Ambo University, Woliso Campus
School of Technology and Informatics
Department of Information Systems

Object Oriented System Analysis and Design (OOSAD)

COMPILED BY: HABTAMU KENO

DEPARTMENT OF INFORMATION SYSTEMS

3, June, 2O

WOLISO, ETHIOPIA

Contents

Chapter 1: Understanding the Basics: Object oriented concepts

1.1 A Brief History

The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.

- The first object–oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARK created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL)-Smalltalk, for programming the Dynabook.
- In the 1980s, Grady Booch published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object– oriented design method.

In the 1990s, Coad incorporated behavioral ideas to object-oriented methods.

The other significant innovations were Object Modelling Techniques (OMT) by James Rumbaugh and Object-Oriented Software Engineering (OOSE) by Ivar Jacobson.

1.2. 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.

Grady Booch has defined OOA as, *"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain"*.

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

The common models used in OOA are use cases and object models.

1.3. 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*".

1.4. **Introduction**

Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterised by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, one such model is Unified
Modeling Language (UML).

1.4.1. THE OBJECT MODEL

Object oriented development offers a different model from the traditional software development approach, which is based on functions and procedures.

- An Object-Oriented environment, software is a collection of discrete objects that encapsulate their data and the functionality to model real world "Objects".
- Object are defined, it will perform their desired functions and seal them off in our mind like black boxes.
- The object- Oriented life cycle encourages a view of the world as a system of cooperative and collaborating agents.
- An objective orientation producers system that are easier evolve, move flexible more robust, and more reusable than a top-down structure approach.
- An object orientation allows working at a higher level of abstraction.
- It provides a seamless transition among different phases of software development.
- It encourages good development practices.
- It promotes reusability.

The unified Approach (UA) is the methodology for software development proposed and used and the following concepts consist of Unified Approach

1.4.2.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.

Grady Booch has defined object–oriented programming as *"a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships"*.

OBJECT MODEL

The object model visualizes the elements in a software application in terms of objects. In this chapter, we will look into the basic concepts and terminologies of object–oriented systems.

Objects and Classes

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object–oriented paradigm.

Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has:

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modeled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are:

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Example

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows:

- x–coord, to denote x–coordinate of the center
- y–coord, to denote y–coordinate of the center
- a, to denote the radius of the circle

Some of its operations can be defined as follows:

- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a :4 to depict its state. Now, if

the operation scale() is performed on my_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my_circle, i.e., the object has exhibited certain behavior.

Encapsulation and Data Hiding

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

Data Hiding

Typically, a class is designed such that its data (attributes) can be accessed only by its class methods and insulated from direct outside access. This process of insulating an object's data is called data hiding or information hiding.

Example

In the class Circle, data hiding can be incorporated by making attributes invisible from outside the class and adding two more methods to the class for accessing class data, namely:

- setValues(), method to assign values to x-coord, y-coord, and a
- getValues(), method to retrieve values of x-coord, y-coord, and a

Here the private data of the object my_circle cannot be accessed directly by any method that is not encapsulated within the class Circle. It should instead be accessed through the methods setValues() and getValues().

Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

The features of message passing are:

- Message passing between two objects is generally unidirectional.
- Message passing enables all interactions between objects.
- Message passing essentially involves invoking class methods.
- Objects in different processes can be involved in message passing.

Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an "is – a" relationship.

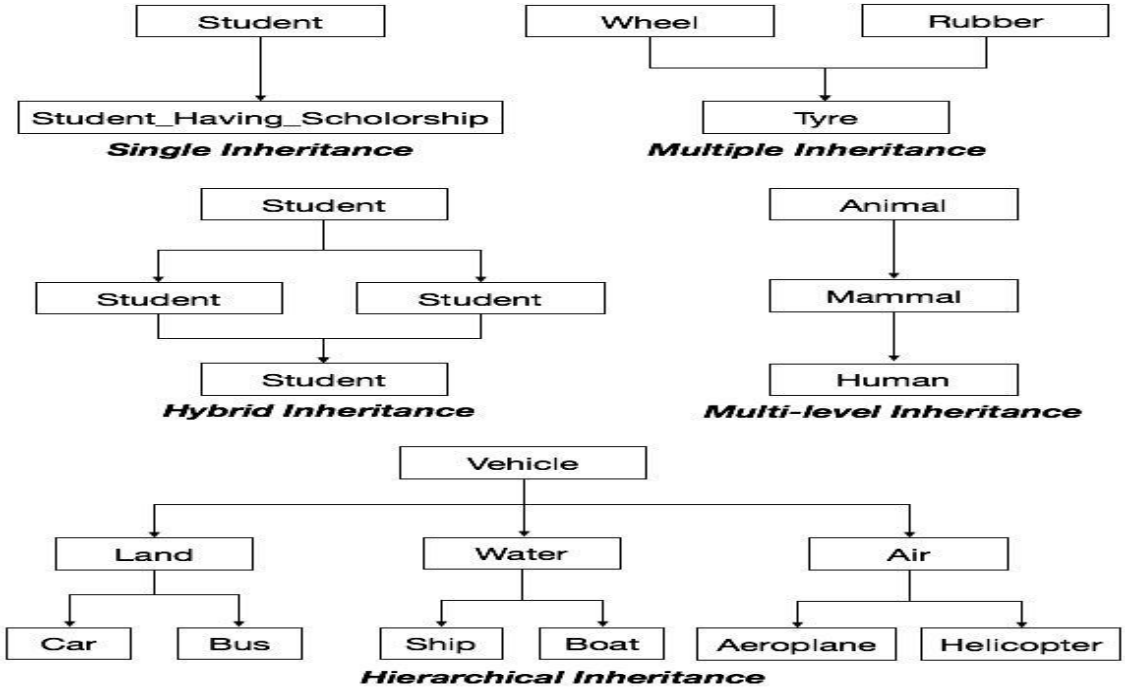
Example

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow "is – a" mammal.

Types of Inheritance

- **Single Inheritance** : A subclass derives from a single super-class.
- **Multiple Inheritance** : A subclass derives from more than one super-classes.
- **Multilevel Inheritance** : A subclass derives from a super-class which in turn is derived from another class and so on.
- **Hierarchical Inheritance** : A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
- **Hybrid Inheritance** : A combination of multiple and multilevel inheritance so as to form a lattice structure.

The following figure depicts the examples of different types of inheritance.



Polymorphism

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

Example

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

Generalization and Specialization

Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

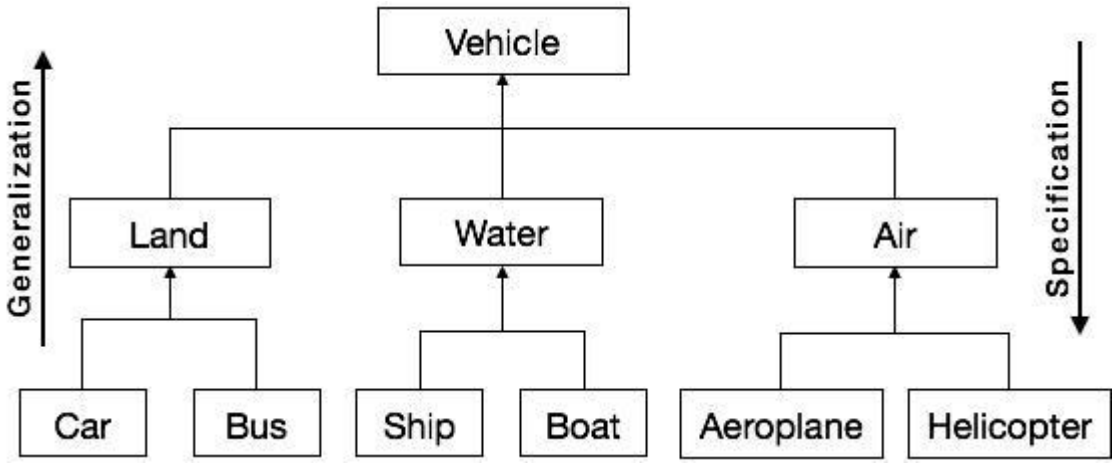
Generalization

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an "is – a – kind – of" relationship. For example, "car is a kind of land vehicle", or "ship is a kind of water vehicle".

Specialization

Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes. It can be said that the subclasses are the specialized versions of the super-class.

The following figure shows an example of generalization and specialization.



Links and Association

Link

A link represents a connection through which an object collaborates with other objects. Rumbaugh has defined it as "a physical or conceptual connection between objects". Through a link, one object may invoke the methods or navigate through another object. A link depicts the relationship between two or more objects.

Association

Association is a group of links having common structure and common behavior. Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association.

Degree of an Association

Degree of an association denotes the number of classes involved in a connection. Degree may be unary, binary, or ternary.

- A **unary relationship** connects objects of the same class.
- A **binary relationship** connects objects of two classes.
- A **ternary relationship** connects objects of three or more classes.

Cardinality Ratios of Associations

Cardinality of a binary association denotes the number of instances participating in an association. There are three types of cardinality ratios, namely:

- **One–to–One** : A single object of class A is associated with a single object of class B.
- **One–to–Many** : A single object of class A is associated with many objects of class B.
- **Many–to–Many** : An object of class A may be associated with many objects of class B and conversely an object of class B may be associated with many objects of class A.

Aggregation or Composition

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a

"part–of" or "has–a" relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

Example

In the relationship, "a car has–a motor", car is the whole object or the aggregate, and the motor is a "part–of" the car. Aggregation may denote:

- **Physical containment** : Example, a computer is composed of monitor, CPU, mouse, keyboard, and so on.
- **Conceptual containment** : Example, shareholder has–a share.

1.5. Benefits of Object Model

Now that we have gone through the core concepts pertaining to object orientation, it would be worthwhile to note the advantages that this model has to offer.

The benefits of using the object model are:

- It helps in faster development of software.
- It is easy to maintain. Suppose a module develops an error, then a programmer can fix that particular module, while the other parts of the software are still up and running.
- It supports relatively hassle-free upgrades.
- It enables reuse of objects, designs, and functions.
- It reduces development risks, particularly in integration of complex systems.

3. OBJECT-ORIENTED SYSTEM

We know that the Object-Oriented Modeling (OOM) technique visualizes things in an application by using models organized around objects. Any software development approach goes through the following stages:

- ***Analysis,***
- ***Design, and***
- ***Implementation.***

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools.

Phases in Object-Oriented Software Development

The major phases of software development using object–oriented methodology are object-oriented analysis, object-oriented design, and 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.

The associations between the identified classes are established and the hierarchies of classes are identified. Besides, the developer designs the internal details of the classes and their associations, i.e., the data structure for each attribute and the algorithms for the operations.

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.

1.6. Object-Oriented Principles

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

Abstraction

Abstraction means to focus on the essential features of an element or object in OOP, ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used.

Grady Booch has defined abstraction as follows:

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

Example : When a class Student is designed, the attributes enrolment_number, name, course, and address are included while characteristics like pulse_rate and size_of_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. The class has methods that provide user interfaces by which the services provided by the class may be used.

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.

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.

1.7.Object-Oriented Analysis

In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified.
The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modeling, dynamic modeling, and functional modeling.

Object Modeling
Object modeling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.
The process of object modeling 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 Modeling
After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.
Dynamic modelling can be defined as "a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world".
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
Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

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

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

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	The initial cost of constructing the system is high, since the whole system

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.	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.

States and State Transitions

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.

Initial and Final States

The default starting state of an object is called its initial state. The final state indicates the completion of execution of the state machine. The initial and the final states are pseudo-states, and may not have the parts of a regular state except name. In state transition diagrams, the initial state is represented by a filled black circle. The final state is represented by a filled black circle encircled within another unfilled black circle.

Transition

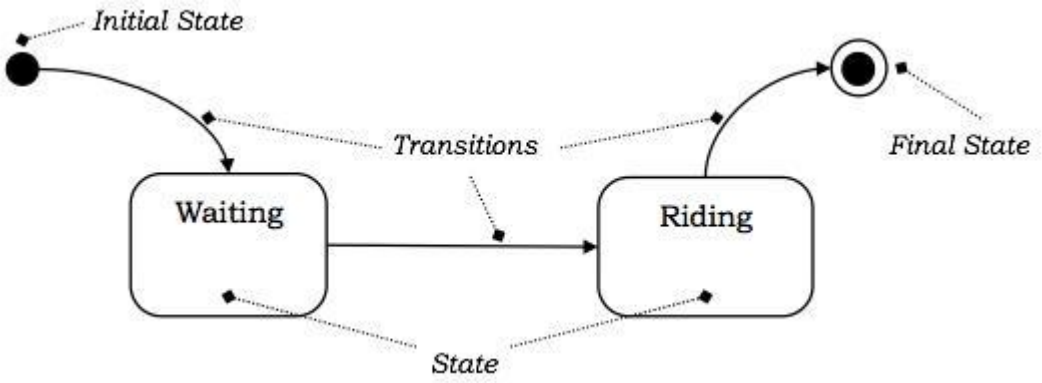
A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state. In this case, a state−transition is said to have occurred. The transition gives the relationship between the first state and the new state. A transition is graphically represented by a solid directed arc from the source state to the destination 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.



Events

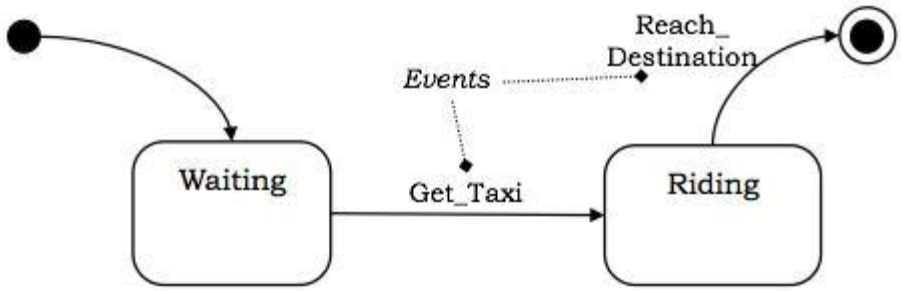
Events are some occurrences that can trigger state transition of an object or a group of objects. Events have a location in time and space but do not have a time period associated with it. Events are generally associated with some actions.

Examples of events are mouse click, key press, an interrupt, stack overflow, etc.

Events that trigger transitions are written alongside the arc of transition in state diagrams.

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.



External and Internal Events

External events are those events that pass from a user of the system to the objects within the system. For example, mouse click or key−press by the user are external events.

Internal events are those that pass from one object to another object within a system. For example, stack overflow, a divide error, etc.

Deferred Events

Deferred events are those which are not immediately handled by the object in the current state but are lined up in a queue so that they can be handled by the object in some other state at a later time.

Event Classes

Event class indicates a group of events with common structure and behavior. As with classes of objects, event classes may also be organized in a hierarchical structure. Event classes may have attributes associated with them, time being an implicit attribute. For example, we can consider the events of departure of a flight of an airline, which we can group into the following class:

Flight_Departs (Flight_No, From_City, To_City, Route)

Activity

Activity is an operation upon the states of an object that requires some time period. They are the ongoing executions within a system that can be interrupted. Activities are shown in activity diagrams that portray the flow from one activity to another.

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. An action may operate upon an object on which an event has been triggered or on other objects that are visible to this object. A set of actions comprise an activity.

Entry and Exit Actions

Entry action is the action that is executed on entering a state, irrespective of the transition that led into it.

Likewise, the action that is executed while leaving a state, irrespective of the transition that led out of it, is called an exit action.

Scenario

Scenario is a description of a specified sequence of actions. It depicts the behavior of objects undergoing a specific action series. The primary scenarios depict the essential sequences and the secondary scenarios depict the alternative sequences.

Diagrams for Dynamic Modeling

There are **two** primary diagrams that are used for dynamic modeling:

Interaction Diagrams

Interaction diagrams describe the dynamic behavior among different objects. It comprises of a set of objects, their relationships, and the message that the objects send and receive. Thus, an interaction models the behavior of a group of interrelated objects. The two types of interaction diagrams are:

- **Sequence Diagram** : It represents the temporal ordering of messages in a tabular manner.
- **Collaboration Diagram** : It represents the structural organization of objects that send and receive messages through vertices and arcs.

State Transition Diagram

State transition diagrams or state machines describe the dynamic behavior of a single object. It illustrates the sequences of states that an object goes through in its lifetime, the transitions of the states, the events and conditions causing the transition and the responses due to the events.

Concurrency of Events

In a system, two types of concurrency may exist. They are discussed below.

System Concurrency

Here, concurrency is modelled in the system level. The overall system is modelled as the aggregation of state machines, where each state machine executes concurrently with others.

Concurrency within an Object

Here, an object can issue concurrent events. An object may have states that are composed of sub-states, and concurrent events may occur in each of the sub-states.

Concepts related to concurrency within an object are as follows:

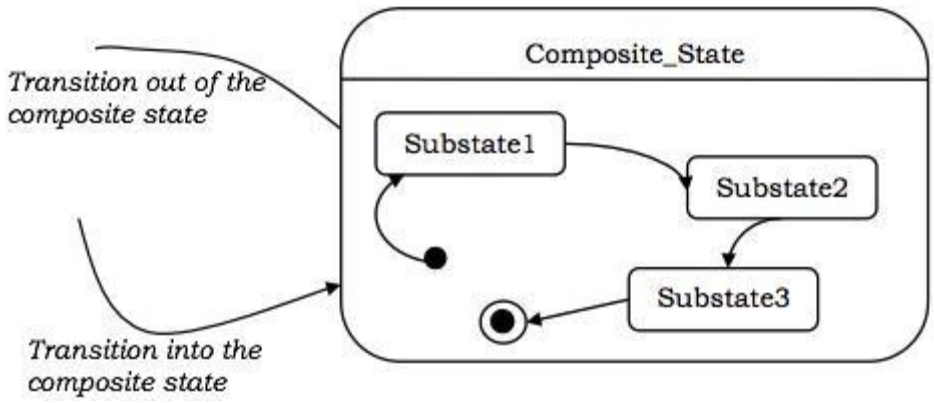
Simple and Composite States

A simple state has no sub-structure. A state that has simpler states nested inside it is called a composite state. A sub-state is a state that is nested inside another state. It is generally used to reduce the complexity of a state machine. Sub-states can be nested to any number of levels.
Composite states may have either sequential sub-states or concurrent sub-states.

Sequential Sub-states

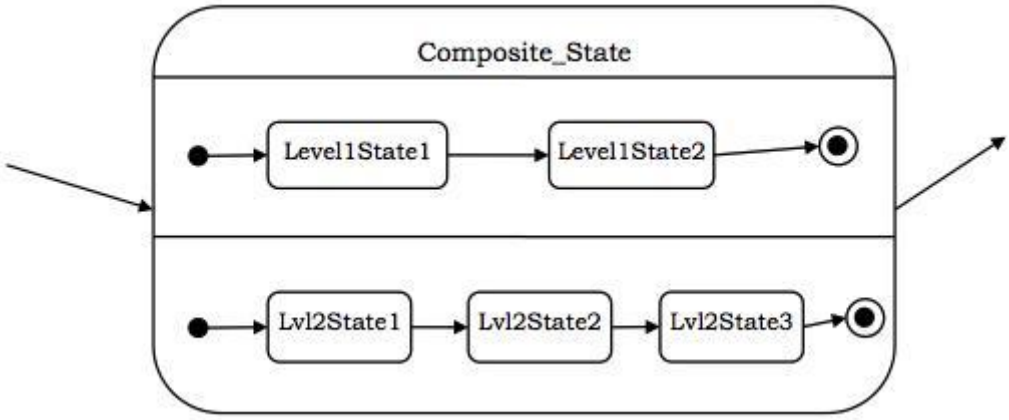
In sequential sub-states, the control of execution passes from one sub-state to another sub-state one after another in a sequential manner. There is at most one initial state and one final state in these state machines.

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



Concurrent Sub-states

In concurrent sub-states, the sub-states execute in parallel, or in other words, each state has concurrently executing state machines within it. Each of the state machines has its own initial and final states. If one concurrent sub-state reaches its final state before the other, control waits at its final state. When all the nested state machines reach their final states, the sub-states join back to a single flow.
The following figure shows the concept of concurrent sub-states.



Chapter Two: Object Orientation the new software paradigm

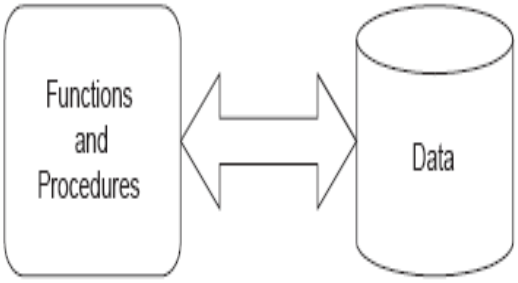
2. Structured vs. Object Orientation paradigm

Structured paradigm

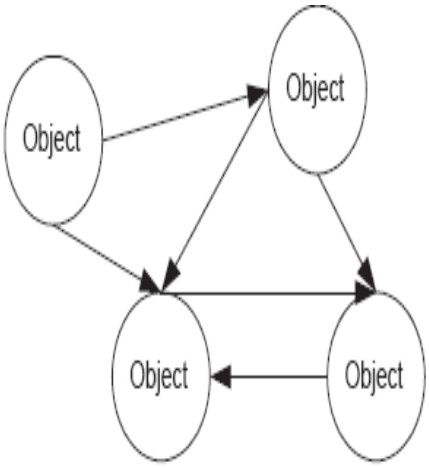
- The structured *paradigm is* a development strategy based on the concept that a system should be separated into two parts:
    - Data and functionality (modeled using a process model). Using the structured approach, you develop applications in which data is separated from behavior in both the design model and in the system implementation (that is, the program).
- Example: Consider the design of an information system for a university. Taking the structured approach, you would define the layout of a data initially as a separate system and the design of a program to access that data as another. The programs have the ability to change the data states.

Object oriented Paradigm

- The main concept behind the object-oriented paradigm is that instead of defining systems as two separate parts (data and functionality), system defined as a collection of interacting objects.
    - *Describes and build system that consists object*
- An object-oriented system comprises a number of software objects that interact to achieve the system objective.



A Structured Application



An Object Application

2.1. The Potential Benefits of the Object Oriented paradigm

Increased reusability:

- The OO paradigm provides opportunities for reuse through the concepts of inheritance, polymorphism, encapsulation, modularity, coupling and cohesion.
- It provides more opportunities for reuse than the structured paradigm

Increased extensibility

Because classes have both data and functionality, when you add new features to the system you need to make changes in one place, the class;

Improved Quality

- Quality systems are on time, on budget and meet or exceed the expectations of their users.
- Improved quality comes from increased participation of users in systems development.
- OO systems development techniques provide greater opportunity for users to participate in the development process.

Financial benefits

- Reusability, extensibility, and improved quality are all technical benefits.
- Object orientation enables you to build systems better, faster and cheaper (BFC)
- The benefits OO are realized through out the entire development life cycle, not just programming

Increased Chance of Project success

- A project is successful if it is on time, on budget and meets the needs of the its users.
- Users are expert at business and they are the only ones who can tell you what they need.
- You need to know the right question to ask, know the business very well.
- You need models that communicate the required information and that users understand.
- You need to work closely with users
- Time invested in defining requirements and modeling pays off in the long run.
- You can use a wide variety of artifacts including code, model and components.

Reduce maintenance Burdon

- Software organizations currently spend significant resources (80%) maintaining and operating software, and because of the long waiting list of work to be done, it takes significant time to get new projects started. These two problems are respectively called
  - the maintenance Burdon and
  - The application backlog
- These are problems that object orientation can help you to overcome

2.2. The Potential Drawbacks of OO

Nothing is perfect including OO. While many exiting benefits exist to OO, they come at a price:

1. OO requires greater concentration on requirements analysis and design
   - You cannot build a system that meets users needs unless you know what those needs are(you need to do requirements)
   - You cannot built a system unless you know how it all fit together (you need to do analysis and design)
   - But this fact is often ignored by many developers
2. Developers must closely work with users
   - Users are the experts but they have their own jobs to do (busy)
3. OO requires a complete change in the mindset on the part of individuals
   - they should understand the benefits of OO
4. OO requires the development culture of the IS dept to change
   - The change in the mind set of individual developers actually reflect an over all change in the development culture
   - Do more analysis and design but (less programming) and working with users
5. OO is just more than programming
6. Many OO benefits are long term
   - OO truly pays off when you extend and enhance your system
7. OO demands up front investments in training education and tools
   - Organizations must train and educate their development staff.
   - Buy books, development tools and magazines
8. OO techniques do not guarantee you will build the right system

- While OO increases the probability of project success ,it still depends on the ability of individuals involved.
- developers, users, mangers must be working together to have a good working atmosphere .

9. OO necessitates increased testing
   - OO is typically iterative in nature, and probably developing complex system using the objects, the end result is you need to spend more time in testing.
10. OO is only part of the solution
    - You still need CASE tools
    - Need to perform quality assurance (QA)
    - You still need usable interface so the users can work with the systems effectively

2.3. Object Standards

- OO orientation today becomes the significant part of the software development .
- Objects are the primary enabling technology for components.
- It also stays in the future because of the standard set by the OMG.
- OO orientation today becomes the significant part of the software development .
- Objects are the primary enabling technology for components.
- It also stays in the future because of the standard set by the OMG.
  - **CORBA**(Common object request broker architecture) – the standard architecture for supporting distributed objects.
  - **UML** (Unified modeling language)-the standard modeling language for the object oriented software.
  - **ANSI** (Americans National Standards Institute)-Defined standards for C++. Http://www.ansi.org
  - **Sun Microsystems** ,Http://www.sum.com actively maintains, enhances and supports a de facto standard definition for java and related standards such as Enterprise Java Beans (EJB).
  - The Object Database Management group (**ODMG**)- Http://www.odmg.org actively maintains, enhances and supports a standard definition for object oriented databases and object query language (OQL).
  - **ANSI** (Americans National Standards Institute)-Defined standards for C++. Http://www.ansi.org
  - **Sun Microsystems** ,Http://www.sum.com actively maintains, enhances and supports a de facto standard definition for java and related standards such as Enterprise Java Beans (EJB).
  - **The Object Database Management group (ODMG**)- Http://www.odmg.org actively maintains, enhances and supports a standard definition for object oriented databases and object query language (OQL).

Chapter 3: Gathering user requirements

3. An Overview of Requirements Elicitation

Requirements elicitation focuses on describing the purpose of the system. The client, the developers, and the users identify a problem area and define a system that addresses the problem. Such a definition is called a **requirements specification** and serves as a contract between the client and the developers. The requirements specification is structured and formalized during analysis (Chapter 5, *Analysis*) to produce an **analysis model**. Both requirements specification and analysis model represent the same information. They differ only in the language and notation they use; the requirements specification is written in natural language, whereas the analysis model is usually expressed in a formal or semiformal notation. The requirements specification supports the communication with the client and users. The analysis model supports the communication among developers. They are both models of the system in the sense that they attempt to represent accurately the external aspects of the system. Given that both models represent the same aspects of the system, requirements elicitation and analysis occur concurrently and iteratively

Requirements elicitation and analysis focus only on the user's view of the system. For example, the system functionality, the interaction between the user and the system, the errors that the system can detect and handle, and the environmental conditions in which the system functions are part of the requirements. The system structure, the implementation technology selected to build the system, the system design, the development methodology, and other aspects not directly visible to the user are not part of the requirements.

3.1. Requirements elicitation includes the following activities:

- ***Identifying actors***. During this activity, developers identify the different types of users the future system will support.
- ***Identifying scenarios*.** During this activity, developers observe users and develop a set of detailed scenarios for typical functionality provided by the future system. Scenarios are concrete examples of the future system in use. Developers use these scenarios to communicate with the user and deepen their understanding of the application domain.
- ***Identifying use cases*.** Once developers and users agree on a set of scenarios, developers derive from the scenarios a set of use cases that completely represent the future system. Whereas scenarios are concrete examples illustrating a single case, use cases are abstractions describing all possible cases. When describing use cases, developers determine the scope of the system.

- ***Refining use cases***. During this activity, developers ensure that the requirements specification is complete by detailing each use case and describing the behavior of the system in the presence of errors and exceptional conditions.
- ***Identifying relationships among use cases*.** During this activity, developers identify dependencies among use cases. They also consolidate the use case model by factoring out common functionality. This ensures that the requirements specification is consistent.
- ***Identifying nonfunctional requirements*.** During this activity, developers, users, and clients agree on aspects that are visible to the user, but not directly related to functionality. These include constraints on the performance of the system, its documentation, the resources it consumes, its security, and its quality.

During requirements elicitation, developers access many different sources of information, including client-supplied documents about the application domain, manuals and technical documentation of legacy systems that the future system will replace, and most important, the users and clients themselves. Developers interact the most with users and clients during requirements elicitation. We focus on two methods for eliciting information, making decisions with users and clients, and managing dependencies among requirements and other artifacts:

- **Joint Application Design (JAD)** focuses on building consensus among developers, users, and clients by jointly developing the requirements specification.
- **Traceability** focuses on recording, structuring, linking, grouping, and maintaining dependencies among requirements and between requirements and other work products.

3.2. Requirements Elicitation Concepts

In this section, we describe the main requirements elicitation concepts used in this chapter. In particular, we describe

- Functional Requirements
- Nonfunctional Requirements

3.3. Functional Requirements

Functional requirements describe the interactions between the system and its environment independent of its implementation. The environment includes the user and any other external system with which the system interacts.

3.4. Nonfunctional Requirements

Nonfunctional requirements describe aspects of the system that are not directly related to the functional behavior of the system. Nonfunctional requirements include a broad variety of requirements that apply to many different aspects of the system, from usability to performance.
The FURPS+ model2 used by the Unified Process [Jacobson et al., 1999] provides the following categories of nonfunctional requirements:

- **Usability** is the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. Usability requirements include, for example, conventions adopted by the user interface, the scope of online help, and the level of user documentation. Often, clients address usability issues by requiring the developer to follow user interface guidelines on color schemes, logos, and fonts.
- **Reliability** is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Reliability requirements include, for example, an acceptable mean time to failure and the ability to detect specified faults or to withstand specified security attacks. More recently, this category is often replaced by **dependability**, which is the property of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability includes reliability, **robustness** (the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions), and **safety** (a measure of the absence of catastrophic consequences to the environment).
- **Performance** requirements are concerned with quantifiable attributes of the system, such as **response time** (how quickly the system reacts to a user input), **throughput** (how much work the system can accomplish within a specified amount of time), **availability** (the degree to which a system or component is operational and accessible when required for use), and **accuracy**.
- **Supportability** requirements are concerned with the ease of changes to the system after deployment, including for example, **adaptability** (the ability to change the system to deal with additional application domain concepts), **maintainability** (the ability to change the system to deal with new technology or to fix defects), and internationalization (the ability to change the system to deal with additional international conventions, such as languages, units, and number formats). The ISO 9126 standard on software quality [ISO Std. 9126], similar to the FURPS+ model, replaces this category with two categories: **maintainability** and **portability** (the ease with which a system or component can be transferred from one hardware or software environment to another). 2. FURPS+ is an acronym using the first letter of the

requirements categories: Functionality, Usability, Reliability, Performance, and Supportability. The + indicates the additional subcategories. The FURPS model was originally proposed by [Grady, 1992].

3.5. Fundamental requirements gathering techniques

Individually interview people informed about the operation and issues of the current system and future systems needs

- **Interview groups** of people with diverse needs to fid synergies and contrasts among system requirements
- **Observe** workers at selected times to see how data are handled and what information people need to do their jobs
- **Study business documents/document analysis** to discover reported issues, policies, rules, and directions as well as concrete examples of the use of data and information in the org. What can the analysis of documents tell you about the requirements for a new system? In documents you can find information about the following:
    - Problems with existing systems (e.g., missing information or redundant steps
    - Opportunities to meet new needs if only certain information or information processing were available (e.g., analysis of sales based on customer type
    - Organizational direction that can influence information system requirements (e.g., trying to link customers and suppliers more closely to the organization)
    - Titles and names of key individuals who have an interest in relevant existing systems (e.g., the name of a sales manager who led a study of the buying behavior of key customers)
    - Values of the organization or individuals who can help determine priorities for different capabilities desired by different users (e.g., maintaining market share even if it means lower short-term profits)

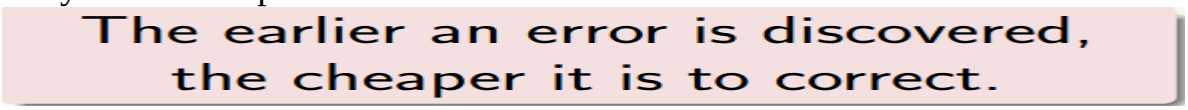
4. Requirements Validation

Validation denotes **checking** whether **inputs**, performed **activities**, and created **outputs** (requirements artifacts) of the requirements engineering core activities fulfill defined **quality criteria**. Validation is performed by involving relevant **stakeholders**, other requirement **sources** (standards, laws, etc.) as well as external **reviewers**, if necessary.

4.1. Quality Criteria

- **Completeness**
  -The requirement must contain all relevant information (template).
- **Consistency**
  - The requirements must be compatible with each other.
- **Adequacy**
  - The requirements must address the actual needs of the system.
- **Un ambiguity**
  -Every requirement must be described in a way that precludes different interpretations.
- **Comprehensibility**
  - The requirements must be understandable by the stakeholders.
- **Importance**
  -Each requirement must indicate how essential it is for the success of the project.
- **Measurability**
  - The requirement must be formulated at a level of precision that enables to evaluate its satisfaction.
- **Necessity**
  - The requirements must all contribute to the satisfaction of the project goals.
- **Viability**
  -All requirements can be implemented with the available technology, human resources and budget.
- **Traceability**
  - The context in which a requirement was created should be easy to retrieve.

In System development :



4.2. The 6 Principles of Validation

1. **Involving the Right Stakeholders**
   Ensure that relevant company-internal as well as relevant external stakeholders participate in validation. Pay attention to the reviewers' independence and appoint external, independent stakeholders, if necessary.
2. **Defect Detection vs. Defect Correction**
   Separate defect detection from the correction of the detected defects.
3. **Leveraging Multiple Independent Views**
   Whenever possible, try to obtain independent views that can b e integrated during requirements validation in order to detect defects more reliably.

4. **Use of Appropriate Documentation Formats**
   Consider changing the documentation format of the requirements into a format that matches the validation goal and the preference s of the stakeholders who actually perform the validation.
5. **Creation of Development Artifacts during Validation**
   If your validation approach generates p o or results, try to supp ort defect detection by creating development artifacts such as architectural artifacts, test artifacts, user manuals, or goals and scenarios during validation.
6. **Repeated Validation**
   Establish guidelines that clearly determine when or under what conditions an already released requirements artifact has to b e validated again.

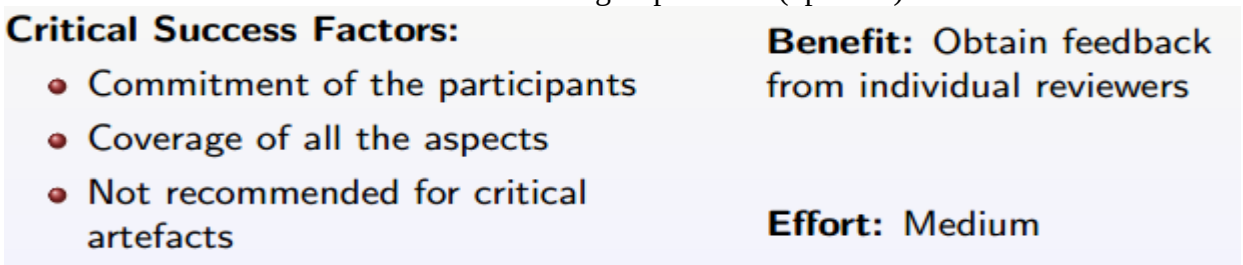
4.3. Validation Techniques

- Inspections
- Desk-Checks
- Walkthroughs
- Prototypes

Inspection: an organized examination process of the requirements

Desk-Checks

- The author of a requirement artifact distributes the artifact to a set of stakeholders.
- The stakeholders check the artifact individually.
- The stakeholders report the identified defects to the author.
- The collected issues are discussed in a group session (optional)



Walkthrough

A walkthrough does not have formally defined procedure and does not require a differentiated role assignment.

-Checking early whether an idea is feasible or not.

-Obtaining the opinion and suggestions of other people.
- Checking the approval of others and reaching agreement.

Critical Success Factors:

- Involving stakeholders from different contexts
- Comprehensible presentation of the artefact

Benefit: Validation of ideas and sketches

Effort: Medium-Low

Prototypes

A prototype allows the stakeholders to try out the requirements for the system and experience them thereby.

- Develop the prototype (tool support).
- Training of the stakeholders.
- Observation of prototype usage.
- Collect issues.

Critical Success Factors:

- Effort
- Level of detail of the prototype
- Quality of the review

Benefit:

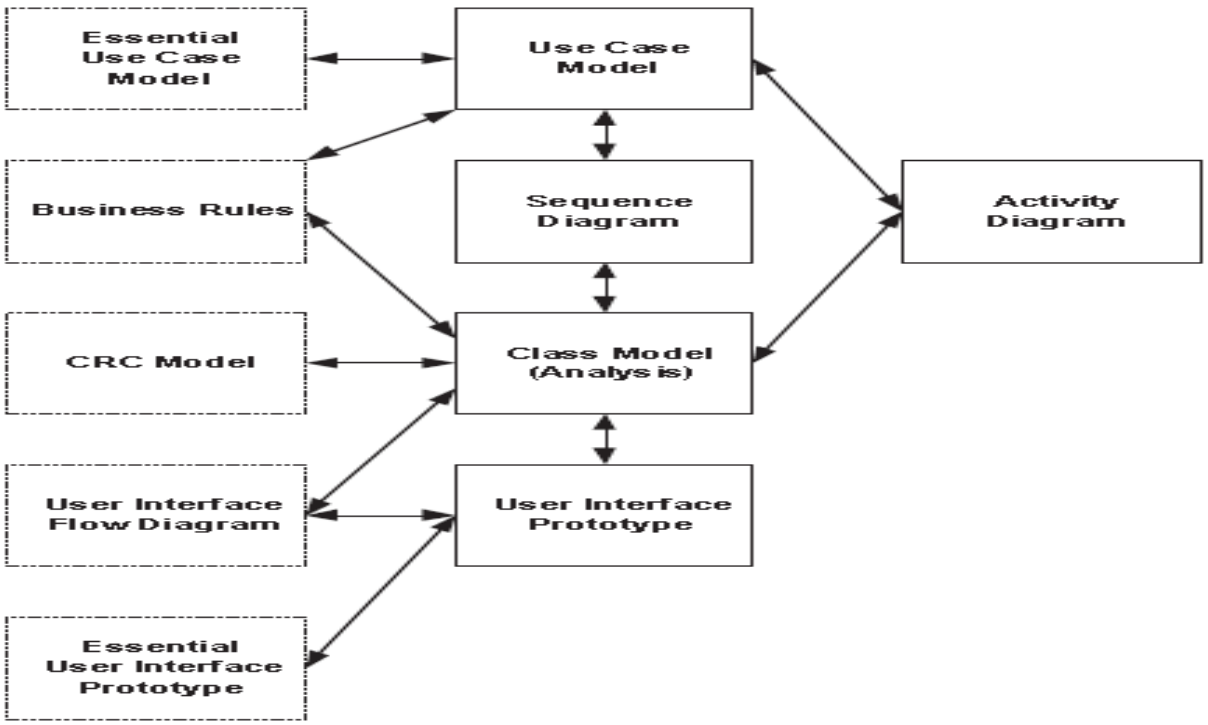
- Highly effective defect detection
- Proof of feasibility

Effort: Very High-High

Chapter 5: Determining What to Build: OO Analysis

The purpose of analysis is to understand what will be built. This is similar to requirements gathering. The main difference is that the focus of requirements gathering is on understanding your users and their potential usage of the system, whereas the focus of analysis shifts to understanding the system itself. The following picture depicts the main artefacts of your analysis efforts and the relationships between them.

5.1. Overview of Analysis artefacts and their Relationships



- The picture has three important implications.
    1. Analysis is an iterative process.
    2. Requirements gathering and analysis are highly interrelated and iterative.
    3. "essential" models, such as essential use case model and essential user interface prototype, evolve into corresponding analysis artefacts—respectively, in to system use case model and user interface prototype.

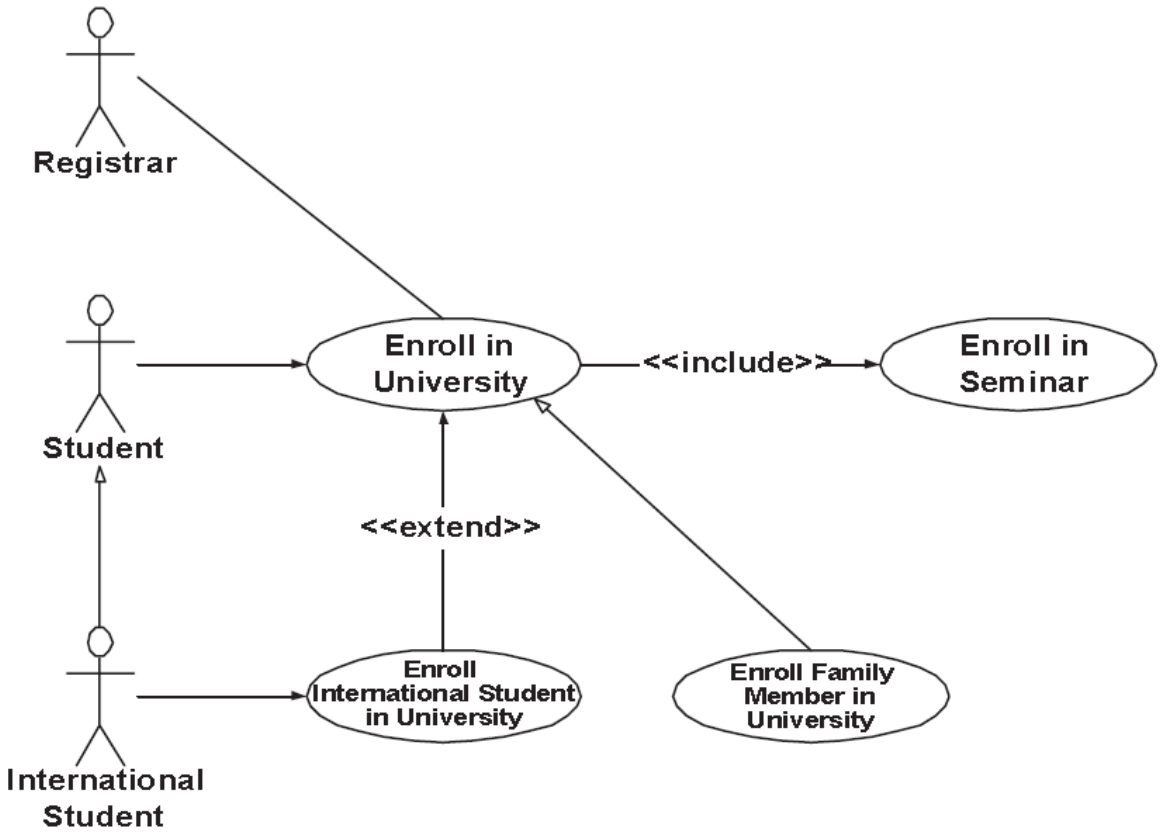
During analysis, your main goal is to evolve your essential use cases into system use cases. The main difference between an essential use case and a system use case is, in the system use case, you include high-level implementation decisions.

- For example, a system use case refers to specific user interface components—such as screens, HTML pages, or reports—something you wouldn't do in an essential use case.

During analysis, you make decisions regarding what will be built or design.

- For example, a design decision is whether your user interface is implemented using browser-based technology, such as HTML pages or graphical user interface (GUI) technology such as Windows.

Because your user interface will work differently depending on the implementation technology, the logic of your system use cases, which reflect the flow of your user interface, will also be affected. Likewise an essential use case model a system use case model is composed of a use case diagram and the accompanying documentation describing the use cases, actors, and associations.The following figures provides an example of a use case diagram, depicts a collection of use cases, actors, their associations, a system boundary box (optional), and packages (optional).



The rectangle around the use cases is called the system boundary box and, as the name suggests, it delimits the scope of your system. The use cases inside the rectangle represent the functionality you intend to implement. Finally, packages are UML constructs that enable you to organize model elements (such as use cases) into groups.

Reuse in Use Case Models: <<extend>>, <<include>>, and Inheritance

- Potential reuse can be modelled through four generalization relationships supported by the UML use case models:
  - extend relationships between use cases,
  - include relationships between use cases,
  - inheritance between use cases,
  - inheritance between actors.

Good Things to Know About Use Case Modelling

An important thing to understand about use case models is that the associations between actors and use cases indicate the need for interfaces. When the actor is a person, then to support the association, you need to develop user interface components, such as screens and reports.

5.2. The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a graphical language for OOAD that gives a standard way to write a software system's blueprint. It helps to visualize, specify, construct, and document the artifacts of an object-oriented system. It is used to depict the structures and the relationships in a complex system.

Brief History

It was developed in 1990s as an amalgamation of several techniques, prominently OOAD technique by Grady Booch, OMT (Object Modeling Technique) by James Rumbaugh, and OOSE (Object Oriented Software Engineering) by Ivar Jacobson. UML attempted to standardize semantic models, syntactic notations, and diagrams of OOAD.

Systems and Models in UML

System: A set of elements organized to achieve certain objectives form a system. Systems are often divided into subsystems and described by a set of models.

Model : Model is a simplified, complete, and consistent abstraction of a system, created for better understanding of the system.

View : A view is a projection of a system's model from a specific perspective.

Conceptual Model of UML

The Conceptual Model of UML encompasses three major elements:

- Basic building blocks
- Rules
- Common mechanisms

Basic Building Blocks

The **three** building blocks of UML are:

- Things
- Relationships
- Diagrams

Things

There are **four** kinds of things in UML, namely:

- **Structural Things** :These are the nouns of the UML models representing the static elements that may be either physical or conceptual. The structural things are class, interface, collaboration, use case, active class, components, and nodes.

- **Behavioral Things** : These are the verbs of the UML models representing the dynamic behavior over time and space. The two types of behavioral things are interaction and state machine.
- **Grouping Things** : They comprise the organizational parts of the UML models. There is only one kind of grouping thing, i.e., package.
- **Notational Things** : These are the explanations in the UML models representing the comments applied to describe elements.

Relationships

Relationships are the connection between things. The four types of relationships that can be represented in UML are:

- **Dependency** : This is a semantic relationship between two things such that a change in one thing brings a change in the other. The former is the independent thing, while the latter is the dependent thing.
    - **Association** : This is a structural relationship that represents a group of links having common structure and common behavior.

Generalization: This represents a generalization/specialization relationship in which subclasses inherit structure and behavior from super-classes.

Realization: This is a semantic relationship between two or more classifiers such that one classifier lays down a contract that the other classifiers ensure to abide by.

Diagrams

A diagram is a graphical representation of a system. It comprises of a group of elements generally in the form of a graph. UML includes nine diagrams in all, namely:

- Class Diagram
- Object Diagram
- Use Case Diagram
- Sequence Diagram
- Collaboration Diagram
- State Chart Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram

Rules

UML has a number of rules so that the models are semantically self-consistent and related to other models in the system harmoniously. UML has semantic **rules** for the following:

- Names
- Scope
- Visibility
- Integrity
- Execution

Common Mechanisms

UML has **four** common mechanisms:

- Specifications
- Adornments
- Common Divisions
- Extensibility Mechanisms

Specifications

In UML, behind each graphical notation, there is a textual statement denoting the syntax and semantics. These are the specifications. The specifications provide a semantic backplane that contains all the parts of a system and the relationship among the different paths.

Adornments

Each element in UML has a unique graphical notation. Besides, there are notations to represent the important aspects of an element like name, scope, visibility, etc.

Common Divisions

Object-oriented systems can be divided in many ways. The two common ways of division are:

- **Division of classes and objects** : A class is an abstraction of a group of similar objects. An object is the concrete instance that has actual existence in the system.
- **Division of Interface and Implementation** : An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

Extensibility Mechanisms

UML is an open-ended language. It is possible to extend the capabilities of UML in a controlled manner to suit the requirements of a system. The extensibility mechanisms are:

- **Stereotypes** : It extends the vocabulary of the UML, through which new building blocks can be created out of existing ones.
- **Tagged Values** : It extends the properties of UML building blocks.
- **Constraints** : It extends the semantics of UML building blocks.

5.3. UML Basic Notations

UML defines specific notations for each of the building blocks.

Class

A class is represented by a rectangle having three sections:

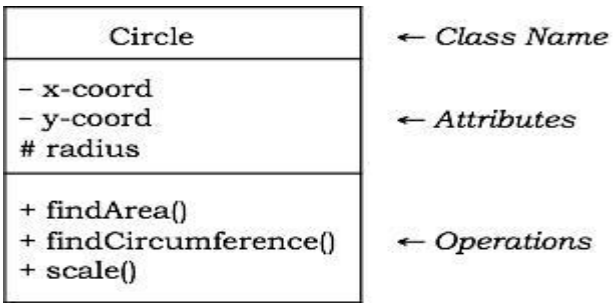
- the top section containing the name of the class
- the middle section containing class attributes
- the bottom section representing operations of the class

The visibility of the attributes and operations can be represented in the following ways:

- **Public** : A public member is visible from anywhere in the system. In class diagram, it is prefixed by the symbol '+'.
- **Private** : A private member is visible only from within the class. It cannot be accessed from outside the class. A private member is prefixed by the symbol '−'.
- **Protected** : A protected member is visible from within the class and from the subclasses inherited from this class, but not from outside. It is prefixed by the symbol '#'.

An abstract class has the class name written in italics.

Example : Let us consider the Circle class introduced earlier. The attributes of Circle are x-coord, y-coord, and radius. The operations are findArea(), findCircumference(), and scale(). Let us assume that x-coord and y-coord are private data members, radius is a protected data member, and the member functions are public. The following figure gives the diagrammatic representation of the class.

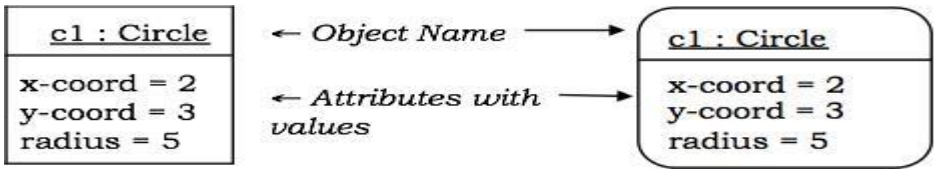


Object

An object is represented as a rectangle with two sections:

- The top section contains the name of the object with the name of the class or package of which it is an instance of. The name takes the following forms:
  - **object-name** : class-name
  - **object-name** : class-name :: package-name
  - **class-name** : in case of anonymous objects
- The bottom section represents the values of the attributes. It takes the form attribute-name = value.
- Sometimes objects are represented using rounded rectangles.

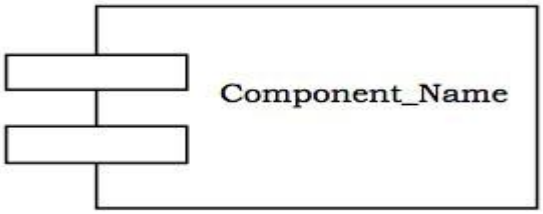
Example : Let us consider an object of the class Circle named c1. We assume that the center of c1 is at (2, 3) and the radius of c1 is 5. The following figure depicts the object.



Component

A component is a physical and replaceable part of the system that conforms to and provides the realization of a set of interfaces. It represents the physical packaging of elements like classes and interfaces.

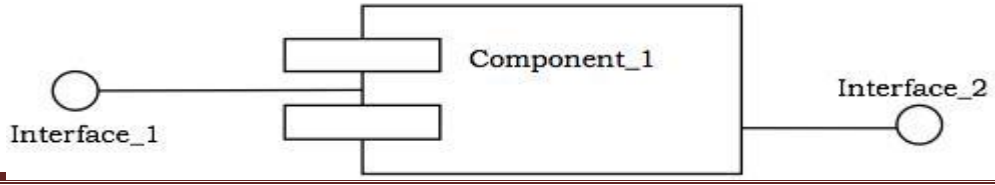
Notation : In UML diagrams, a component is represented by a rectangle with tabs as shown in the figure below.



Interface

Interface is a collection of methods of a class or component. It specifies the set of services that may be provided by the class or component.

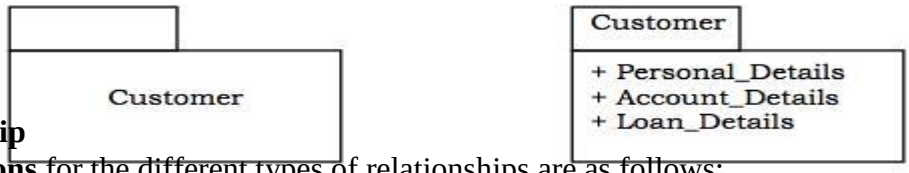
Notation : Generally, an interface is drawn as a circle together with its name. An interface is almost always attached to the class or component that realizes it. The following figure gives the notation of an interface.



Package

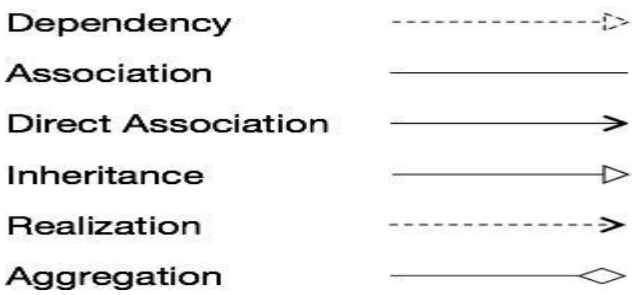
A package is an organized group of elements. A package may contain structural things like classes, components, and other packages in it.

Notation:Graphically, a package is represented by a tabbed folder. A package is generally drawn with only its name. However it may have additional details about the contents of the package. See the following figures.



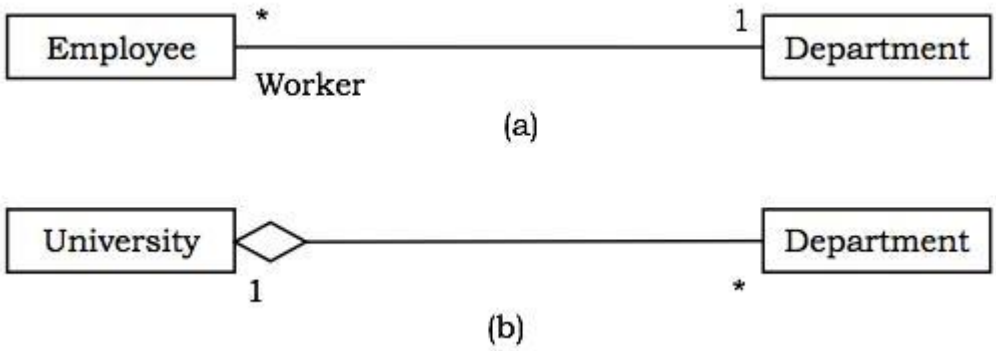
Relationship

The **notations** for the different types of relationships are as follows:



Usually, elements in a relationship play specific roles in the relationship. A role name signifies the behavior of an element participating in a certain context.

Example : The following figures show examples of different relationships between classes. The first figure shows an association between two classes, Department and Employee, wherein a department may have a number of employees working in it. Worker is the role name. The '1' alongside Department and '*' alongside Employee depict that the cardinality ratio is one–to–many. The second figure portrays the aggregation relationship, a University is the "whole– of" many Departments.



5.4. UML STRUCTURED DIAGRAMS

UML structural diagrams are categorized as follows: ***class diagram, object diagram, component diagram, and deployment diagram.***

Class Diagram

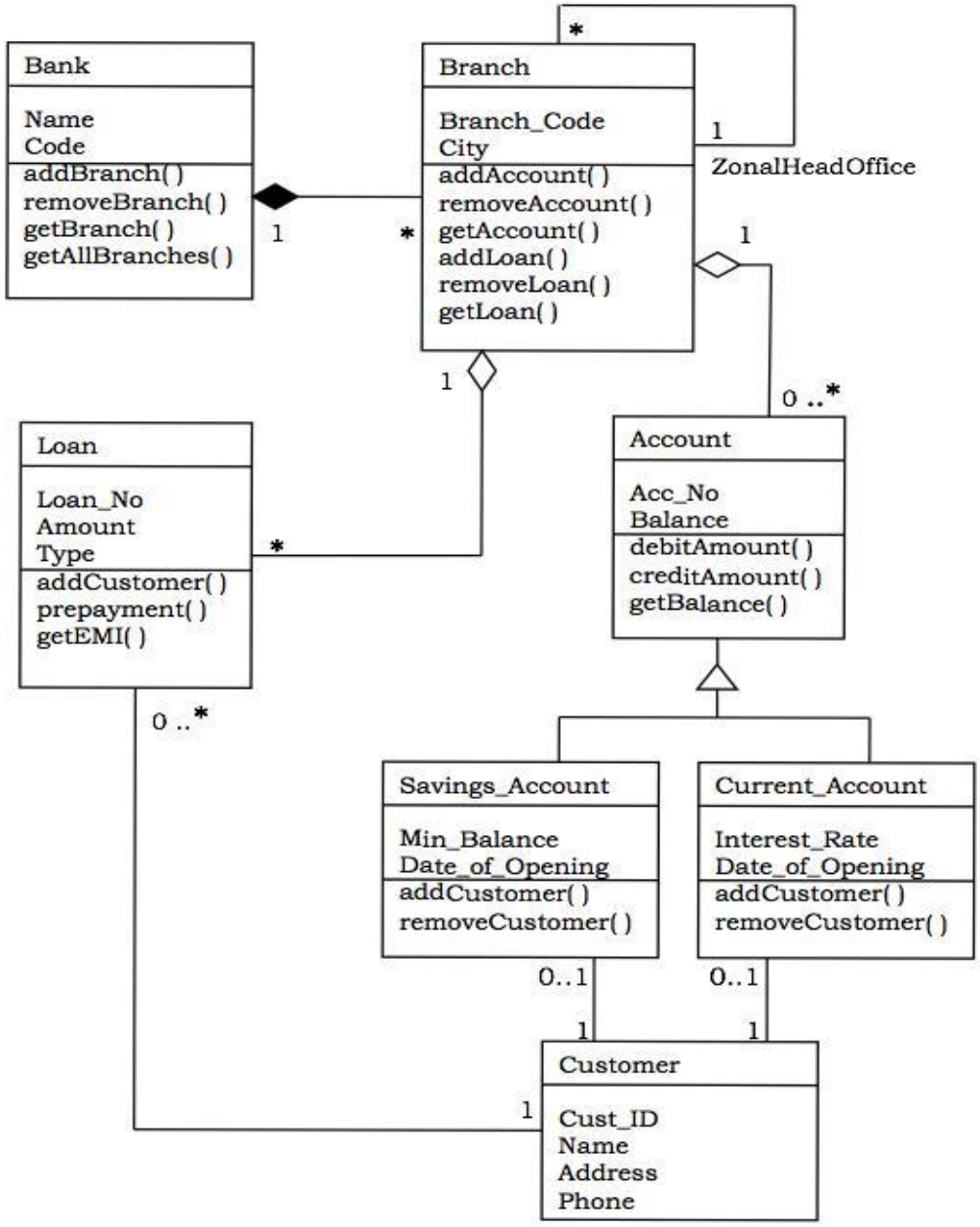
A class diagram models the static view of a system. It comprises of the classes, interfaces, and collaborations of a system; and the relationships between them.

Class Diagram of a System

Let us consider a simplified Banking System.

A bank has many branches. In each zone, one branch is designated as the zonal head office that supervises the other branches in that zone. Each branch can have multiple accounts and loans. An account may be either a savings account or a current account. A customer may open both a savings account and a current account. However, a customer must not have more than one savings account or current account. A customer may also procure loans from the bank.

The following figure shows the corresponding class diagram.



Classes in the System

Bank, Branch, Account, Savings Account, Current Account, Loan, and Customer.

Relationships

- **A Bank "has–a" number of Branches**: composition, one–to–many
- A Branch with role Zonal Head Office supervises other Branches : unary association, one–to-many
- A Branch "has–a" number of accounts : aggregation, one–to–many

From the class Account, two classes have inherited, namely, Savings Account and Current Account.

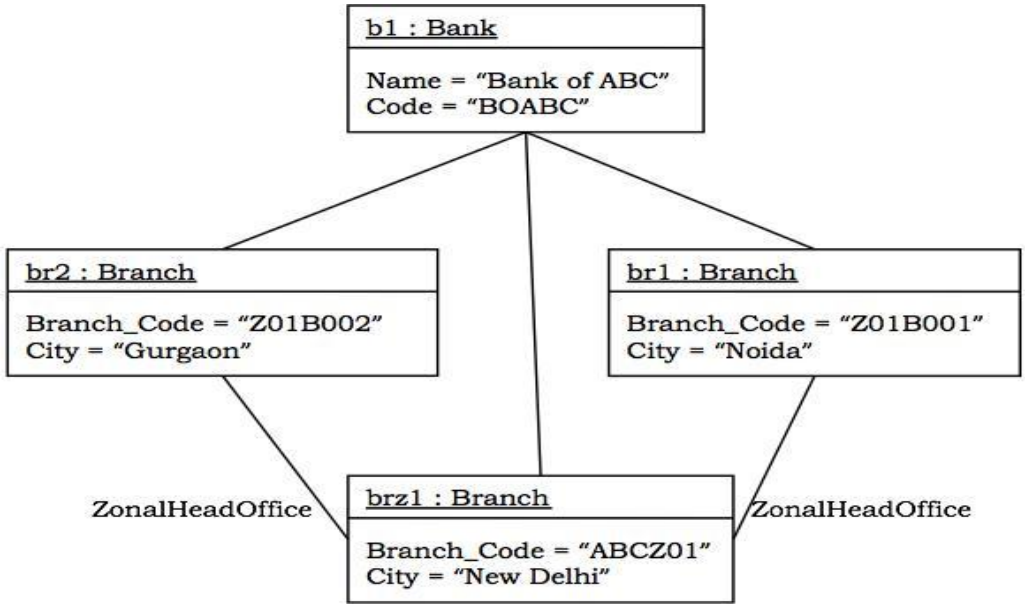
- **A Customer can have one Current Account** : association, one–to–one

- **A Customer can have one Savings Account** : association, one–to–one
- **A Branch "has–a" number of Loans** : aggregation, one–to–many
- **A Customer can take many loans** : association, one–to–many

Object Diagram

An object diagram models a group of objects and their links at a point of time. It shows the instances of the things in a class diagram. Object diagram is the static part of an interaction diagram.

Example : The following figure shows an object diagram of a portion of the class diagram of the Banking System.



Component Diagram

Component diagrams show the organization and dependencies among a group of components.

Component diagrams comprise of:

- Components
- Interfaces
- Relationships
- Packages and Subsystems (optional) Component diagrams are used for:
- Constructing systems through forward and reverse engineering.
- Modeling configuration management of source code files while developing a system using an object-oriented programming language.
- Representing schemas in modeling databases.
- Modeling behaviors of dynamic systems.

Deployment Diagram

A deployment diagram puts emphasis on the configuration of runtime processing nodes and their components that live on them. They are commonly comprised of nodes and dependencies, or associations between the nodes.

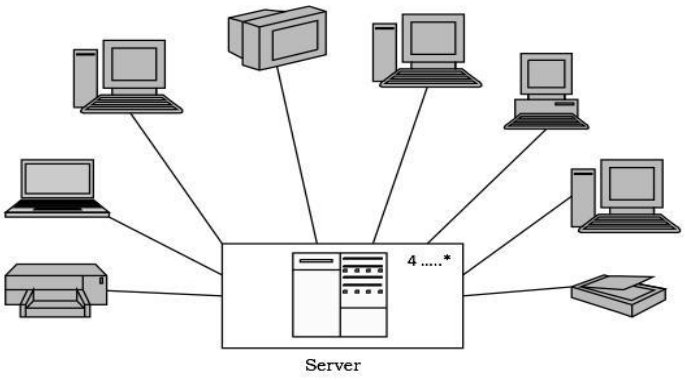
Deployment diagrams are used to:

- model devices in embedded systems that typically comprise of software-intensive collection of hardware.
- represent the topologies of client/server systems.
- model fully distributed systems.

Example

The following figure shows the topology of a computer system that follows client/server architecture. The figure illustrates a node stereotyped as server that comprises of processors. The figure indicates that four or more servers are deployed at the system. Connected to the server are the client nodes, where each node represents a terminal device such as workstation, laptop,

scanner, or printer. The nodes are represented using icons that clearly depict the real-world equivalent.



5.5. UML Behavioral Diagrams

UML behavioral diagrams visualize, specify, construct, and document the dynamic aspects of a system. The behavioral diagrams are categorized as follows: use case diagrams, interaction diagrams, state–chart diagrams, and activity diagrams.

Use Case Model

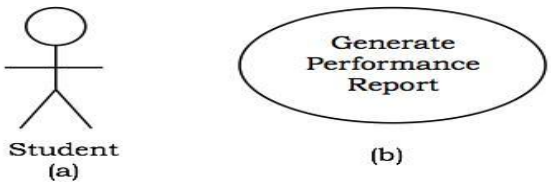
Use Case

A use case describes the sequence of actions a system performs yielding visible results. It shows the interaction of things outside the system with the system itself. Use cases may be applied to the whole system as well as a part of the system.

Actor

An actor represents the roles that the users of the use cases play. An actor may be a person (e.g. student, customer), a device (e.g. workstation), or another system (e.g. bank, institution).

The following figure shows the notations of an actor named Student and a use case called Generate Performance Report.



Use Case Diagrams

Use case diagrams present an outside view of the manner the elements in a system behave and how they can be used in the context.

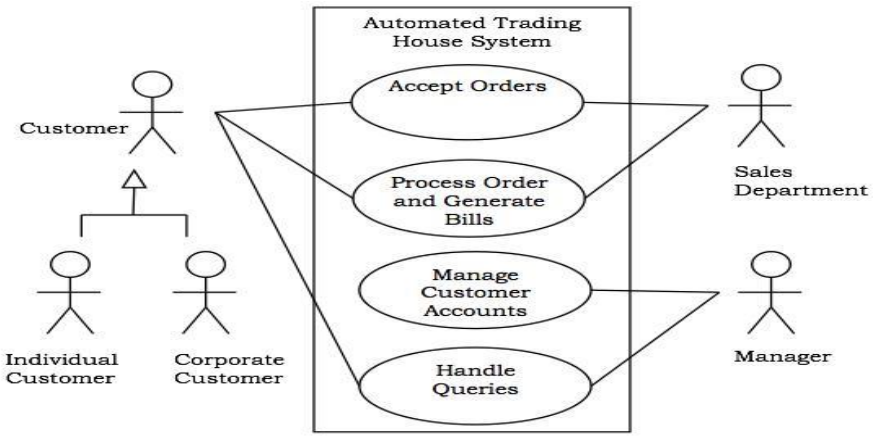
Use case diagrams comprise of:

- Use cases
- Actors
- Relationships like dependency, generalization, and association Use case diagrams are used:
  o model the context of a system by enclosing all the activities of a system within a rectangle and focusing on the actors outside the system by interacting with it.To model the requirements of a system from the outside point of view.

Example

Let us consider an Automated Trading House System. We assume the following features of the system:

- The trading house has transactions with two types of customers, individual customers and corporate customers.
- Once the customer places an order, it is processed by the sales department and the customer is given the bill.
- The system allows the manager to manage customer accounts and answer any queries posted by the customer.



Interaction Diagrams

Interaction diagrams depict interactions of objects and their relationships. They also include the messages passed between them. There are two types of interaction diagrams:

- Sequence Diagrams
- Collaboration Diagrams

Interaction diagrams are used for modeling:

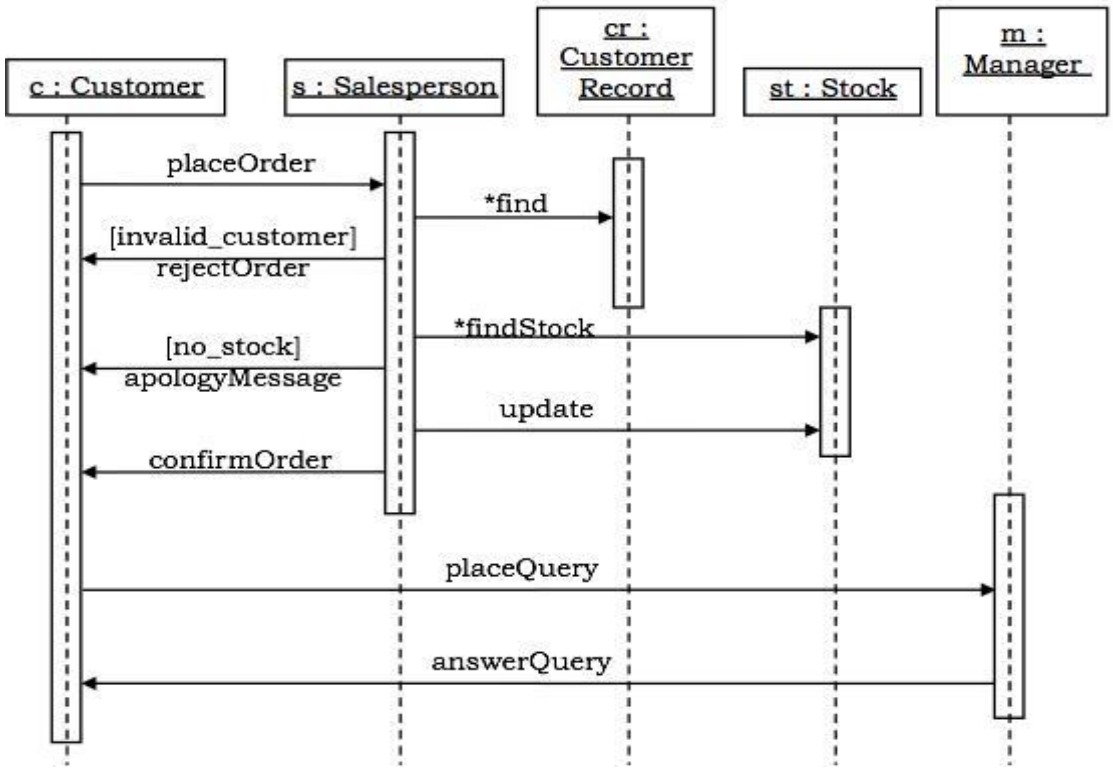
- the control flow by time ordering using sequence diagrams.
- the control flow of organization using collaboration diagrams.

Sequence Diagrams

Sequence diagrams are interaction diagrams that illustrate the ordering of messages according to time.

Notations: These diagrams are in the form of two-dimensional charts. The objects that initiate the interaction are placed on the x–axis. The messages that these objects send and receive are placed along the y–axis, in the order of increasing time from top to bottom

Example : A sequence diagram for the Automated Trading House System is shown in the following figure.

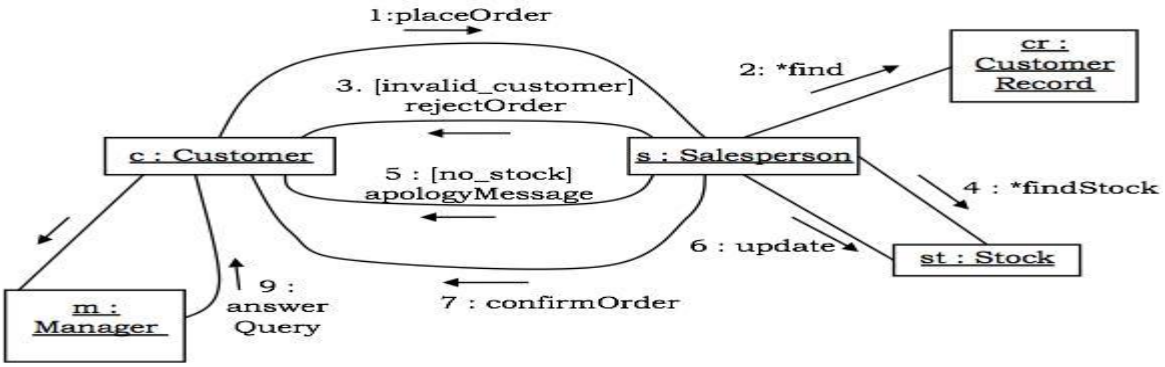


Collaboration Diagrams

Collaboration diagrams are interaction diagrams that illustrate the structure of the objects that send and receive messages.

Notations : In these diagrams, the objects that participate in the interaction are shown using vertices. The links that connect the objects are used to send and receive messages. The message is shown as a labeled arrow.

Example : Collaboration diagram for the Automated Trading House System is illustrated in the figure below.



State–Chart Diagrams

A state–chart diagram shows a state machine that depicts the control flow of an object from one state to another. A state machine portrays the sequences of states which an object undergoes due to events and their responses to events.

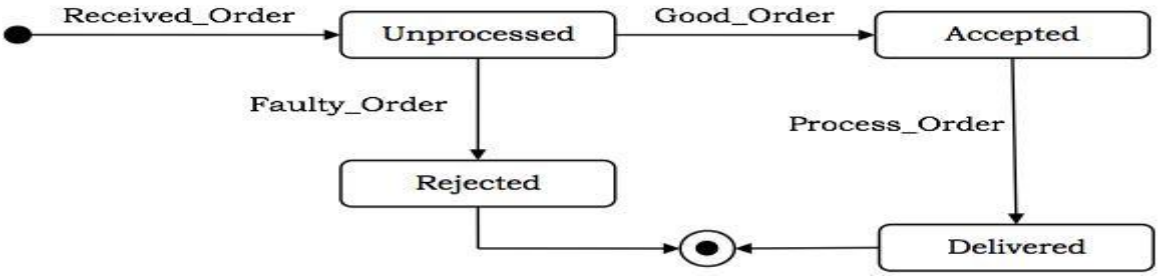
State–Chart Diagrams comprise of:

- States: Simple or Composite
- Transitions between states
- Events causing transitions
- Actions due to the events

State-chart diagrams are used for modeling objects which are reactive in nature.

Example

In the Automated Trading House System, let us model Order as an object and trace its sequence. The following figure shows the corresponding state–chart diagram.



Activity Diagrams

An activity diagram depicts the flow of activities which are ongoing non-atomic operations in a state machine. Activities result in actions which are atomic operations.

Activity diagrams comprise of:

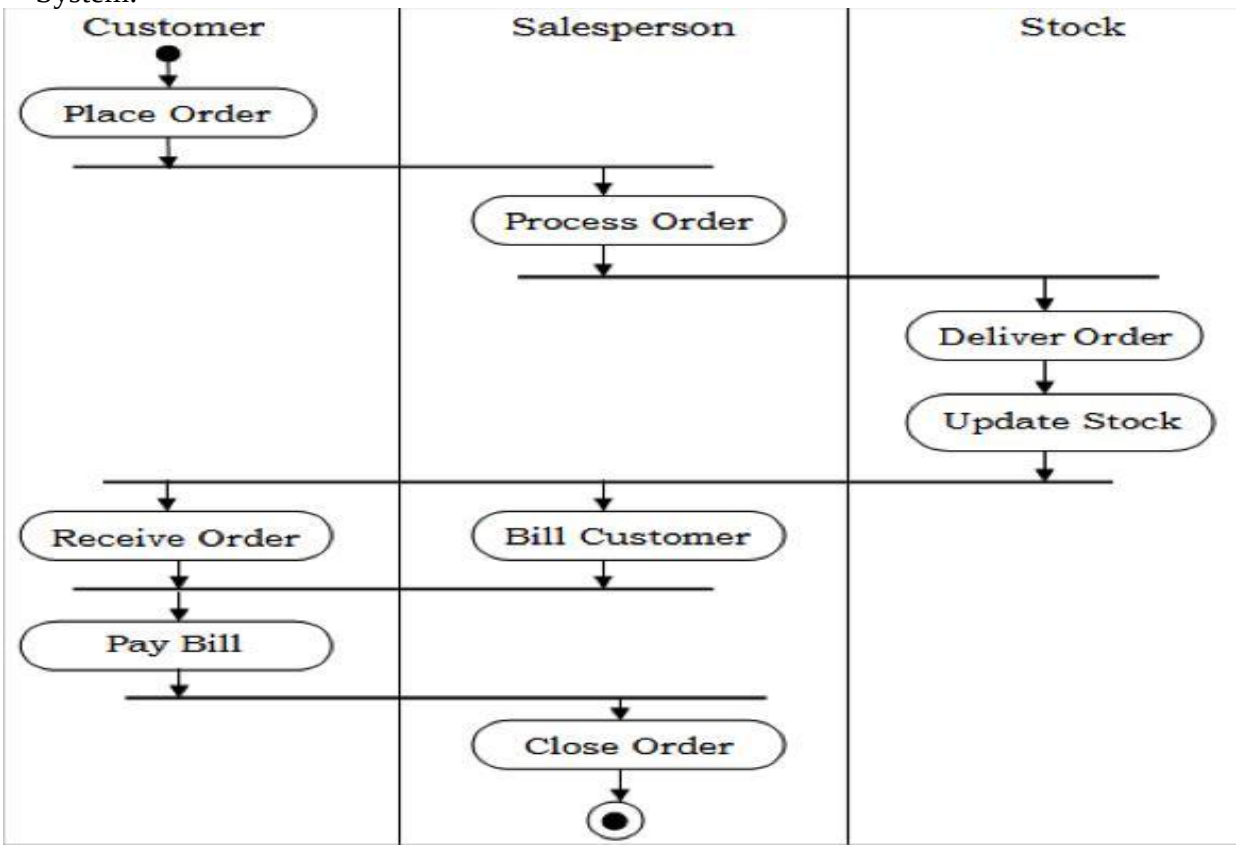
- Activity states and action states
- Transitions
- Objects

Activity diagrams are used for modeling:

- Workflows as viewed by actors, interacting with the system.
- details of operations or computations using flowcharts.

Example

The following figure shows an activity diagram of a portion of the Automated Trading House System.



Chapter 6: Determining How to Build Your System: OO 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

6.1. 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.

6.2. 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.

6.2.1 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.

6.2.2. 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

6.2.3. 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.

6.2.4. 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 modeled using boundary use cases.

6.3. 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

6.3.1. 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

6.3.2. Object Representation

Once the classes are identified, they need to be represented using object modeling 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.

6.3.3. 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 eveloped.
- 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.

6.3.4. 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.

6.3.5. 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.

6.3.6. 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.

6.3.7. 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.

6.4. 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.

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.

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** : Behavioral diagrams

Features

The features of a good documentation are:

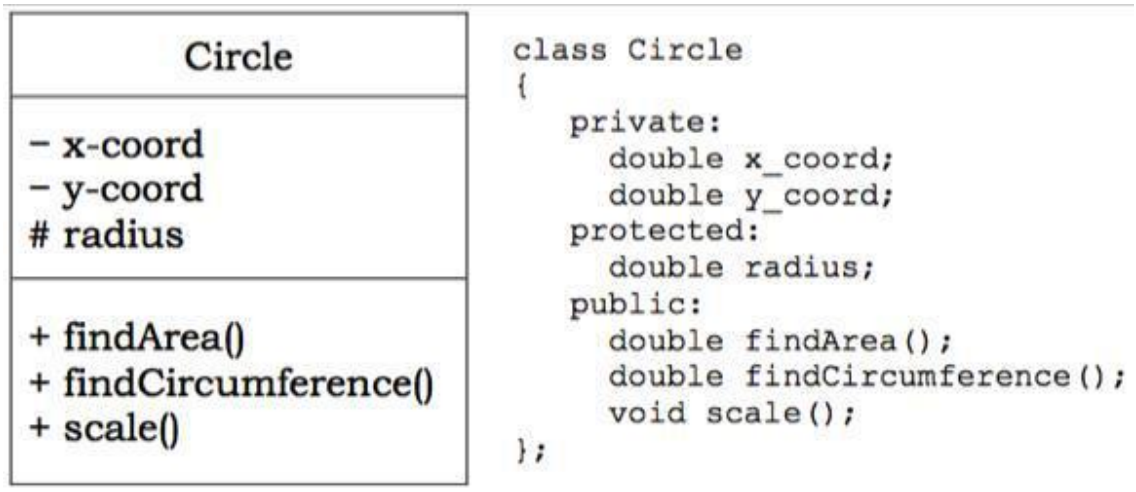
- Concise and at the same time, unambiguous, consistent, and complete
- Traceable to the system's requirement specifications
- Well-structured
- Diagrammatic instead of descriptive

6.5. IMPLEMENTATION STRATEGIES

Implementing an object-oriented design generally involves using a standard object oriented programming language (OOPL) or apping object designs to databases. In most cases,it involves both.

Implementation using Programming Languages

Usually, the task of transforming an object design into code is a straightforward process. Any object-oriented programming language like C++,Java,Smalltalk,C# and Python, includes provision for representing classes. In this chapter, we exemplify the concept using C++.The following figure shows the representation of the class Circle using C++.



Implementing Associations

Most programming languages do not provide constructs to implement associations directly. So the task of implementing associations needs considerable thought.

Associations may be either unidirectional or bidirectional. Besides, each association may be either one–to–one, one–to–many, or many–to–many.

Unidirectional Associations

For implementing unidirectional associations, care should be taken so that unidirectionality is maintained. The implementations for different multiplicity are as follows:

- **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.

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. **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.

Bi-directional Associations

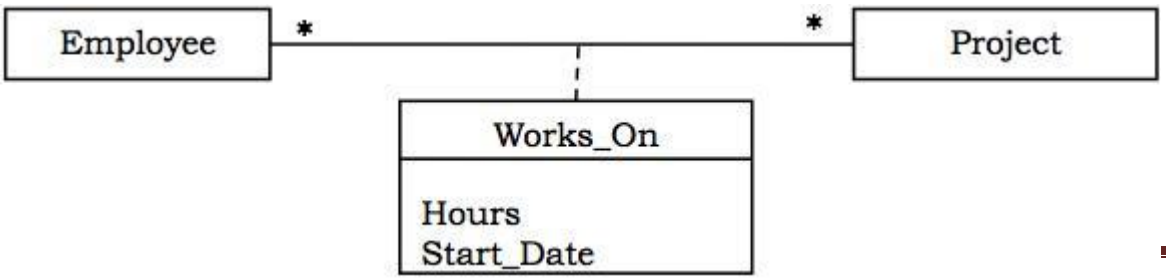
To implement bi-directional association, links in both directions require to be maintained.

- **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.

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

Implementing Associations as Classes

If an association has some attributes associated, it should be implemented using a separate class. For example, consider the one–to–one association between Employee and Project as shown in the figure below.



Implementing Constraints

Constraints in classes restrict the range and type of values that the attributes may take. In order to implement constraints, a valid default value is assigned to the attribute when an object is instantiated from the class. Whenever the value is changed at runtime, it is checked whether the value is valid or not. An invalid value may be handled by an exception handling routine or other methods.

Implementing State Charts

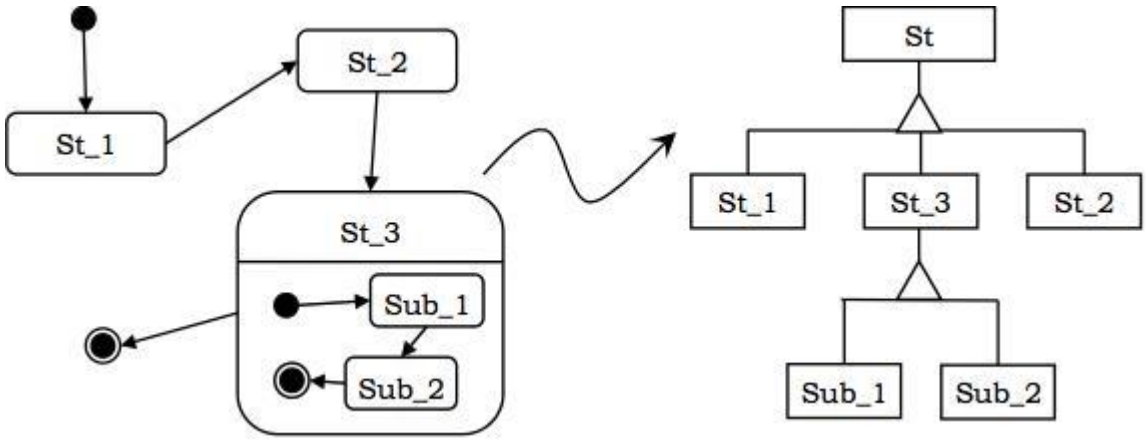
There are two alternative implementation strategies to implement states in state chart diagrams.

Enumerations within Class

In this approach, the states are represented by different values of a data member (or set of data members). The values are explicitly defined by an enumeration within the class. The transitions are represented by member functions that change the value of the concerned data member.

Arrangement of Classes in a Generalization Hierarchy

In this approach, the states are arranged in a generalization hierarchy in a manner that they can be referred by a common pointer variable. The following figure shows a transformation from state chart diagram to a generalization hierarchy.



Object Mapping to Database System

Persistency of Objects

An important aspect of developing object-oriented systems is persistency of data. Through persistency, objects have longer lifespan than the program that created it. Persistent data is saved on secondary storage medium from where it can be reloaded when required.

Overview of RDBMS

A database is an ordered collection of related data.

A database management system (DBMS) is a collection of software that facilitates the processes of defining, creating, storing, manipulating, retrieving, sharing, and removing data in databases.

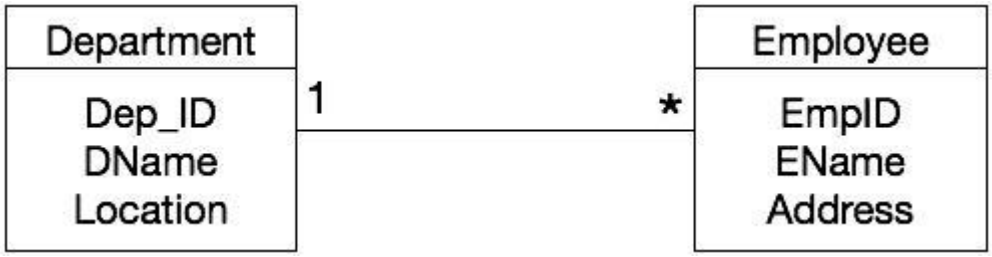
In relational database management systems (RDBMS), data is stored as relations or tables, where each column or field represents an attribute and each row or tuple represents a record of an instance.

Each row is uniquely identified by a chosen set of minimal attributes called **primary key**.

A **foreign key** is an attribute that is the primary key of a related table.

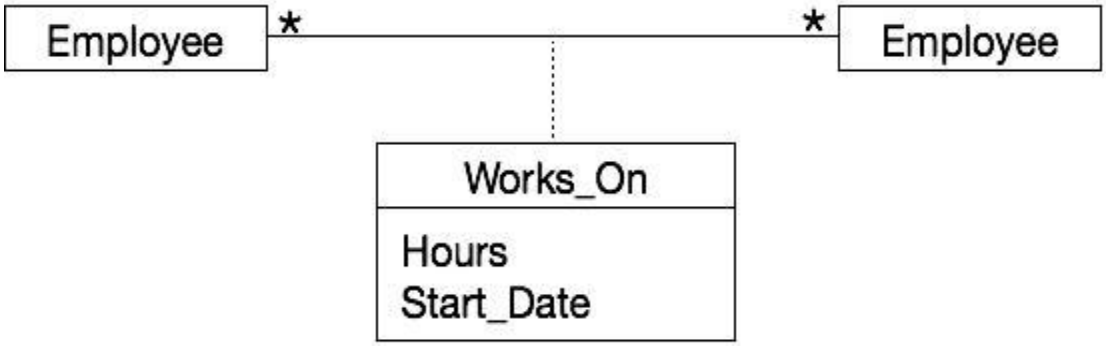
One–to–Many Associations

To implement 1:N associations, the primary key of the table in the 1-side of the association is assigned as the foreign key of the table at the N-side of the association. For example, consider the association between Department and Employee:



Many–to–Many Associations

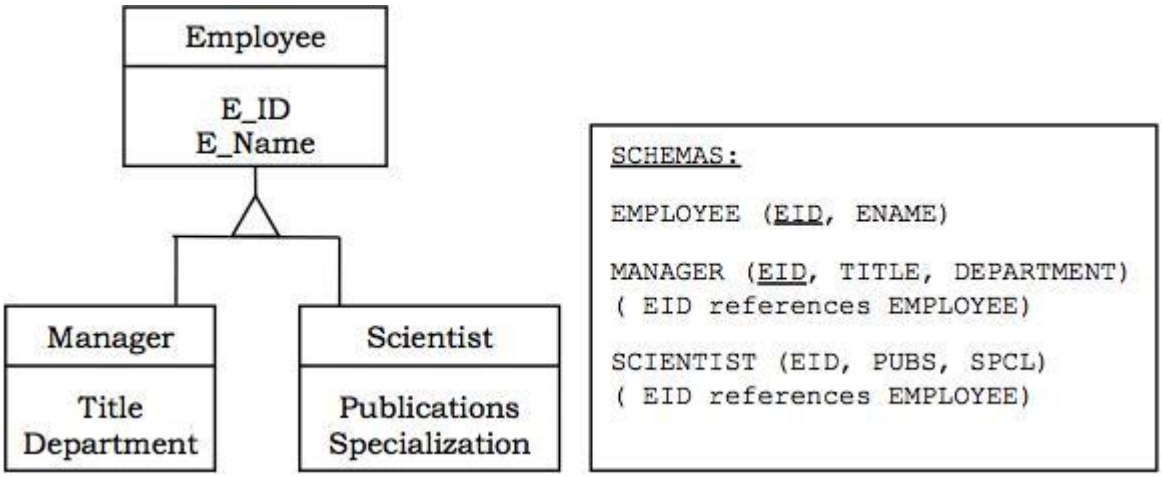
To implement M:N associations, a new relation is created that represents the association. For example, consider the following association between Employee and Project:



Mapping Inheritance to Tables

To map inheritance, the primary key of the base table(s) is assigned as the primary key as well as the foreign key in the derived table(s).

Example



7.1 TESTING AND QUALITY ASSURANCE

Once a program code is written, it must be tested to detect and subsequently handle all errors in it. A number of schemes are used for testing purposes.
Another important aspect is the fitness of purpose of a program that ascertains whether the program serves the purpose which it aims for. The fitness defines the software quality.

7.1.1. Testing Object-Oriented Systems

Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.

7.1.2. Unit Testing

In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free. Unit testing is the responsibility of the application engineer who implements the structure.

7.1.3. Subsystem Testing

This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside. Subsystem tests can be used as regression tests for each newly released version of the subsystem.

7.1.4.System Testing

System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases.

7.2. Categories of System Testing

- **Alpha testing** : This is carried out by the testing team within the organization that develops software.
- **Beta testing** : This is carried out by select group of co-operating customers.
- **Acceptance testing** : This is carried out by the customer before accepting the deliverables.

7.3. Object-Oriented Testing Techniques

Grey Box Testing

The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are:

- **State model based testing** :This encompasses state coverage, state transition coverage, and state transition path coverage.
- **Use case based testing: Each** scenario in each use case is tested.
- **Class diagram based testing:** Each class, derived class, associations, and aggregations are tested.
- **Sequence diagram based testing** : The methods in the messages in the sequence diagrams are tested.

7.4. Techniques for Subsystem Testing

The two main approaches of subsystem testing are:

- **Thread based testing** : All classes that are needed to realize a single use case in a subsystem are integrated and tested.
- **Use based testing** : The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

7.5. The Full-Lifecycle Object-Oriented Testing (FLOOT)

The Full-Lifecycle Object-Oriented Testing (FLOOT) methodology is a collection of testing techniques to verify and validate object-oriented software. The FLOOT lifecycle is depicted in Figure 1, indicating a wide variety of techniques (described in Table 1 are available to you throughout all aspects of software development. The list of techniques is not meant to be complete: instead the goal is to make it explicit that you have a wide range of options available to you. It is important to understand that although the FLOOT method is presented as a collection of serial phases it does not need to be so: the techniques of FLOOT can be applied with evolutionary/agile processes as well. The reason why I present the FLOOT in a "traditional" manner is to make it explicit that you can in fact test throughout all aspects of software development, not just during coding.

- **Figure 1. The FLOOT Lifecycle.**

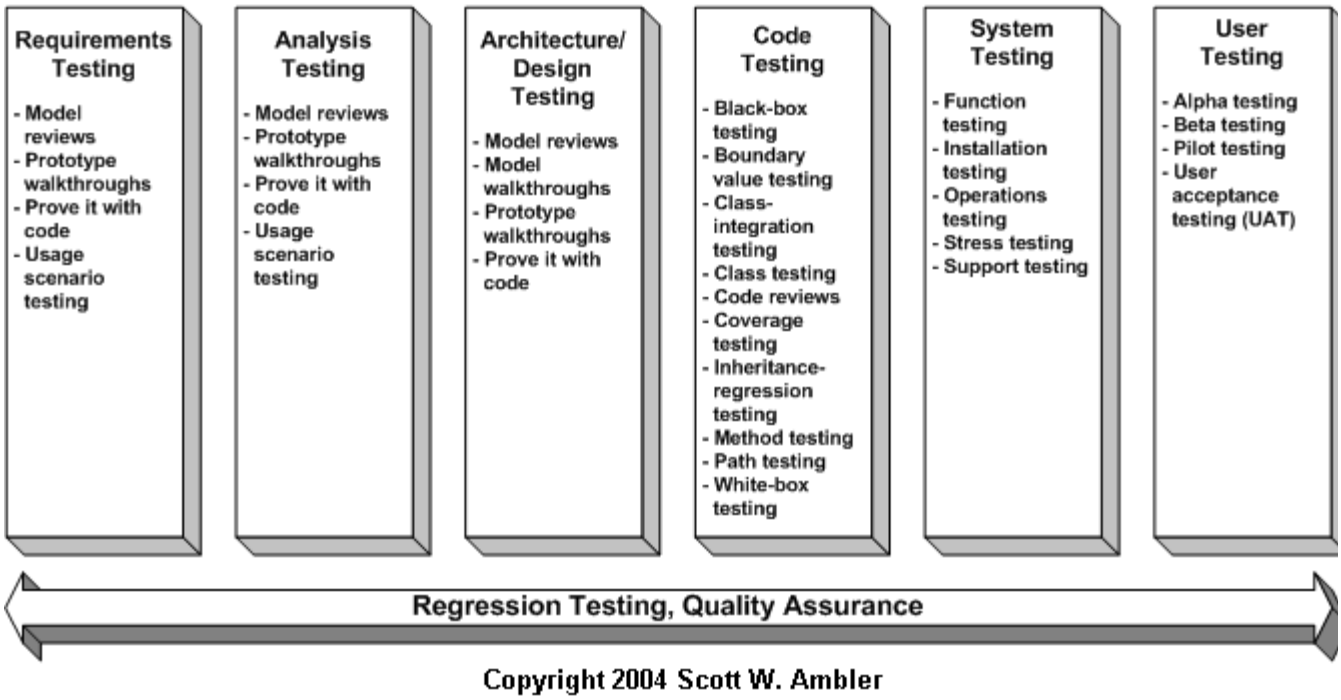


Table 1. Testing techniques.

FLOOT Technique	Description
Black-box testing	Testing that verifies the item being tested when given the appropriate input provides the expected results.
Boundary-value testing	Testing of unusual or extreme situations that an item should be able to handle.
Class testing	The act of ensuring that a class and its instances (objects) perform as defined.
Class-integration testing	The act of ensuring that the classes, and their instances, form some software perform as defined.
Code review	A form of technical review in which the deliverable being reviewed is source code.
Component testing	The act of validating that a component works as defined.
Coverage testing	The act of ensuring that every line of code is exercised at least once.
Design review	A technical review in which a design model is inspected.
Inheritance-regression testing	The act of running the test cases of the super classes, both direct and indirect, on a given subclass.
Integration testing	Testing to verify several portions of software work together.
Method testing	Testing to verify a method (member function) performs as defined.
Model review	An inspection, ranging anywhere from a formal technical review to an informal walkthrough, by others who were not directly involved with the development of the model.
Path testing	The act of ensuring that all logic paths within your code are exercised at least once.
Prototype review	A process by which your users work through a collection of use cases, using a prototype as if it was the real system. The main goal is to test whether the design of the prototype meets their needs.
Prove it with code	The best way to determine if a model actually reflects what is needed, or what should be built, is to actually build software based on that model that show that the model works.
Regression testing	The acts of ensuring that previously tested behaviors still work as expected after changes have been made to an application.
Stress testing	The act of ensuring that the system performs as expected under high volumes of transactions, users, load, and so on.
Technical review	A quality assurance technique in which the design of your application is examined critically by a group of your peers. A review typically focuses on accuracy, quality, usability, and completeness. This process is often referred to as a walkthrough, an inspection, or a peer review.
Usage scenario testing	A testing technique in which one or more person(s) validate a model by acting through the logic of usage scenarios.
User interface testing	The testing of the user interface (UI) to ensure that it follows accepted UI standards and meets the requirements defined for it. Often referred to as graphical user interface (GUI) testing.
White-box testing	Testing to verify that specific lines of code work as defined. Also referred to as clear-box testing.

7.6.Software Quality Assurance

Software Quality

Schulmeyer and McManus have defined software quality as “the fitness for use of the total software product”. A good quality software does exactly what it is supposed to do and is interpreted in terms of satisfaction of the requirement specification laid down by the user.

7.6.1. Quality Assurance

Software quality assurance is a methodology that determines the extent to which a software product is fit for use. The activities that are included for determining software quality are:

- Auditing
- Development of standards and guidelines
- Production of reports
- Review of quality system

7.6.2. Quality Factors

- **Correctness** : Correctness determines whether the software requirements are appropriately met.
- **Usability** : Usability determines whether the software can be used by different categories of users (beginners, non-technical, and experts).
- **Portability** : Portability determines whether the software can operate in different platforms with different hardware devices.
- **Maintainability** : Maintainability determines the ease at which errors can be corrected and modules can be updated.
- **Reusability** : Reusability determines whether the modules and classes can be reused for developing other software products.

Object-Oriented Metrics

Metrics can be broadly classified into three categories: project metrics, product metrics, and process metrics.

Project Metrics

Project Metrics enable a software project manager to assess the status and performance of an ongoing project. The following metrics are appropriate for object-oriented software projects:

- Number of scenario scripts
- Number of key classes
- Number of support classes
- Number of subsystems

Product Metrics

Product metrics measure the characteristics of the software product that has been developed. The product metrics suitable for object-oriented systems are:

- **Methods per Class** : It determines the complexity of a class. If all the methods of a class are assumed to be equally complex, then a class with more methods is more complex and thus more susceptible to errors.
- **Inheritance Structure** : Systems with several small inheritance lattices are more well–structured than systems with a single large inheritance lattice. As a thumb rule, an inheritance tree should not have more than 7 (± 2) number of levels and the tree should be balanced.
- **Coupling and Cohesion** : Modules having low coupling and high cohesion are considered to be better designed, as they permit greater reusability and maintainability.
- **Response for a Class** : It measures the efficiency of the methods that are called by the instances of the class.

Process Metrics

Process metrics help in measuring how a process is performing. They are collected over all projects over long periods of time. They are used as indicators for long-term software process improvements. Some process metrics are:

- Number of KLOC (Kilo Lines of Code)
- Defect removal efficiency
- Average number of failures detected during testing
- Number of latent defects per KLOC

Chapter 8: Software Process

8.1. Process

According to Webster, the term *process* means "a particular method of doing something, generally involving a number of steps or operations." In software engineering, the phrase *software process* refers to the methods of developing software.

8.2. Software Process

A software process is a set of activities, together with ordering constraints among them, such that if the activities are performed properly and in accordance with the ordering constraints, the desired result is produced. The basic desired result is high quality and productivity.

The process that deals with the technical and management issues of software development is called a *software process.* Software process is that a set of activities whose goal is the development or evolution of software.

In an organization whose major business is software development, there are typically many processes executing simultaneously. Many of these do not concern software engineering, though they do impact software development. These could be considered non software engineering process. Business processes, social processes, and training processes, are all examples of processes that come under this. These processes also affect the software development activity.

Software process as consisting of many component processes, each consisting of a certain type of activity Each of these component processes typically has a different objective, though they obviously cooperate with each other to satisfy the overall software engineering objective.

8. 3. Processes and Process Models

Software process is that a set of activities whose goal is the development or evolution of software. A simplified representation of a software process, presented from a specific perspective. A successful project is the one that satisfies the expectations on all the three goals of cost, schedule, and quality.

When planning and executing a software project, the decisions are mostly taken with a view to ultimately reduce the cost or the cycle time, or for improving the quality. *Software projects utilize a process to organize the execution of tasks to achieve the goals on the cost, schedule, and quality*.
A project's process specification defines the tasks the project should perform, and the order in which they should be done. The actual process exists when the project is actually executed.
A process model specifies a general process, usually as a set of stages in which a project should be divided, the order in which the stages should be executed, and any other constraints and conditions on the execution of stages.

The basic premise behind a process model is that, in the situations for which the model is applicable, using the process model as the projects process will lead to low cost, high quality, reduced time. A project's process may utilize some process model. That is, the project's process has a general resemblance to the process model with the actual tasks being specific to the project. However, using a process model is not simply translating the tasks in the process model to tasks in the project.

That is, a process specifies the steps, the project executes these steps, and during the course of execution products are produced. A process limits the degrees of freedom for a project by specifying what types of activities must be undertaken and in what order, such that the "shortest" (or the most efficient) path is obtained from the user needs to the software satisfying these needs. It should be clear that it is the process that drives a project and heavily influences the expected outcomes of a project.

8.3.1. Component Software Processes

Generally, the development process is the central process which specifies the tasks to be done in a project. Planning and scheduling the tasks and monitoring their execution fall in the domain of project management process. Hence, there are clearly two major components in a software process a development process and a project management.

As development processes generally do not focus on evolution and changes, to handle them another process called *software configuration control process,* is often used. The objective of this component process is to primarily deal with managing change, so that the integrity of the products is not violated despite changes. Sometimes, changes in requirements may be handled separately by a *requirements change management process.*

These three constituent processes focus on the projects and the products and can be considered as comprising the *product engineering processes,* as their main objective is to produce the desired product. If the software process can be viewed as a static entity, then these three component processes will suffice.

The whole process of understanding the current process, analyzing its properties, determining how to improve, and then affecting the improvement is dealt with by the *process management process.* These component processes are distinct not only in the type of activities performed in them, but typically also in the people who perform the activities specified by the process.

- In a project, development activities are performed by programmers, designers, testers, etc.
- The project management process activities are performed by the project management.
- Configuration control process activities are performed by a group generally called the *configuration controller.*

The process management process activities are performed by the *software engineering process group (SEPG).*

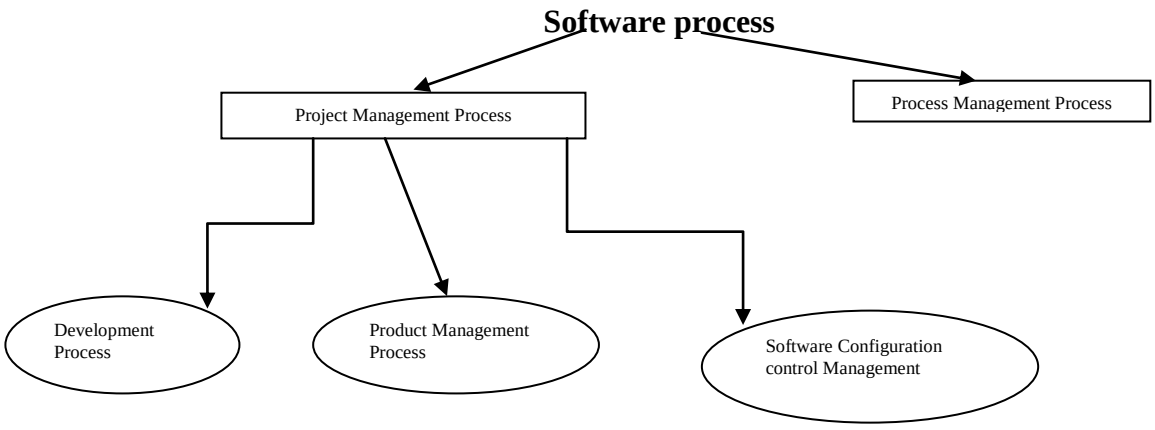


Fig. show Software processes.

8.3.2.ETVX Approach for Process Specification

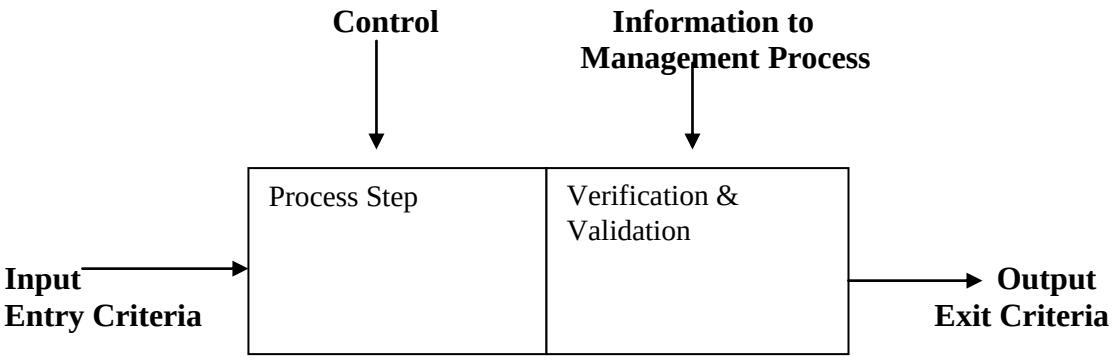
(Entry criteria, Task, Verification, and eXit criteria)

A process has a set of phases, each phase performing a well-defined task which leads a project towards satisfaction of its goals. To reduce the cost, a process should aim to detect defects in the phase in which they are introduced.

This requires that there be some verification at the end of each step, which in turn requires that there is a clearly defined output of a phase, which can be verified by some means.

Such outputs of a development process, which are not the final output, are frequently called the *work products*. In software, a work product can be the requirements document, design document, code, prototype, etc. This restriction that the output of each step be some work product that can be verified suggests that the process should have a small number of steps. Having too many steps results in too many work products or documents.

A process typically consists of a few steps, each satisfying a clear objective and producing a document which can be verified. How to perform the activity of the particular step or phase is generally addressed by *methodologies* for that activity.



A step in a development process

As a process typically contains a sequence of steps, the next issue to address is when a phase should be initiated and terminated. This is frequently done by specifying the entry criteria and exit criteria for a phase. The *entry criteria* of a phase specifies the conditions that the input to the phase should satisfy to initiate the activities of that phase. The *exit criteria* specifies the conditions that the work product of this phase should satisfy to terminate the activities of the phase.

The entry and exit criteria specify constraints of when to start and stop an activity. It should be clear that the entry criteria of a phase should be consistent with the exit criteria of the previous phase. In addition to the entry and exit criteria, the inputs and outputs of a step also need to be clearly specified.

8.3.3.Characteristics of Software Process

As a process may be used by many projects, it needs characteristics beyond satisfying the project goals.

- **Predictability:** Predictability of a process determines how accurately the outcome of following that process in a project can be predicted before the project is completed. Predictability can be considered a fundamental property of any process. In fact, if a process is not predictable, it is of limited use.

However, even this simple method implies that the process that will be used to develop project A will be same as the process used for project B, *and* that following the process the second time will produce similar results as the first time. That is, this assumes that the process is *predictable.* If it was not predictable, then there is no guarantee that doing a similar project using the process will incur a similar cost. The fundamental basis for quality prediction is that quality of the product is determined largely by the process used to develop it.

Testability and Maintainability

In the life of software the maintenance costs generally exceed the development costs. The goal of development should be to reduce the maintenance effort. That is, one of the important objectives of the development project should be to produce software that is easy to maintain. And the process used should ensure this maintainability.

Even in development, coding is frequently given a great degree of importance. We have seen that a process consists of phases, and a process generally includes requirements, design, coding, and testing phases. Of the development cost, an example distribution of effort with the different phases could be:

- Requirements 10%
- Design 10%
- Coding 30%
- Testing 50%

Both testing and maintenance depend heavily on the quality of design and code, and these costs can be considerably reduced if the software is designed and coded to make testing and maintenance easier.

- **Support Change**

Software changes for a variety of reasons. Besides changing an existing and working software, after all, the needs of the customer may change during the course of the project. And if the project is of any significant duration, considerable changes can be expected. Changes may occur simply because people may change their minds as they think more about possibilities and alternatives.

- **Early Defect Removal**

The notion that programming is the central activity during software development is largely due to programming being considered a difficult task and sometimes an "art." Another consequence of this kind of thinking is the belief that errors largely occur during programming, as it is the hardest activity in software development and offers many opportunities for committing errors. It is now clear that errors can occur at any stage during development.

- **Cost of correcting errors**

As one would expect, the greater the delay in detecting an error after it occurs, the more expensive it is to correct it. As the figure shows, an error that occurs during the requirements phase, if corrected during acceptance testing, can cost up to 100 times more than correcting the error during the requirements phase itself.

If there is an error in the requirements, then the design and the code will be affected by it. To correct the error after the coding is done would require both the design and the code to be changed, thereby increasing the cost of correction. Error detection and correction should be a

continuous process that is done throughout software development. A quality control (QC) activity is one whose main purpose is to identify and remove defects.

- **Process Improvement and Feedback**

A process is not a static entity. Improving the quality and reducing the cost of products are fundamental goals of any engineering discipline. In the context of software, as the productivity and quality are determined largely by the process, to satisfy the objectives of quality improvement and cost reduction, the software process must be improved. Process improvement is also an objective in a large project where feedback from the early parts of the project can be used to improve the execution of the rest of the project.

8.4. Software Development Process Models

In the software development process, the activities directly related to production of the software, for example, design, coding, and testing.

As the development process specifies the major development and quality control activities that need to be performed in the project, the development process really forms the core of the software process.

A) Waterfall Model

The simplest process model is the *waterfall model*, which states that the phases are organized in a linear order. In this model, a project begins with feasibility analysis. Upon successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins.

The design starts after the requirements analysis is complete, and coding begins after the design is complete. Once the programming is completed, the code is integrated and testing is done. Upon successful completion of testing, the system is installed. After this, the regular operation and maintenance of the system takes place.

The requirements analysis phase is mentioned as "analysis and planning." *Planning* is a critical activity in software development. A good plan is based on the requirements of the system and should be done before later phases begin.

Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and the beginning of the next. This is usually done by some verification and validation means that will ensure that the output of a phase is consistent with its input (which is the output of the previous phase), and that the output of the phase is consistent with the overall requirements of the system.

The consequence of the need for certification is that each phase must have some defined output that can be evaluated and certified. That is, when the activities of a phase are completed, there should be some product that is produced by that phase. The outputs of the earlier phases are often called *work products* and are usually in the form of documents like the requirements document or design document.

For the coding phase, the output is the code. Though the set of documents that should be produced in a project is dependent on how the process is implemented, the following documents generally form a reasonable set that should be produced in each project:

• Requirements document
• Project plan
• Design documents
• Test plan and test reports
• Final code
• Software manuals (e.g., user, installation, etc.)

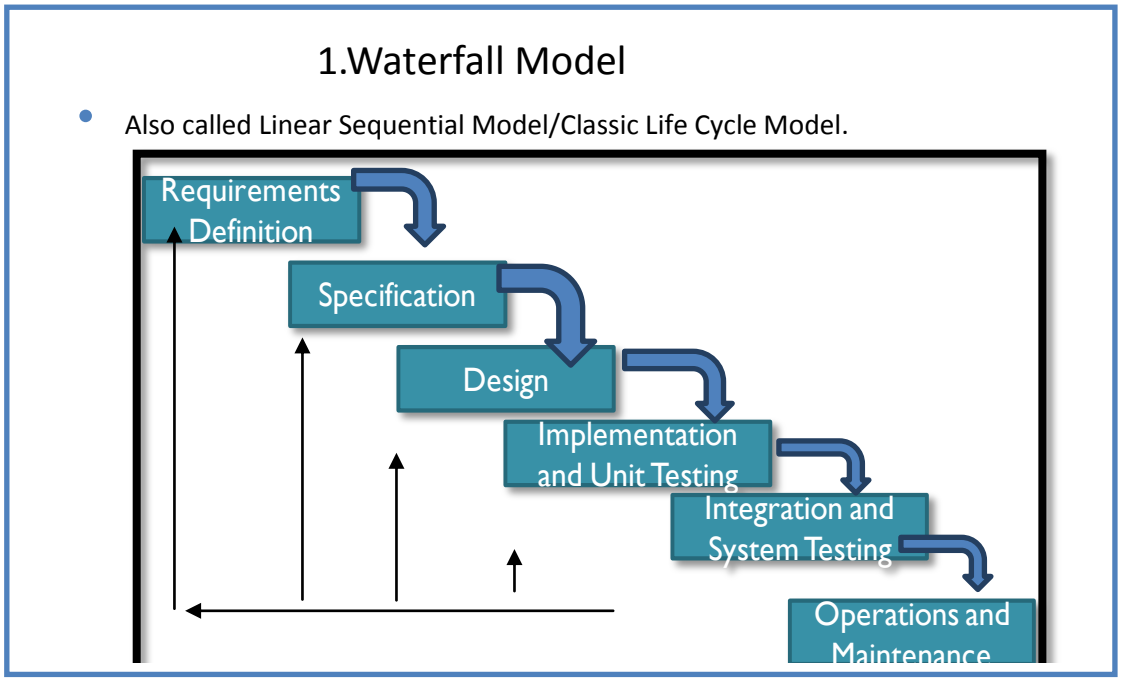


Fig. above show the waterfall model.

Advantages:

1. One of the main advantages of this model is its simplicity.
2. It is conceptually straightforward and divides the large task of building a software system into a series of cleanly divided phases, each phase dealing with a separate logical concern.
3. It is also easy to administer in a contractual setup as each phase is completed and its work product produced, some amount of money is given by the customer to the developing organization.

Limitations are:

1. It assumes that the requirements of a system can be frozen before the design begins. But for new systems, determining the requirements is difficult as the user does not even know the requirements.
2. Freezing the requirements usually requires choosing the hardware, may become obsolete over a period of time.
3. The entire software is delivered in one shot at the end. This entails heavy risks, as the user does not know until the very end what they are getting.

B) Prototyping

The goal of a prototyping-based development process is that instead of freezing the requirements before any design or coding can proceed. This prototype is developed based on the currently known requirements.

Development of the prototype obviously undergoes design, coding, and testing, but each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an actual feel of the system, because the interactions with the prototype can enable the client to, better understand the requirements of the desired system.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements. In both situations, the risks associated with the projects are being reduced through the use of prototyping.

After the prototype has been developed, the end users and clients are given an opportunity to use the prototype. Based on their experience, they provide feedback to the developers.

Based on the feedback, the prototype is modified to incorporate some of the suggested changes that can be done easily, and then the users and the clients are again allowed to use the system. Based on the feedback, the initial requirements are modified to produce the final requirements specification, which is then used to develop the production quality system.

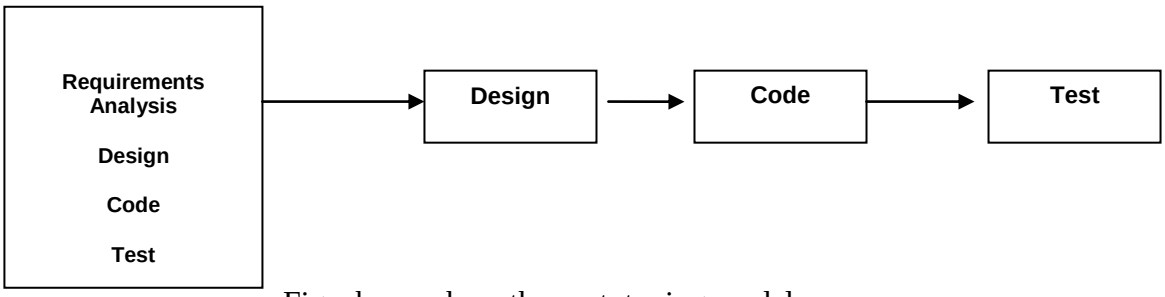


Fig. above show the prototyping model.

The focus of the development is to include those features that are not properly understood. And the development approach is with the focus on quick development rather than quality.

Advantages of Prototyping

1. Users are actively involved in the development
2. It provides a better system to users, as users have natural tendency to change their mind in specifying requirements and this method of developing systems supports this user tendency.
3. Errors can be detected much earlier as the system is made side by side.

Limitations of Prototyping

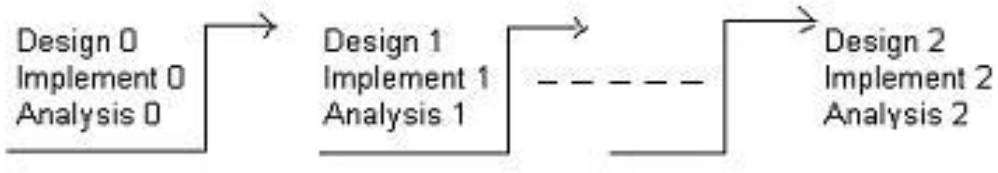
1. This Model Leads to 'implementing and then repairing' way of building systems.
2. This may increase the complexity of the system as scope of the system may expand beyond original plans.
3. Cost of implementing this method for larger or complex systems is more.

C) **Iterative Development:** The iterative development process model tries to combine the benefits of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented.

At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the waterfall model. The prototyping, the increments provide feedback to the client that is useful for determining the final requirements of the system.

D) The iterative enhancement model:

In the first step of this model, a simple initial implementation is done for a subset of the overall problem.



The iterative enhancement model

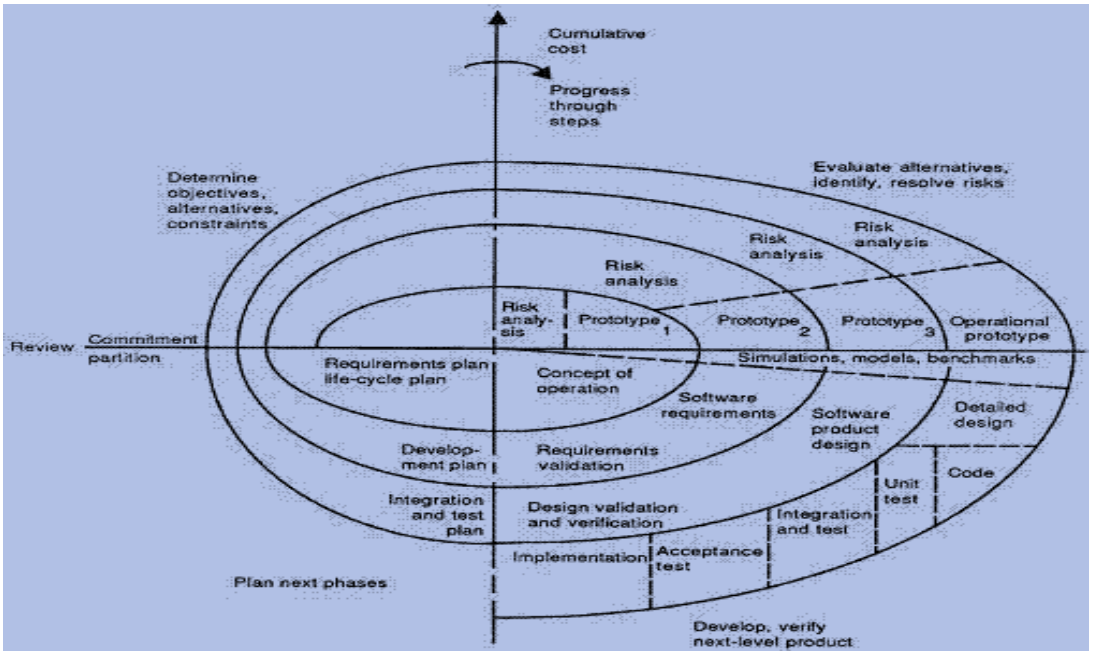
This subset is one that contains some of the key aspects of the problem that are easy to understand and implement and which form a useful and usable system. A project control list is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far along the project is at any given step from the final system.

Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called *the design phase, implementation phase,* and *analysis phase.*

Advantages of Iterative Enhancement Model

1) This approach results in better testing, the increments provides feedback to the client which is useful for determining the final requirements of the system.

The spiral model: The spiral model is another iterative model that has been proposed. As the name suggests, the activities in this model can be organized like a spiral that has many cycles.



The spiral model.

Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exist. The focus of evaluation in this step is based on the risk perception for the project. The next step is to develop strategies that resolve the uncertainties and risks.

Time boxing Model

In this model to speed up development, parallelism between the different iterations can be employed. That is, a new iteration commences before the system produced by the current iteration is released, and hence development of a new release happens in parallel with the development of the current release.

By starting an iteration before the previous iteration has completed, it is possible to reduce the average delivery time for iterations. In the time boxing model, the basic unit of development is a time box, which is of fixed duration. Since the duration is fixed, a key factor in selecting the requirements or features to be built in a time box is what can be fit into the time box.

Each time box is divided into a sequence of stages, like in the waterfall model. Each stage performs some clearly defined task for the iteration and produces a clearly defined output.

The model also requires that the duration of each stage, that is, the time it takes to complete the task of that stage, is approximately the same. Furthermore, the model requires that there be a dedicated team for each stage. This is quite different from other iterative models where the implicit assumption is that the same team performs all the different tasks of the project or the iteration.

Having time boxed iterations with stages of equal duration and having dedicated teams renders itself to pipelining of different iterations. (Pipelining is a concept from hardware in which different instructions are executed in parallel, with the execution of a new instruction starting once the first stage of the previous instruction is finished.)

As an example, consider a time box consisting of three stages: requirement specification, build, and deployment. The requirement stage is executed by its team of analysts and ends with a prioritized list of requirements to be built in in this iteration along with a high level design.

The build team develops the code for implementing the requirements, and performs the testing. The tested code is then handed over to the deployment team, which performs pre deployment tests, and then installs the system for production use. These three stages are such that they can be done in approximately equal time in an iteration.

8.4.1. Project Management Process

Proper management is an integral part of software development. A large software development project involves many people working for a long period of time. A development process typically partitions the problem of developing software into a set of phases.

To meet the cost, quality, and schedule objectives, resources have to be properly allocated to each activity for the project, and progress of different activities has to be monitored and corrective actions taken. All these activities are part of the project management process.

The project management process specifies all activities that need to be done by the project management to ensure that cost and quality objectives are met. The focus is on issues like

planning a project, estimating resource and schedule, and monitoring and controlling the project. The basic task is to plan the detailed implementation of the process for the particular project and then ensure that the plan is followed.

The activities in the management process for a project can be grouped broadly into three phases: planning, monitoring and control, and termination analysis. Project management begins with planning, which is perhaps the most critical project management activity.

The goal of this phase is to develop a *plan* for software development following which the objectives of the project can be met successfully and efficiently. A software plan is usually produced before the development activity begins and is updated as development proceeds and data about progress of the project becomes available.

The Inspection Process
The main goal of the inspection process is to detect defects in work products.

Software inspections were first proposed by Fagan. Earlier inspections were focused on code, but over the years its use has spread to other work products too. In other words, the inspection process is used throughout the development process. Software inspections are now a recognized industry best practice with considerable data to support that they help in improving quality and also improve productivity

An inspection is a review of a software work product by a group of peers following a clearly defined process. The basic goal of inspections is to improve the quality of the work product by finding defects.

Some of the characteristics of inspections are:

• An inspection is conducted by technical people for technical people
• It is a structured process with defined roles for the participants
• The focus is on identifying problems, not resolving them
• The review data is recorded and used for monitoring the effectiveness of the inspection process

As inspections are performed by a group of people, they can be applied to any work product, something that cannot be done with testing. Inspections are performed by a team of reviewers (or inspectors) including the author, with one of them being the *moderator*.

- **Planning**

The objective of the planning phase is to prepare for inspection. The author of the work product ensures that the work product is ready for inspection. The moderator checks that the entry criteria are satisfied by the work product. The entry criteria for different work products will be different. The package that needs to be distributed to the review team is prepared.

- **Overview and Preparation**

In this phase the package for review is given to the reviewers. The moderator may arrange an opening meeting, if needed, in which the author may provide a brief overview of the product and any special areas that need to be looked at carefully.

The objective and overview of the inspection process might also be given in this meeting. The meeting is optional and can be omitted. In that case, the moderator provides a copy of the group review package to the reviewers.

- **Group Review Meeting**

The basic purpose of the group review meeting is to come up with the final defect list, based on the initial list of defects reported by the reviewers and the new ones found during the discussion in the meeting. The entry criterion for this step is that the moderator is satisfied that all the reviewers are ready for the meeting.

The main outputs of this phase are the defect log and the defect summary report. The moderator first checks to see if all the reviewers are prepared. The moderator is in-charge of the meeting and has to make sure that the meeting stays focused on its basic purpose of defect identification and does not degenerate into a general brainstorming session or personal attacks on the author.

Software Configuration Management Process

Changes continuously take place in a software project changes due to the evolution of work products as the project proceeds, changes due to defects (bugs) being found and then fixed, and changes due to requirement changes. Configuration management (CM) or *software configuration management (SCM)* is the discipline for systematically controlling the changes that take place during development.

The IEEE defines SCM as "the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items"

- **CM Functionality**

To better understand CM, let us consider some of the functionality that a project requires from the CM process. *Give latest version of a program.* Suppose that a program has to be modified. Clearly, the modification has to be carried out in the latest copy of that program; otherwise, changes made earlier may be lost.

• *Undo a change or revert back to a specified version.* A change is made to a program, but later it becomes necessary to undo this change request.

- **CM Mechanisms**

The main purpose of CM is to provide various mechanisms that can support the functionality needed by a project to handle the types of scenarios discussed above that arise due to changes. The mechanisms commonly used to provide the necessary functionality include the following

• Configuration identification and base lining
• Version control or version management
• Access control

A Software configuration item (SCI), or *item* is a document that is explicitly placed under configuration control and that can be regarded as a basic unit for modification.

CM Process

The CM process defines the set of activities that need to be performed to control change. As with most activities in project management, the first stage in the CM process is planning. Then the process has to be executed, generally by using some tools.

Finally, as any CM plan requires some discipline from the project personnel in terms of storing items in proper locations, and making changes properly, monitoring the status of the configuration items and performing CM audits are therefore other activities in the CM process.

Planning for configuration management involves identifying the configuration items and specifying the procedures to be used for controlling and implementing changes to these configuration items. Identifying configuration items is a fundamental activity in any type of. The configuration controller (CC) is responsible for the implementation of the CM plan. Depending on the size of the system under development, his or her role may be a part-time or full-time job.

8.4.2. Process Management

A software process is not a static entity, it has to change to improve so that the products produced using the process are of higher quality and are less costly.

As we have seen, improving quality and productivity are fundamental goals of engineering. To achieve these goals the software process must continually be improved, as quality and productivity are determined to a great extent by the process. As stated earlier, improving the quality and productivity of the process is the main objective of the process management process.

It should be emphasized that process management is quite different from project management. In process management the focus is on improving the process which in turn improves the general quality and productivity for the products produced using the process. In project management the focus is on executing the current project and ensuring that the objectives of the project are met.

The time duration of interest for project management is typically the duration of the project, while process management works on a much larger time scale as each project is viewed as providing a data point for the process. To improve its software process, an organization needs to first understand the status of the current status and then develop a plan to improve the process.
The reason is that it takes time to internalize and truly follow any new methods that may be introduced. And only when the new methods are properly implemented will their effects be visible. Introducing too many new methods for the software process will make the task of implementing the change very hard. Software process capability describes the range of expected results that can be achieved by following the process.

The process capability of an organization determines what can be expected from the organization in terms of quality and productivity. The goal of process improvement is to improve the process capability. A *maturity level* is a well-defined evolutionary plateau towards achieving a mature software process. Based on the empirical evidence found by examining the processes of many organizations, the CMM suggests that there are five well-defined maturity levels for a software process. These are initial (level 1), repeatable, defined, managed, and optimizing.

The CMM framework says that as process improvement is best incorporated in small increments, processes go from their current levels to the next higher level when they are improved.

The *initial process* (level 1) is essentially an ad hoc process that has no formalized method for any activity. Basic project controls for ensuring that activities are being done properly, and that the project plan is being adhered to, are missing. In crisis the project plans and development processes are abandoned in favour of a code-and-test type of approach.

In a *repeatable process* (level 2), policies for managing a software project and procedures to implement those policies exist.

That is, project management is well developed in a process at this level. Some of the characteristics of a process at this level are: project commitments are realistic and based on past experience with similar projects, cost and schedule are tracked and problems resolved when they arise, formal configuration control mechanisms are in place, and software project standards are defined and followed. Essentially, results obtained by this process can be repeated as the project planning and tracking is formal.

At the *defined level* (level 3) the organization has standardized a software process, which is properly documented. A software process group exists in the organization that owns and manages the process. In the process each step is carefully defined with verifiable entry and exit criteria, methodologies for performing the step, and verification mechanisms for the output of the step.

At the *managed level* (level 4) quantitative goals exist for process and products. Data is collected from software processes, which is used to build models to characterize the process. Hence, measurement plays an important role in a process at this level. Due to the models built, the organization has a good insight of the process capability and its deficiencies. The results of using such a process can be predicted in quantitative terms.

At the *optimizing level* (level 5), the focus of the organization is on continuous process improvement. Data is collected and routinely analysed to identify areas that can be strengthened to improve quality or productivity.

8.5. The Unified Process

The Unified Process (UP) is a popular iterative and incremental software development process framework. The Unified Process (UP) has emerged as a popular software development process for building object-oriented systems.

The Unified Process is not simply a process, but rather an extensible framework which can and should be customized for specific organizations and/or projects. The *Rational Unified Process* is, similarly, a customizable framework.

The most well known and extensively documented refinement of the Unified Process is the *Rational Unified Process* (RUP). The other process arising from UP is the Enterprise Unified Process(EUP).

The Rational Unified Process (RUP**)** is a detailed refinement of the Unified Process. It is a software engineering process. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high quality software that meets the needs of its end users within a predictable schedule and budget.

RUP is based on a set of building blocks or content elements describing what is to be produced, the necessary skills required and the step-by-step explanation describing how specific development goals are to be achieved. The main building blocks, or content elements, are the following:

- Roles (who) – a *role* defines a set of related skills, competencies and responsibilities.
- Artifacts/Work Products (what) – a *work product* represents something resulting from a task, including all the documents and models produced while working through the process.
- Activities/Tasks (how) – a t*ask* describes a unit of work assigned to a Role that provides a meaningful result.

The Rational Unified Process (and in fact the Unified Process) is:

- Iterative and incremental
- Use case driven
- Architecture-centric
- Risk-driven

Iterative and Incremental Development

1. Incremental Development

In incremental development, we break up the work into smaller pieces and schedule them to be developed over time and integrated as they are completed.

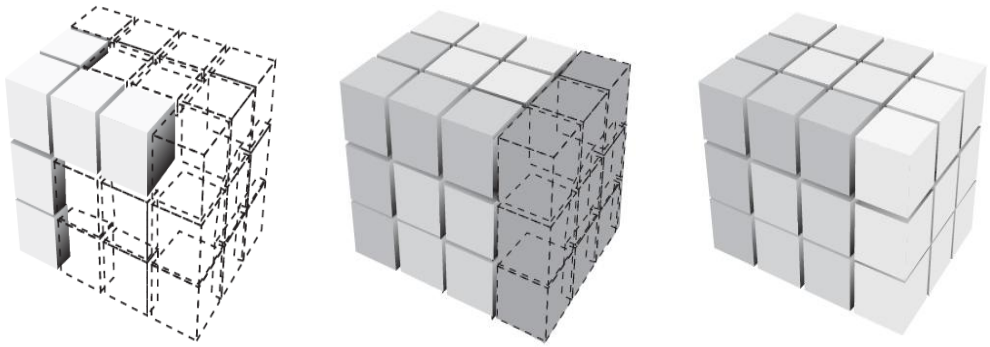


Fig Incremental development stage 1, 2, and 3

In Incremental approach, we concentrate on those aspects that are currently the most important and postpone until later those that are less critical. Eventually, every aspect is handled but in order of importance. We start off by constructing an artifact that solves only a small part of what we are trying to achieve. Then we consider further aspects of the problem and add the resulting new pieces to the existing artefact.

2. Iterative Development

In iterative development, we set aside time to improve what we have. Requirements and user interfaces are the most notorious places where we historically have had to revise our work, but they are not the only ones. Technology, architecture, and algorithms are also likely to need inspection and revision.

Fig Iterative development of Mona Lisa

Leonardo draws a sketch of what he intends to do and goes to the patron, asking, "How's this going to work for you?" The patron says, "No, no, no. She can't be looking right, she has to be looking left!" Fortunately, Leonardo has not done too much work yet, so this is easy to change. Leonardo goes away, reverses the picture and does some color and detail. He goes back to the patron and says "By cost, I'm about one-third done. What do you think now?" The patron says, "No, you can't make her head look that big! Make it look more balanced with her body size." Leonardo goes away and finishes the painting. The patron says, "Really, I'd rather have her eyes bigger, but okay, let's call it done."

Iterative development is a rework scheduling strategy in which time is set aside to revise and improve parts of the system. It does not presuppose incremental development, but works very well with it. A typical difference is that the output from an increment is not necessarily subject to further refinement, and its testing or user feedback is not used as input for revising the plans or specifications of the successive increments. On the contrary, the output of an iteration is examined for modification, and especially for revising the targets of the successive iterations.

In iterative approach, we produce the first version of the artifact and then we revise it and produce the second version, and so on. Our intent is that each version will be closer to our target than its predecessor, and finally a version that is satisfactory.

The unified process groups iterations into phases: inception, elaboration, construction, and transition.

3. Iterative and Incremental Approach
In practice, iteration and incrementation are used in conjunction with one another. An artifact is constructed piece by piece (incrementation), and each piece/increment goes through multiple versions (iteration).

An iteration is a "mini-project". The result of an iteration is an increment.

Characteristics of RUP
The RUP has the following characteristics:

1. Iterative and Incremental
The Unified Process is an iterative and incremental development process. The Elaboration, Construction and Transition phases are divided into a series of time-boxed iterations. The Inception phase may also be divided into iterations for a large project. Each iteration results in an *increment*, which is a release of the system that contains added or improved functionality compared with the previous release.

Although most iterations will include work in most of the process disciplines (*e.g.* Requirements, Design, Implementation, Testing) the relative effort and emphasis will change over the course of the project.

2. Use Case Driven
In the Unified Process, Use Cases are used to capture the functional requirements and to define the contents of the iterations. Each iteration takes a set of Use Cases or scenarios from requirements all the way through implementation, test and deployment. The process employs use cases to drive the development process from inception to deployment.

3. Architecture Centric
The architecture of an information system includes the various component modules and how they fit together. The architecture of an information system can be described as object oriented,

pipes and filters (Unix or Linux components), or client-server. The architecture for iterative and incremental should be extendable continually to incorporate next increment. This is called robustness.

The Unified Process insists that architecture sit at the heart of the project team's efforts to shape the system. Since no single model is sufficient to cover all aspects of a system, the Unified Process supports multiple architectural models and views.

4. Risk Focused

The Unified Process requires the project team to focus on addressing the most critical risks early in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first.

There are different kinds of risks to projects. Some of them include:

- Technical risks: does the development team have all the necessary skill? Are all the technology required available? Etc.
- Risk of not getting the requirements right
- Risk of software architecture problem

Phases of the Rational Unified Process

The Unified Process divides the project into four phases:

- Inception
- Elaboration
- Construction
- Transition

Through all phases, activities known as *workflows* specify the details of the work that needs to be done. A workflow is a sequence of activities that produces a result which can be measured. There are nine core process workflows in the Rational Unified Process, which represent a partitioning of all workers and activities into logical groupings. The core process workflows are divided into six core *engineering workflows*:

1. Business modeling workflow – understanding the business
2. Requirements workflow – requirement gathering
3. Analysis & Design workflow – behavioral and structural modeling
4. Implementation workflow – building the system
5. Test workflow – quality assurance
6. Deployment workflow – environmental modeling

And there are three core *supporting workflows*:

1. Project Management workflow
2. Configuration and Change Management workflow
3. Environment workflow

I. Inception Phase

Inception is the first phase of the UP lifecycle. The purpose of Inception is phase is to determine whether it is worthwhile to develop the system.

Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it is usually an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process.

II. Elaboration Phase

During the Elaboration phase, the project team captures the majority of the system requirements. However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture.

The architecture is validated primarily through the implementation of an *Executable Architecture Baseline*. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, time-boxed iterations. By the end of the Elaboration phase the system architecture must have stabilized and the executable architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost.

III. Construction Phase

Construction is the largest phase in the project. In this phase, the remainder of the system is built on the foundation laid in Elaboration. System features are implemented in a series of short, time-boxed iterations. Each iteration results in an executable release of the software.

The *Initial Operational Capability Milestone* marks the end of the Construction phase.

Objectives of the Construction Phase

Construction is really about cost-efficient development of a complete product (an operational version of your system that can be deployed in the user community. This translates into the following objectives:

- *Minimize development costs and achieve some degree of parallelism in the work of the development teams.* Optimize resources and avoid unnecessary scrap and rework. Even smaller projects generally have components that can be developed independently of one another, allowing for natural parallelism between developers or teams of developers (resources permitting).
- *Iteratively develop a complete product that is ready to transition to its user community.* Develop the first operational version of the system by describing the remaining use cases and other requirements, filling in the design details, completing the implementation, and testing the software. Determine whether the software, the sites, and the users are all ready for the application to be deployed.

IV. Transition Phase

The Transition phase is the fourth and last phase of the RUP life cycle. In Transition phase, the system is deployed to the target users. Feedback received from an initial release may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training. The *Product Release Milestone* marks the end of the Transition phase.

The Transition phase has the following objectives:

- *Beta test to validate that user expectations are met:* this typically requires some tuning activities such as bug fixing and making enhancements for performance and usability.
- *Train users and maintainers to achieve user self-reliability*: these activities ensure that the adopting organization is qualified to use the system and has moved any necessary data from earlier systems or taken any other measures required to operate the new system successfully.
- *Prepare deployment site and convert operational databases:* to get the new system up and running, you may have to purchase new hardware, add space for new hardware, or convert data from earlier systems to the new system.
- *Prepare for launch-packaging, production, and marketing rollout; release to distribution and sales forces; field personnel training.* Especially when developing a commercial product, these activities should take place to ensure a successful launch.

- *Achieve stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision.*