processes, and data storage that are needed to move a data structure through a system.
- Do not try to put everything you know on the data flow diagram. The diagram should serve as index and outline. The index/outline will be "fleshed out" in the data dictionary, data structure diagrams, and procedure specification techniques.

4.2.4 Different Types of Keys

This concept is related to relational data base management systems. A relation is two dimensional table that has rows called tuples and columns called domains. There are different types of keys, namely Primary keys, alternate keys, etc. , which are described below.

Primary key: Within a given relation, there can be one attribute with values that are unique within the relation that can be used to identify the tuples of that relation. That attribute is said to be primary key for that relation. For example, in a Student relation roll no is a primary key.

Composite primary key: Not every relation will have a single-attribute primary key. There can be a possibility that some combination of attributes when taken together have the unique identification property. These attributes as a group is called composite primary key. A combination consisting of a single attribute is a special case.

Existence of such a combination is guaranteed by the fact that a relation is a set. Since sets don’t contain duplicate elements, each tuple of a relation is unique with respect to that relation. Hence, at least the combination of all attributes has the unique identification property.

- In practice it is not usually necessary to involve all the attributes-some lesser combination is normally sufficient. Thus, every relation does have a primary (possibly composite) key.
- Tuples represent entities in the real world. Primary key serves as a unique identifier for those entities.

Candidate key: In a relation, there can be more than one attribute combination possessing the unique identification property. These combinations, which can act as primary key, are called candidate keys.

EmpNo	SocSecurityNo	Name	Age
1011	2364236	Harry	21
1012	1002365	Sympson	19
1013	1056300	Larry	24

Table: having “EmpNo” and “SocSecurityNo” as candidate keys

Alternate key: A candidate key that is not a primary key is called an alternate key. In fig. 8.6 if EmpNo is primary key then SocSecurityNo is the alternate key.

4.3 Summary

- The functional model describes computations and specifies those aspects of the system concerned with transformations of values - functions, mappings, constraints, and functional dependencies.
- The functional model is represented graphically with multiple data flow diagrams, which show the flow of values from external inputs, through operations and internal data stores, to external outputs.
- DFDs play a major role in designing of the software and also provide the basis for other design-related issues. Some of these issues are addressed in this chapter. All the basic elements of DFD are further addressed in the designing phase of the development procedure

- Metadata is structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. Metadata is often called data about data or information about information.

- Within a given relation, there can be one attribute with values that are unique within the relation that can be used to identify the tuples of that relation. That attribute is said to be primary key for that relation.

4.4 Suggested Readings/Reference Materials

1. Object-Oriented Modeling and Design with UML, M. Blaha, J. Rumbaugh, Pearson Education-2007
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson-2007
3. Object Oriented Analysis & Design, Grady Booch, Addison Wesley-1994
4. Timothy C. Lethbridge, Robert Laganiere, Object Oriented Software Engineering, TMH, 2004

4.5 Self-Assessment Questions

1. What is functional model of OMT? How is it related to other models of OMT? Explain.
2. What is DFD? What are elements of DFD? Explain the purpose of DFD with a suitable example.
3. Distinguish between control flow and data flow.
4. Discuss the steps to draw a DFD systematically.
5. What is metadata? Explain its usefulness.
6. What is data dictionary? How is it created?
7. What is a key? Explain different types of keys with suitable examples.

Writer: Dr. Rajender Nath

Vetter: Dr. Dharminder Kumar

Chapter 5

System Design

Structure

5.0 Introduction

After analyzing the problem, one must decide how to approach the system design. During system design developers devise the high-level strategy (system architecture) for solving the problem and building a solution and make decisions about the organization of the system into subsystems, the allocation of subsystems to hardware and software and major policy decisions that form the basis for class design.

5.1 Objectives

In this chapter you will learn what is system design? What is physical and logical design? What are different types of design decisions made by the designer? What is software architecture? What are different types of architectural styles?

5.2 Presentation of Contents

5.2.1 System Design

Systems design is the process or art of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. It is a process where developers decide the overall structure and style of the system. System design is the first stage for solving any problem. The emphasis is given on how the problem is to be solved. The system design process is generally divided into two sub-phases – logical design and physical design.

The logical design of a system pertains to an abstract representation of the data flows, inputs and outputs of the system. This is often conducted via modeling, which involves a simplistic (and sometimes graphical) representation of an actual system. In the context of systems design, modeling can undertake the following forms, including: Data flow diagram, Entity Life Histories and Entity-relationship diagrams etc.

The physical design relates to the actual input and output processes of the system. This is laid down in terms of how data is input into a system, how it is verified/authenticated, how it is processed, and how it is displayed as output.

Physical design, in this context, does not refer to the tangible physical design of an information system. To use an analogy, a personal computer's physical design involves input via a keyboard, processing within the CPU, and output via a monitor, printer, etc. It would not concern the actual layout of the tangible hardware, which for a PC would be a monitor, CPU, motherboard, hard drive, modems, video/graphics cards, USB slots, etc.

In the OMT methodology, system design is one of the phases of the software development life cycle. During this phase, developers decide the overall structure and style of the system. The system architecture determines the organization of the system into subsystems. During system design, the following design decisions are to be made.

i. Estimation of system performance
ii. Make a reuse plan.
iii. Organize the system into subsystems
iv. Identify concurrency inherent in the problem
v. Allocate subsystems to hardware.
vi. Manage data stores.
vii. Handle global resources.
viii. Choose a global control strategy
ix. Handle boundary conditions
x. Set trade off priorities
xi. Select an architectural style.

Now, each of these design decision will be discussed in detail.

5.2.1.1 Estimating System Performance

A rough performance estimate (generally referred as “Back of the envelop”) should be calculated for a new system. The purpose is not to achieve high accuracy, but merely to determine if the system is feasible. The calculation should be fast and should also involve common sense. You can estimate, number of transactions to be processed by the system, response time needed, storage requirements etc.

5.2.1.2 Making a Reuse Plan

Reuse is often cited as an advantage of OO technology. But reuse does not happen automatically. There are two different aspects – using existing things and creating reusable new things. It is much easier to use existing things than to design new things for uncertain use to come. Reusable things include models, libraries, frameworks and patterns. Reuse of models is often the post practical form of reuse. The logic in a model can apply to multiple problems. Now, we discuss library, framework and pattern based reuse.

a) Library based Reuse

It is a collection of prewritten, ready-made software routines that act as templates for programmers to use in writing object-oriented application programs. Class libraries are typically used to provide GUI functions like buttons, scroll bars, icons and windows. Class libraries greatly simplify the work of the programmer who can use the pre-tested code instead of having to write new code. The classes in the library are useful in many contexts. The collection of classes must be carefully organized, so that users can find them.

For example, the .NET Framework class library is a library of classes, interfaces, and value types that are included in the Microsoft .NET Framework SDK. This

library provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built.

A good library satisfies the following qualities:

- Coherence
- Completeness
- Consistency
- Efficiency
- Extensibility
- Genericity

Many problems may arise when classes are reused from many sources. Such problems are sometime very difficult to solve by specializing a class and adding codes. These problems effectively limit the reuse of classes from the class libraries. The following types of problems may be encountered in reusing class libraries.

Argument validation: Arguments can be validated in two ways – collectively or individually. In collective validation, first user enters all the arguments and then they are checked collectively. The collective validation is appropriate for command interfaces. In individual validation, the argument is checked as and when it is entered. Combination of the two approaches should be avoided.

Error handling: Different class libraries may use different error-handling techniques. For example, a method in one class may return the error code to the calling routine while a method in another class in another library may deal with the error directly.

Control paradigms: Different class libraries may us different control paradigms. For example, one library uses procedure-driven control and another uses event-driven control. Combining these two types of control into one application is difficult and awkward.

Group operations: If you want to delete objects in groups then that library must have group-delete function. If that function is not available then you can not reuse the classes of that library. Group operations are often inefficient and incomplete.

Garbage collection: Different libraries may use different techniques to allocate memory and to collect garbage. In Java, there is automatic garbage collection and while in C++, it is the responsibility of the application.

Name collisions: classes from different libraries may have same name for public identifiers causing names collisions. To resolve this problem, most of the class libraries add a distinguishing prefix to names to reduce the chance of names collision.

b) Frameworks based Reuse

In general, a framework is a real or conceptual structure intended to serve as a support or guide for the building of something that expands the structure into something useful. In computer science, a framework is skeletal structure of a program that must be elaborated to build a complete application. It is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code, thus providing specific functionality. A framework is generally more comprehensive than a protocol and more prescriptive than a structure.

Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined Application Programming Interface (API), yet they contain some key distinguishing features that separate them from normal libraries. Software frameworks typically contain considerable housekeeping and utility code in order to facilitate user applications, but generally focus on specific problem domains.

Following are some of the problem domains where frameworks have been used:

- Artistic drawing, music composition, and mechanical
- Compilers for different programming languages and target machines
- Financial modeling applications
- Earth system modeling applications
- Decision support systems
- Media playback and authoring
- Web applications
- Middleware

Examples of frameworks that are currently used or offered by standards bodies or companies include:

- Resource Description Framework – it is a set of rules from the World Wide Web Consortium for how to describe any Internet resource such as a Web site and its content.
- Internet Business Framework – it is a group of programs that form the technological basis for the mySAP product from SAP, the German company that markets an enterprise resource management line of products.
- Sender Policy Framework: it is a defined approach and programming for making e-mail more secure.
- Zachman framework – it is a logical structure intended to provide a comprehensive representation of an information technology enterprise that is independent of the tools and methods used in any particular IT business

In an object-oriented environment, a framework consists of abstract and concrete classes. Instantiation of such a framework consists of composing and subclassing the existing classes. When developing software based on framework reuse, the new system is built by customizing and/or extending the generic architecture defined by the framework.

For example, The Microsoft .NET Framework is a software framework for Microsoft Windows operating systems. It includes a large library, and it supports several programming languages which allows language interoperability (each language can utilize code written in other languages.) The .NET library is available to all the programming languages that .NET supports.

Software frameworks have these distinguishing features that separate them from libraries or normal user applications:

- **Inversion of control:** In a framework, unlike in libraries the overall program's flow of control is not dictated by the caller, but by the framework.
- **Default behavior** - A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-operations.
- **Extensibility** - A framework can be extended by the user usually by selective overriding or specialized by user code providing specific functionality.
- **Non-modifiable framework code** - The framework code, in general, is not allowed to be modified. Users can extend the framework, but not modify its code.
- There are different types of software frameworks: conceptual, application, domain, platform, component, service, development, etc.

b) Pattern based Reuse

A pattern is a proven solution to general problem. Various patterns target different phases of the software development life cycle. There are patterns for analysis, architecture, design and implementation.

Design patterns are general solutions to problems in object-oriented programming. They will not solve a specific problem, but they provide a sort of architectural outline that may be reused in order to speed up the development process of a program. Design patterns have provided the stepping stone for computer science to truly enter the engineering field.

There are many types of design patterns: structural design patterns, computational patterns, algorithm strategy patterns, implementation strategy patterns and execution patterns. Structural patterns address concerns related to the high level structure of an application being developed. Computational patterns address concerns related to the identification of key computations. Algorithm strategy patterns address concerns related to high-level strategies that describe how to exploit application characteristic on a computation platform. Implementation strategy patterns related to the realization of the source code to support (i) how the program itself is organized and (ii) the common data structures specific to parallel programming. Execution patterns address concerns related to the support of the execution of an application, including the strategies

Examples of structural patterns are: Adapter Pattern – it converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Bridge Pattern – it decouples an abstraction from its implementation allowing the two to vary independently.

Examples of creational patterns are: Abstract Factory – it provides an interface for creating families of related or dependent objects without specifying their concrete classes. Builder Pattern – it separates the construction of a complex object from its representation allowing the same construction process to create various representations.

A pattern is different from a framework. A pattern is typically a small number of classes and relationships where as a framework is much broader in scope and covers an entire subsystem or application.

5.2.1.3 Organizing a system into Subsystems

A subsystem may be defined as a group of classes, associations, operations, events and constraints that are interrelated and have a well-defined interface. For example file system is a subsystem of an operating system. The interface of a subsystem defines the form of all interactions and how the information will flow across subsystem boundaries. A system may be divided into smaller subsystems and each subsystem may further be divided in to smaller subsystems of its own.

Relationships between subsystems are: there are two types of relationships between subsystems: a) Client-Server and b) Peer-to-peer

a) Client–Server relationship: In client-server relationship client calls on the server for performing certain task and server replies back with the result. The client–server characteristic describes the relationship of cooperating programs in an application. The server component provides a function or service to one or many clients, which initiate requests for such services.

For instance, when someone checks their bank account from a computer, the computer acts as the client and forwards the request to an on-line bank (server). The bank's program (server) then serves a response to the user in which the

requested information is displayed. The client/server model has become a predominate function in network computing. Many applications intended for both personal and business use were designed with this concept in mind. In most cases, both the client and server are part of a larger program or system. As it relates to the Internet, a web browser acts as a client that requests specific services to a web server, a process known as HTTP (Hypertext Transfer Protocol). In similar fashion, a computer that relies on TCP/IP for a connection enables a user to make client requests to FTP servers found in other computers connected to the Internet.

Peer-to-peer relationship: In peer-to-peer relationship, each subsystem may call on the others. The communication in this case can be much complex because individual subsystems may not be aware about each other. Designing such type of systems may lead to subtle design errors.

Layers and Partitions: The decomposition of systems into subsystems may be organized as a sequence of horizontal layers or vertical partitions.

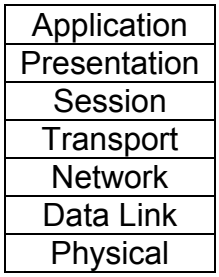
Layers: A layered system is an ordered set of virtual worlds, each built in terms of the ones below it and providing the implementation basis for the ones above it. Layered architecture comes in closed or open.

In a closed architecture, each layer is built only in terms of the immediate lower layer. This reduces dependencies between layers and allows changes to be made most easily. In an open architecture, a layer can use the features of any lower layer to any depth. This reduces the need to redefine operations at each level.

In a layered architecture model, classes within each subsystem layer provide services to the layer above it. Ideally, this knowledge is one-way: each layer knows about the layer below it, but the converse is not true. An example of a

layered system architecture is the ISO Reference Model for Open Systems Interconnection (OSI) as shown in Figure below.

Example of Subsystem Layers
(ISO Reference Model for OSI)



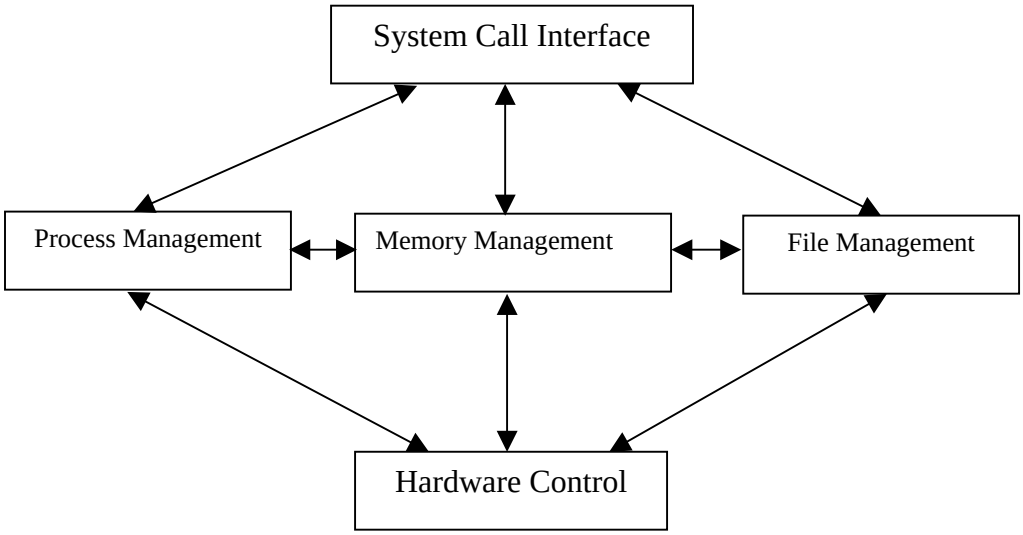
Partitions: Partitions vertically divide a system into several independent or weakly subsystems, each providing one kind of service. One difference between layers and partitions is that layers vary in their level of abstraction, but partitions merely divide a system into pieces, all of which have a similar level of abstraction. You can decompose a system into subsystems by combining layers and partitions. Layers can be partitioned and partitions can be layered. Most large systems require a mixture of layers and partitions.

In a partitioned architecture model, subsystems collaborate with peer subsystems at the same level of abstraction. Like layers, each subsystem partition represents a group of classes that share similar functionality at the same level of abstraction. Classes within each subsystem partition can communicate with other classes in that partition. Selected subsystem classes (assuming weak coupling) can communicate with designated classes from subsystem partitions that are at the same level as the subject subsystem.

Partitioning effects a vertical structuring of peer subsystems. For example, many operating system (OS) architectures partition OS services into subsystems for file management, process control, and memory management as shown in Figure below. Since the OS subsystems are at the same level of abstraction, they can interact with each other in a peer-to-peer manner. For example, when an

executable file is loaded into memory for execution, the file system, the memory management system, and the process control system all interoperate.

Example of Subsystem Partitions
(OS Kernel Service)



Identifying subsystems: The following steps can be used to identify subsystems:

- Look at the analysis model as the staring point
- Look for separation of design concerns
- Look for functionally related classes and interfaces
- Look for classes interfaces and subsystems with many dependencies
- Look for large-grained components
- Look for reusable components
- Look for interfaces
- Look for software products that need to be wrapped up as a subsystem
- Look for legacy systems which need to be wrapped up as a subsystem
- Look for service subsystems
- Look for software products that need to be wrapped up as a subsystem.

5.2.1.4 Identifying Concurrency

Concurrency (or distribution) is an important issue during design process as it may affect the design of classes and their interfaces. Concurrency can be very important for improving the efficiency of a system. In the analysis model, as the real world and in hardware, all objects are concurrent. In an implementation, not all software objects are concurrent, because one processor may support many objects. One important goal of the system design is to identify the objects that must be active concurrently and the objects that have mutually exclusive activity

For concurrent applications, such as distributed and real-time applications, the following activities are performed:

- Identify inherent concurrency: - Two objects are inherently concurrent if they can receive events at the same time without interacting. Inherently concurrent subsystems need not be implemented as separate hardware units.

- Define concurrent Tasks: Many objects in a system are dependent on each other. By examining the state diagrams of individual objects and exchange of events among them, many objects can be folded in to a single thread of model. A thread of control is a path through a set of state diagrams on which only a single object at a time is active.

- Make decisions about subsystem structure and interfaces. Develop the overall software architecture. Structure the application into subsystems.

- Make decisions about how to structure the distributed application into distributed subsystems, in which subsystems are designed as configurable components. Design the distributed software architecture by

decomposing the system into distributed subsystems and defining the message communication interfaces between the subsystems.

- Make decisions about the characteristics of objects, in particular, whether they are active or passive. For each subsystem, structure the system into concurrent tasks (active objects). During task structuring, tasks are structured using the task structuring criteria, and task interfaces are defined. Make decisions about the characteristics of messages, in particular, whether they are asynchronous or synchronous (with or without reply).

- Make decisions about class interfaces. For each subsystem, design the information hiding classes (passive classes). Design the operations of each class and the parameters of each operation. Use inheritance to develop class hierarchies. . Develop the detailed software design, addressing detailed issues concerning task synchronization and communication, and the internal design of concurrent tasks.

- For real-time applications, analyze the performance of the design. Apply real-time scheduling to determine if the concurrent real-time design will meet performance goals. If not, investigate alternative software architectures.

5.2.1.5 Allocation of Subsystems

You must allocate each concurrent subsystem to a hardware unit, either a general-purpose processor or specialized functional unit. The system designer must do the following:

- Estimate performance needs and the resources needed to satisfy them.
- Choose hardware or software implementation for subsystems

- Allocate software subsystems to processors to satisfy performance needs and minimize inter processor communication
- Determine the connectivity of the physical units that implement the subsystems.
- Consider the connection between nodes and communication protocols to be used.
- Consider the need for redundant processing.
- Identify any interface implied by deployment.

UML deployment diagram can be used to present the above-mentioned steps. A deployment diagram shows how the systems will be physically distributed on the hardware.

5.2.1.6 Estimating Hardware Resource Requirements

To increase the performance of system, multiple processors or hardware functional units may be used. The number of processors required depends on the volume of computations and the speed of the machine. The system designer must estimate the required CPU processing powered by computing the steady state as the product of the number of transactions per second and the time required to process a transaction.

5.2.1.7 Making Hardware-Software Trade-offs

While implementing the system a decision has to be made regarding which subsystems will be implemented as hardware and which in software. There are two main reasons for implementing the subsystems in hardware.

- **Cost:** Following the advances in technology**,** the cost of hardware has drastically come down. Today it is easier to buy a floating-point chip than to implement the same functionality in the software.

- **Performance:** Dedicated hardware gives high performance than the same task carried using software instructions. For example chips that perform Fourier transformation are widely used in signal-processing applications.

5.2.1.8 Allocating Tasks to Processors

The system designer must allocate the tasks for various subsystems to processors. Each concurrent subsystem should be allocated to an independent process or processor. The system designer must do the following:

There are several reasons for assigning tasks to processors.

- **Logistics:** Certain tasks can be carried out at only specific physical locations to avail the benefits of specialized and dedicated hardware.

- **Communication Limits:** The response time or data flow rate exceeds the available communication bandwidth between a task and a piece of hardware. For example, high performance graphics devices require tightly coupled controllers because of their high internal data generation rates.

- **Computation Limits:** The amount of computation that can be carried out by any processor is limited only; so several processors may be employed to carry out the tasks. Highly interactive subsystems can be assigned to same processors to minimize the cost of communication. The independent subsystems can be allocated to separate processors.

5.2.1.9 Determining Physical Connectivity

System designer must decide the arrangement and form of connections among various units. The following decisions must be made.

- **Connection Topology:** Topology refers to the type of arrangements for physical units. The topology must be chosen keeping in mind the no. of physical units and to minimize the cost of communications.

- **Repeated Units:** System designer can enhance the system performance by using several copies of same physical unit. In the topology, the repeated units can be chosen as a regular pattern such as a linear sequence, a matrix, a tree, or a star. System designer must consider the expected arrival patterns of data and the proposed parallel algorithm for processing it.

- **Communications**: It is more appropriate to make a decision during system design phase about the type of interactions and protocols for communication between physical units. For example the interactions may be asynchronous, synchronous, or blocking. Bandwidth and latency of communication channels also plays an important role for deciding about communication channels.

5.2.1.10 Management of Data Storage

System designer must decide from among several alternatives for data storage that can be used separately or in combination of data structures, files and databases. This involves identifying the complexity of the data, the size of the data, the type of access to data (single user or multiple user), access times and portability. Having considered these issues, the designer must make the decision about whether he data can be held in flat files or in relational or object databases.

The following points must be considered when selecting an appropriate approach to data storage:

- Data persistence: Does data need to be persistent? If so then files, serialization or databases must be considered.

- Purchase cost: If your system requires a database system then it is likely that this will increase the cost of the system. It may also involve licensing agreements.
- Life cycle cost: This reflects costs such as purchase, development, deployment, operating and maintenance costs.
- Amount of data: The more data the designer has the more carefully he needs to think about how it should be stored and accessed.
- Performance: In main memory, the storage provides the faster data access while files are likely to provide the poorest performance.
- Extensibility: How easy it will be to extend your application in the future for a given the method of data storage.
- Concurrent access: Whether the designer needs concurrent access or not.
- Crash recovery: How will you recover your data if the system crashes.
- Distribution: Will the data need to be distributed among a number of sites? If yes, then make the decision accordingly.

Different kinds of data stores provide trade-offs among cost, access time, capacity and reliability.

Files provide cheap, simple and permanent storage and are easy to work with. However, file on one system may not be useful when transported to another system because of varying file implementations over different hardware types. Files may be used in random access mode or sequential access mode. Sequential file format is mostly a standard format and is easy to handle. Whereas, the commands and storage formats for random access files and index files vary in their formats.

The kind of data that belongs to files can be characterized as follows:

- Data with high volumes and low information density (such as archival files or historical data)

- Modest quantities of data with simple structure.
- Data that are accessed sequentially.
- Data that can be fully read into the memory.

Another alternative to store data is to use database management systems (DBMSs). There are various kinds of DBMSs like relational, object oriented, network and hierarchical etc. Databases make applications easier to port to different hardware and operating system platforms. One disadvantage of DBMSs is their complex interface. The kind of data that belongs to a database can be characterized as follows:

- Data to be stored exists in large quantity.
- Data that is to be kept in store for a very longer period of time.
- Data that must be secured against unauthorized and malicious access.
- Data that must be accessed by multiple application programs.
- Data that require updates at fine levels of detail by multiple users

5.2.1.11 Handling Global Resources

The system designer must identify global resources and determine mechanisms for controlling access to them. There are several kinds of global resources:

- **Physical system**: Example includes processors, tape drives and communication channels.
- **Space**: Example includes keyboard, buttons on a mouse, display screen
- **Logical names**: Example includes object IDs, filenames, and class names.
- **Access to shared data:** Example includes Databases

Physical resource such as processors, tape drives etc. can control their own access by establishing a protocol for obtaining access. For a logical resource like

Object ID or a database, there arises a need to provide access in a shared environment without any conflicts. One strategy to avoid conflict may be to employ a guardian object which controls access to all other resources. Any request to access a resource has to pass through a guardian object only. Another strategy may be to partition a resource logically and assign subsets to different guardian objects for independent control. In a critical real time application passing the entire access requests or resources through a guardian object may be not be desirable and it may become necessary to provide direct access to resources. In this case, to avoid conflicts, a lock can be placed on the subsets of resources. A lock is a logical object associated with a shared resource which gives the right of accessing the resource to the lock holder. Guardian object can still exist to allocate the lock. However, direct access to resources must not be implemented unless it is absolutely desirable.

5.2.1.12 Choosing a Software Control Strategy

It is best to choose a single control style for the whole system. There are two kinds of control flows in a software system: External control and internal control. External control concerns the flow of externally visible events among the objects in the system. There are three kinds of external events: procedural-driven sequential, event-driven sequential and concurrent.

Procedural-driven Control: In a procedure-driven system, the control lies within the program code. Procedures request external input and then wait for it, when input arrives, control resumes within the procedure that made the call.

The major advantage of procedure-driven control is that it is easy to implement with conventional languages, the disadvantage is that it requires the concurrency inherent in objects to be mapped into a sequential flow of control.

Event-driven Control: In the sequential model, the control resides within a dispatcher or monitor that the language, subsystem or operating system provides. In event-driven, the developers attach application procedures to events and the dispatcher calls the procedures when the corresponding events occur. Usually event driven systems are used for external control in preference to procedure driven systems, because the mapping from events to program constructs is simpler and more powerful. Event driven systems are more modular and can handle error conditions better than procedure-driven systems.

For example, the GUI consists of many built in objects (like text boxes, tool icons menus etc). The user interacts with these GUI objects either through mouse clicks or through pressing keys on keyboard. These user interactions with GUI objects are called events and these event notifications are given to the program (whom the user is interacting with) by Windows operating system. Now the programmer's task is to write the code that executes on the occurrence of these events automatically, and this type of control is called event-driven control.

Concurrent system Control: Here control resides concurrently in several independent objects, each as a separate task. A task can wait for input, but other tasks continue execution. The operating system keeps track of the raised events while a task is being performed so that events are not lost. Scheduling conflicts among tasks are also resolved by the operating system.

Internal control refers to the flow of control within a process. It exists only in the implementation and therefore is neither inherently concurrent nor sequential.

5.2.1.13 Handling boundary Conditions

Although most of the system design concerns steady-state behavior system designer must consider boundary conditions as well and address issues like initialization, termination and failure (the unplanned termination of the system).

- **Initialization:** It refers to initialization of constant data, parameters, global variables, tasks, guardian objects, and classes as per their hierarchy. Initialization of a system containing concurrent tasks must be done in a manner so that tasks can be started without prolonged delays. There is quite possibility that one object has been initialized at an early stage and the other object on which it is dependent is not initialized even after considerable time. This may lead to halting of system tasks.

- **Termination:** Termination requires that objects must release the reserved resources. In case of concurrent system, a task must intimate other tasks about its termination.

- **Failure:** Failure is the unplanned termination of the system, which can occur due to system fault or due to user errors or due to exhaustion of system resources, or from external breakdown or bugs from external system. The good design must not affect remaining environment in case of any failure and must provide mechanism for recording details of system activities and error logs.

5.2.1.14 Setting trade-off Priorities

The system designer must set priorities that will be used to guide trade-offs for the rest of the design. For example system can be made faster using extra memory. Design trade-offs involve not only the software itself but also the process of developing it. System designer must determine the relative importance of the various criteria as a guide to making design trade-offs. Design trade-offs affect entire character of the system. Setting trade-offs priorities is at best vague. Priorities are generally specified as a statement of design philosophy.

5.2.2 Common Architectural Styles

Software architecture, according to ANSI/IEEE Standard 1471-2000, is defined as the “fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.”

The architecture of a system describes its gross structure. This structure illuminates the top-level design decisions, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal.

Several prototypical architectural styles are common in existing systems. Each of these is well suited to a certain kind of system. Some of the styles are discussed below in brief.

5.2.2.1 Batch Transformation

A data transformation executed once on an entire input set. It performs sequential computations. The application receives the inputs and the goal is to compute an answer; there is no ongoing interaction with the outside world. The steps involved in batch transformation are as follows.

- Break the overall transformation into stages, with each stage performing one part of the transformation.

- Prepare class models for the input, output and between each pair of successive stages. Each state knows only about the models on either side of it.

- Expand each stage in turn until the operations are straightforward to implement.

- Restructure the final pipeline for optimization.

Compilers are examples of batch transformation systems. A simplified view of the architecture of the compiler is depicted in Figure below.

There are six phases in a typical compiler:

- Lexical Analysis
- Syntax Analysis
- Error Recovery
- Scope Analysis
- Type Analysis
- Code Generation

The process of lexical analysis is called lexical analyzer, scanner, or tokenizer. Its purpose is to break a sequence of characters into a subsequences called tokens. The syntax analysis phase, called parser, reads tokens and validates them in accordance with a grammar. Vocabulary, i.e., a set of predefined tokens, is composed of word symbols (reserved words), names (identifiers), numerals (constants), and special symbols (operators). During compilation, a compiler will find errors such as lexical, syntax, semantic, and logical errors.

In this system input is to the first phase and output is from the last phase. Input will be the source program and output will be the object program.

5.2.2.2 Continuous Transformation

A data transformation performed continuously as inputs change. It is a system in which the outputs actively depend on changing inputs. It updates outputs frequently. One way to implement continuous transformation is with a pipeline of functions. Through the pipeline any change in input is propagated. Examples of continuous transformation applications are windowing systems, signal processing, process monitoring systems, incremental compilers etc.

The steps in designing a pipeline for a continuous transformation are as follows:

- Break the overall transformation into stages performing one part of the transformation.
- Define input, output and intermediate models between each pair of successive stages, as for the batch transformation.
- Differentiate each operation to obtain incremental changes to each stage. That is, propagate the incremental effects of each change to an input through the pipeline as a series of incremental updates.
- Add additional intermediate objects for optimization.

The main difference between batch transformation and continuous transformation is that in former computation is done once while in later continuous transformation updates outputs frequently.

For example, in a graphics application, first it maps geometric figures in user-defined coordinates to window coordinates. Then it clips the figures to fit the window bounds. Lastly, it offsets each figure by its window position to yield its screen position.

5.2.2.3 Interactive Interface

It is a system dominated by interactions between the system and external agents called actors. An actor can be a person, a device or another system/subsystem. The actors are not the part of the system rather they are supposed to lie on the boundary of the system. The main concerns of interactive interface are the communication protocols between the system and the actors, the flopw of control within the system, performance, error handling etc.

The steps in designing an interactive interface are as follows.

- Isolate interface classes from the application classes.
- Use predefined classes to interact with the actors, if possible.
- Use the state model as the structure of the program.
- Isolate physical events from logical events. Often a logical event corresponds to multiple physical events.
- Fully specify the application functions that are invoked by the interface.

5.2.2.4 Dynamic Simulation

Dynamic simulation is the use of a computer program to model the time varying behavior of a system. It models real-world objects. In this objects and operations directly come from the application and hence it is easy to implement. Controls can be implemented in two ways in dynamic simulation systems – external control and internal control. In former, objects simulate a state machine and in later objects can exchange messages among themselves. Examples of dynamic simulation are video games, spacecraft trajectory computation system, moleculer motion mode etc.

The steps in designing a dynamic simulation are as follows:

- Identify active real word objects from the class model. These objects have attributes that are periodically updated.
- Identify a discrete event, which corresponds to discrete interactions with the object.
- Identify continuous dependencies. Real world attributes may be dependent on other real-world attribute.
- Generally a simulation driven by a timing loop at a fine time scale. Discrete events between objects can often be exchanged as part of the timing loop.

5.2.2.5 Real time system

A system dominated by strict timing constrains. It is an interactive system with tight time constraints on actions. Real-Time systems span several domains of computer science. They are defense and space systems, networked multimedia systems, embedded automative electronics etc. In a real-time system, the correctness of the system behavior depends not only the logical results of the computations, but also on the physical instant at which these results are produced. A real-time system changes its state as a function of physical time, e.g., a chemical reaction continues to change its state even after its controlling computer system has stopped.

Based on this a real-time system can be decomposed into a set of subsystems i.e., the controlled object, the real-time computer system and the human operator. A real-time computer system must react to stimuli from the controlled object (or the operator) within time intervals dictated by its environment. The instant at which a result is produced is called a deadline. If the result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If a catastrophe could result if a firm deadline is missed, the deadline is hard. Commands and Control systems, Air traffic control systems are examples for hard real-time systems. On-line transaction systems, airline reservation systems are soft real-time systems.

5.2.2.6 Transaction Manager

It is a system concerned with storing and retrieving data, often including concurrent accesses from different physical locations. Most of the transaction managers deal with multiple users who read and write data at the same time. So they need to secure the data against the unauthorized access and accidental loss. They are generally built on the top of DBMS. Examples of such systems are inventory control, sales and purchase order processing, airline reservation systems etc.

The steps involved in the design of such systems are:

- Map the class model to database structures.
- Determine the units of concurrency that is the resources that inherently or by specification cannot be shared.
- Determine the unit of transaction that is the set of resources that must be accessed together during a transaction.
- Design concurrency control for transactions. This feature is provided by most database systems.

5.3 Summary

- Systems design is the process or art of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. The system design process is generally divided into two sub-phases – logical design and physical design.
- The logical design of a system pertains to an abstract representation of the data flows, inputs and outputs of the system.

- The physical design relates to the actual input and output processes of the system. This is laid down in terms of how data is input into a system, how it is verified/authenticated, how it is processed, and how it is displayed as output.
- It is much easier to use existing things than to design new things for uncertain use to come. Reusable things include models, libraries, frameworks and patterns.
- A subsystem may be defined as a group of classes, associations, operations, events and constraints that are interrelated and have a well-defined interface.
- In a layered architecture model, classes within each subsystem layer provide services to the layer above it. Ideally, this knowledge is one-way: each layer knows about the layer below it, but the converse is not true
- Partitions vertically divide a system into several independent or weakly subsystems, each providing one kind of service.
- To increase the performance of system, multiple processors or hardware functional units may be used. The number of processors required depends on the volume of computations and the speed of the machine
- The system designer must allocate the tasks for various subsystems to processors. Each concurrent subsystem should be allocated to an independent process or processor.
- System designer must decide from among several alternatives for data storage that can be used separately or in combination of data structures, files and databases.
- The system designer must identify global resources and determine mechanisms for controlling access to them
- There are two kinds of control flows in a software system: External control and internal control. External control concerns the flow of externally visible events among the objects in the system. There are three kinds of external events: procedural-driven sequential, event-driven sequential and concurrent.
- Although most of the system design concerns steady-state behavior system designer must consider boundary conditions as well and address issues like initialization, termination and failure.

- The architecture of a system describes its gross structure. This structure illuminates the top-level design decisions, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts.

5.4 Suggested Readings/Reference Materials

1. Object-Oriented Modeling and Design with UML, M. Blaha, J. Rumbaugh, Pearson Education-2007
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson-2007
3. Object Oriented Analysis & Design, Grady Booch, Addison Wesley-1994
4. Timothy C. Lethbridge, Robert Laganiere, Object Oriented Software Engineering, TMH, 2004

5.5 Self-Assessment Questions

1. What is system design? What are two types of system design? Explain.
2. What are different types of design decisions taken by a system designer? Explain in detail.
3. What is reuse plan? Discuss three models for reusability.
4. What is library-based reuse? Discuss problems faced in library-based reuse.
5. What is framework based reuse? How is it different from pattern?
6. What is pattern based reuse? Explain.
7. What is subsystem? What are different types of relationships between subsystems? Explain.
8. What are partitions and layers in a system?
9. How can you identify subsystems?
10. How do you identify concurrency?
11. How do you decide allocation of subsystems to processors?

12. How do you estimate hardware resource requirements?
13. How do you allocate tasks to processors?
14. How do you determine physical connectivity among subsystems?
15. Write a short note on data storage management.
16. Discuss the suitability of different types of data storages.
17. What are global resources in a system? How can you handle them?
18. How do you choose a software control strategy? Explain.
19. Distinguish between procedure driven and event driven control.
20. Distinguish between external and internal control.
21. What are boundary conditions? How do you handle them?
22. How do you set trade-off priorities?
23. What is software architecture? Discuss some commonly used architectural styles.
24. What is batch transformation architectural style? Enumerate the steps to be followed for this style.
25. What is continuous transformation architectural style? Enumerate the steps to be followed for this style.
26. What is interactive interface architectural style? Enumerate the steps to be followed for this style.
27. What is dynamic simulation architectural style? Enumerate the steps to be followed for this style.
28. What is transaction manager architectural style? Enumerate the steps to be followed for this style.
29. What is real time system?

Writer: Dr. Rajender Nath

Vetter: Dr. Dharminder Kumar

Chapter 6

Object Design

Structure

6.0 Introduction

The object design is the next phase in the OMT methodology. Object oriented design is a process of refinement or adding details. The object-designer works to implement the objects discovered during analysis phase. All the operations identified during analysis are expressed as algorithms, with complex operations expressed as internal operations. The classes, attributes and associations from analysis must be implemented as specific data structures. New object classes may be introduced to store intermediate results in this phase. The following steps are performed in the object design phase:

- Combine the three models to get the operations of the classes
- Design algorithms for operations
- Optimize design
- Implementation of control
- Maximize inheritance
- Design associations
- Determine object representation
- Package classes and associations into modules

6.1 Objectives

After having discussed high-level design decisions made during the system design phase as has been discussed in the last chapter, in this chapter you will learn how to combine three models developed during analysis phase to obtain operations on objects. How can you design and optimize algorithms? How can you implement software controls? The way you can maximize inheritance tree. How can you implement associations and represent objects? Packaging of classes and association will be discussed in this chapter. Documentation and implementation of design will also be discussed in this chapter.

6.2 Presentation of Contents

6.2.1 Object Design

6.2.1.1Combine The Three Models To Get The Operations Of The Classes

The outcomes of the analysis phase are three models – object model, dynamic model and functional model. Operations for the classes may not be specified in the object model. These operations are obtained from dynamic and functional model. The object designer has convert actions and activities of the dynamic model and processes of the functional model into operations of the classes in the object model.

Each state diagram constructed in the dynamic model gives the life history of an object. A transition in the state diagram represents a change of state of the object, which can be mapped into an operation on the object. One operation can be associated with an each event received by an object. The action performed by a transition depends on both the event and the state of the object. This implies that the algorithm implementing an operation depends upon the state of the object.

An event generated by an object may represent an operation on another object. Events often occur in pairs – the first event triggers an action and the second event returns the result or indicates the completion of the action. In such cases, the event pair can be mapped into an operation performing the action and returning control provided that the events are on a single thread of control passing from one object to another.

An action or activity initiated by a transition in a state diagram may expand into a full-fledged DFD in the functional model. The collection of processes within the DFD represents the body of an operation. The data flows in the DFD show intermediate values in the operation. The object designer must convert the graph structure of the DFD into a linear sequence of steps in an algorithm. The processes in the DFD constitute suboperations. These may be on the original target object or on other objects.

The target objects of a suboperation can be determined as follows:

- If a process extracts a value from an input flow then the output flow is the target.
- If a process takes an output value from several input flows, then the operation is a class operation on the output class.
- If a process has an input flow and an output flow of the same type and the output value is considerably modified version of the input flow then input/output flow is the target.
- If a process has an input from or output to a data store or an actor then the data store or an actor is the target of the process.

6.2.1.2 Design Algorithms For Operations

An algorithm is designed for each operation specified in the DFD. The DFD tells what the operation does but the algorithm tells how it is done. The data store or an actor is the target of the process.

The algorithm designer must follow the following steps for designing algorithms for operations:

- Choose an appropriate algorithm
- Choose an appropriate data structure
- Add new classes and operations as necessary
- Assign appropriate responsibility to classes

Choose an appropriate algorithm: The following criteria can be applied in choosing an appropriate algorithm:

Computational complexity: Determine the complexity of the algorithm and choose an algorithm, which is, takes lesser time. For example, computational complexity of the

bubble sort is of the order of n^2 while merge sort has the computational complexity of the order of n log n. So, we can choose merge sort over the bubble sort algorithm.

Ease of implementation: If operation is non-critical then it is worth giving up some performance, if the operation can be implemented with a simpler algorithm.

Flexibility: Most of the programs are bound to be extended in future. If an algorithm is highly optimized then it is very difficult to understand and modify. In such cases, we can provide two implementations for an operation – one algorithm, which is simple and inefficient that can be developed quickly. This algorithm can be used to validate the other algorithm, which is optimized and very efficient.

Choose an appropriate data structure: Every algorithm uses some data structures such as strings, array, lists, queues, stacks, trees, graphs, sets, bags etc. To make an algorithm efficient, an appropriate data structure should be chosen. For example, stack can be implemented by using an array or by using a linked list. The algorithm designer has to choose an appropriate data structure.

Add new classes and operations as necessary: To hold intermediate results, we may need to add new classes. A complex operation can be decomposed into many low-level operations. Such low-level operations are defined during object design.

Assign appropriate responsibility to classes: Most of the operations have an obvious target but there are some operations, which can be placed at many places. This problem gets more complicated if the operation involves many objects. The following guidelines can be used to decide where to attach the operation.

If one object is acted upon while the other object performs the action then associate the operation with the target of the operation.
If one object is modified by the operation while other objects are read only assign the operation with the object, which is modified, by the operation.

If the classes and associations form a star network then assign the operation to the central class.

6.2.1.3 Optimize The Design

The analysis model captures the logical information about the system, while the design model must add details to support efficient information accesses. The designer must strike the balance between efficiency and clarity.

The designer can optimize the design by:

- Adding redundant associations to minimize access cost and maximize convenience
- Rearranging the computation for greater efficiency
- Saving the derived attributes to avoid recomputation of complicated expressions

Adding Redundant Associations for efficient access: During design, the structure of the object model is evaluated for an implementation. However, in some cases the associations that were useful during analysis may not be useful during design. For an example consider the design of a company's employee skills database. A portion of the object model from analysis phase is shown in figure below.

For this example, suppose that company has 1000 employees each of whom has 10 skills on an average. To know about how many employees speak French, a simple nested loop will traverse employees 1000 times and has-skill 10,000 times. In cases where number of hits from a query is low because only a fraction of objects satisfy the test, we can add a qualified association - speaks language - from company to employee, where qualifier is the language spoken as shown in Figure below. This permits us to find all employees who speak a particular language with no wasted accesses.

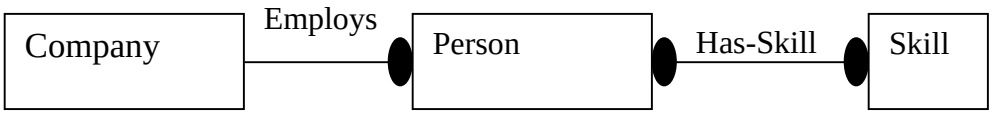


Figure: Chain of associations

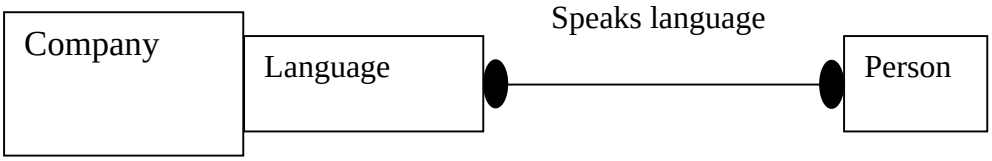
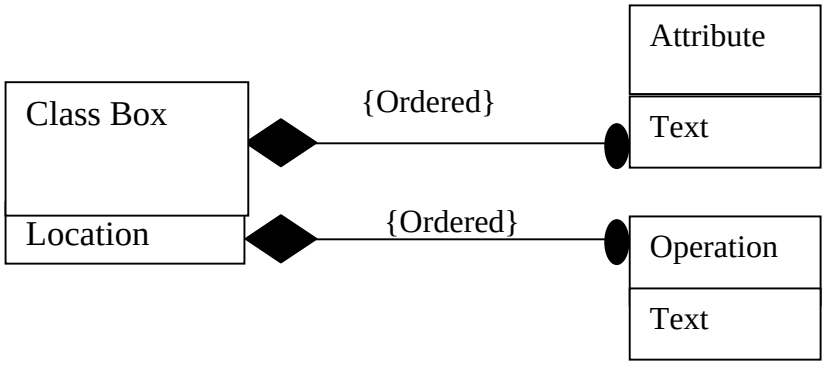


Figure: Qualified association

Rearranging Execution Order for Efficiency: After adjusting the structure of the object model for optimization the next thing to optimize is the algorithm itself. There may exist certain paths, which may be termed as dead path that means that chances of execution along those paths are very less.

One key to algorithm optimization is to eliminate dead paths as early as possible. For example, suppose we want to find all employees who speak both French and Chinese. Further, suppose 10 employees speak French and 150 employees speak Chinese. In this case, it is better to test French speaker first then test whether they speak Chinese. This results in narrowing the search.

***Saving* Derived Attributes to Avoid Recomputation:** There may exist some data which can be derived from other data. This kind of data may be termed as redundant data and it can be cached or stored in its computed form to avoid overhead of recomputing again. The class that contains redundant data must be updated if any of the objects that it depends upon are changed.



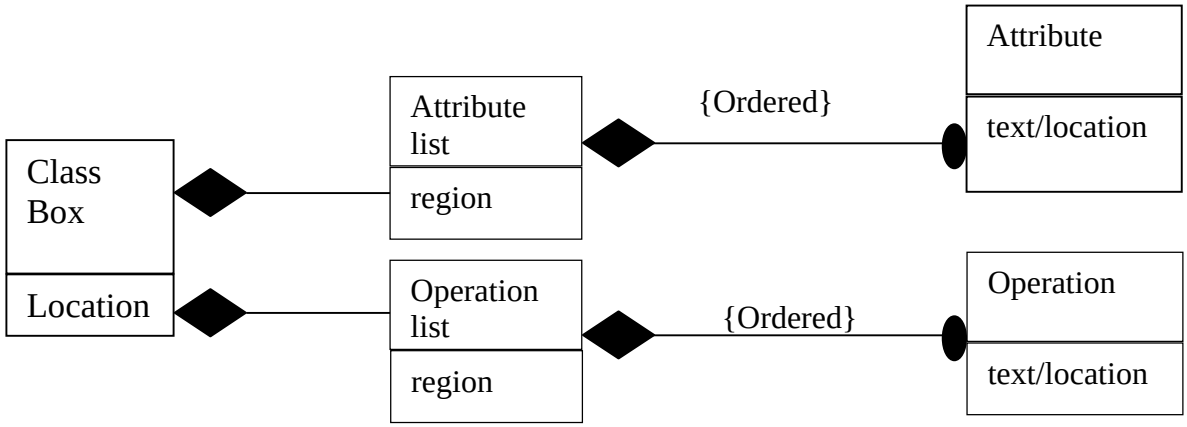


Figure above shows a use of derived object and the derived attribute. Each class box contains an ordered list of attributes and operations, each represented as a text string. Given the location of class box itself, the location of each attribute can be computed by adding up the size of all the elements in front of it. Since the location of each element is needed frequently, the location of each attribute string is computed and stored. The region containing the entire attribute list is also computed and stored so that input points need not be tested against attribute text elements in other boxes. Derived attributes must be updated when base values change. There are three ways to recognize when an update is needed: by explicit code, by periodic recomputation, or by using active values.

Explicit Update: Each derived attribute is defined in terms of one or more fundamental base objects. The designer determines which derived attributes are affected by each change to a fundamental attribute and inserts code into update operation on the base object to explicitly update the derived attributes that depend on it.

Periodic Recomputation: Derived attributes are recomputed periodically instead of recomputing derived attributes on each change in base value. Periodic update is simpler than explicit update and less prone to bugs.

Active Values: An active value is a value that has dependent values. An operation to update the base value triggers update of all the dependent values.

6.2.1.4 Implementation of Control

As part of the system design, the designer must implement state diagram of the dynamic model. There are three basic approaches to implement it:

- To use location within the program to hold state.
- Direct implementation of state machine mechanism
- Using concurrent tasks

State as location within a program: The location of control within a program can be used to represent state of the program. Each state transition corresponds to an input statement. The control of the program branches depending upon the input event received. In highly nested procedural code, there must be a procedure, which can handle the supplied input.

The state diagram can be converted to code as follows:

1. Identify the main path through the diagram that leads to normal execution of events. Identify the names of states along this path. This becomes a sequence of statements in the program.
2. Identify alternate paths, which branch off the main path. This becomes the conditional statements in the program.
3. To identify the loops, find out backward paths that branch off the main path. Multiple backward paths become nested loops in the programs.
4. Remaining states and conditions correspond to exceptional conditions. These exceptional conditions can be handled through exception handling or including error subroutines.

However, in this approach the lack of modularity is biggest drawback.

Direct implementation of state machine mechanism: The most direct approach to implement control is to have a state machine engine class that keeps track of execution of states and actions. Each object instance maintains its own independent variables but would call on the state engine to determine the next state and action. The basic flow of control can be traced by creating stubs of the action routines. A stub contains the minimal piece of information regarding functions or subroutines. Teach stub can provide its print out to which it can be combined with other printouts to find the skeleton of the application.

Using concurrent tasks: An object can be implemented as a task in the programming language or operating system. Events are implemented as inter-task calls using the features of the language or operating system. Languages, which allow concurrency such as ADA, Concurrent C++ etc., can be used to execute tasks concurrently.

6.2.1.5 Maximize Inheritance

The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization coin: Mammal

generalizes what is common between dogs and cats, and dogs and cats specialize mammals to their own specific subtypes. These relationships are hierarchical because they create a relationship tree, with specialized types branching off from more generalized types. As you move "up" the hierarchy, you achieve greater generalization. You move up toward Mammal to generalize that dogs, cats, and horses all bear live young, nurses with milk, have hair. As you move "down" the hierarchy you specialize. Thus, the cat specializes Mammal in having claws (a characteristic) and purring (a behavior).

Similarly, when you say that ListBox and Button are Windows, you indicate that there are characteristics and behaviors of Windows that you expect to find in both of these types. In other words, Window generalizes the shared characteristics of both ListBox and Button, while each specializes its own particular characteristics and behaviors.

The definition of classes and operations can be adjusted to make the inheritance tree as large as possible. It can be done in the following ways.

- Rearranging and adjustment of classes and operations to increase inheritance.
- Abstracting common behavior out of group of classes.
- Use of delegation to share behavior when inheritance is semantically invalid.

Rearranging and adjustment of classes and operations to increase inheritance

Inheritance leads to reusability of already existing functionality. Inheriting more and more functions can increase the level of reusability. However, greater inheritance requires that functions definitions must be modified in a sense that same function can serve the purpose for multiple classes. The following adjustments can be used to increase the level of inheritance.

- Some operations may have fewer arguments than others. The missing arguments can be supplied in the function definition and they may be ignored.

- Similar attributes in different classes may have the different names. These attributes may be moved to a common ancestor class.
- Some operations may have fewer arguments because they are special cases of more general arguments. These special operations can be implemented by calling the general operations with appropriate parameters
- An operation may be defined on several different classes in a group but may not be required in other classes. In that case the operation can be defined as a common ancestor class and can be declared as no-operation on the classes that don't care about it.

Abstracting common behavior out of group of classes

During the design phase, new classes and operations are often added which generally may lead to commonality between classes. In this case, a common super class can be created that implements the shared features of the classes. The subclasses may be derived from super class and may implement their own specialized features. Usually the resulting super class is an abstract class, meaning that there are no direct instances of it, but all its operations can be defined and implemented by subclasses. This leads to reusability of the code.

Sometimes there may be only one subclass, even in that case it is advisable to abstract out a super class. It is quite possible that new subclasses may be added during project development life cycle or in future the scope of the project may be increased. The splitting of a class into two classes that separate the specific aspects from the more general aspects is a form of modularity. The creation of abstract super classes also improves the extensibility of a software product.

Use of delegation to share behavior when inheritance is semantically invalid

Inheritance is the most commonly used method of implementation of generalization where the behavior of a super class is shared by all its sub classes. Operations of the subclass that override the corresponding operation of the super class have a responsibility to provide the same services as provided by the super class.

Sometimes the concept of inheritance is used to utilize already existing functionality with no intention to provide similar behavior of sub class as of super class. For example there may exist a class list, which implements a no. of operations i.e addition, deletion or modification of elements at any place in the list. Any programmer who wants to create a stack class may be tempted to use already existing list class. However, stack just requires two main operations i.e. push and pop. Extra operations already existing in the list class will also be inherited in the stack class, which may cause some side effects.

This can be avoided 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, using delegation rather than inheritance.

6.2.1.6 Design Associations

During the object design phase, all the associations in the object model should be implemented. To make a decision for implementation, one must analyze the way the associations are used.

Analyzing Association Traversal: Associations may be traversed unidirectional or bi-directional. The unidirectional associations can be traversed in forward direction only but they are easier to implement. Bi-directional associations allow reverse and forward traversal. Therefore bi-directional associations are always preferred over unidirectional associations so that we can add new behavior or expand or modify the application rapidly.

One-way Associations: A unidirectional association can be implemented as a pointer. If the multiplicity is one as shown in figure below, then a simple pointer is used. If the multiplicity is many, then a set of pointers is used. 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.

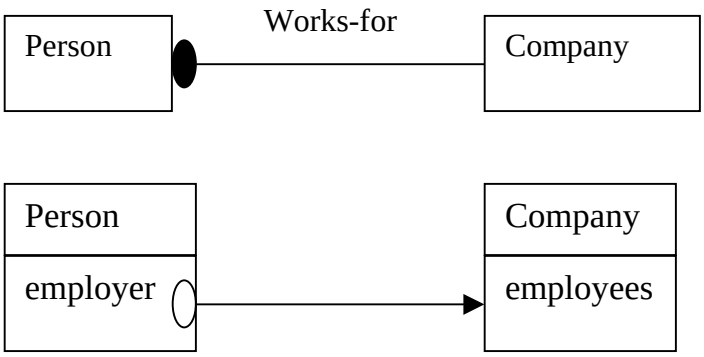


Figure: One-way association using pointer

Two-way Associations: There are three approaches to implement bi-directional associations as discussed below:

- Implement as an attribute in one direction only and perform a search when a backward traversal is required. This works fine when there is great disparity in traversal frequency in the two directions. Implementation in one direction will reduce the cost of storage and updation.

- Implement as attributes in both the directions as shown in figure below. This results in fast access but if either attribute is updated then the other attribute must also be updated to keep the link consistent.

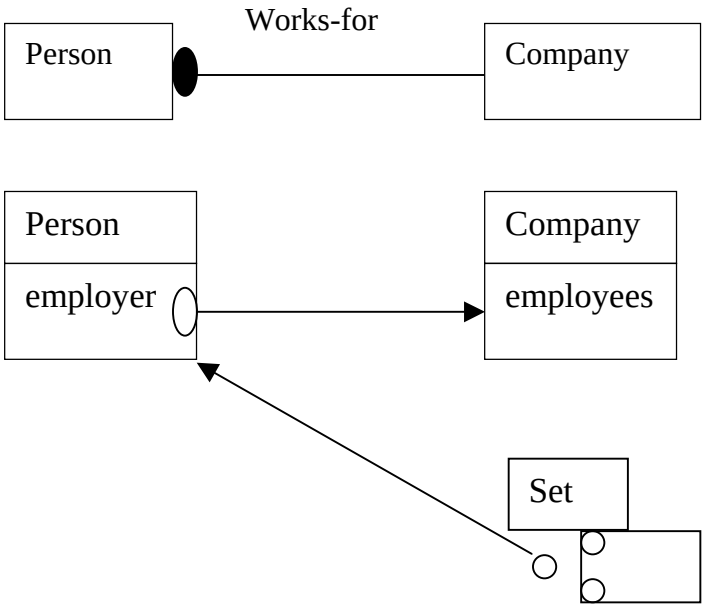
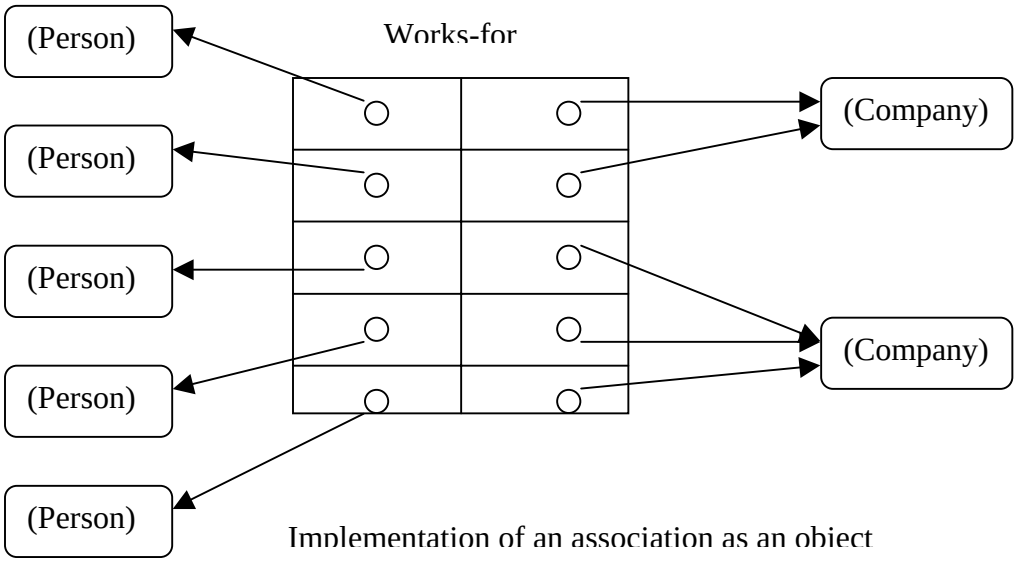


Figure: Two-way association using pointers



Implementation of an association as an object

- Implement as a distinct association object, independent of either class as shown in figure below. An association object is a set of pairs of associated objects stored in a single variable size object. For efficiency, an association object can be implemented using two dictionary objects, one for the forward

direction and one for the backward direction. Access is slightly slower than with attribute pointers, but if hashing is used then the access is still of constant time.

Link Attributes: If an association has link attributes, then its implementation depends on the multiplicity. If the association is one-to-one, the link attributes can be stored as attribute of either object. If the association is many-to-one, the link attributes can be stored as attributes of the many object, since many object appears only once in the association. If the association is many to many, the link attributes cannot be associated with either of the objects.

6.2.1.7 Determine Object Representation

The object designer has to choose when to use primitive types in representing the objects or when to combine the groups of objects. A class can be defined in terms of other classes but ultimately all data members have to be defined in terms of built-in data types supported by a programming language. For example, roll no. can be implemented as integer or string. In another example, a two dimensional can be represented as one class or it can be implemented as two classes – Line class and Point class.

6.2.1.8 Packaging of Classes and Associations into Modules

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Modules serve as the physical containers in which we declare the classes and objects of our logical designs. A module can be edited, compiled or imported separately. Different object-oriented programming languages support the packing in different ways. For example, Java supports in the form of package, C++ in the form of header files etc.

Modules are program units that manage the visibility and accessibility of names. Following purposes can be solved by modularity.

- A module typically groups a set of class definitions and objects to implement some service or abstraction.
- A module is frequently a unit of division of responsibility within a programming team. A module provides an independent naming environment that is separate from other modules within the program.
- Modules support team engineering by providing isolated name spaces.

Packaging involves the following three issues:

- Information Hiding
- Coherence of Entities
- Constructing Physical Modules

Information Hiding: During analysis phase we are not concerned with information hiding. So, visibilities of class members are not specified during analysis phase. It is done during object design phase. In a class, data members and internal operations should be hidden, so, they should be specified as private. External operations form the interface so they should be specified as public.

The following design principles can be used to design classes:

- A class should be given the responsibilities of performing the operations and proving information contained in it to other classes.
- Calling a method of that class should access attributes of other class.
- Avoid traversing associations that are not connected to this class.
- Define interfaces at the highest level possible.
- External objects should be hidden. Defining interface classes could do this.
- Avoid applying a method to the result of another class unless the result class is already a supplier of methods to the caller class.

Coherence of Entities: Module, class, method etc. are entities. An entity is said to coherent, if it is organized on a consistent plan and all its parts fit together toward a common goal.

Policy is the making of context-dependent decisions while implementation is the execution of fully specified algorithms. Policy involves making decisions, gathering global information and interacting with the external agents. Policy and implementation should be separated in different methods i.e. both should not be combined into a single method.

A policy method does not have complex algorithms rather invokes various implementation methods. It can contain I/O statements, conditional statements and can access data stores.

An implementation method does exactly one operation, without making any decision, assumption, default or deviation. All its information is supplied by arguments. These methods do not contain any context-dependent decision so they are likely to be reusable.

Constructing Physical Modules: Modules of analysis phase have changed as more classes and associations have been added during object design phase. Now, the object designer has to create modules with well-defined and minimal interfaces. The classes in a module should have similar kinds of things in the system. There should be cohesiveness or unity of the purpose in a module. So the classes, which are strongly associated, should be put into a single module.

6.2.2 Documentation

All design decisions must be properly documented when they are complete. Documentation is the best practice to remember design details and for transmitting the designs to others and for recording it for future references during maintenance

tasks. The design document should be documented by extending requirements analysis document, by adding details to the object and functional models.

The design document must include a revised and much more detailed description of the object model, both in graphical and textual forms. The traversal directions of associations can be shown using proper arrow pointers. It is a good idea to keep design document distinct from analysis document. The design document includes many optimizations and implementations artifacts. Therefore, it is important to keep detailed information about internal operations and user oriented description of the system.

The clear and well-written documentation helps to trace an element in the original analysis to the corresponding element in the design document. The design document must be straightforward and must be an evolution of the analysis model.

6.2.3 Implementation Using A Programming Language

The design of the system can be implemented using an object-oriented language or a non object-oriented language. Even if non-object oriented language is used, the object-oriented design can be beneficial in terms of easiness in preserving the object-oriented structure of the program.

There are mainly three aspects which most of the executable languages are able to implement.

Data Structures: Whereas control structures organize algorithms, data structures organize information. In particular, data structures specify types and structures of data, and set of operations that can be performed on them. Simple data structures include integers, real numbers, Booleans (true/false), and characters or character strings. Combining one or more data types forms compound data structures. The most important compound data structures are the array, a homogeneous collection

of data, and the record, a heterogeneous collection. An array may represent a vector of numbers, a list of strings.

Data structures can be expressed in a non-procedural language. The statements used to describe data structures can be mixed with procedural statements but are not executable. In some languages like Ada, a sharp distinction is made between external specification, which is purely declarative and internal specification, which is often combined with procedural statements.

Flow of control: Programming languages provide statements for control flow. Control flows can be expressed either procedurally (conditions, loops and calls) or non procedurally (rules, constraints, tables and state machines). Non-procedural languages such as rule based systems, real time systems etc. support entirely different ways of organizing programs.

Functional Transformations: These are expressed in terms of primitive operators of the language, as well as calls to subprograms. Some languages like LISP permit construction of functions during run time, which allows more sophisticated operations to be performed.

6.2.4 Implementation Using A Database System

Database systems offer the more appropriate form of implementation where the main concern is to access data in a persistent manner. Most of the database systems have built-in features for concurrent execution, which provides a high degree of parallelism. Database commands operate on sets of records leading to faster execution and reduced access time. Generally databases provide a data definition language to declare the structure of the data. A query language may also be provided which allows querying, updation, deletion, modification etc. The query language may also offer the features of a programming language such as defining and execution of stored procedures.

6.3 Summary

- Object oriented design is a process of refinement or adding details. The object-designer works to implement the objects discovered during analysis phase.
- Operations for the classes may not be specified in the object model. These operations are obtained from dynamic and functional model.
- Each state diagram constructed in the dynamic model gives the life history of an object. A transition in the state diagram represents a change of state of the object, which can be mapped into an operation on the object.
- Events often occur in pairs – the first event triggers an action and the second event returns the result or indicates the completion of the action. In such cases, the event pair can be mapped into an operation performing the action and returning control provided that the events are on a single thread of control passing from one object to another.
- Computational complexity determines the complexity of the algorithm and chooses an algorithm, which takes lesser time.
- There are three basic approaches to implement controls: to use location within the program to hold state, direct implementation of state machine mechanism and using concurrent tasks.
- The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization.
- During the design phase, new classes and operations are often added which generally may lead to commonality between classes. In this case, a common super class can be created that implements the shared features of the classes.
- The unidirectional associations can be traversed in forward direction only but they are easier to implement. Bi-directional associations allow reverse and forward traversal.
- If an association has link attributes, then its implementation depends on the multiplicity.

- The object designer has to choose when to use primitive types in representing the objects or when to combine the groups of objects.
- Modules serve as the physical containers in which we declare the classes and objects of our logical designs. A module can be edited, compiled or imported separately.
- An entity is said to coherent, if it is organized on a consistent plan and all its parts fit together toward a common goal.
- A policy method does not have complex algorithms rather invokes various implementation methods. It can contain I/O statements, conditional statements and can access data stores.
- An implementation method does exactly one operation, without making any decision, assumption, default or deviation. All its information is supplied by arguments. These methods do not contain any context-dependent decision so they are likely to be reusable.
- Documentation is the best practice to remember design details and for transmitting the designs to others and for recording it for future references during maintenance tasks.
- Most of the database systems have built-in features for concurrent execution, which provides a high degree of parallelism.
- The query language may also offer the features of a programming language such as defining and execution of stored procedures.

6.4 Suggested Readings/Reference Materials

1. Object-Oriented Modeling and Design with UML, M. Blaha, J. Rumbaugh, Pearson Education-2007
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson-2007
3. Object Oriented Analysis & Design, Grady Booch, Addison Wesley-1994
4. Timothy C. Lethbridge, Robert Laganiere, Object Oriented Software Engineering, TMH, 2004

6.5 Self-Assessment Questions

1. What do you mean by object design? What are the steps followed during the object design?
2. How can you combine object model, dynamic model and functional model to obtain operations on classes?
3. What are the steps an object designer has to follow during algorithm design?
4. How do you optimize the design during the object design phase?
5. Discuss the basic strategies to implement control.
6. How do you design associations?
7. Discuss different ways of increasing inheritance.
8. How can you implement one-way, two-way and link attribute associations?
9. How do you document your design decisions?
10. What are partitions and layers in a system?