UNIT – 3

Object oriented Design

Unit-03/Lecture-01

Overview of object design [RGPV /DEC-2012(10),June-2012(10)]

- Object-oriented analysis, design and programming are related but distinct.
- OOA is concerned with developing an object model of the application domain.
- OOD is concerned with developing an object-oriented system model to implement requirements.
- OOP is concerned with realising an OOD using an OO programming language such as Java or C++.

Characteristics of OOD
- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated.
- Objects communicate by message passing.
- Objects may be distributed and may execute sequentially or in parallel.

Advantages of OOD
- Easier maintenance. Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

Object Design
The object design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations. The object design phase adds internal objects for implementation and optimizes data structures and algorithms.

Overview of Object Design
During object design, the designer carries out the strategy chosen during the system design and fleshes out the details. There is a shift in emphasis from application domain concepts toward computer concepts. The objects discovered during analysis serve as the skeleton of the design, but the object designer must choose among different ways to implement them with an eye toward minimizing execution time, memory and other measures of cost. The operations identified during the analysis must be expressed as algorithms, with complex operations decomposed into simpler internal operations.

The classes, attributes and associations from analysis must be implemented as specific data structures. New object classes must be introduced to store intermediate results during program execution and to avoid the need for recomputation. Optimization of the design should not be carried to excess, as ease of implementation, maintainability, and extensibility are also important concerns.

Steps of Design:

During object design, the designer must perform the following steps:

1. Combining the three models to obtain operations on classes.

2. Design algorithms to implement operations.

3. Optimize access paths to data.

4. Implement control for external interactions

5. Adjust class structure to increase inheritance. 6. Design associations.

7. Determine object representation.

8. Package classes and associations into modules.

Combining the three models to obtain operations on classes.

After analysis, we have object, dynamic and functional model, but the object model is the main framework around which the design is constructed. The object model from analysis may not show operations. The designer must convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model. Each state diagram describes the life history of an object. A transition is a change of state of the object and maps into an operation on the object.
We can associate an operation with each event received by an object. In the state diagram, the action performed by a transition depends on both the event and the state of the object. Therefore, the algorithm implementing an operation depends on the state of the object. If the same event can be received by more than one state of an object, then the code implementing the algorithm must contain a case statement dependent on the state. An event sent by an object may represent an operation on another object.
Events often occur in pairs, with the first event triggering an action and the second event returning the result on indicating the completion of the action. In this case, the event pair can be mapped into an operation performing the action and returning the control provided that the events are on a single thread. An action or activity initiated by a transition in a state diagram may expand into an entire dfd in the functional model .The network of processes within the dfd represents the body of an operation.
The flows in the diagram are intermediate values in operation. The designer convert the graphic structure of the diagram into linear sequence of steps in the algorithm .The process in the dfd represent sub operations. Some of them, but

not necessarily all may be operations on the original target object or on other objects. Determine the target object of a sub operation as follows:

* If a process extracts a value from input flow then input flow is the target.

* Process has input flow or output flow of the same type, input output flow is the target.

* Process constructs output value from several input flows, then the operation is a class operation on output class.

* If a process has input or an output to data store or actor, data store or actor is the target.

S.NO	**RGPV QUESTIONS**	**Year**	**Marks**
1.	What is object design? Explain the idea behind designing the object	RGPV, Dec. 2012	10
2.	What are various steps involved in object oriented design? Explain in brief	RPPV, JUNE 2012	10

UNIT-03

TOPIC: DESIGNING ALGORITHMS

UNIT-03/LECTURE-02

Designing algorithms [RGPV /DEC-2012(10),DEC-2011(10)]

Each operation specified in the functional model must be formulated as an algorithm. The analysis specification tells what the operation does from the view point of its clients, but the algorithm shows how it is done. The analysis specification tells what the operation does from the view point of its clients, but the algorithm shows how it is done. An algorithm may be subdivided into calls on simpler operations, and so on recursively, until the lowest-level operations are simple enough to implement directly without refinement .The algorithm designer must decide on the following:

i) Choosing algorithms

Many operations are simple enough that the specification in the functional model already constitutes a satisfactory algorithm because the description of what is done also shows how it is done. Many operations simply traverse paths in the object link network or retrieve or change attributes or links.

Non trivial algorithm is needed for two reasons:

- To implement functions for which no procedural specification
- To optimize functions for which a simple but inefficient algorithm serves as a definition.

Some functions are specified as declarative constraints without any procedural definition. In such cases, you must use your knowledge of the situation to invent an algorithm. The essence of most geometry problems is the discovery of appropriate algorithms and the proof that they are correct. Most functions have simple mathematical or procedural definitions. Often the simple definition is also the best algorithm for computing the function or else is also so close to any other algorithm that any loss in efficiency is the worth the gain in clarity. In other cases, the simple definition of an operation would be hopelessly inefficient and must be implemented with a more efficient algorithm.

For example, let us consider the algorithm for search operation .A search can be done in two ways like binary search (which performs log n comparisons on an average) and a linear search (which performs n/2 comparisons on an average).Suppose our search algorithm is implemented using linear search , which needs more comparisons. It would be better to implement the search with a much efficient algorithm like binary search.

Considerations in choosing among alternative algorithm include:

a) Computational Complexity:

It is essential to think about complexity i.e. how the execution time (memory) grows with the number of input values.

For example: For a bubble sort algorithm, time ∞ n^2

Most other algorithms, time ∞ n log n

b) Ease of implementation and understand ability:

It is worth giving up some performance on non critical operations if they can be implemented quickly with a simple algorithm.

c) Flexibility:

Most programs will be extended sooner or later. A highly optimized algorithm often sacrifices readability and ease of change. One possibility is to provide two Implementations of critical applications, a simple but inefficient algorithm that can be implemented, quickly and used to validate the system, and a complicated but efficient algorithm whose correct implementation can be checked against the simple one.

d) Fine Timing the Object Model:

We have to think, whether there would be any alternatives, if the object model were structured differently.

ii) Choosing Data Structures

Choosing algorithms involves choosing the data structures they work on. We must choose the form of data structures that will permit efficient algorithms. The data structures do not add information to the analysis model, but they organize it in a form convenient for the algorithms that use it.

iii) Defining Internal Classes and Operations

During the expansion of algorithms, new classes of objects may be needed to hold intermediate results. New, low level operations may be invented during the decomposition of high level operations. A complex operation can be defined in terms of lower level operations on simpler objects. These lower level operations must be defined during object design because most of them are not externally visible. Some of these operations were found from “shopping –list”. There is a need to add new internal operations as we expand high –level functions. When you reach this point during the design phase, you may have to add new classes that were not mentioned directly in the client’s description of the problem. These low-level classes are the implementation elements out of which the application classes are built.

iv) Assigning Responsibility for Operations

Many operations have obvious target objects, but some operations can be performed at several places in an algorithm, by one of the several places, as long as they eventually get done. Such operations are often part of a complex high-level operation with many consequences. Assigning responsibility for such operations can be easy to overlook in laying out object classes because they are not an inherent part of one class. When a class is meaningful in the real world, then the operations on it are usually clear. During implementation, internal classes are introduced.

How do you decide what class owns an operation?

When only one object is involved in the operation, tell the object to perform the operation. When more than one object is involved, the designer must decide which object plays the lead role in the operation. For that, ask the following questions:

Is one object acted on while the other object performs the action? It is best to associate the operation with the target of the operation, rather than the initiator.

Is one object modified by the operation, while other objects are only queried for the information they contain? The object that is changed is the target. Looking at the classes and associations that are involved in the operation, which class is the most centrally-located in this sub network of the object model? If the classes and associations form a star about a single central class, it is the target of the operation.

If the objects were not software, but the real world objects represented internally, what real world objects would you push, move, activate or manipulate to initiate operation?

Assigning an operation within a generalization hierarchy can be difficult. Since the definitions of the subclasses within the hierarchy are often fluid and can be adjusted during design as convenient. It is common to move an operation up and down in the hierarchy during design, as its scope is adjusted

S.NO	RGPV QUESTIONS	Year	Marks
1.	What do you understand by algorithm designing?	RGPV, Dec. 2012	10
2.	Discuss in detail the process of designing algorithms in object oriented design	RGPV, Dec. 2011	10

UNIT-03
TOPIC: DESIGN OPTIMIZATION
UNIT-03/LECTURE-03

Design Optimization [RGPV /DEC-2010(10),DEC-2011(10)]

The basic design model uses the analysis model as the framework for implementation.
The analysis model captures the logical information about the system, while the design model must add details to support efficient information access. The inefficient but semantically correct analysis model can be optimized to make the implementation more efficient, but an optimized system is more obscure and less likely to be reusable in another context. The designer must strike an appropriate balance between efficiency and clarity. During design optimization, the designer must
Add Redundant Associations for Efficient Access During analysis, it is undesirable to have redundancy in association network because redundant associations do not add any information. During design, however we evaluate the structure of the object model for an implementation. For that, we have to answer the following questions:

* Is there a specific arrangement of the network that would optimize critical aspects of the completed system?
* Should the network be restructured by adding new associations? * Can existing associations be omitted?
The associations that were useful during analysis may not form the most efficient network when the access patterns and relative frequencies of different kinds of access are considered. In cases where the number of hits from a query is low because only a fraction of objects satisfy the test, we can build an index to improve access to objects that must be frequently retrieved.

i) Analyze the use of paths in the association network as follows:
Examine each operation and see what associations it must traverse to obtain its information. Note which associations are traversed in both directions, and which are traversed in a single direction only, the latter can be implemented efficiently with one way pointers.

For each operation note the following items:

How often is the operation called? How costly is 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, don’t worry about such small effects.
What is the fraction of “hits” on the final class, that is, objects that meet selection criteria (if any) and is operated on? If most objects are rejected during the traversal for some reason, then a simple nested loop may be inefficient at finding target objects. Provide indexes for

frequent, costly operations with a low hit ratio because such operations are inefficient to implement using nested loops to traverse a path in the network.

ii) Rearranging Execution Order for Efficiency

After adjusting the structure of the object model to optimize frequent traversal, the next thing to optimize is the algorithm itself. Algorithms and data structures are directly related to each other, but we find that usually the data structure should be considered first. One key to algorithm optimization is to eliminate dead paths as early as possible. Sometimes the execution order of a loop must be inverted.

iii) Saving Derived Attributes to Avoid Re computation:

Data that is redundant because it can be derived from other data can be “cached” or store in its computed form to avoid the overhead of re computing it. The class that contains the cached data must be updated if any of the objects that it depends on are changed.
Derived attributes must be updated when base values change. There are 3 ways to recognize when an update is needed:

Explicit update: Each 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 the update operation on the base object to explicitly update the derived attributes that depend on it.

Periodic Re computation: Base values are updated in bunches. Re compute all derived attributes periodically without re computing derived attributes after each base value is changed. Re computation 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 incremental approach. Periodic re computation is simpler than explicit update and less prone to bugs. On the other hand, if the data set changes incrementally a few objects at a time, periodic re computation 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 all dependent values, but the calling code need not explicitly invoke the updates. It provides modularity

S.NO	RGPV QUESTIONS	Year	Marks
1.	What do you understand by “ Design Optimization”	RGPV, Dec. 2010	10
2.	Explain what you understand by “Design Optimization” with the help of suitable examples.	RGPV, Dec. 2011	10

UNIT-03

TOPIC:IMPLEMENTATION OF CONTROL

UNIT-03/LECTURE-04

Implementation of Control [RGPV /June-2005(10)]

The designer must refine the strategy for implementing the state – event models present in the dynamic model. As part of system design, you will have chosen a basic strategy for realizing dynamic model, during object design flesh out this strategy.

There are three basic approaches to implementing the dynamic model:

i) State as Location within a Program:

This is the traditional approach to representing control within a program. The location of control within a program implicitly defines the program state. Any finite state machine can be implemented as a program. Each state transition corresponds to an input statement. After input is read, the program branches depending on the input event received. Each input statement need to handle any input value that could be received at that point. In highly nested procedural code, low –level procedures must accept inputs that they may know nothing about and pass them up through many levels of procedure calls until some procedure is prepared to handle them. One technique of converting state diagram to code is as follows:

1. Identify the main control path. Beginning with the initial state, identify a path through the diagram that corresponds to the normally expected sequence of events. Write the name of states along this path as a linear sequence of events. Write the names of states along this path as a linear sequence .This becomes a sequence of statements in the program.
2. Identify alternate paths that branch off the main path and rejoin it later. These become conditional statements in the program.
3. Identify backward paths that branch off the main loop and rejoin it earlier .These become loops in program. If multiple backward paths that do not cross, they become nested loops. Backward paths that cross do not nest and can be implemented with goto if all else fails, but these are rare.
4. The status and transitions that remain correspond to exception conditions. They can be

handled using error subroutines , exception handling supported by the language , or setting and testing of status flags. In the case of exception handling, use goto statements.

ii) State machine engine

The most direct approach to control is to have some way of explicitly representing and executing state machine. For example, state machine engine class helps execute state machine represented by a table of transitions and actions provided by the application.

Each object instance would contain its own independent state variables but would call on the state engine to determine next state and action.

This approach allows you to quickly progress from analysis model to skeleton prototype of the system by defining classes from object model state machine and from dynamic model and creating “stubs” of action routines.

A stub is a minimal definition of function /subroutine without any internal code. Thus if each stub prints out its name, technique allows you to execute skeleton application to verify that basic flow of control is correct. This technique is not so difficult.

iii) Control as Concurrent Tasks

An object can be implemented as task in programming language /operating system. It preserves inherent concurrency of real objects.

Events are implemented as inter task calls using facilities of language/operating system. Concurrent C++/Concurrent Pascal support concurrency. Major Object Oriented languages do not support concurrency.

“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 state 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.

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.

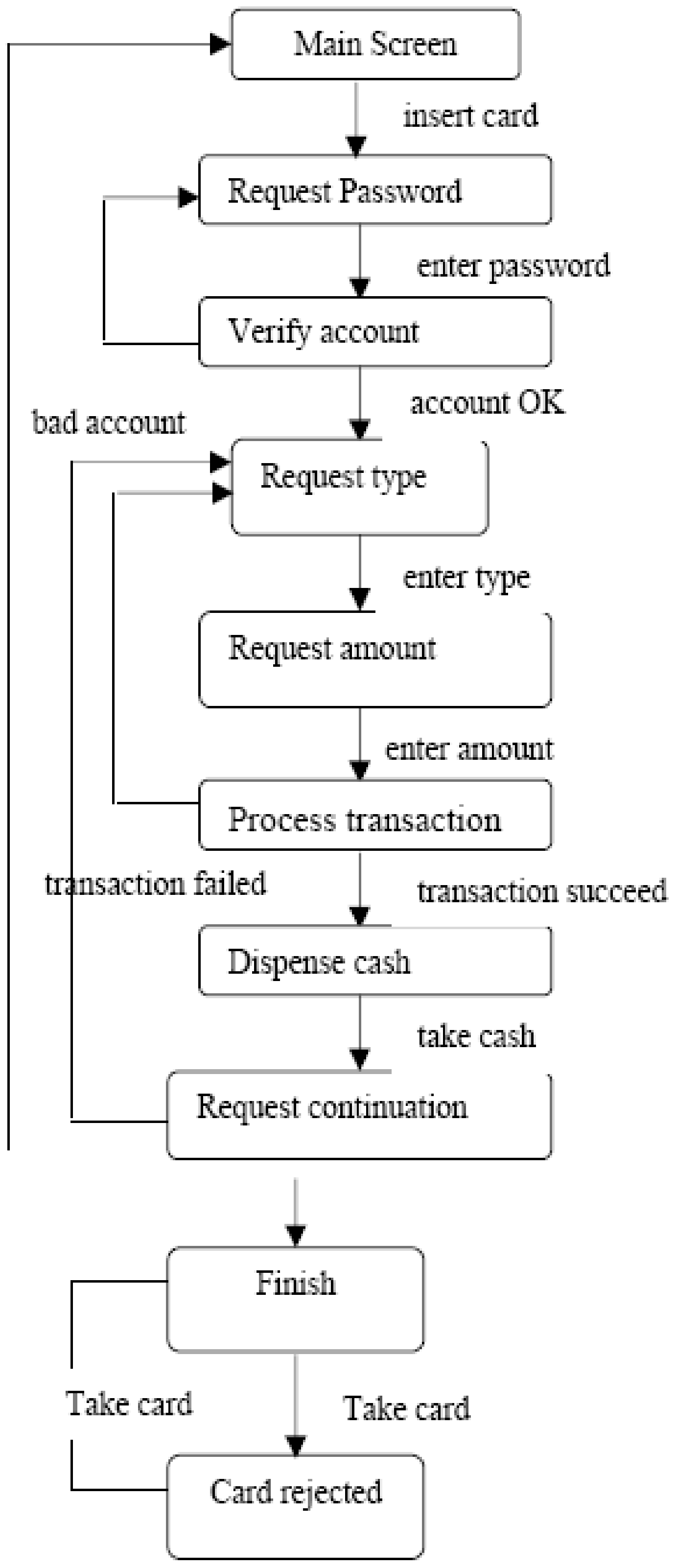
c) Identify all backward paths that branch off the main loop of the program and rejoin it earlier. This could be the loop in the program. All non-intersecting backward paths become nested loops in the program.

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 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.



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.

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.

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.

S.NO	**RGPV QUESTIONS**	**Year**	**Marks**
1.	What are the important issues in implementing object-oriented systems?	RGPV, June 2005	10

UNIT-03

TOPIC: DESIGN OF AASSOCIATIONS

UNIT-03/LECTURE-05

Design of Associations [RGPV /DEC-2003(10),DEC-2007(10)]

During object design phase, we must formulate a strategy for implementing all associations in the object model. We can either choose a global strategy for implementing all associations uniformly, or a particular technique for each association.

i) Analyzing Association Traversal

Associations are inherently bidirectional. If association in your application is traversed in one direction, their implementation can be simplified. The requirements on your application may change , you may need to add a new operation later that needs to traverse the association in reverse direction. For prototype work, use bidirectional association so that we can add new behaviour and expand /modify. In the case of optimization work, optimize some associations.

ii) One-way association

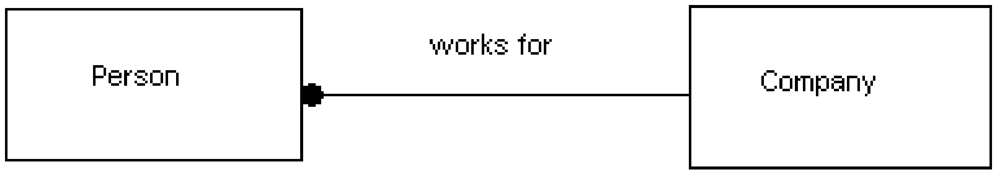
* If an association is only traversed in one direction it may be implemented as pointer.
* If multiplicity is „many‟ then it is implemented as a set of pointers. * If the “many” is ordered, use list instead of set .
* A qualified association with multiplicity one is implemented as a dictionary object(A dictionary is a set of value pairs that maps selector values into target values. * Qualified association with multiplicity “many” are rare.(it is implemented as dictionary set of objects).

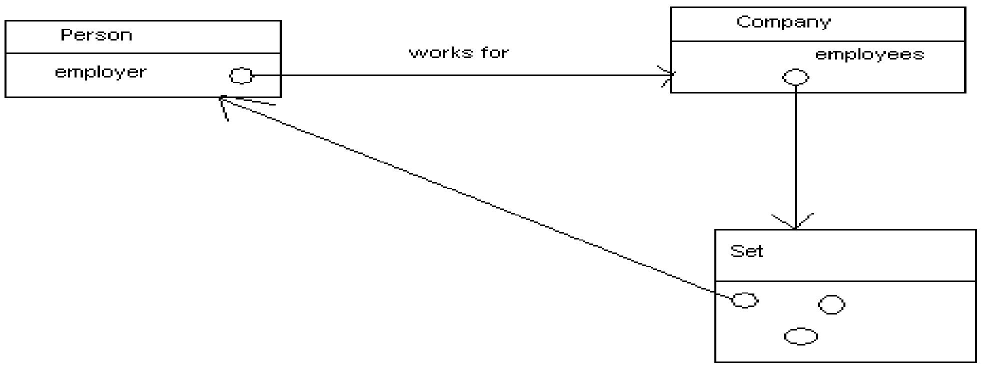
iii) Two-way associations

Many associations are traversed in both directions, although not usually with equal frequency. There are three approaches to their implementation:

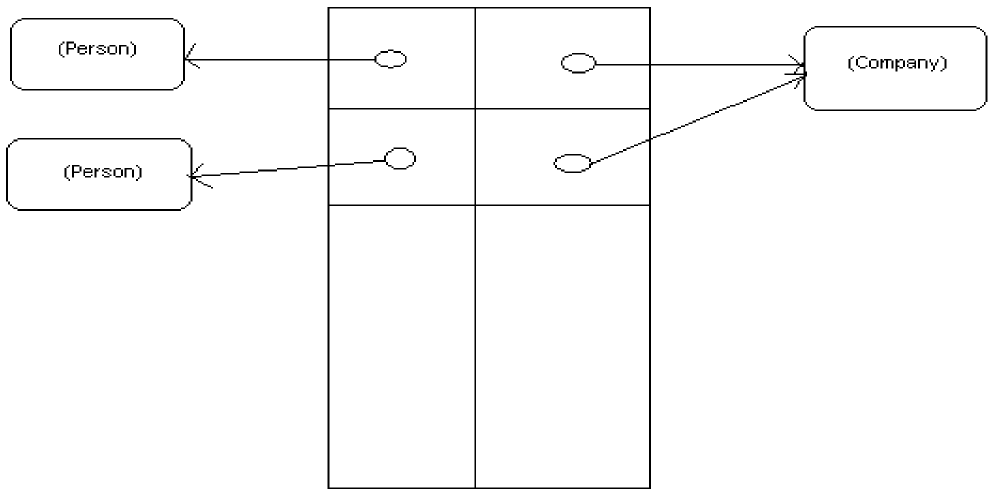
Implement as an attribute in one direction only and perform a search when a backward traversal is required. This approach is useful only if there is great disparity in traversal frequency and minimizing both the storage cost and update cost are important.

Implement as attributes in both directions. 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 if accesses outnumber updates





Implement as a distinct association object independent of either class. An association object is a set of pairs of associated objects stored in a single variable size object. An association object can be implemented using two dictionary object one for forward direction and other for reverse direction.



iv) Link Attributes

Its implementation depends on multiplicity.

- If it is a one-one association, link attribute is stored in any one of the classes involved.

- If it is a many-one association, the link attribute can be stored as attributes of many object, since each " many object appears only once in the association.

- If it is a many-many association, the link attribute can‟t be associated with either object; implement association as distinct class where each instance is one link and its attributes.

S.NO	RGPV QUESTIONS	Year	Marks
1.	Write Short note on design of associations	RGPV, Dec. 2003	10
2.	Write explanatory short note on design of association	RGPV, Dec. 2007	10

UNIT-03

TOPIC: INHERITANCE ADJUSMENT

UNIT-03/LECTURE-06

INHERITANCE ADJUSTMENT [RGPV /DEC-2011(10)]

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 behaviour out of groups of classes
- Use delegation to share behaviour 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 colour parameter, but the parameter can be accepted and ignored for consistency with colour 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 Behaviour 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 seem 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 behaviour 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 behaviour. Usually, the resulting super class is abstract, meaning that there are no direct instances of it, but the behaviour 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 colour, 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

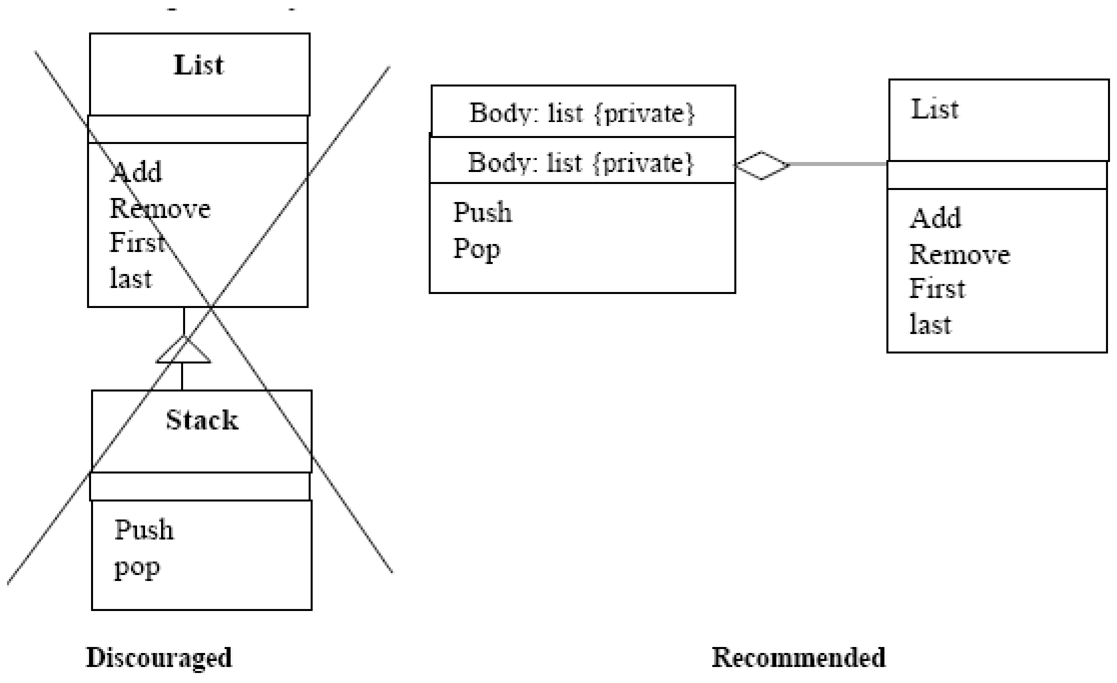
As we now know, inheritance means the sharing of to the behaviour of a super class by its subclass. Let us see how delegation could be used for this purpose. Before we use delegation, let us try to understand that what actually delegation can do.

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 accide*ntally". It is true that Inheritance is a mechanism for implementing generalization, in which the behaviour of super class is shared by all its subclasses. But, sharing of behaviour 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. 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.



Alternative implementations of a Stack using inheritance (left) and delegation (right)
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.

S.NO	RGPV QUESTIONS	Year	Marks
1.	Explain the adjustment of inheritance in classes and operations	RGPV, Dec. 2011	10

UNIT-03

TOPIC:OBJECT REPRESENTATION

UNIT 3/LECTURE 7

_Object Representation [RGPV /DEC-2012(10)]

Implementing objects is mostly straight forward, but the designer must choose when to use primitive types in representing objects and when to combine groups of related objects. Classes can be defined in terms of other classes, but eventually everything must be implemented in terms of built-in-primitive data types, such as integer strings, and enumerated types. For example, consider the implementation of a social security number within an employee object. It can be implemented as an attribute or a separate class.
Defining a new class is more flexible but often introduces unnecessary indirection. In a similar vein, the designer must often choose whether to combine groups of related objects 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.

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.

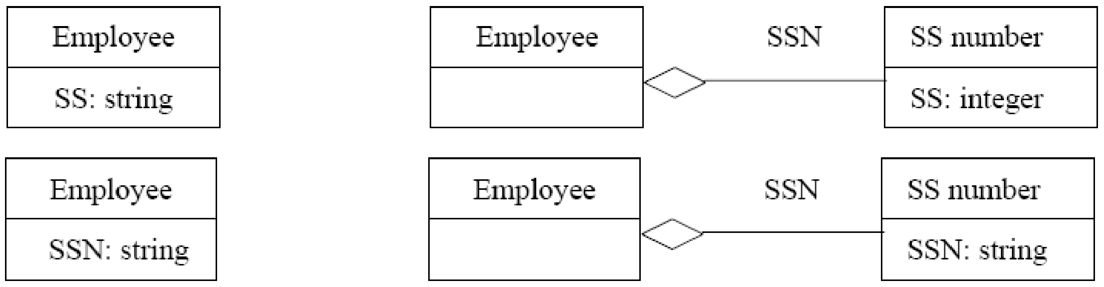


Figure 6: Alternative representations for an attribute

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

S.NO	RGPV QUESTION	YEAR	MARKS
1.	How objects are represented? Explain use-case driven approach.	RGPV, Dec. 2012	10

UNIT-03

TOPIC: PHYSICAL PACKAGING

UNIT 3/LECTURE 8

Physical Packaging [RGPV /June-2011(10),June-2012(10)]

Programs are made of discrete physical units that can be edited, compiled, imported, or otherwise manipulated. In C and Fortran the units are source files; In Ada, it is packages. In object oriented languages, there are various degrees of packaging. In any large project, careful partitioning of an implementation into packages is important to permit different persons to cooperatively work on a program.

Packaging involves the following issues:

i) Hiding internal information from outside view.

One design goal is to treat classes as „black boxes‟ , whose external interface is public but whose internal details are hidden from view. Hiding internal information permits implementation of a class to be changed without requiring any clients of the class to modify code. Additions and changes to the class are surrounded by “fire walls” that limit the effects of any change so that changes can be understood clearly. Trade off between information hiding and optimization activities. During analysis, we are concerned with information hiding. During design, the public interface of each class must be defined carefully. The designer must decide which attributes should be accessible from outside the class. These decisions should be recorded in the object model by adding the annotation {private} after attributes that are to be hidden, or by separating the list of attributes into 2 parts. Taken to an extreme a method on a class could traverse all the associations of the object model to locate and access another object in the system .This is appropriate during analysis, but methods that know too much about the entire model are fragile because any change in representation invalidates them. During design we try to limit the scope of any one method. We need top define the bounds of visibility that each method requires.

Specifying what other classes a method can see defines the dependencies between classes. Each operation should have a limited knowledge of the entire model, including the structure of Classes, associations and operations. The fewer things that an operation knows about, the less likely it will be affected by any changes. The fewer operations know about details of a class, the easier the class can be changed if needed.

The following design principles help to limit the scope of knowledge of any operation:

- Allocate to each class the responsibility of performing operations and providing information that pertains to it.
- Call an operation to access attributes belonging to an object of another class Avoid traversing associations that are not connected to the current class. Define interfaces at as high a level of abstraction as possible.
- Hide external objects at the system boundary by defining abstract interface classes, that is, classes that mediate between the system and the raw external objects.
- Avoid applying a method to the result of another method, unless the result class is already a supplier of methods to the caller. Instead consider writing a method to combine the two operations.

ii) Coherence of entities.

One important design principle is coherence of entities. An entity, such as a class, an operation, or a module, is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal. It shouldn't be a collection of unrelated parts. A method should do one thing well .a single method should not contain both policy and implementation.

"A policy is the making of context dependent decisions." "Implementation is the execution of fully specified algorithms."

Policy involves making decisions, gathering global information, interacting with outside world and interpreting special cases. Policy methods contain input output statements, conditionals and accesses data stores. It doesn't contain complicated algorithms but instead calls various implementation methods. An implementation method does exactly one operation without making any decisions, assumptions, defaults or deviations .All information is supplied as arguments (list is long). Separating policy and implementation increase reusability. Therefore implementation methods don't contain any context dependency. So they are likely to be reusable Policy method need to be rewritten in an application , they are simple and consists of high level decisions and calls on low-level methods. A class shouldn't serve too many purposes.

iii) Constructing physical modules.

During analysis and system design phases we partitioned the object model into modules.

* The initial organization may not be suitable for final packaging of system implementation

New classes added to existing module or layer or separate module.

Modules should be defined so that interfaces are minimal and well defined. Connectivity of object model can be used as a guide for partitioning modules. Classes that are closely connected by associations should be in the same module. Loosely connected classes should be grouped in separate modules. Classes in a module should represent similar kinds of things in the application or should be components of the same composite object.

Try to encapsulate strong coupling within a module. Coupling is measured by number of different operations that traverse a given association. The number expresses the number of different ways the association is used, not the frequency.

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.

S.NO	RGPV QUESTION	YEAR	MARKS
1.	Describe the various issues involved in packaging in programs.	RGPV, June 2011	10
2.	Discuss the issues involved in packaging	RGPV, June. 2012	10

UNIT 3

TOPIC:DOCUMENTING DESIGN DECISIONS

UNIT 3/LECTURE 9

Documenting Design Decisions [RGPV /DEC-2011(10),DEC-2012(10)]

The above design decisions must be documented when they are made, or you will become confused. This is especially true if you are working with other developers. It is impossible to remember design details for any non trivial software system, and documentation is the best way of transmitting the design to others and recording it for reference during maintenance.

The design document is an extension of the Requirements Analysis Document.

-> The design document includes revised and much more detailed description of the object model-both graphical and textual. Additional notation is appropriate for showing implementation decisions, such as arrows showing the traversal direction of associations and pointers from attributes to other objects.

-> Functional model will also be extended. It specifies all operation interfaces by giving their arguments, results, input-output mappings and side effects.

-> Dynamic model – if it is implemented using explicit state control or concurrent tasks then the analysis model or its extension is adequate. If it is implemented by location within program code, then structured pseudocode for algorithms is needed.

Keep the design document different from analysis document .The design document includes many optimizations and implementation artefacts. It helps in validation of software and for reference during maintenance. Traceability from an element in analysis to element in design document should be straightforward. Therefore the design document is an evolution of analysis model and retains same names.

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 artefacts. 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 modelling

S.NO	RGPV QUESTION	YEAR	MARKS
1.	Discuss the documentation of design decisions	RGPV, Dec. 2011	10
2.	Explain physical packaging and documenting design decision	RGPV, Dec. 2012	10