Unit 3

Object-Oriented Analysis

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

The primary tasks in object-oriented analysis (OOA) are

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

Object-Oriented Analysis

Object Oriented Analysis (OOA):

Object Oriented Analysis (OOA) is the first technical activity performed as part of object oriented software engineering. OOA introduces new concepts to investigate a problem. It is based in a set of basic principles, which are as follows-

1. The information domain is modeled.
2. Behavior is represented.
3. Function is described.
4. Data, functional, and behavioral models are divided to uncover greater detail.
5. Early models represent the essence of the problem, while later ones provide implementation
   details.

Object-Oriented Design

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include −

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

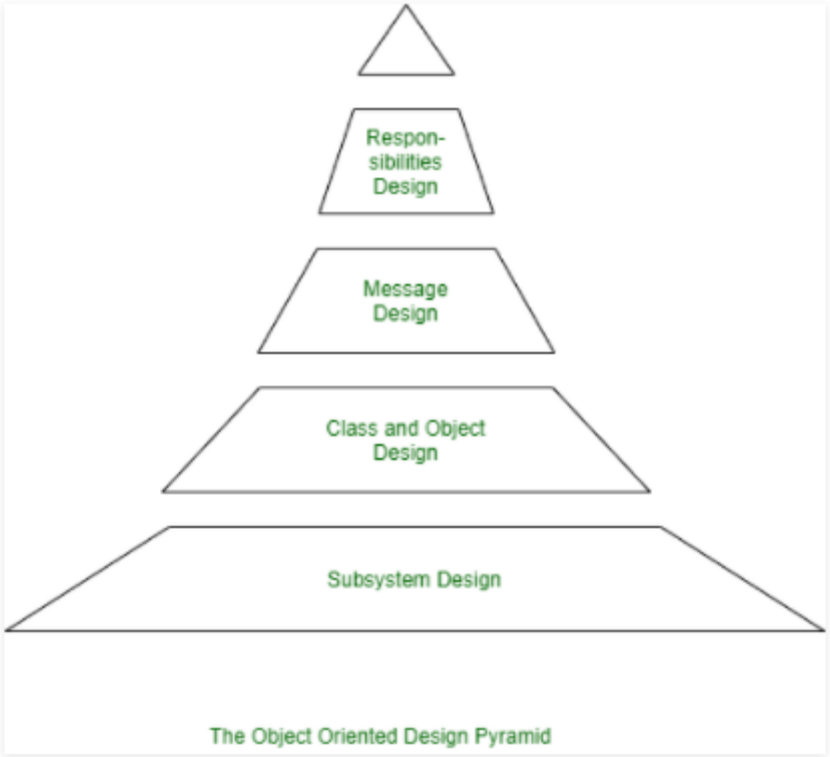
Grady Booch has defined object-oriented design as *"a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design"*.

Object Oriented Design (OOD):

An analysis model created using object oriented analysis is transformed by object oriented design into a design model that works as a plan for software creation. OOD results in a design having several different levels of modularity i.e., The major system components are partitioned into subsystems (a system level "modular"), and data their manipulation operations are encapsulated into objects (a modular form that is the building block of an OO system.).

Object-Oriented Design

Shows a design pyramid for object oriented systems. It is having the following four layers.



The Object Oriented Design Pyramid

Object-Oriented Design

1. **The Subsystem Layer :**
   It represents the subsystem that enables software to achieve user requirements and implement technical frameworks that meet user needs.

2. **The Class and Object Layer :**
   It represents the class hierarchies that enable the system to develop using generalization and specialization. This layer also represents each object.

3. **The Message Layer :**
   It represents the design details that enable each object to communicate with its partners. It establishes internal and external interfaces for the system.

4. **The Responsibilities Layer :**
   It represents the data structure and algorithmic design for all the attributes and operations for each object.

Object-Oriented Programming

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are −

- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Phases in Object oriented software development

1. Object – oriented analysis
2. Object – oriented design
3. Object – oriented implementation

Object–Oriented Analysis

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects. The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

Object–Oriented Design

Object-oriented design includes two main stages, namely, system design and object design.

System Design

In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model and the proposed system architecture. Here, the emphasis is on the objects comprising the system rather than the processes in the system.

Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. All the classes required are identified. The designer decides whether –

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- new classes should be inherited from the existing classes.

Phases in Object oriented software development (contd..)

Object–Oriented Implementation and Testing

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code.

Principles of Object oriented systems

The conceptual framework of object–oriented systems is based upon the object model. There are two categories of elements in an object-oriented system –

Major Elements – By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are –

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Minor Elements – By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are –

- Typing
- Concurrency
- Persistence

Principles of Object oriented systems

The conceptual framework of object–oriented systems is based upon the object model. There are two categories of elements in an object-oriented system –

Major Elements – By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are –

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Minor Elements – By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are –

- Typing
- Concurrency
- Persistence

Principles of Object oriented systems (contd..)

Modularity

Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem. Booch has defined modularity as –

"Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."

Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter–module interaction or coupling is low.

Hierarchy

In Grady Booch's words, "Hierarchy is the ranking or ordering of abstraction". Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of "divide and conquer". Hierarchy allows code reusability.

The two types of hierarchies in OOA are –

- **"IS–A" hierarchy** – It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose "is–a" flower.
- **"PART–OF" hierarchy** – It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a "part–of" flower.

Principles of Object oriented systems (contd..)

Typing

According to the theories of abstract data type, a type is a characterization of a set of elements. In OOP, a class is visualized as a type having properties distinct from any other types. Typing is the enforcement of the notion that an object is an instance of a single class or type. It also enforces that objects of different types may not be generally interchanged; and can be interchanged only in a very restricted manner if absolutely required to do so.

The two types of typing are –

- **Strong Typing** – Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.
- **Weak Typing** – Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

Concurrency

Concurrency in operating systems allows performing multiple tasks or processes simultaneously. When a single process exists in a system, it is said that there is a single thread of control. However, most systems have multiple threads, some active, some waiting for CPU, some suspended, and some terminated. Systems with multiple CPUs inherently permit concurrent threads of control; but systems running on a single CPU use appropriate algorithms to give equitable CPU time to the threads so as to enable concurrency.

In an object-oriented environment, there are active and inactive objects. The active objects have independent threads of control that can execute concurrently with threads of other objects. The active objects synchronize with one another as well as with purely sequential objects.

Persistence

An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

Three analysis techniques (Models) used in OOA

Object Modelling

The process of object modelling can be visualized in the following steps −

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modelling

The process of dynamic modelling can be visualized in the following steps −

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

The process of functional modelling can be visualized in the following steps −

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are –

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

Advantages/Disadvantages of Object Oriented Analysis

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Advantages/Disadvantages of Structured Analysis

Advantages	Disadvantages
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

Dynamic modeling

The dynamic model represents the time–dependent aspects of a system. It is concerned with the temporal changes in the states of the objects in a system. The main concepts are −

- State, which is the situation at a particular condition during the lifetime of an object.
- Transition, a change in the state
- Event, an occurrence that triggers transitions
- Action, an uninterrupted and atomic computation that occurs due to some event, and
- Concurrency of transitions.

Dynamic modelling (contd..)

State: The state is an abstraction given by the values of the attributes that the object has at a particular time period. It is a situation occurring for a finite time period in the lifetime of an object, in which it fulfils certain conditions, performs certain activities, or waits for certain events to occur. In state transition diagrams, a state is represented by rounded rectangles.

Parts of a state

Name − A string differentiates one state from another. A state may not have any name.
Entry/Exit Actions − It denotes the activities performed on entering and on exiting the state.
Internal Transitions − The changes within a state that do not cause a change in the state.
Sub–states − States within states.
Initital state and Final state.

Dynamic modelling (contd..)

Transition:A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state.

The five parts of a transition are −

Source State − The state affected by the transition.
Event Trigger − The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.
Guard Condition − A Boolean expression which if True, causes a transition on receiving the event trigger.
Action − An un-interruptible and atomic computation that occurs on the source object due to some event.
Target State − The destination state after completion of transition.

Dynamic modelling (contd..)

Transition:A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state.

The five parts of a transition are –

Source State − The state affected by the transition.

Event Trigger − The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.

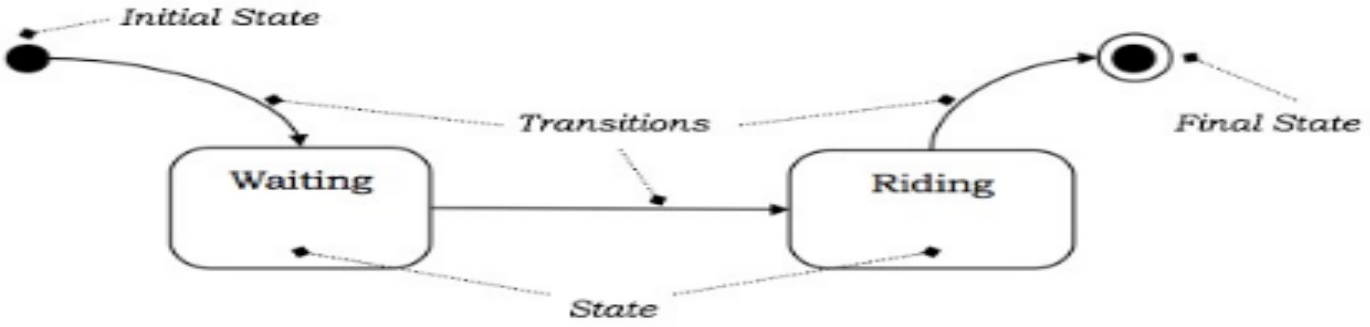
Guard Condition − A Boolean expression which if True, causes a transition on receiving the event trigger.

Action − An un-interruptible and atomic computation that occurs on the source object due to some event.

Target State − The destination state after completion of transition.

Example

Suppose a person is taking a taxi from place X to place Y. The states of the person may be: Waiting (waiting for taxi), Riding (he has got a taxi and is travelling in it), and Reached (he has reached the destination). The following figure depicts the state transition.



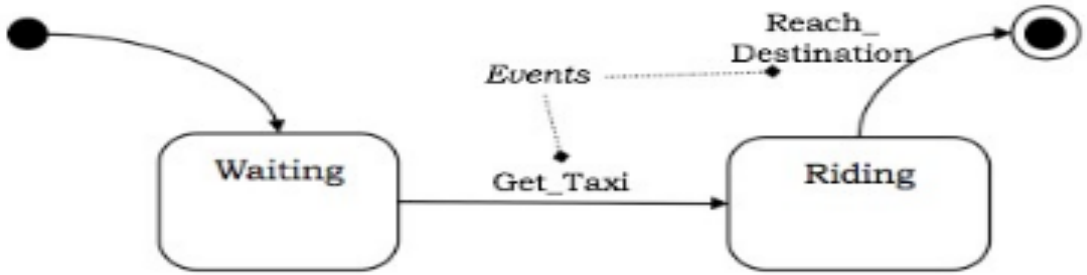
Dynamic modelling (contd..)

Events: Events are some occurrences that can trigger state transition of an object or a group of objects. Events have a location in time and space but do not have a time period associated with it. Events are generally associated with some actions.
Examples of events are mouse click, key press, an interrupt, stack overflow, etc.

Types of events:
External events
Internal events
Deferred events

Example

Considering the example shown in the above figure, the transition from Waiting state to Riding state takes place when the person gets a taxi. Likewise, the final state is reached, when he reaches the destination. These two occurrences can be termed as events Get_Taxi and Reach_Destination. The following figure shows the events in a state machine.



Dynamic modelling (contd..)

Actions

Activity: Activity is an operation upon the states of an object that requires some time period.

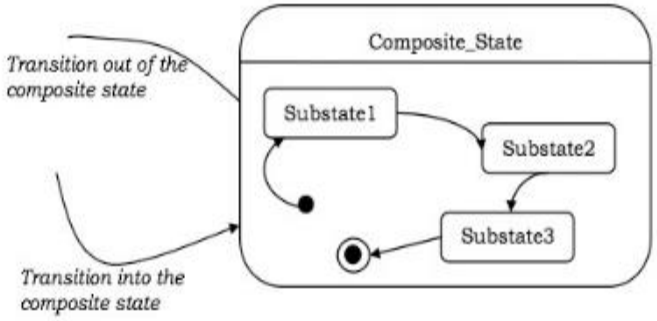
Action: An action is an atomic operation that executes as a result of certain events. By atomic, it is meant that actions are un-interruptible, i.e., if an action starts executing, it runs into completion without being interrupted by any event.

Entry and exit action: Entry action is executed at entering state and exit at leaving state.

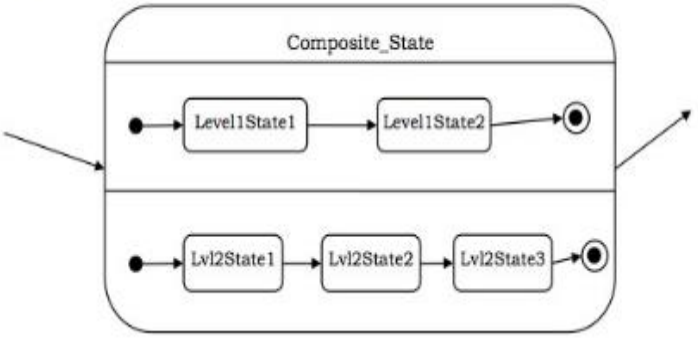
Scenerio: Scenario is a description of a specified sequence of actions. It depicts the behavior of objects undergoing a specific action series.

Concurrency of events

The following figure illustrates the concept of sequential sub-states.



The following figure shows the concept of concurrent sub-states.



Dynamic modelling (contd..)

Diagrams

1. **Interaction diagram**

    1) Sequence diagram
    2) Collaboration diagram

2. State transition diagram

Functional modeling

Functional Modelling gives the process perspective of the object-oriented analysis model and an overview of what the system is supposed to do. It defines the function of the internal processes in the system with the aid of Data Flow Diagrams (DFDs). It depicts the functional derivation of the data values without indicating how they are derived when they are computed, or why they need to be computed.

Data Flow Diagrams

Functional Modelling is represented through a hierarchy of DFDs. The DFD is a graphical representation of a system that shows the inputs to the system, the processing upon the inputs, the outputs of the system as well as the internal data stores. DFDs illustrate the series of transformations or computations performed on the objects or the system, and the external controls and objects that affect the transformation.

Rumbaugh et al. have defined DFD as, "A data flow diagram is a graph which shows the flow of data values from their sources in objects through processes that transform them to their destinations on other objects."

The four main parts of a DFD are −

- Processes,
- Data Flows,
- Actors, and
- Data Stores.

The other parts of a DFD are −

- Constraints, and
- Control Flows.

Functional modeling (contd..)

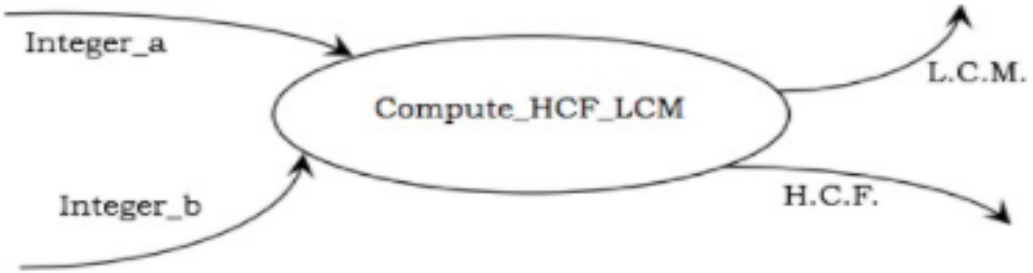
Features of a DFD

Processes

Processes are the computational activities that transform data values. A whole system can be visualized as a high-level process. A process may be further divided into smaller components. The lowest-level process may be a simple function.

Representation in DFD – A process is represented as an ellipse with its name written inside it and contains a fixed number of input and output data values.

Example – The following figure shows a process Compute_HCF_LCM that accepts two integers as inputs and outputs their HCF (highest common factor) and LCM (least common multiple).



Data Flows

Data flow represents the flow of data between two processes. It could be between an actor and a process, or between a data store and a process. A data flow denotes the value of a data item at some point of the computation. This value is not changed by the data flow.

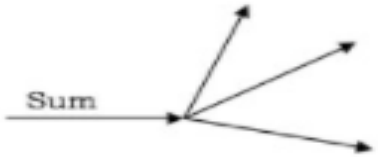
Functional modeling (contd..)

Representation in DFD – A data flow is represented by a directed arc or an arrow, labelled with the name of the data item that it carries.

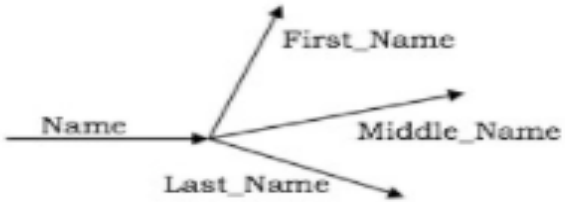
In the above figure, Integer_a and Integer_b represent the input data flows to the process, while L.C.M. and H.C.F. are the output data flows.

A data flow may be forked in the following cases –

- The output value is sent to several places as shown in the following figure. Here, the output arrows are unlabelled as they denote the same value.
- The data flow contains an aggregate value, and each of the components is sent to different places as shown in the following figure. Here, each of the forked components is labelled.



(a)



(b)

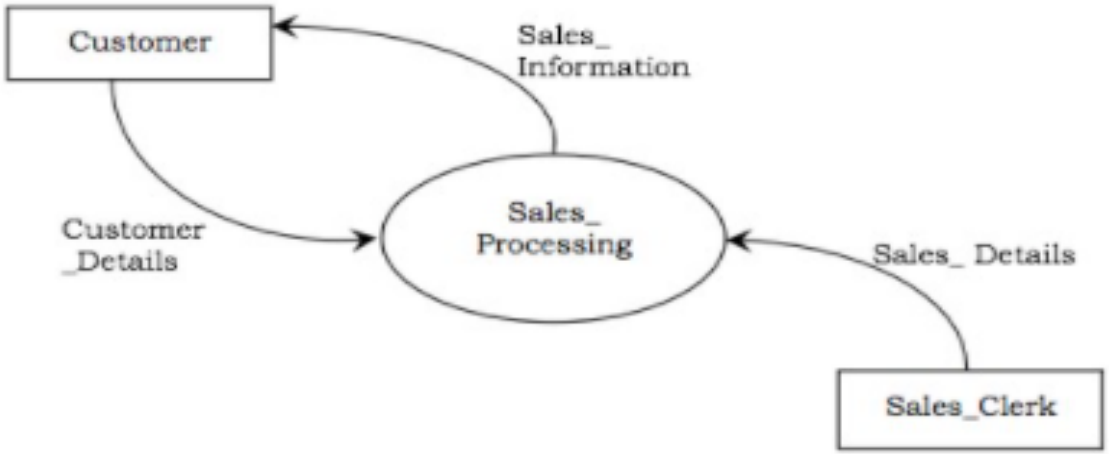
Actors

Actors are the active objects that interact with the system by either producing data and inputting them to the system, or consuming data produced by the system. In other words, actors serve as the sources and the sinks of data.

Representation in DFD – An actor is represented by a rectangle. Actors are connected to the inputs and outputs and lie on the boundary of the DFD.

Functional modeling (contd..)

Example − The following figure shows the actors, namely, Customer and Sales_Clerk in a counter sales system.



Data Stores

Data stores are the passive objects that act as a repository of data. Unlike actors, they cannot perform any operations. They are used to store data and retrieve the stored data. They represent a data structure, a disk file, or a table in a database.

Representation in DFD − A data store is represented by two parallel lines containing the name of the data store. Each data store is connected to at least one process. Input arrows contain information to modify the contents of the data store, while output arrows contain information retrieved from the data store. When a part of the information is to be retrieved, the output arrow is labelled. An unlabelled arrow denotes full data retrieval. A two-way arrow implies both retrieval and update.

Functional modeling (contd..)

Advantages and Disadvantages of DFD

Advantages	Disadvantages
DFDs depict the boundaries of a system and hence are helpful in portraying the relationship between the external objects and the processes within the system.	DFDs take a long time to create, which may not be feasible for practical purposes.
They help the users to have a knowledge about the system.	DFDs do not provide any information about the time-dependent behavior, i.e., they do not specify when the transformations are done.
The graphical representation serves as a blueprint for the programmers to develop a system.	They do not throw any light on the frequency of computations or the reasons for computations.
DFDs provide detailed information about the system processes.	The preparation of DFDs is a complex process that needs considerable expertise. Also, it is difficult for a non-technical person to understand.
They are used as a part of the system documentation.	The method of preparation is subjective and leaves ample scope to be imprecise.

Relationship between Object, dynamic & functional model

The Object Model, the Dynamic Model, and the Functional Model are complementary to each other for a complete Object-Oriented Analysis.

- Object modelling develops the static structure of the software system in terms of objects. Thus it shows the "doers" of a system.
- Dynamic Modelling develops the temporal behavior of the objects in response to external events. It shows the sequences of operations performed on the objects.
- Functional model gives an overview of what the system should do.

Functional Model and Object Model

The four main parts of a Functional Model in terms of object model are –

- **Process** – Processes imply the methods of the objects that need to be implemented.
- **Actors** – Actors are the objects in the object model.
- **Data Stores** – These are either objects in the object model or attributes of objects.
- **Data Flows** – Data flows to or from actors represent operations on or by objects. Data flows to or from data stores represent queries or updates.

Functional Model and Dynamic Model

The dynamic model states when the operations are performed, while the functional model states how they are performed and which arguments are needed. As actors are active objects, the dynamic model has to specify when it acts. The data stores are passive objects and they only respond to updates and queries; therefore the dynamic model need not specify when they act.

Object Model and Dynamic Model

The dynamic model shows the status of the objects and the operations performed on the occurrences of events and the subsequent changes in states. The state of the object as a result of the changes is shown in the object model.

Object Oriented design

After the analysis phase, the conceptual model is developed further into an object-oriented model using object-oriented design (OOD). In OOD, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain. In a nutshell, a detailed description is constructed specifying how the system is to be built on concrete technologies

The stages for object–oriented design can be identified as –

- Definition of the context of the system
- Designing system architecture
- Identification of the objects in the system
- Construction of design models
- Specification of object interfaces

Object Oriented design (contd..)

System Design

Object-oriented system design involves defining the context of a system followed by designing the architecture of the system.

- **Context** – The context of a system has a static and a dynamic part. The static context of the system is designed using a simple block diagram of the whole system which is expanded into a hierarchy of subsystems. The subsystem model is represented by UML packages. The dynamic context describes how the system interacts with its environment. It is modelled using **use case diagrams**.
- **System Architecture** – The system architecture is designed on the basis of the context of the system in accordance with the principles of architectural design as well as domain knowledge. Typically, a system is partitioned into layers and each layer is decomposed to form the subsystems.

Object-Oriented Decomposition

Decomposition means dividing a large complex system into a hierarchy of smaller components with lesser complexities, on the principles of divide–and–conquer. Each major component of the system is called a subsystem. Object-oriented decomposition identifies individual autonomous objects in a system and the communication among these objects.

The advantages of decomposition are –

- The individual components are of lesser complexity, and so more understandable and manageable.
- It enables division of workforce having specialized skills.
- It allows subsystems to be replaced or modified without affecting other subsystems.

Object Oriented design (contd..)

Identifying Concurrency

Concurrency allows more than one objects to receive events at the same time and more than one activity to be executed simultaneously. Concurrency is identified and represented in the dynamic model.

To enable concurrency, each concurrent element is assigned a separate thread of control. If the concurrency is at object level, then two concurrent objects are assigned two different threads of control. If two operations of a single object are concurrent in nature, then that object is split among different threads.

Concurrency is associated with the problems of data integrity, deadlock, and starvation. So a clear strategy needs to be made whenever concurrency is required. Besides, concurrency requires to be identified at the design stage itself, and cannot be left for implementation stage.

Identifying Patterns

While designing applications, some commonly accepted solutions are adopted for some categories of problems. These are the patterns of design. A pattern can be defined as a documented set of building blocks that can be used in certain types of application development problems.

Some commonly used design patterns are –

- Façade pattern
- Model view separation pattern
- Observer pattern
- Model view controller pattern
- Publish subscribe pattern
- Proxy pattern

Object Oriented design (contd..)

Controlling Events

During system design, the events that may occur in the objects of the system need to be identified and appropriately dealt with.

An event is a specification of a significant occurrence that has a location in time and space.

There are four types of events that can be modelled, namely –

- **Signal Event** – A named object thrown by one object and caught by another object.
- **Call Event** – A synchronous event representing dispatch of an operation.
- **Time Event** – An event representing passage of time.
- **Change Event** – An event representing change in state.

Handling Boundary Conditions

The system design phase needs to address the initialization and the termination of the system as a whole as well as each subsystem. The different aspects that are documented are as follows –

- The start–up of the system, i.e., the transition of the system from non-initialized state to steady state.
- The termination of the system, i.e., the closing of all running threads, cleaning up of resources, and the messages to be sent.
- The initial configuration of the system and the reconfiguration of the system when needed.
- Foreseeing failures or undesired termination of the system.

Boundary conditions are modelled using boundary use cases.

Object Oriented design (contd..)

Object Design

After the hierarchy of subsystems has been developed, the objects in the system are identified and their details are designed. Here, the designer details out the strategy chosen during the system design. The emphasis shifts from application domain concepts toward computer concepts. The objects identified during analysis are etched out for implementation with an aim to minimize execution time, memory consumption, and overall cost.

Object design includes the following phases –

- Object identification
- Object representation, i.e., construction of design models
- Classification of operations
- Algorithm design
- Design of relationships
- Implementation of control for external interactions
- Package classes and associations into modules

Object Identification

The first step of object design is object identification. The objects identified in the object–oriented analysis phases are grouped into classes and refined so that they are suitable for actual implementation.

The functions of this stage are –

- Identifying and refining the classes in each subsystem or package
- Defining the links and associations between the classes
- Designing the hierarchical associations among the classes, i.e., the generalization/specialization and inheritances
- Designing aggregations

Object Oriented design (contd..)

Object Representation

Once the classes are identified, they need to be represented using object modelling techniques. This stage essentially involves constructing UML diagrams.

There are two types of design models that need to be produced –

- **Static Models** – To describe the static structure of a system using class diagrams and object diagrams.
- **Dynamic Models** – To describe the dynamic structure of a system and show the interaction between classes using interaction diagrams and state–chart diagrams.

Classification of Operations

In this step, the operation to be performed on objects are defined by combining the three models developed in the OOA phase, namely, object model, dynamic model, and functional model. An operation specifies what is to be done and not how it should be done.

The following tasks are performed regarding operations –

- The state transition diagram of each object in the system is developed.
- Operations are defined for the events received by the objects.
- Cases in which one event triggers other events in same or different objects are identified.
- The sub–operations within the actions are identified.
- The main actions are expanded to data flow diagrams.

Object Oriented design (contd..)

Algorithm Design

The operations in the objects are defined using algorithms. An algorithm is a stepwise procedure that solves the problem laid down in an operation. Algorithms focus on how it is to be done.

There may be more than one algorithm corresponding to a given operation. Once the alternative algorithms are identified, the optimal algorithm is selected for the given problem domain. The metrics for choosing the optimal algorithm are –

- **Computational Complexity** – Complexity determines the efficiency of an algorithm in terms of computation time and memory requirements.
- **Flexibility** – Flexibility determines whether the chosen algorithm can be implemented suitably, without loss of appropriateness in various environments.
- **Understandability** – This determines whether the chosen algorithm is easy to understand and implement.

Design of Relationships

The strategy to implement the relationships needs to be chalked out during the object design phase. The main relationships that are addressed comprise of associations, aggregations, and inheritances.

The designer should do the following regarding associations –

- Identify whether an association is unidirectional or bidirectional.
- Analyze the path of associations and update them if necessary.
- Implement the associations as a distinct object, in case of many–to-many relationships; or as a link to other object in case of one–to-one or one–to-many relationships.

Regarding inheritances, the designer should do the following –

- Adjust the classes and their associations.
- Identify abstract classes.
- Make provisions so that behaviors are shared when needed.

Object Oriented design (contd..)

Implementation of Control

The object designer may incorporate refinements in the strategy of the state–chart model. In system design, a basic strategy for realizing the dynamic model is made. During object design, this strategy is aptly embellished for appropriate implementation.

The approaches for implementation of the dynamic model are −

- **Represent State as a Location within a Program** − This is the traditional procedure-driven approach whereby the location of control defines the program state. A finite state machine can be implemented as a program. A transition forms an input statement, the main control path forms the sequence of instructions, the branches form the conditions, and the backward paths form the loops or iterations.
- **State Machine Engine** − This approach directly represents a state machine through a state machine engine class. This class executes the state machine through a set of transitions and actions provided by the application.
- **Control as Concurrent Tasks** − In this approach, an object is implemented as a task in the programming language or the operating system. Here, an event is implemented as an inter-task call. It preserves inherent concurrency of real objects.

Object Oriented design (contd..)

Packaging Classes

In any large project, meticulous partitioning of an implementation into modules or packages is important. During object design, classes and objects are grouped into packages to enable multiple groups to work cooperatively on a project.

The different aspects of packaging are –

- **Hiding Internal Information from Outside View** – It allows a class to be viewed as a "black box" and permits class implementation to be changed without requiring any clients of the class to modify code.
- **Coherence of Elements** – An element, such as a class, an operation, or a module, is coherent if it is organized on a consistent plan and all its parts are intrinsically related so that they serve a common goal.
- **Construction of Physical Modules** – The following guidelines help while constructing physical modules –
    - Classes in a module should represent similar things or components in the same composite object.
    - Closely connected classes should be in the same module.
    - Unconnected or weakly connected classes should be placed in separate modules.
    - Modules should have good cohesion, i.e., high cooperation among its components.
    - A module should have low coupling with other modules, i.e., interaction or interdependence between modules should be minimum.

Object Oriented design (contd..)

Design Optimization

The analysis model captures the logical information about the system, while the design model adds details to support efficient information access. Before a design is implemented, it should be optimized so as to make the implementation more efficient. The aim of optimization is to minimize the cost in terms of time, space, and other metrics.

However, design optimization should not be excess, as ease of implementation, maintainability, and extensibility are also important concerns. It is often seen that a perfectly optimized design is more efficient but less readable and reusable. So the designer must strike a balance between the two.

The various things that may be done for design optimization are –

- Add redundant associations
- Omit non-usable associations
- Optimization of algorithms
- Save derived attributes to avoid re-computation of complex expressions

Addition of Redundant Associations

During design optimization, it is checked if deriving new associations can reduce access costs. Though these redundant associations may not add any information, they may increase the efficiency of the overall model.

Omission of Non-Usable Associations

Presence of too many associations may render a system indecipherable and hence reduce the overall efficiency of the system. So, during optimization, all non-usable associations are removed.

Object Oriented design (contd..)

Optimization of Algorithms

In object-oriented systems, optimization of data structure and algorithms are done in a collaborative manner. Once the class design is in place, the operations and the algorithms need to be optimized.

Optimization of algorithms is obtained by −

- Rearrangement of the order of computational tasks
- Reversal of execution order of loops from that laid down in the functional model
- Removal of dead paths within the algorithm

Saving and Storing of Derived Attributes

Derived attributes are those attributes whose values are computed as a function of other attributes (base attributes). Re-computation of the values of derived attributes every time they are needed is a time–consuming procedure. To avoid this, the values can be computed and stored in their computed forms.

However, this may pose update anomalies, i.e., a change in the values of base attributes with no corresponding change in the values of the derived attributes. To avoid this, the following steps are taken −

- With each update of the base attribute value, the derived attribute is also re-computed.
- All the derived attributes are re-computed and updated periodically in a group rather than after each update.

Object Oriented design (contd..)

Design Documentation

Documentation is an essential part of any software development process that records the procedure of making the software. The design decisions need to be documented for any non–trivial software system for transmitting the design to others.

Usage Areas

Though a secondary product, a good documentation is indispensable, particularly in the following areas –

- In designing software that is being developed by a number of developers
- In iterative software development strategies
- In developing subsequent versions of a software project
- For evaluating a software
- For finding conditions and areas of testing
- For maintenance of the software.

Contents

A beneficial documentation should essentially include the following contents –

- **High–level system architecture** – Process diagrams and module diagrams
- **Key abstractions and mechanisms** – Class diagrams and object diagrams.
- **Scenarios that illustrate the behavior of the main aspects** – Behavioural diagrams

Features

The features of a good documentation are –

- Concise and at the same time, unambiguous, consistent, and complete
- Traceable to the system's requirement specifications
- Well-structured
- Diagrammatic instead of descriptive

SA/SD

Structured Analysis and Structured Design (SA/SD) is diagrammatic notation which is design to help people understand the system. The basic goal of SA/SD is to improve quality and reduce the risk of System failure. It establishes concrete management specification and documentation. It focuses on solidity, pliability and maintainability of system.

The approach of SA/SD is based on the **Data Flow Diagram**. It is easy to understand SA/SD but it focuses on well defined system boundary whereas JSD approach is too complex and does not have any graphical representation.

SA/SD is combined known as SAD and it mainly focuses on following 3 points:

1. System
2. Process
3. Technology

SA/SD involves 2 phases:

1. **Analysis Phase:** It uses Data Flow Diagram, Data Dictionary, State Transition diagram and ER diagram.
2. **Design Phase:** It uses Structure Chart and Pseudo Code.

SA/SD (contd..)

1. Analysis Phase:

Analysis Phase involves data flow diagram, data dictionary, state transition diagram and entity relationship diagram.

1. **Data Flow Diagram:**

   In the data flow diagram model describe how the data flows through the system. We can incorporate the Boolean operators and & or to link data flows when more than one data flow may be input or output from a process.

For example, if we have to choose between two paths of a process we can add an operator or and if two data flows are necessary for a process we can add and operator. The input of the process "check-order" needs the credit information and order information whereas the output of the process would be a cash-order or a good-credit-order.

2. **Data Dictionary:**

   The content that are not described in the DFD are described in data dictionary. It defines the data store and relevant meaning. A physical data dictionary for data elements which flow between processes, between entities, and between processes and entities may be included. This would also include descriptions of data elements that flow external to the data stores.

SA/SD (contd..)

3. **State Transition Diagram:**

   State transition diagram is similar to dynamic model. It specifies how much time function will take to execute and data access triggered by events. It also describes all of the states that an object can have, the events under which an object changes state, the conditions that must be fulfilled before the transition will occur and the activities undertaken during the life of an object.

4. **ER Diagram:**

   ER diagram specifies the relationship between data store. It is basically used in database design. It basically describes the relationship between different entities.

2. Design Phase:

Design Phase involves structure chart and pseudo code.

1. **Structure Chart:**

   It is created by the data flow diagram. Structure Chart specifies how DFS's processes are grouped into task and allocate to CPU.The structured chart does not show the working and internal structure of the processes or modules, and does not show the relationship between data or data-flows. Similar to other SASD tools, it is time and cost independent and there is no

SA/SD (contd..)

The modules of a structured chart are arranged arbitrarily and any process from a DFD can be chosen as the central transform depending on the analysts' own perception. The structured chart is difficult to amend, verify, maintain, and check for completeness and consistency.

2. **Pseudo Code:**

   It is actual implementation of system.It is a informal way of programming which doesn't require any specific programming language or technology.

JSD

Jackson System Development (JSD) is a method of system development that covers the software life cycle either directly or by providing a framework into which more specialized techniques can fit. JSD can start from the stage in a project when there is only a general statement of requirements.
However many projects that have used JSD actually started slightly later in the life cycle, doing the first steps largely from existing documents rather than directly with the users.

Phases of JDS:

JSD has 3 phases:

1. **Modelling Phase:**
   In the modelling phase of JSD the designer creates a collection of entity structure diagrams and identifies the entities in the system, the actions they perform, the attributes of the actions and time ordering of the actions in the life of the entities.
2. **Specification Phase:**
   This phase focuses on actually what is to be done? Previous phase provides the basic for this phase. An sufficient model of a time-ordered world must itself be time-ordered. Major goal is to map progress in the real world on progress in the system that models it.

3. **Implementation Phase:**
   In the implementation phase JSD determines how to obtain the required functionality. Implementation way of the system is based on transformation of specification into efficient set of processes. The processes involved in it should be designed in such a manner that it would be possible to run them on available software and hardware.

JSD (contd..)

JSD Steps:

Initially there were six steps when it was originally presented by Jackson, they were as below:

1. Entity/action step
2. Initial model step
3. Interactive function step
4. Information function step
5. System timing step
6. System implementation step

Later some steps were combined to create method with only three steps:

1. Modelling Step
2. Network Step
3. Implementation Step

JSD (contd..)

Merits of JSD:

- It is designed to solve real time problem.
- JSD modelling focuses on time.
- It considers simultaneous processing and timing.
- It is a better approach for micro code application.

Demerits of JSD:

- It is a poor methodology for high level analysis and data base design.
- JSD is a complex methodology due to pseudo code representation.
- It is less graphically oriented as compared to SA/SD or OMT.
- It is a bit complex and difficult to understand.

How do you map the object oriented concept using non object oriented languages?

Object orientation, as a concept, means to create logical representations of physical or conceptual items: a vehicle, a person, a manager, etc. Each object type (commonly called a class) has a bundle of attributes (data items, properties) and verbs (methods, functions). These bundles, together, form the class.

It is thus possible to create constructs very close to modern classes as those known in C++ using structures in C.

In C, a structure can

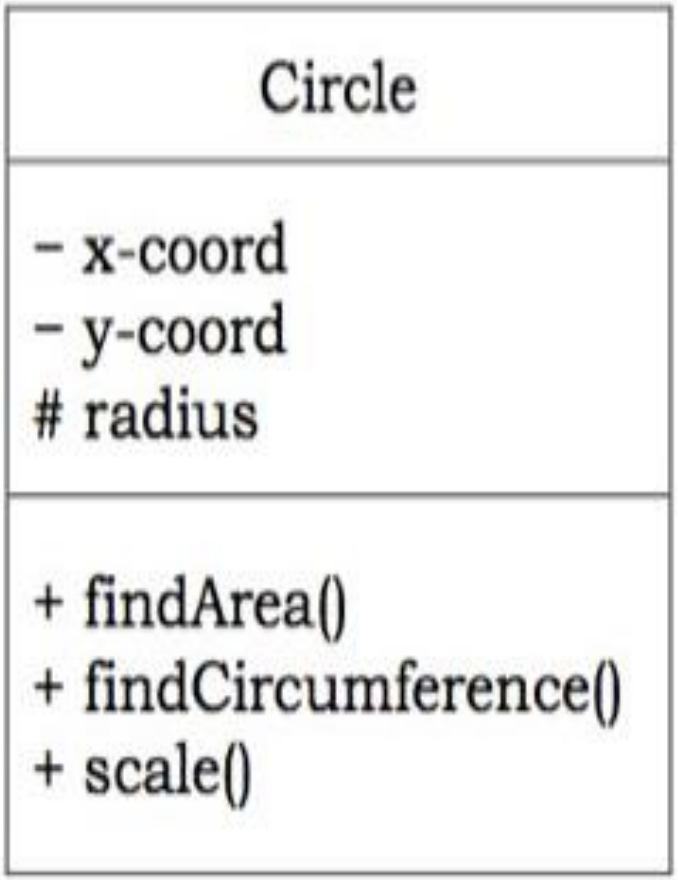
- contain variables (data members)
- pointers to functions (member functions)
- pointers to pointers of functions (virtual functions)

Among the many standard object-oriented techniques which can not be modeled in C are

- privacy through visibility keywords (public, private, protected)
- inheritance (but a "derived" class can embed the "superclass" for a similar effect)
- polymorphism
- contracts (interfaces)
- operator overloading

Translating classes into data structure

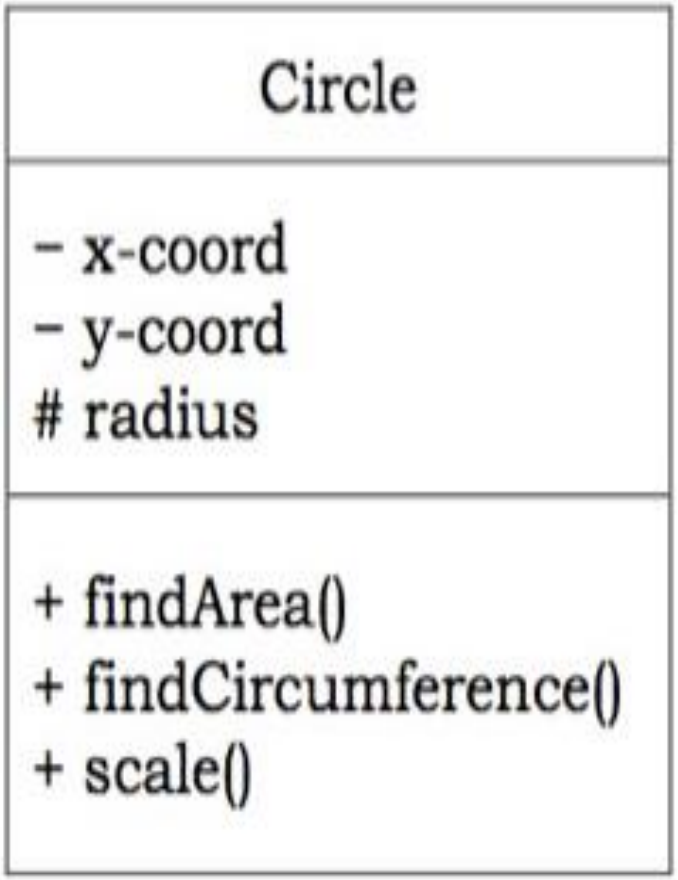
The following figure shows the representation of the class Circle using C++.




```
class Circle
{
   private:
     double x_coord;
     double y_coord;
   protected:
     double radius;
   public:
     double findArea();
     double findCircumference();
     void scale();
};
```

Translating classes into data structure

The following figure shows the representation of the class Circle using C++.




```
class Circle
{
   private:
     double x_coord;
     double y_coord;
   protected:
     double radius;
   public:
     double findArea();
     double findCircumference();
     void scale();
};
```

Implementing Association

1. Unidirectional Association
    i. Optional Association
    ii. One to one association
    iii. One to many association
2. Bi – directional Associations
    i. Optional or one to one
    ii. One to many association
3. Implementing associations as classes
4. Implementing constraints

1. Unidirectional Association

i. Optional Association

- **Optional Associations** – Here, a link may or may not exist between the participating objects. For example, in the association between Customer and Current Account in the figure below, a customer may or may not have a current account.

For implementation, an object of Current Account is included as an attribute in Customer that may be NULL. Implementation using C++ –

```
class Customer {
   private:
   // attributes
   Current_Account c; //an object of Current_Account as attribute

   public:

   Customer() {
      c = NULL;
   } // assign c as NULL

   Current_Account getCurrAc() {
      return c;
   }

   void setCurrAc( Current_Account myacc) {
      c = myacc;
   }

   void removeAcc() {
      c = NULL;
   }
};
```

1. Unidirectional Association

i. Optional Association

- **Optional Associations** – Here, a link may or may not exist between the participating objects. For example, in the association between Customer and Current Account in the figure below, a customer may or may not have a current account.

For implementation, an object of Current Account is included as an attribute in Customer that may be NULL. Implementation using C++ –

```
class Customer {
   private:
   // attributes
   Current_Account c; //an object of Current_Account as attribute

   public:

   Customer() {
      c = NULL;
   } // assign c as NULL

   Current_Account getCurrAc() {
      return c;
   }

   void setCurrAc( Current_Account myacc) {
      c = myacc;
   }

   void removeAcc() {
      c = NULL;
   }
};
```

1. Unidirectional Association

ii. One to one associations

- **One–to–one Associations** – Here, one instance of a class is related to exactly one instance of the associated class. For example, Department and Manager have one–to–one association as shown in the figure below.

This is implemented by including in Department, an object of Manager that should not be NULL. Implementation using C++ –

```
class Department {
   private:
   // attributes
   Manager mgr; //an object of Manager as attribute

   public:
   Department (/*parameters*/, Manager m) { //m is not NULL
      // assign parameters to variables
      mgr = m;
   }

   Manager getMgr() {
      return mgr;
   }
};
```

1. Unidirectional Association

iii. One to many associations

- **One–to–many Associations** – Here, one instance of a class is related to more than one instances of the associated class. For example, consider the association between Employee and Dependent in the following figure.

This is implemented by including a list of Dependents in class Employee. Implementation using C++ STL list container –

```
class Employee {
   private:
   char * deptName;
   list <Dependent> dep; //a list of Dependents as attribute

   public:
   void addDependent ( Dependent d) {
      dep.push_back(d);
   } // adds an employee to the department

   void removeDeoendent( Dependent d) {
      int index = find ( d, dep );
      // find() function returns the index of d in list dep
      dep.erase(index);
   }
};
```

1. Unidirectional Association

iii. One to many associations

- **One–to–many Associations** – Here, one instance of a class is related to more than one instances of the associated class. For example, consider the association between Employee and Dependent in the following figure.

This is implemented by including a list of Dependents in class Employee. Implementation using C++ STL list container –

```
class Employee {
   private:
   char * deptName;
   list <Dependent> dep; //a list of Dependents as attribute

   public:
   void addDependent ( Dependent d) {
      dep.push_back(d);
   } // adds an employee to the department

   void removeDeoendent( Dependent d) {
      int index = find ( d, dep );
      // find() function returns the index of d in list dep
      dep.erase(index);
   }
};
```

1. Bidirectional Association

i. Optional or one to one association

- **Optional or one–to–one Associations** – Consider the relationship between Project and Project Manager having one–to–one bidirectional association as shown in the figure below.

Implementation using C++ –

```
Class Project {
   private:
   // attributes
   Project_Manager pmgr;
   public:
   void setManager ( Project_Manager pm);
   Project_Manager changeManager();
};

class Project_Manager {
   private:
   // attributes
   Project pj;

   public:
   void setProject(Project p);
   Project removeProject();
};
```

2. Bidirectional Association

i. Optional or one to one association

- **Optional or one–to–one Associations** – Consider the relationship between Project and Project Manager having one–to–one bidirectional association as shown in the figure below.

Implementation using C++ –

```
Class Project {
   private:
   // attributes
   Project_Manager pmgr;
   public:
   void setManager ( Project_Manager pm);
   Project_Manager changeManager();
};

class Project_Manager {
   private:
   // attributes
   Project pj;

   public:
   void setProject(Project p);
   Project removeProject();
};
```

2. Bidirectional Association

ii. One to many association

- **One–to–many Associations** – Consider the relationship between Department and Employee having one–to–many association as shown in the figure below.

Implementation using C++ STL list container

```
class Department {
   private:
   char * deptName;
   list <Employee> emp; //a list of Employees as attribute

   public:
   void addEmployee ( Employee e) {
      emp.push_back(e);
   } // adds an employee to the department

   void removeEmployee( Employee e) {
      int index = find ( e, emp );
      // find function returns the index of e in list emp
      emp.erase(index);
   }
};

class Employee {
   private:
   //attributes
   Department d;

   public:
   void addDept();
   void removeDept();
};
```

2. Bidirectional Association

ii. One to many association

- **One–to–many Associations** – Consider the relationship between Department and Employee having one–to–many association as shown in the figure below.

Implementation using C++ STL list container

```
class Department {
   private:
   char * deptName;
   list <Employee> emp; //a list of Employees as attribute

   public:
   void addEmployee ( Employee e) {
      emp.push_back(e);
   } // adds an employee to the department

   void removeEmployee( Employee e) {
      int index = find ( e, emp );
      // find function returns the index of e in list emp
      emp.erase(index);
   }
};

class Employee {
   private:
   //attributes
   Department d;

   public:
   void addDept();
   void removeDept();
};
```

Implementing Association as classes:

Example

Consider an Employee class where age is an attribute that may have values in the range of 18 to 60. The following C++ code incorporates it –

```
class Employee {
   private: char * name;
   int age;
   // other attributes

   public:
   Employee() {                  // default constructor
      strcpy(name, "");
      age = 18;                 // default value
   }

   class AgeError {};           // Exception class
   void changeAge( int a) {     // method that changes age

      if ( a < 18 || a > 60 )   // check for invalid condition
      throw AgeError();         // throw exception
      age = a;
   }
};
```