UNIT 3 ADVANCE OBJECT DESIGN

Structure **Page Nos.**

3.0 INTRODUCTION

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

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

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

3.1 OBJECTIVES

After studying this unit, you should be able to:

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

3.2 CONTROL AND ITS IMPLEMENTATION

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

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

3.2.1 Control as State within Program

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

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

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

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

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

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

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

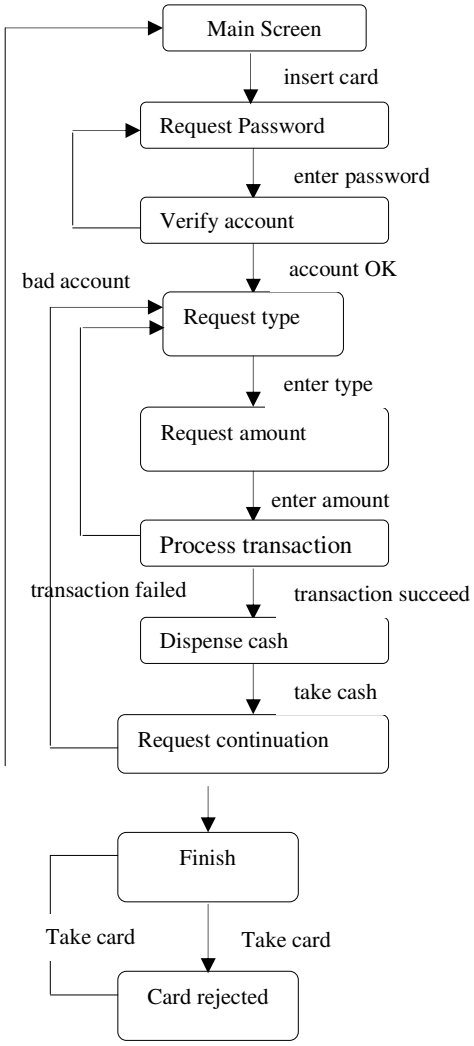

Figure 1: Control of states and events in ATM

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

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

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

3.2.2 Control as a State Machine Engine

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

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

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

3.2.3 Control as Concurrent Tasks

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

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

☞ Check Your Progress 1

1) Briefly explain state diagram by taking one example.

……

……..

2) Explain concurrent task by taking a suitable example.

……

……..

……..……

3) Explain the following terms.

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

……

……

3.3 INHERITANCE ADJUSTMENT

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

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

Rearrange Classes and Operations

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

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

Making Common Behavior Abstract

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

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

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

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

Use Delegation to Share Implementation

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

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

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

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

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

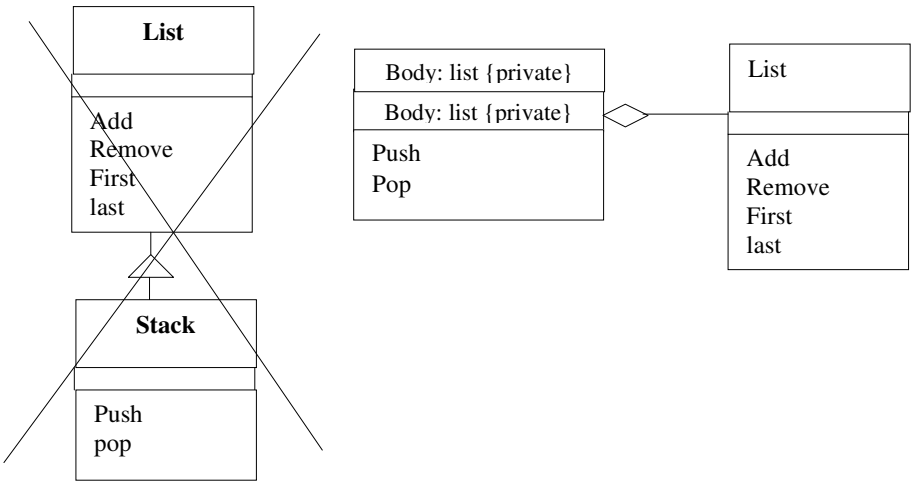

Figure 2: Alternative implementations of a Stack using inheritance (left) and delegation (right)

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

3.4 ASSOCIATION: DESIGN

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

Analyzing Association Traversal

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

One-way Associations

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

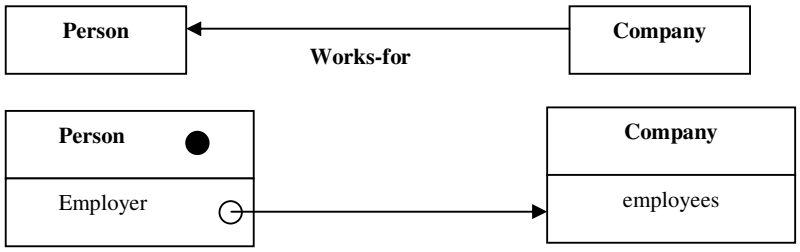

Figure 3: Implementation of one-way association using pointers

Two-way Associations

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

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

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

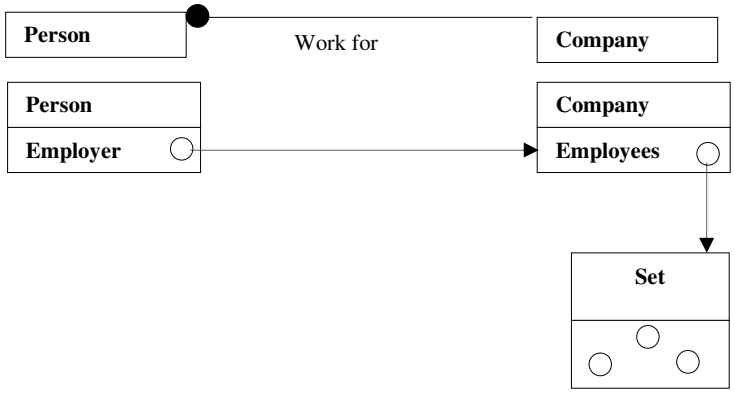

Figure 4: Implementation of two-way association using pointers

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

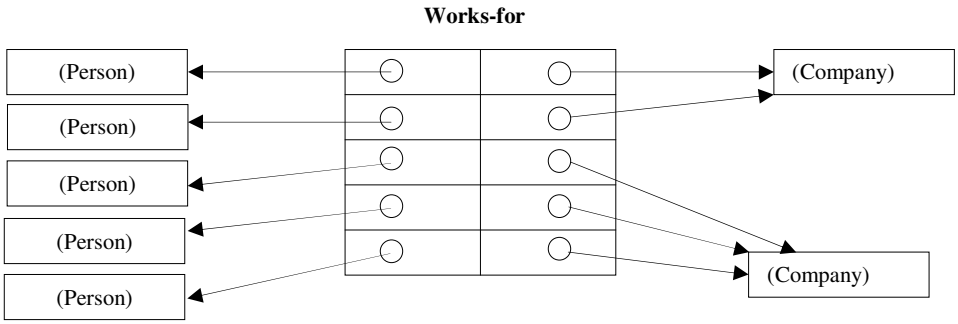

Figure 5: Implementation of association as an object

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

3.5 OBJECT REPRESENTATION

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

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

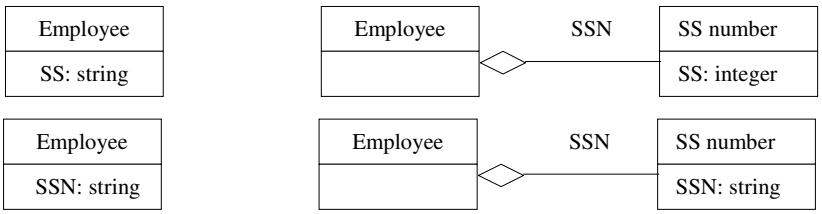

Figure 6: Alternative representations for an attribute

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

☞ Check Your Progress 2

1) Explain inheritance with support of suitable example.

……

……

……

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

……

……

……

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

……

……

……

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

3.6 DESIGN OPTIMIZATION

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

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

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

Now let us discuss these issues one by one:

Adding Redundant Associations for Efficient Access

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

This can be done by asking questions:

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

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

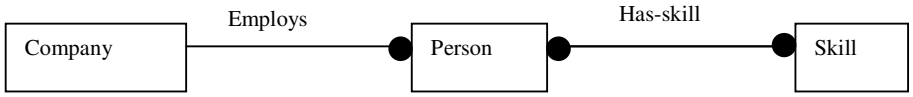

Figure 7: Chain of associations between objects

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

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

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

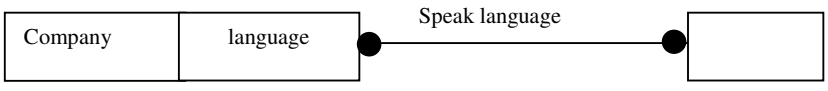

Figure 8: Index for personal skills database

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

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

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

For each operation, you should note the following points:

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

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

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

Rearrange the Execution Order for Efficiency

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

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

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

Saving Derived Attributes to Avoid Recomputation

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

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

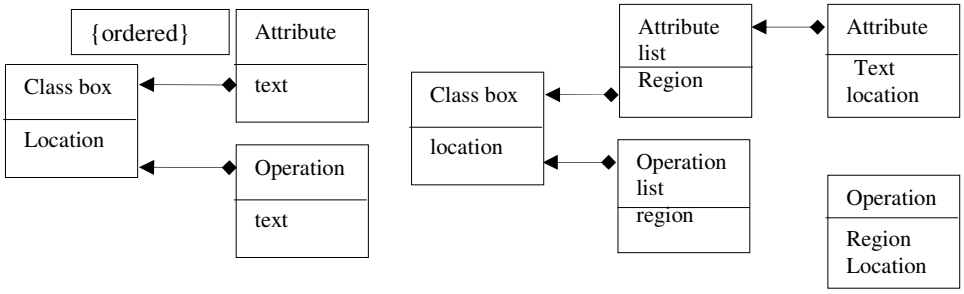

Figure 9 (a) **Figure 9 (b)**

Figure 9: Derived attribute to avoid recomputation

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

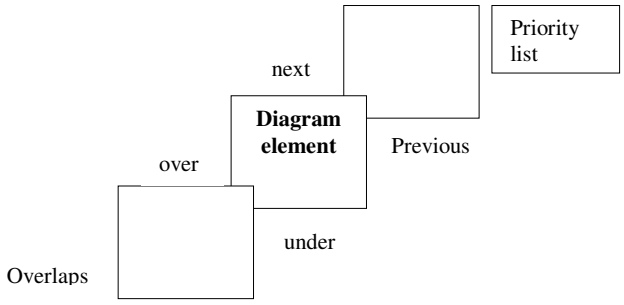

Figure 10: Association as a cache

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

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

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

3.7 DESIGN DOCUMENTATION

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

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

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

☞ Check Your Progress 3

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

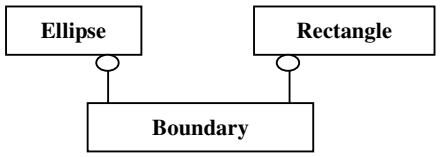

Figure 11: Portion of an object diagram with a shared class

…….

……..……

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

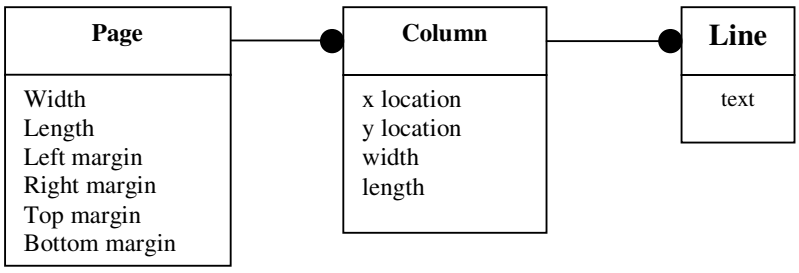

Figure 12: Portion of an object diagram of a newspaper

…….

……..……

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

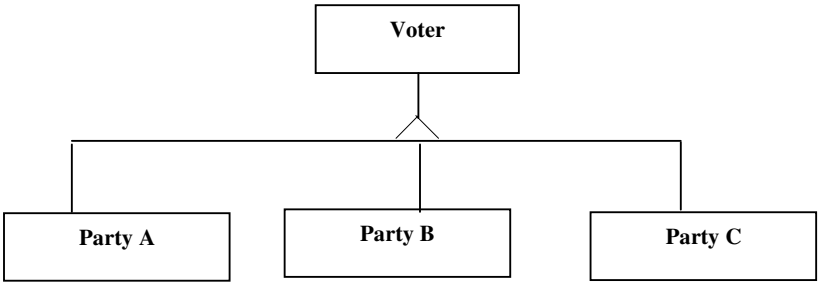

Figure 13: Object diagram representing voter membership in a political party

…….

……..……

……..

……..

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

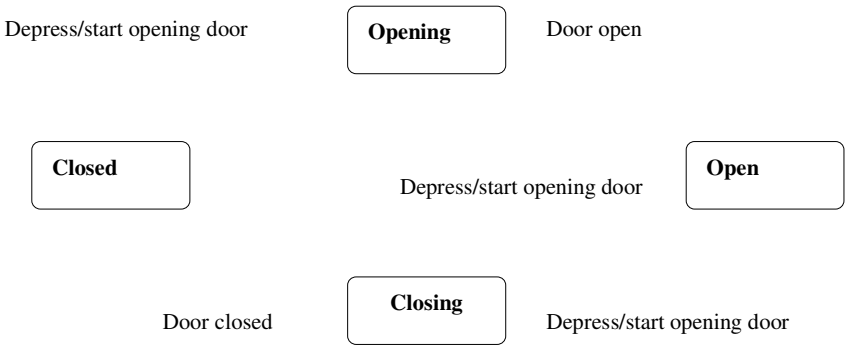

Figure 14: State diagram for a garage door opener

3.8 SUMMARY

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

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

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

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) **State Diagram**

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

2) **Concurrent Task**

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

3) **Event**

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

State

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

Check Your Progress 2

1) **Inheritance**

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

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

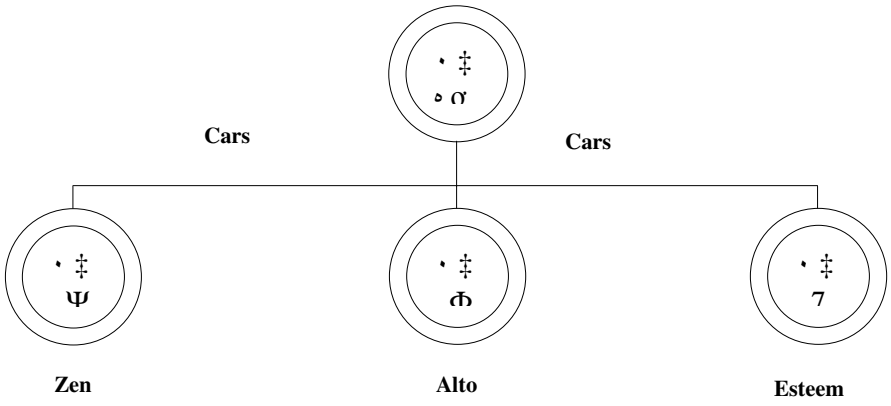

Figure 15: Hierarchy of Classes

Inheritance can be of various types, such as:

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

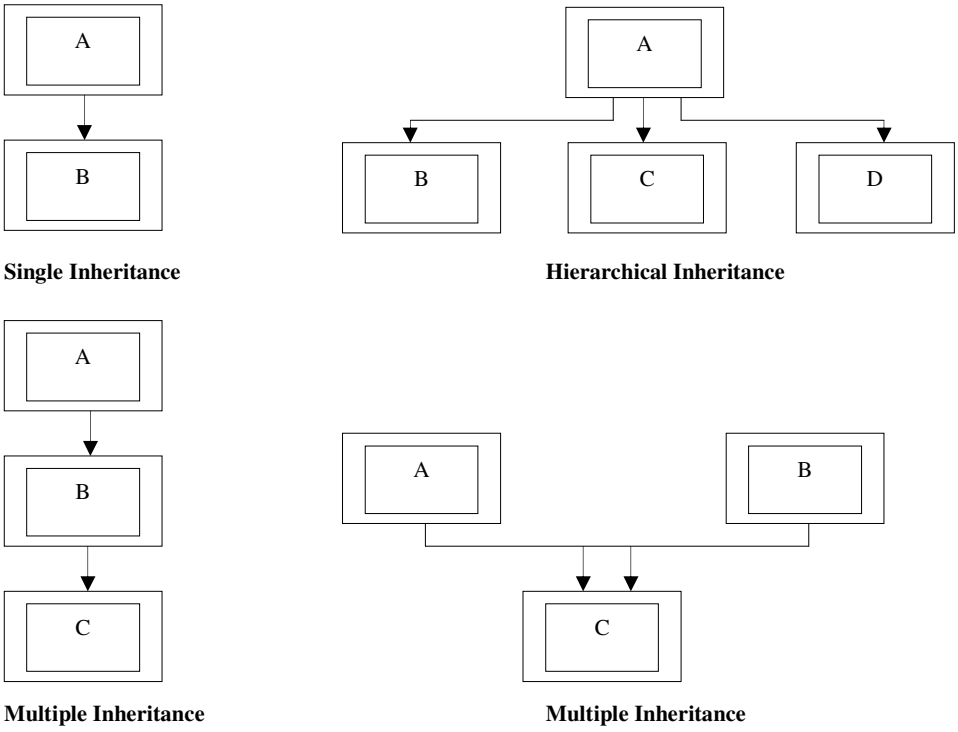

Figure 16: Forms of inheritance

2) The Design of Associations

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

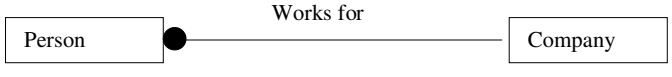

Figure 17: A simple form of Association

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

3) Adjustment of Inheritance

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

Check Your Progress 3

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

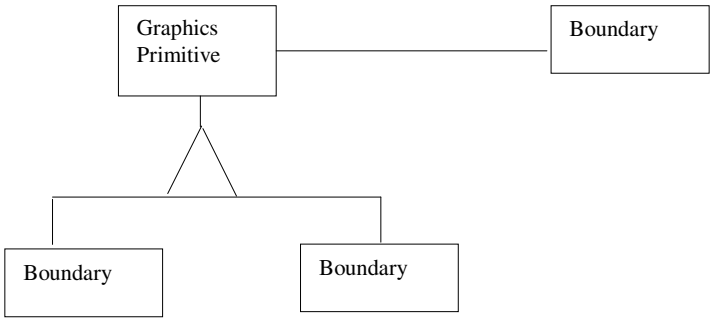

Figure: 18 Object Diagram

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

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

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

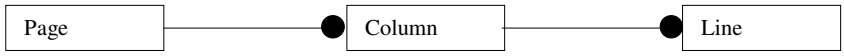

Figure 19: Association between different objects of a news paper.

The data type to the attributes of the Figure is

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

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

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

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

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

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

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