UNIT 2 OBJECT DESIGN

Structure **Page Nos.**

2.0 INTRODUCTION

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

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

2.1 OBJECTIVES

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

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

2.2 OBJECT DESIGN FOR PROCESSING

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

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

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

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

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

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

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

2.3 OBJECT DESIGN STEPS

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

There are certain steps to be followed in this design:

1) **Classify the operations on classes**

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

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

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

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

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

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

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

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

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

The algorithm designer must:

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

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

iii) Define new internal classes and operations, if required

iv) Assign responsibility for operations to appropriate classes.

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

3) Optimization of data access paths

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

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

ii) Rearrange the order of computational tasks for better efficiency

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

4) Implementing software control

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

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

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

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

iii) Using concurrent tasks.

5) Adjustment of inheritance

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

i) Adjust, or rearrange the classes and operations

ii) Abstract common behavior out of groups of classes

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

6) Design of associations

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

i) Analyze the path of associations.

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

7) Determine object representation

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

8) Package classes and associations into models

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

a. Information hiding

b. Collection of entities

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

☞ Check Your Progress 1

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

……

……

……

2) What is object design?

……

……

……

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

……

……

……

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

2.4 CHOOSING ALGORITHMS

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

i) If no procedural specification is given for functions

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

There are a number of metrics for selecting best algorithm:

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

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

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

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

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

2.4.1 Selecting Data Structure

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

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

2.4.2 Defining Internal Classes and Operations

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

2.4.3 Assigning Responsibility for Operation

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

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

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

2.5 DESIGN OPTIMIZATION

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

a) Adding Redundant Associations for Efficient Access

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

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

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

For each operation, we should know the followings:

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

b) Rearranging the Execution Order for Efficiency

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

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

c) Saving Derived Attributes to Avoid Recomputation

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

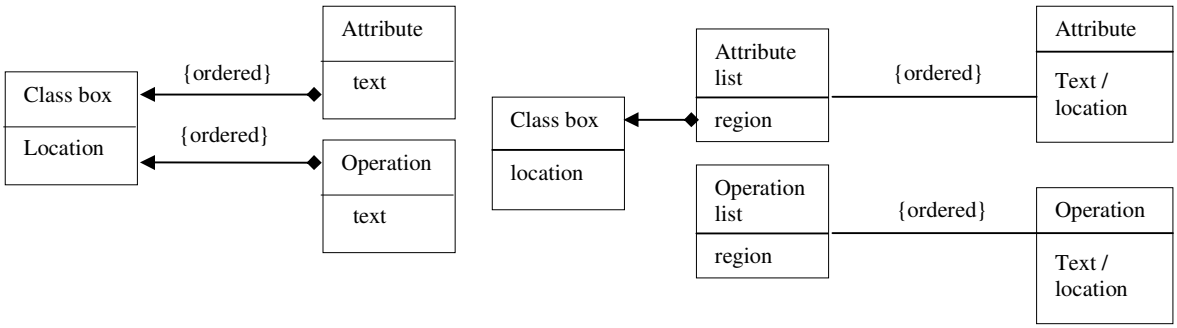

Figure 1: Derived attribute to avoid recomputation

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

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

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

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

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

☞ Check Your Progress 2

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

……

……

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

……

……

3) Give an example of an active value.

……

……

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

2.6 IMPLEMENTATION OF CONTROL

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

2.6.1 State as Location within a Program

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

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

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

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

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

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

2.6.2 State Machine Engine

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

2.6.3 Control as Concurrent Tasks

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

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

2.7 ADJUSTMENT OF INHERITANCE

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

2.7.1 Rearranging Classes and Operations

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

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

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

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

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

2.7.2 Abstracting Out Common Behavior

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

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

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

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

2.8 DESIGN OF ASSOCIATIONS

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

2.8.1 Analyzing Association Traversal

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

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

2.8.2 One-way Associations

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

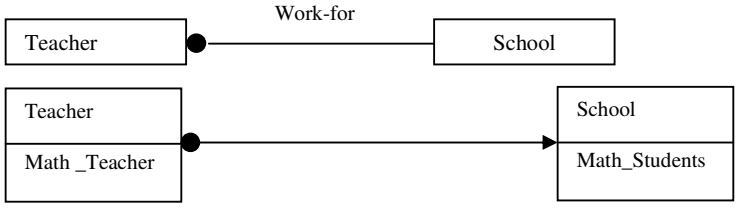

Figure 2: Implementation of a one-way association using pointers

2.8.3 Two-way Associations

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

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

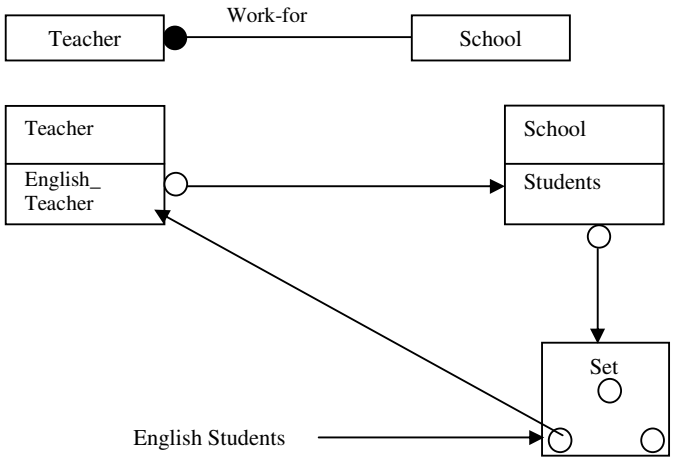

Figure 3: Implementation of two-way association, using pointer

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

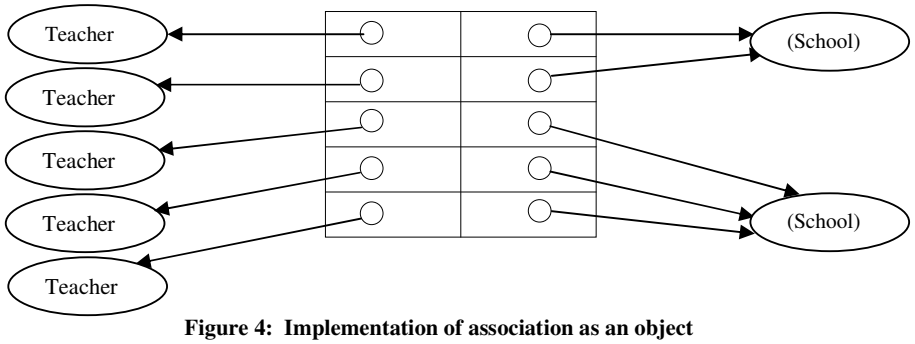

Figure 4: Implementation of association as an object

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

☞ Check Your Progress 3

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

……

……

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

……

……

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

……

……

2.9 SUMMARY

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

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

2.10 SOLUTIONS/ANSWERS

Check Your Progress 1

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

Object Model = Object model diagram + data dictionary

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

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

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

Check Your Progress 2

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

Check Your Progress 3

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