

Object-Oriented Analysis And Design

Pre- requisite:

- Software Engineering
- Object Oriented Programming(OOP)

The Concept Of Object-Orientation

- Object-orientation is what's referred to as a programming paradigm. It's not a language itself but a set of concepts that is supported by many languages.
- If you aren't familiar with the concepts of object-orientation, you may take a look at [The Story of Object-Oriented Programming](#).
- If everything we do in these languages is object-oriented, it means, we are oriented or focused around objects.
- Now in an object-oriented language, this one large program will instead be split apart into self contained objects, almost like having several mini-programs, each object representing a different part of the application.

The Concept Of Object-Orientation

- And each object contains its own data and its own logic, and they communicate between themselves.
- These objects aren't random. They represent the way you talk and think about the problem you are trying to solve in your real life.
- They represent things like employees, images, bank accounts, spaceships, asteroids, video segment, audio files, or whatever exists in your program.

OBJECT

IDENTITY

STATE

BEHAVIOUR

OOAD

OOA- Identify the objects.

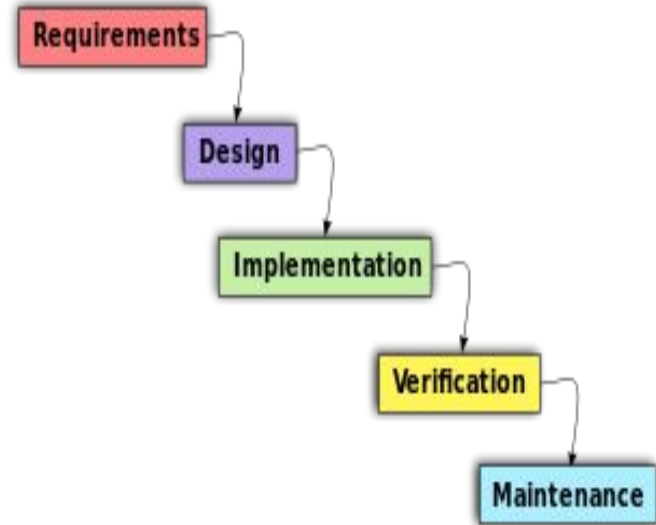
OOD- Relationships of identified objects and model them.

Object-Oriented Analysis And Design (OOAD)

It's a structured method for analyzing, designing a system by applying the object-orientated concepts, and develop a set of graphical system models during the development life cycle of the software.

OOAD In The SDLC

- The software life cycle is typically divided up into stages going from abstract descriptions of the problem to designs then to code and testing and finally to deployment.
- Image for post
- The earliest stages of this process are analysis (requirements) and design.
- The distinction between analysis and design is often described as “what Vs how”.



Object-Oriented Analysis And Design (OOAD)

- In analysis developers work with users and domain experts to define what the system is supposed to do. Implementation details are supposed to be mostly or totally ignored at this phase.
- The goal of the analysis phase is to create a model of the system regardless of constraints such as appropriate technology. This is typically done via use cases and abstract definition of the most important objects using conceptual model.
- The design phase refines the analysis model and applies the needed technology and other implementation constraints.
- It focuses on describing the objects, their attributes, behavior, and interactions. The design model should have all the details required so that programmers can implement the design in code.
- They're best conducted in an iterative and incremental software methodologies. So, the activities of OOAD and the developed models aren't done once, we will revisit and refine these steps continually.

Object-Oriented Analysis

- In the object-oriented analysis, we ...
- **Elicit requirements:** Define what does the software need to do, and what's the problem the software trying to solve.
- **Specify requirements:** Describe the requirements, usually, using use cases (and scenarios) or user stories.
- **Conceptual model:** Identify the important objects, refine them, and define their relationships and behavior and draw them in a simple diagram.
- We're not going to cover the first two activities, just the last one. These are already explained in detail in Requirements Engineering.

Object-Oriented Design

- The analysis phase identifies the objects, their relationship, and behavior using the conceptual model (an abstract definition for the objects).
- While in design phase, we describe these objects (by creating class diagram from conceptual diagram — usually mapping conceptual model to class diagram), their attributes, behavior, and interactions.
- In addition to applying the software design principles and patterns.
- The input for object-oriented design is provided by the output of object-oriented analysis. But, analysis and design may occur in parallel, and the results of one activity can be used by the other.

In the object-oriented design, we ...

- Describe the classes and their relationships using class diagram.
- Describe the interaction between the objects using sequence diagram.
- Apply software design principles and design patterns.
- A class diagram gives a visual representation of the classes you need. And here is where you get to be really specific about object-oriented principles like inheritance and polymorphism.
- Describing the interactions between those objects lets you better understand the responsibilities of the different objects, the behaviors they need to have.

Complexity

- The larger the number of components and relationships between them, higher will be the complexity of the overall system.

Grady Booch

The function of good software is to make the complex appear to be simple.

Simple-

- Largely forgettable applications
- Very limited purpose
- Afford to throw them away

The Structure of Complex systems:

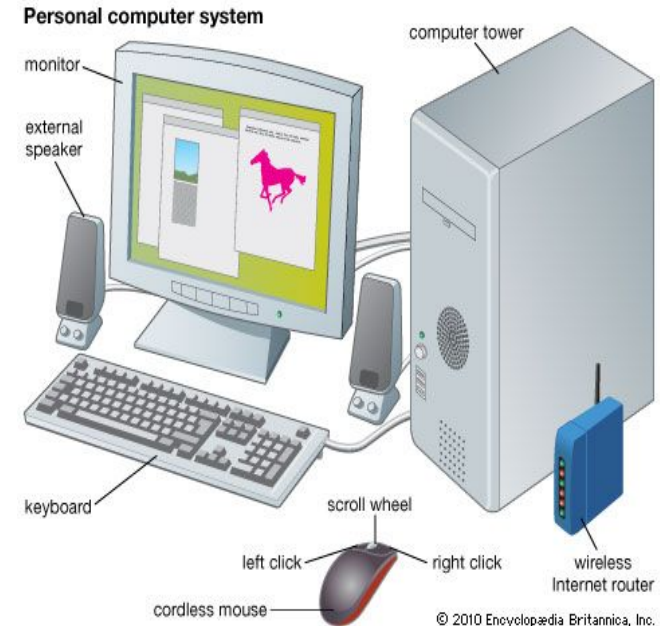
Examples of Complex Systems:

- The Structure of a Personal Computer
- The Structure of Plants and Animals
- The Structure of Matter
- The Structure of Social Institutions

The Structure of Complex systems:

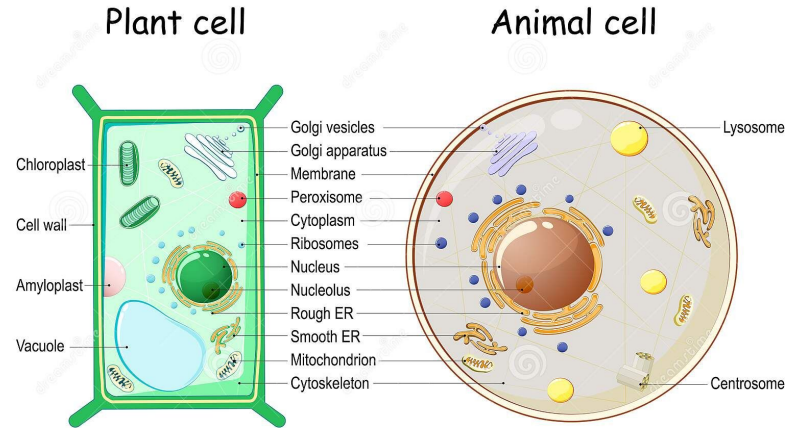
The Structure of a Personal Computer A personal computer is a device of

- moderate complexity. Most of them are composed of the same major elements: a central processing unit (CPU), a monitor, a keyboard, and some sort of secondary storage device,
- usually either a floppy disk or a hard disk drive. We may take any one of these parts and further decompose.



The Structure of Complex systems:

The Structure of Plants and Animals In botany, scientists seek to understand the similarities and differences among plants through a study of their morphology, that is, their form and structure. Plants are complex multicellular organisms, and from the cooperative activity of various plant organ systems arise such complex behaviors as photosynthesis and transpiration. Plants consist of three major structures (**roots, stems, and leaves**), and each of these has its own structure. For example, roots encompass branch roots, root hairs, the root apex, and the root cap. Each of these structures is further composed of a collection of cells, and inside each cell we find yet another level of complexity, encompassing such elements as chloroplasts, a nucleus, and so on.



The Structure of Complex systems:

The Structure of Matter The study of fields as diverse as astronomy and nuclear physics

provides us with many other examples of incredibly complex systems. Spanning these two

disciplines, we find yet another structural hierarchy. Astronomers study galaxies that are

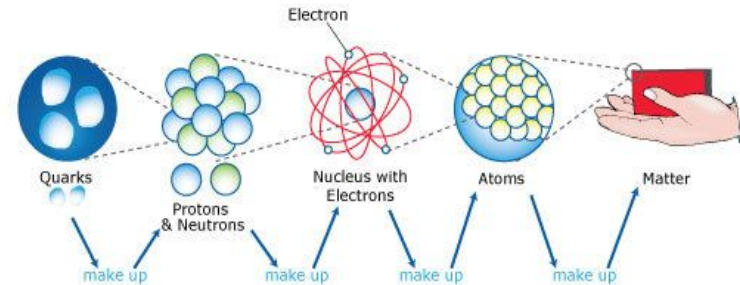
arranged in clusters, and stars, planets, and various debris are the constituents of galaxies.

Likewise, nuclear physicists are concerned with a structural hierarchy, but one on an

entirely different scale. **Atoms are made up of electrons, protons, and neutrons; electrons**

appear to be elementary particles, but protons, neutrons, and other particles are formed

from more basic components called quarks.



The Structure of Complex systems:

The Structure of Social Institutions As a final example of complex systems, we turn to the structure of social institutions. Groups of people join together to accomplish tasks that cannot be done by individuals. Some organizations are transitory, and some endure beyond many lifetimes. As organizations grow larger, we see a distinct hierarchy emerge. Multinational corporations contain companies, which in turn are made up of divisions, which in turn contain branches, which in turn encompass local offices, and so on. If the organization endures, the boundaries among these parts may change, and over time, a new, more stable hierarchy may emerge. The relationships among the various parts of a large organization are just like those found among the components of a computer, or a plant, or even a galaxy. Specifically, the degree of interaction among employees within an individual office is greater than that between employees of different offices. A mail clerk usually does not interact with the chief executive officer of a company but does interact frequently with other people in the mail room. Here too, these different levels are unified by common mechanisms. The clerk and the executive are both paid by the same financial organization, and both share common facilities, such as the company's telephone system, to accomplish their tasks.



The Inherent Complexity of Software:

● Why Software Is Inherently Complex

As Brooks suggests, "The complexity of software is an essential property, not an accidental one" . We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the developmental process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems. The Complexity of the Problem Domain The problems we try to solve in software often involve elements of inescapable complexity, in which we find a myriad of competing perhaps even contradictory, requirements. Consider the requirements for the electronic system of a multi-engine aircraft, a cellular phone switching system, or an autonomous robot. The raw functionality of such systems is difficult enough to comprehend, but now add all of the (often implicit) nonfunctional requirements such as usability, performance, cost, survivability, and reliability. This unrestrained external complexity is what causes the arbitrary complexity about which Brooks writes.

Object oriented methodologies

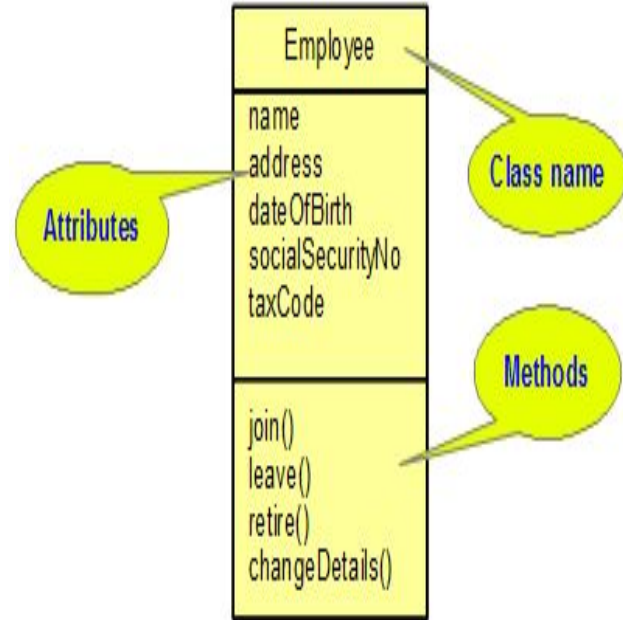
Object-oriented development-

Object-oriented design is part of object-oriented development where an object-oriented strategy is used throughout the development process:

- **Object-oriented analysis.** It is concerned with developing an object-oriented model of the application domain. The objects in that model reflect the entities and operations associated with the problem to be solved.
- **Object-oriented design.** It is concerned with developing an object-oriented model of a software system to implement the identified requirements.
- **Object-oriented programming.** It is concerned with implementing a software design using an object-oriented programming language, such as Java.

Object oriented methodologies

- In the UML, an object class is represented as a named rectangle with two sections. The object attributes are listed in the top section. The operations that are associated with the object are set out in the bottom section. Figure illustrates this notation using an object class that models an employee in an organisation. The UML uses the term operation to mean the specification of an action; the term method is used to refer to the implementation of an operation.



Object oriented methodologies

- The class Employee defines a number of attributes that hold information about employees including their name and address, social security number, tax code, and so on.

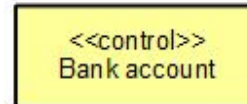
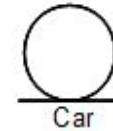
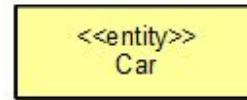
<u>Julien Absolon</u>
Paris
01.12.1966
765453012
8364652524

<u>Steve Davis</u>
London
23.11.1983
365454411
8467654352

Object oriented methodologies

UML supports stereotypes, which are an inbuilt mechanism for logically extending or altering the meaning, display, characteristics or syntax of a basic UML model element such as class and object. Extending classes in UML the next stereotypes are often used:

- <<boundary>> stereotype. A boundary class represents a user interface.(service collaborator)
- <<entity>> stereotype. Entity classes represent manipulated units of information.(model)
- <<control>> (controller)mediate between boundary and entity.glue between boundary and entity elements.
- UML representation of stereotypes applied to classes Car, Login and Bank account are shown in Figure together with an icon for their default representation.



Object oriented methodologies

Four rules apply to their communication-

1. Actor can only talk to boundary objects.
2. Boundary objects can only talk to controller and actors.
3. Entity objects can only talk to controllers.
4. Controllers can talk to boundary objects and entity objects, and to other controllers, but not to actors

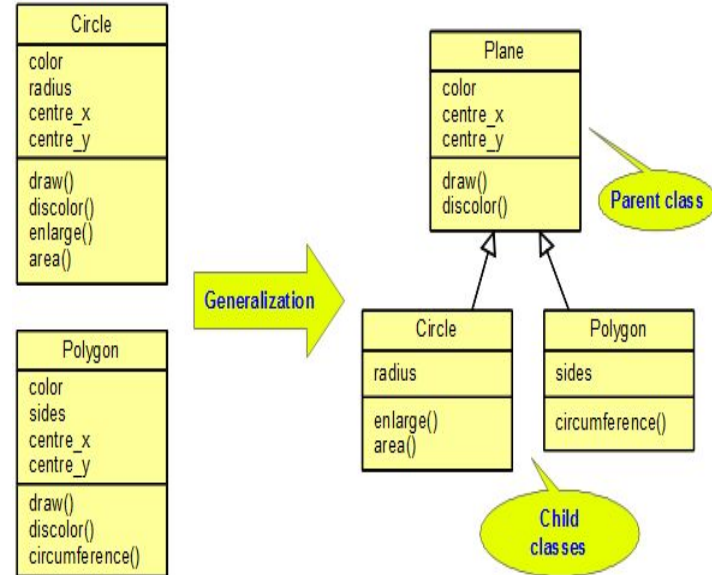
● Communication allowed

	Entity	Boundary	Control
Entity	x		x
Boundary			x
Control	x	x	x

Object oriented methodologies

Generalisation

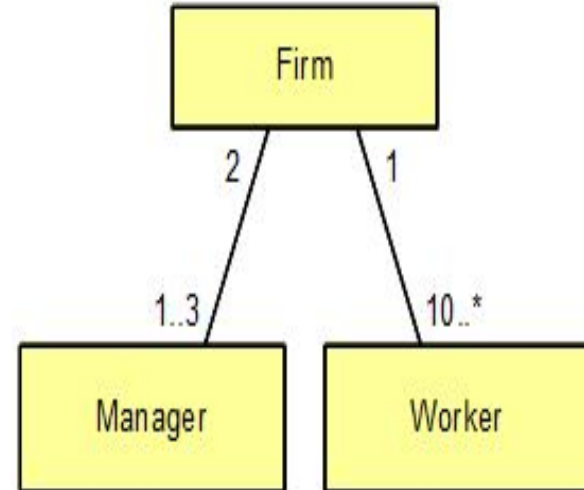
- Object classes can be arranged in a generalisation or inheritance hierarchy that shows the relationship between general and more specific object classes. The more specific object class is completely consistent with its parent class but includes further information. In the UML, an arrow that points from a class entity to its parent class indicates generalisation. In object-oriented programming languages, generalisation is implemented using inheritance. The child class inherits attributes and operations from the parent class.



Object oriented methodologies

Association

Objects that are members of an object class participate in relationships with other objects. These relationships may be modelled by describing the associations between the object classes. In the UML, associations are denoted by a line between the object classes that may optionally be annotated with information about the association. Association is a very general relationship and is often used in the UML to indicate that either an attribute of an object is an associated object or the implementation of an object method relies on the associated object.



Object oriented methodologies

The general process for an object-oriented design has a number of stages:

- Understand and define the context and the modes of use of the system.
- Design the system architecture.
- Identify the principal objects in the system.
- Develop design models.
- Specify object interfaces.

Object oriented methodologies

System context and models of use

The first stage in any software design process is to develop an understanding of the relationships between the software and its external environment. This helps to provide the required system functionality and structure the system to communicate with its environment. The system context and the model of system use represent two complementary models of the relationships between a system and its environment:

- The system context is a static model that describes the other systems in that environment.
- The model of the system use is a dynamic model that describes how the system actually interacts with its environment.

The context model of a system may be represented using associations where a simple block diagram of the overall system architecture is produced. The interactions of a system with its environment can be modelled by use-case models where each use-case represents an interaction with the system. Use-cases can be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do.

Object oriented methodologies

Architectural design

- The system architecture can be represented by a number of architectural views. These views capture the major structural design decisions. Architectural views are the abstractions or simplifications of the entire design, in which important characteristics are made more visible by leaving details aside.
- Design activities are centred around the notion of architecture. **Once the interactions between the software system and its environment have been defined by use case models, this information can be used as a basis for designing the system architecture.** During architectural design the architecturally significant use cases are selected by performing use case analysis on each one and the system is decomposed to a collection of interacting components.
- In UML the architecture of system can be represented by a component diagram. It shows a collection of model elements, such as components, and implementation subsystems, and their relationships, connected as a graph to each other. Component diagrams can be organized into, and owned by implementation sub-systems, which show only what is relevant within a particular implementation subsystem.

Object oriented methodologies

Object and class identification

The analysis process starts with the identification of a set of conceptual classes that are the categories of things which are of significance in the system domain. When identifying appropriate classes, a good understanding of the system domain is important.

Possible classes for inclusion may emerge during the course of the requirements specification process. There are a number of other techniques can be used to help identify appropriate classes from a requirements document. Because a class is conceptually a set of objects of the same kind and objects represent things in the system domain, nouns and noun phrases in the requirements document can be used to identifying possible classes. There have been various proposals to identify object classes:

Grammatical analysis of a natural language description of a system. **Objects and attributes are nouns; operations or services are verbs.** Using tangible entities, things, in the application domain such as car, roles such as worker, events such as request, interactions such as meetings, locations such as offices and so on. Using a behavioural approach where the designer first understands the overall behaviour of the system. The various behaviours are assigned to different parts of the system and an understanding is derived of who initiates and participates in these behaviours. Participants who play significant roles are recognised as objects. Using a scenario-based analysis where various scenarios of system use are identified and analysed in turn. As each scenario is analysed, the team responsible for the analysis must identify the required objects, attributes and operations. These approaches help to get started with object identification. Further information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information may be collected from requirements documents, from discussions with users and from an analysis of existing systems.

Object oriented methodologies

Design models-

Design models show the objects or object classes in a system and the relationships between these entities. The design of system essentially consists of design models. Design models can be considered as a transformation of system requirements to a specification of how to implement the system. Design models have to be abstract enough to hide the unnecessary details. However, they also have to include enough detail for programmers to make implementation decisions.

In general, models are developed at different levels of detail. An important step in the design process, therefore, is to decide which design models has to be developed and the level of detail of these models. This usually depends on the type of system that is being developed. There are two types of models that should be produced to describe an object-oriented design:

- **Static models.** They describe the static structure of the system using object classes and their relationships. Important relationships are the association, generalisation, dependency, aggregation and composition relationships.
- **Dynamic models.** They describe the dynamic structure of the system and show the interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these object interactions.

The UML provides for 9 different static and dynamic models that may be produced to document a design. For examples, some models:

- **Sub-system models.** This model shows logical groupings of objects into coherent sub-systems. These are represented using a form of class diagram where each sub-system is shown as a package. Sub-system models are static models.

Object interface specification

- Software components communicate each other using their interfaces. Object interface design is concerned with specifying the detail of the interface to an object or to a group of objects.
- The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. Equally, a group of objects may all be accessed through a single interface.
- The interface representation should be hidden and object operations are provided to access and update the data. If the representation is hidden, it can be changed without affecting the objects that use these attributes. This leads to a design that is inherently more maintainable.
- Interfaces can be specified in the UML using the same notation as in a class diagram. However, there is no attribute section, and the UML stereotype <<interface>> should be included in the name part.

Characteristics of Objects

The first item in this list is too restrictive. For example, you can think of your bank account as an object, but it is not made of material. (Although you and the bank may use paper and other material in keeping track of your account, your account exists independently of this material.) Although it is not material, your account has properties (a balance, an interest rate, an owner) and you can do things to it (deposit money, cancel it) and it can do things (charge for transactions, accumulate interest).

The last three items on the list seem clear enough. In fact, they have names:

- An object has identity (each object is a distinct individual).
- An object has state (it has various properties, which might change).
- An object has behavior (it can do things and can have things done to it).

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 modelled 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.

Fundamental Concepts of Object Orientetion

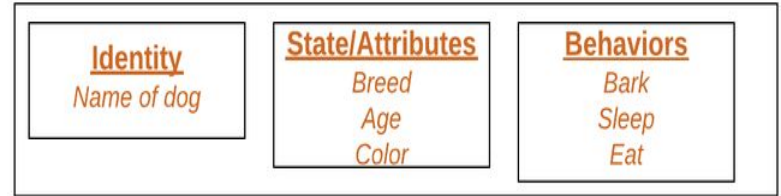
- Class
- Objects
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Message Passing

Class:

- A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.
- For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties.

Object:

- It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.
- For example “Dog” is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

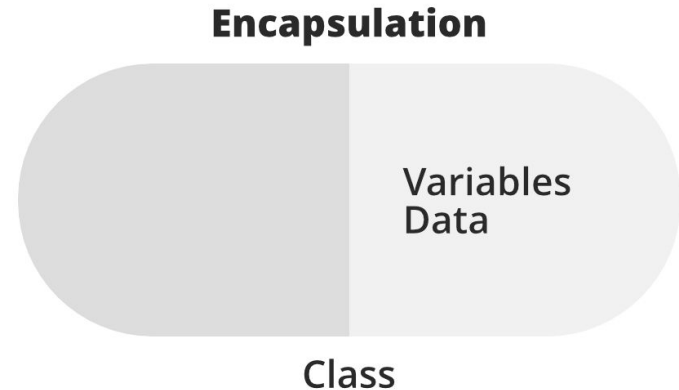


Data Abstraction:

- Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

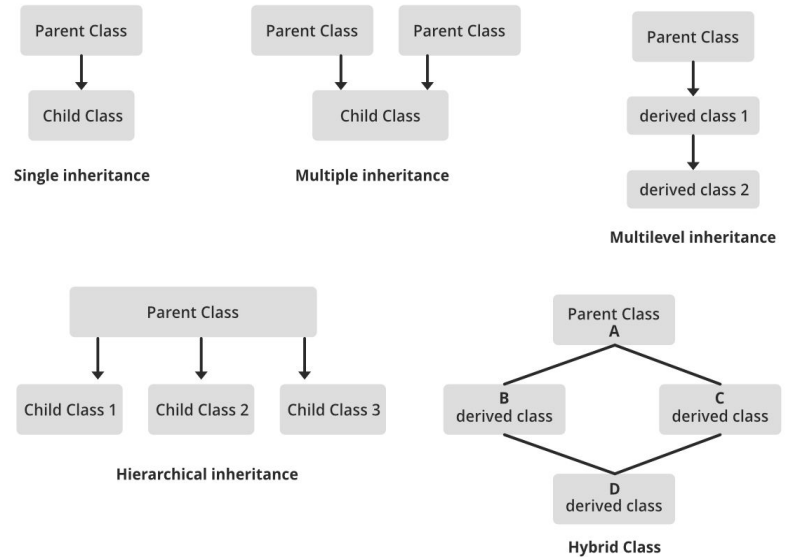
Encapsulation:

- Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance.



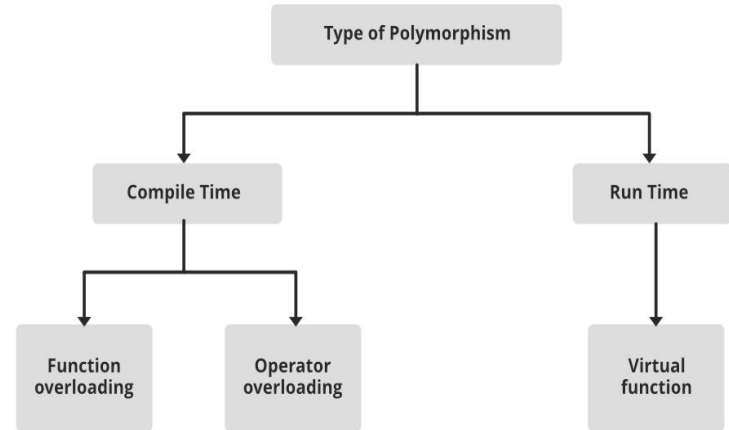
Inheritance:

- Inheritance is an important pillar of OO(Object-Oriented). The capability of a class to derive properties and characteristics from another class is called Inheritance. When we write a class, we inherit properties from other classes. So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.



Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.



Message Passing:

It is a form of communication used in object-oriented programming as well as parallel programming. Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

Unified Modeling Language (UML) | An Introduction

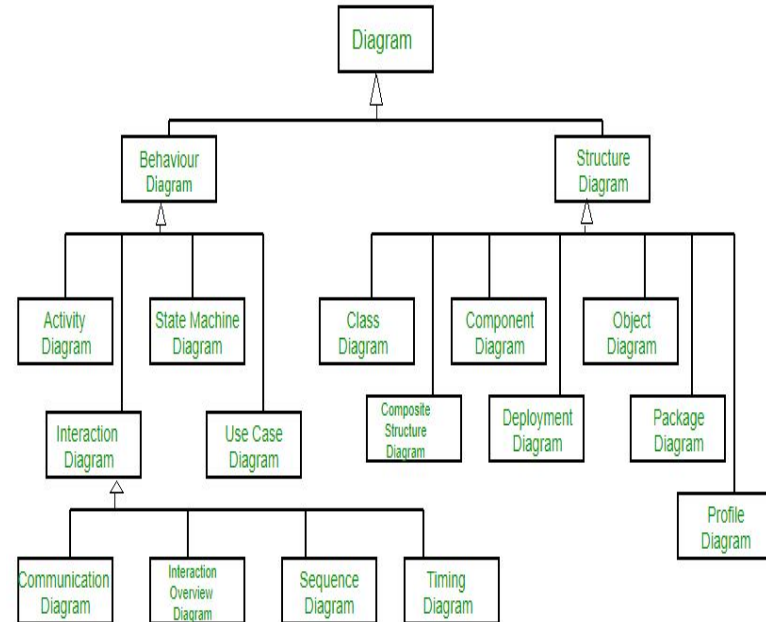
- Unified Modeling Language (UML) is a general purpose modelling language. The main aim of UML is to define a standard way to visualize the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.
- UML is not a programming language, it is rather a visual language. We use UML diagrams to portray the behavior and structure of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. **The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005.** UML has been revised over the years and is reviewed periodically.

Do we really need UML?

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.
- UML is linked with object oriented design and analysis

Diagrams in UML can be broadly classified as:

- Structural Diagrams – Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
- Behavior Diagrams – Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

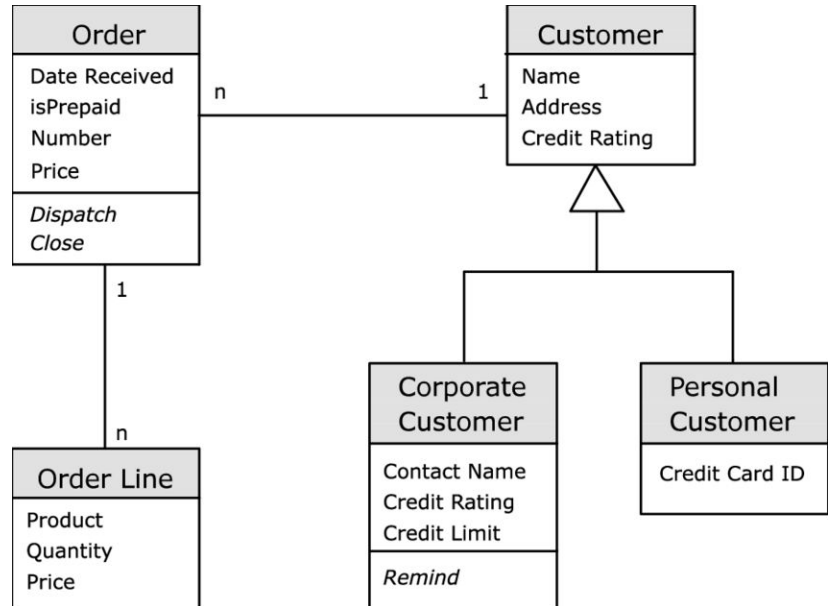


Object Oriented Concepts Used in UML –

- Class – A class defines the blue print i.e. structure and functions of an object.
- Objects – Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.
- Inheritance – Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
- Abstraction – Mechanism by which implementation details are hidden from user.
- Encapsulation – Binding data together and protecting it from the outer world is referred to as encapsulation.
- Polymorphism – Mechanism by which functions or entities are able to exist in different forms.

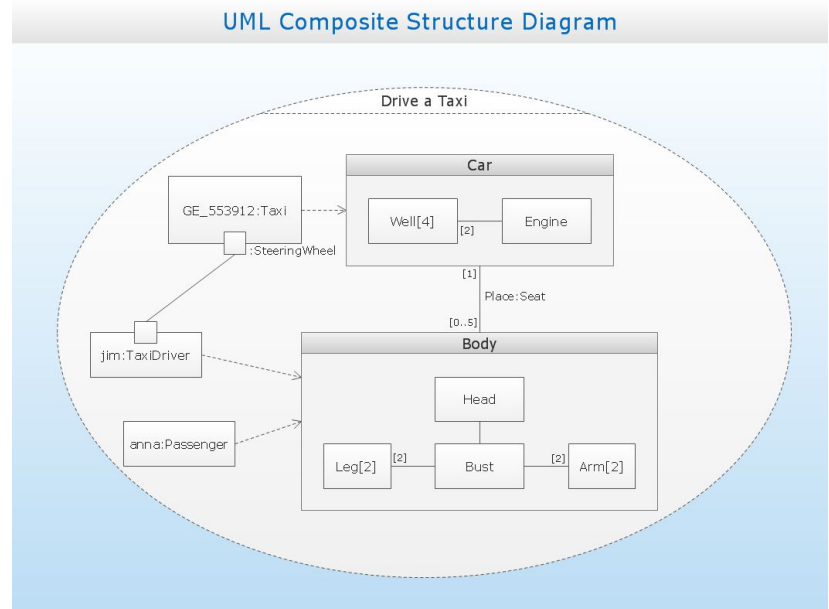
Structural UML Diagrams –

- Class Diagram – The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes, their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.



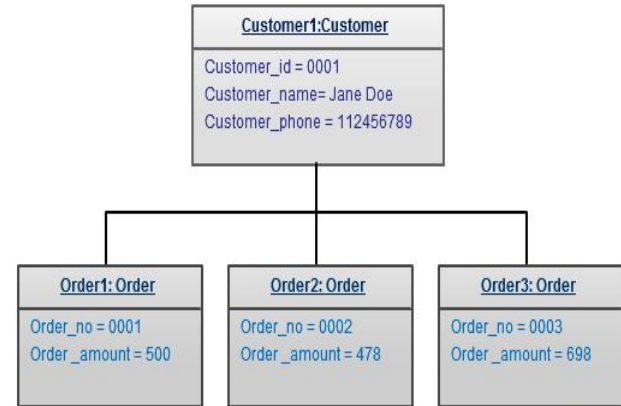
Structural UML Diagrams –

- **Composite Structure Diagram** – We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.



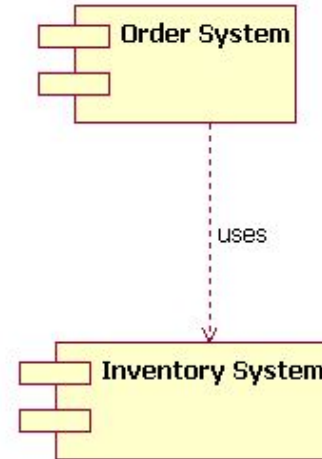
Structural UML Diagrams –

- Object Diagram – An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.



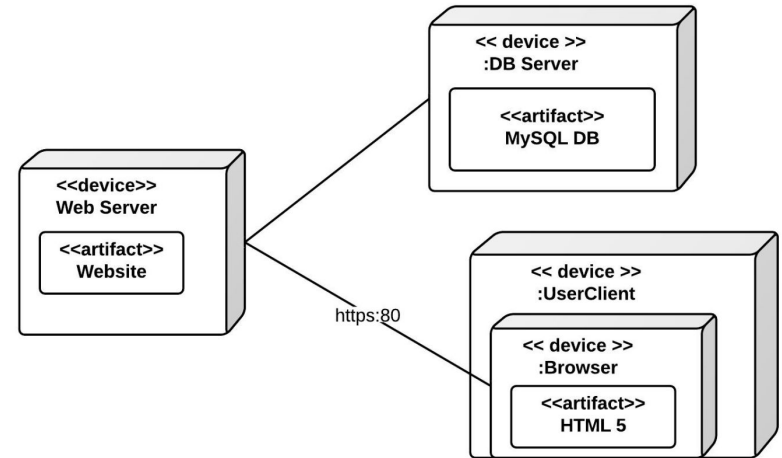
Structural UML Diagrams –

- **Component Diagram** – Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.



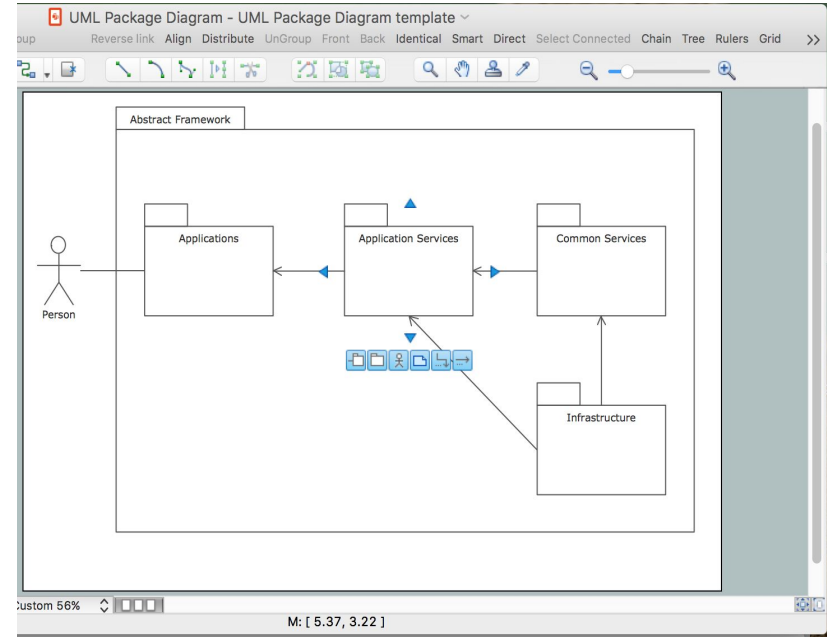
Structural UML Diagrams –

- Deployment Diagram – Deployment Diagrams are used to represent system hardware and its software. It tells us what hardware components exist and what software components run on them. We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.



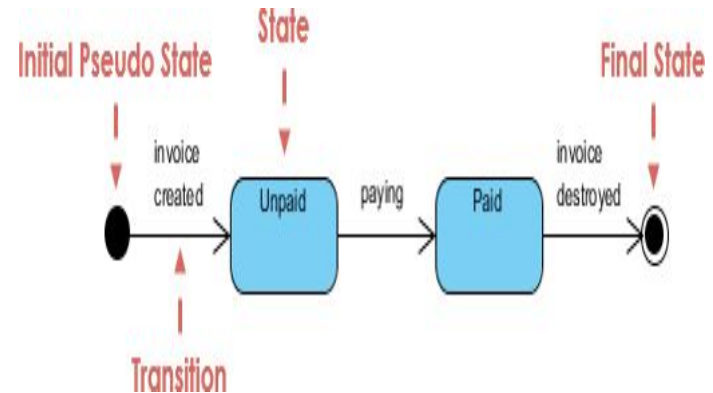
Structural UML Diagrams

- **Package Diagram** – We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organise UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.



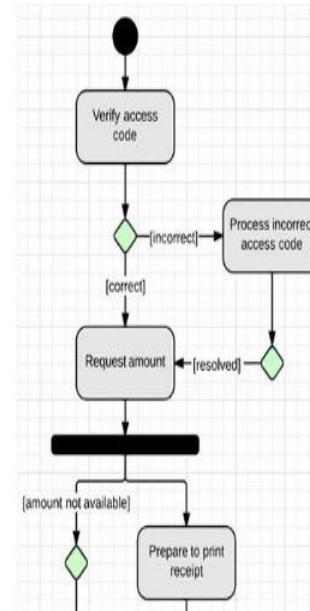
Behavior Diagrams –

- State Machine Diagrams – A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as State machines and State-chart Diagrams . These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.



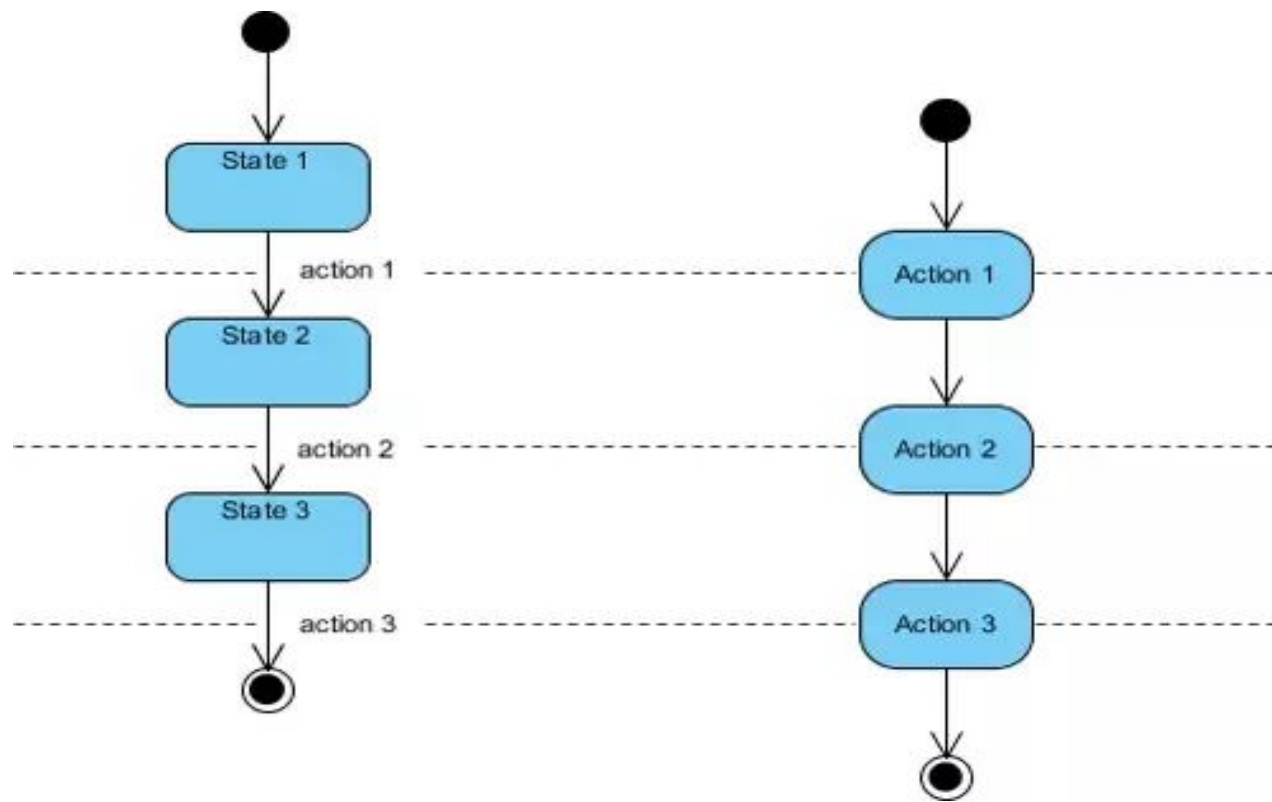
Behavior Diagrams –

- Activity Diagrams – We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.



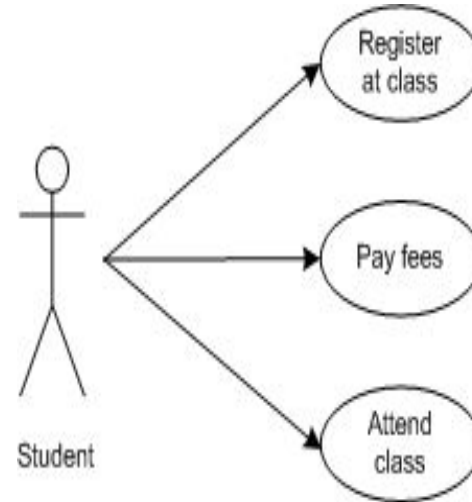
Activity Diagram
Example 1





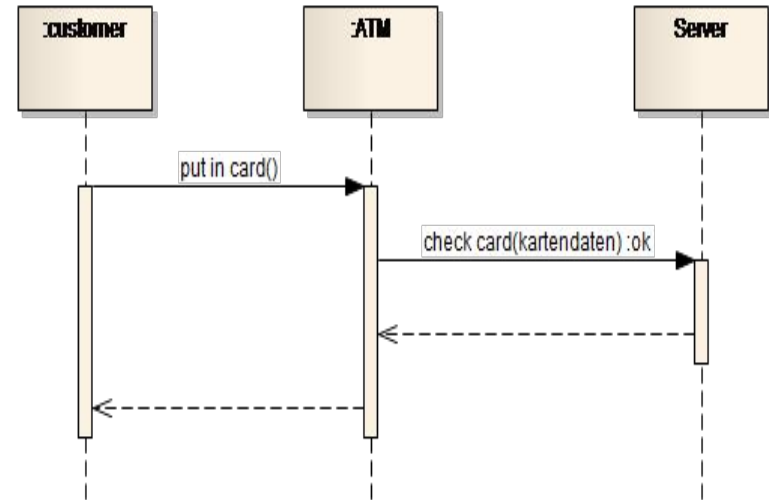
Behavior Diagrams –

- Use Case Diagrams – Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.



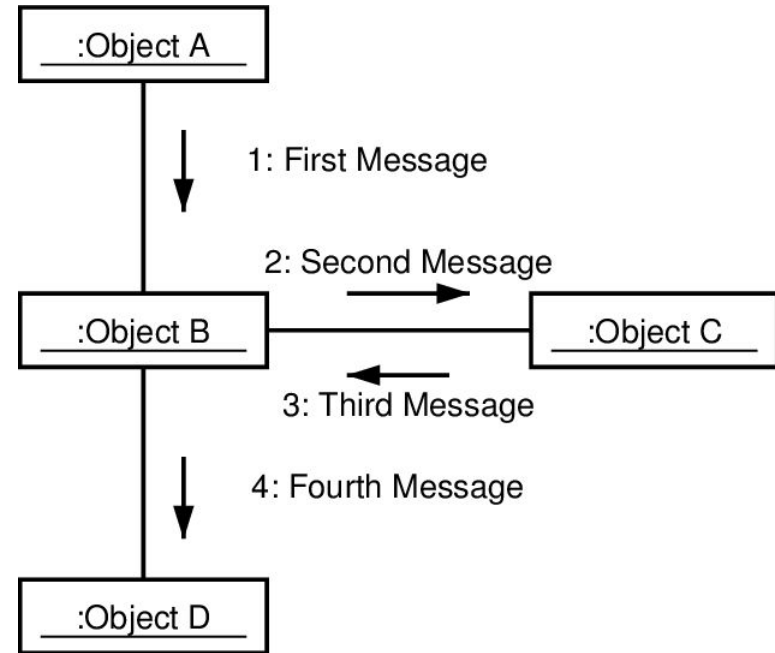
Behavior Diagrams –

- Sequence Diagram – A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.



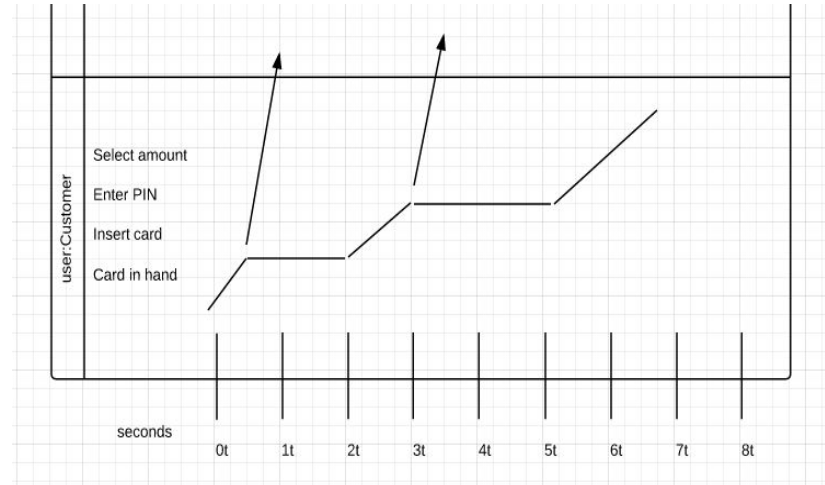
Behavior Diagrams –

- Communication Diagram – A Communication Diagram (known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams, however, communication diagrams represent objects and links in a free form.



Behavior Diagrams –

- Timing Diagram – Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.

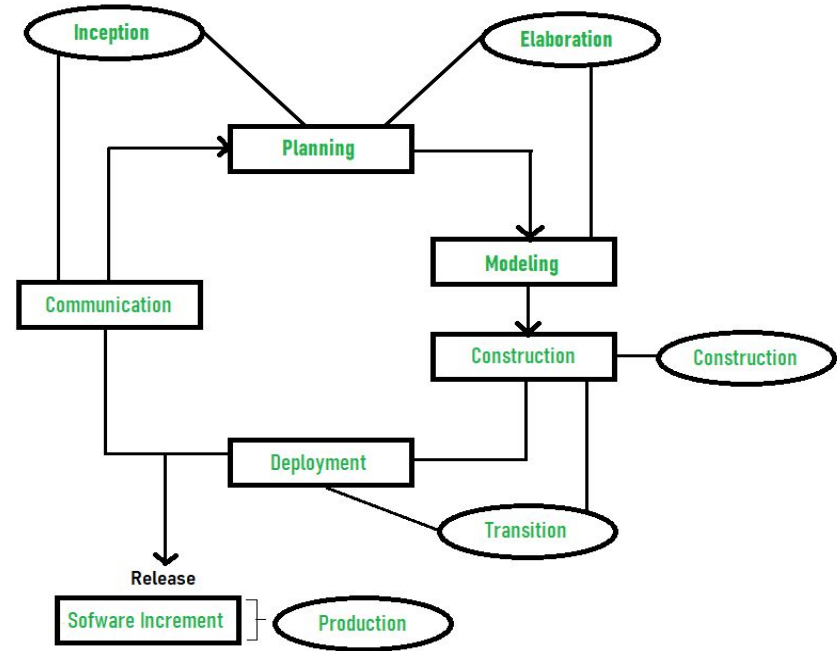


RUP and its Phases

- Rational Unified Process (RUP) is a software development process for object-oriented models. It is also known as the Unified Process Model. It is created by Rational corporation and is designed and documented using UML (Unified Modeling Language). This process is included in IBM Rational Method Composer (RMC) product. IBM (International Business Machine Corporation) allows us to customize, design, and personalize the unified process.
- RUP is proposed by Ivar Jacobson, Grady Bootch, and James Rumbaugh. Some characteristics of RUP include use-case driven, Iterative (repetition of the process), and Incremental (increase in value) by nature, delivered online using web technology, can be customized or tailored in modular and electronic form, etc. RUP reduces unexpected development costs and prevents wastage of resources.

Phases of RUP :

- here are total five phases of life cycle of RUP:



Inception –

- Communication and planning are main.
- Identifies Scope of the project using use-case model allowing managers to estimate costs and time required.
- Customers requirements are identified and then it becomes easy to make a plan of the project.
- Project plan, Project goal, risks, use-case model, Project description, are made.
- Project is checked against the milestone criteria and if it couldn't pass these criteria then project can be either cancelled or redesigned.

Phases of RUP :

Elaboration

- Planning and modeling are main.
- Detailed evaluation, development plan is carried out and diminish the risks.
- Revise or redefine use-case model (approx. 80%), business case, risks.
- Again, checked against milestone criteria and if it couldn't pass these criteria then again project can be cancelled or redesigned.
- Executable architecture baseline.

Construction –

- Project is developed and completed.
- System or source code is created and then testing is done.
- Coding takes place.

Phases of RUP :

Transition –

- Final project is released to public.
- Transit the project from development into production.
- Update project documentation.
- Beta testing is conducted.
- Defects are removed from project based on feedback from public.

Production –

- Final phase of the model.
- Project is maintained and updated accordingly.

What are the six best practices of rational unified process?

Six best practices-

Develop iteratively

It is best to know all requirements in advance; however, often this is not the case. Several software development processes exist that deal with providing solutions to minimize cost in terms of development phases.

Manage requirements

Always keep in mind the requirements set by users.

Use components

Breaking down an advanced project is not only suggested but in fact unavoidable. This promotes ability to test individual components before they are integrated into a larger system. Also, code reuse is a big plus and can be accomplished more easily through the use of object-oriented programming

Six best practices- RUP

Model visually

- Use diagrams to represent all major components, users, and their interaction. "UML", short for Unified Modeling Language, is one tool that can be used to make this task more feasible.

Verify quality

- Always make testing a major part of the project at any point of time. Testing becomes heavier as the project progresses but should be a constant factor in any software product creation.

Control changes

- Many projects are created by many teams, sometimes in various locations, different platforms may be used, etc. As a result, it is essential to make sure that changes made to a system are synchronized and verified constantly. (See Continuous integration).

UNIT 2

Types of Models in Object Oriented Modeling and Design

Intention of object oriented modeling and design is to learn how to apply object -oriented concepts to all the stages of the software development life cycle. Object-oriented modeling and design is a way of thinking about problems using models organized around real world concepts. The fundamental construct is the object, which combines both data structure and behavior.

Purpose of Models:

Testing a physical entity before building it

Communication with customers

Visualization

Reduction of complexity

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.

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

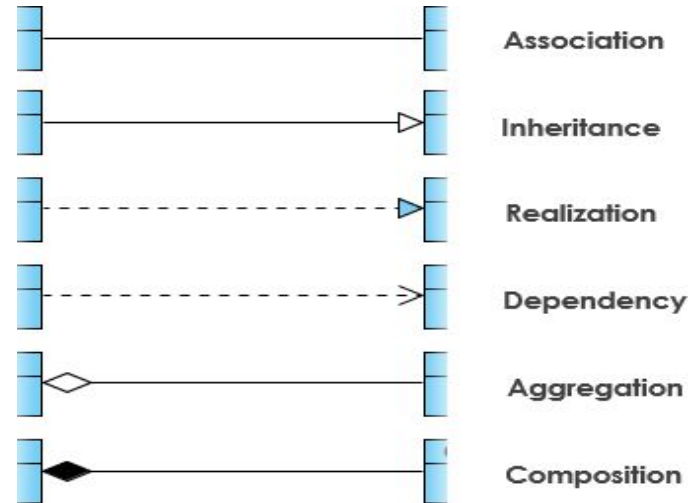
There are four kinds of things in UML, namely

- **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.
- **Annotational 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 –

- Name: Name of relationship, things, and diagrams.
- Visibility: How these names will be seen and used by others.
- Scope: Scope can be defined as an aspect that gives a definite meaning to a name.
- Integrity: Integrity is used to measure how things are associated with each other.
- Execution: It means to simulate the dynamic model.

Common Mechanisms

UML has four common mechanisms –

- Specifications
- Adornments
- Common Divisions
- Extensibility Mechanisms

Common 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

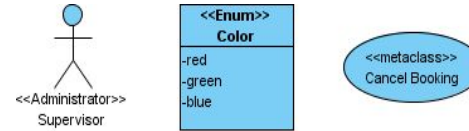
- Each element in UML has a unique graphical notation. Besides, there are notations to represent the important aspects of an element like name, scope, visibility, etc.

Common Mechanisms

Common Divisions

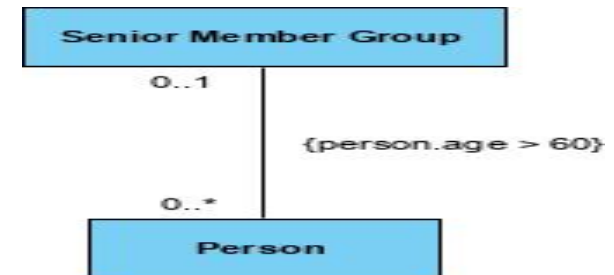
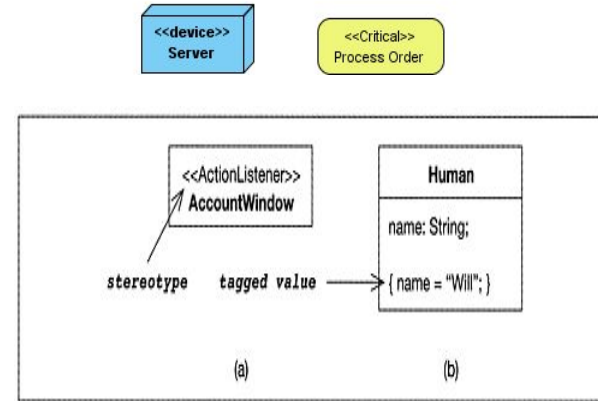
- Object-oriented systems can be divided in many ways. The two common ways of division are –
- Division of classes and objects – A class is an abstraction of a group of similar objects. An object is the concrete instance that has actual existence in the system.
- Division of Interface and Implementation – An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

Common Mechanisms

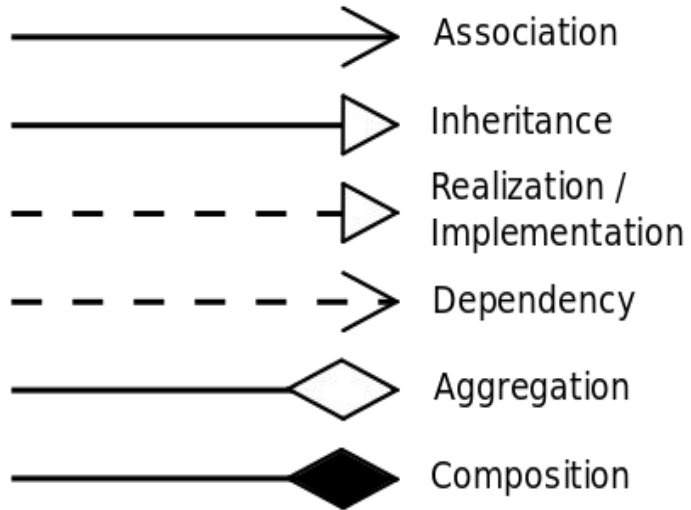


Extensibility Mechanisms

- 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.



Relationships



Relationship

Association is a simple structural connection or channel between classes and is a relationship where all objects have their own lifecycle and there is no owner.

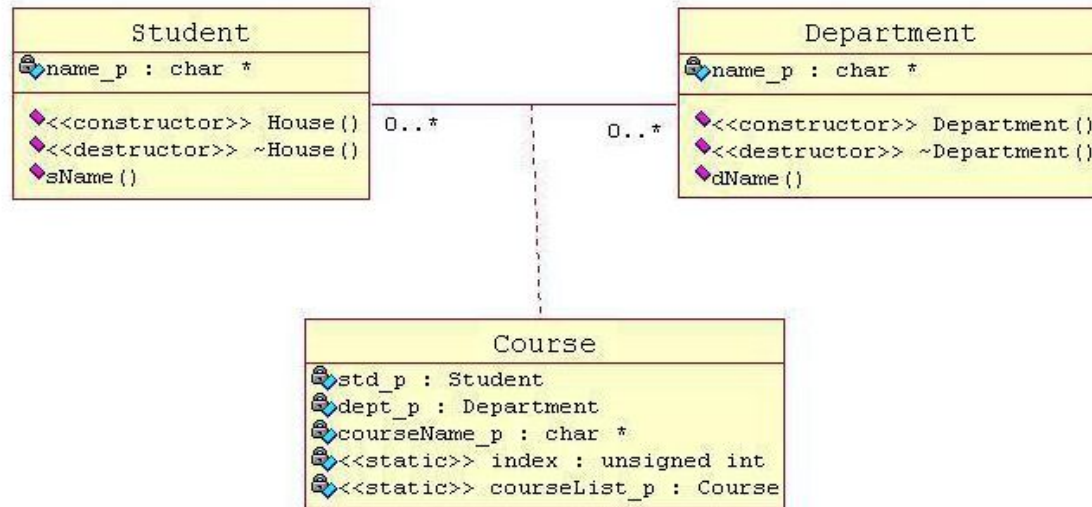
Lets take an example of Department and Student.

Multiple students can associate with a single Department and single student can associate with multiple Departments, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

Here is respective Model and Code for the above example.

Association

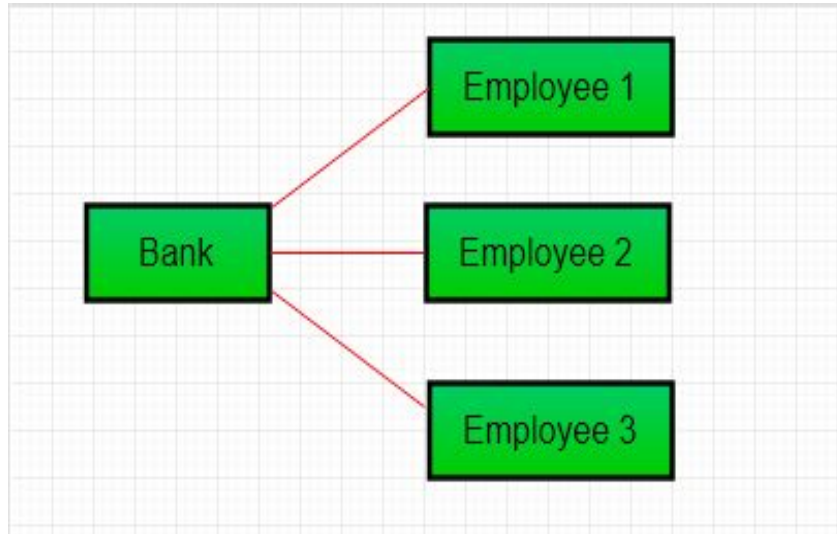
Course class Associates Student and Department classes



Association

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.

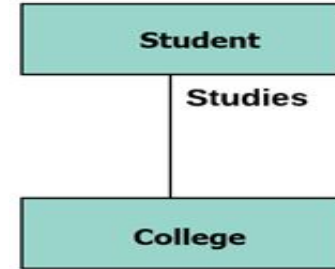
Association



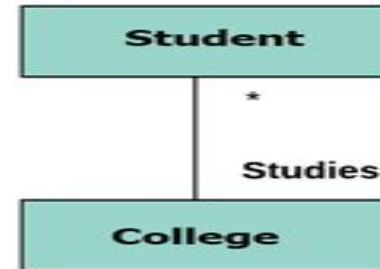
In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.

Association:

- This kind of relationship represents static relationships between classes A and B. For example; an employee works for an organization.
- Association is mostly verb or a verb phrase or noun or noun phrase.
- It should be named to indicate the role played by the class attached at the end of the association path.



Multiplicity



Aggregation

Aggregation is a specialize form of Association where all object have their own lifecycle but there is a ownership like parent and child. Child object can not belong to another parent object at the same time. We can think of it as "has-a" relationship.

Implementation details:

Typically we use pointer variables that point to an object that lives outside the scope of the aggregate class Can use reference values that point to an object that lives outside the scope of the aggregate class Not responsible for creating/destroying subclasses

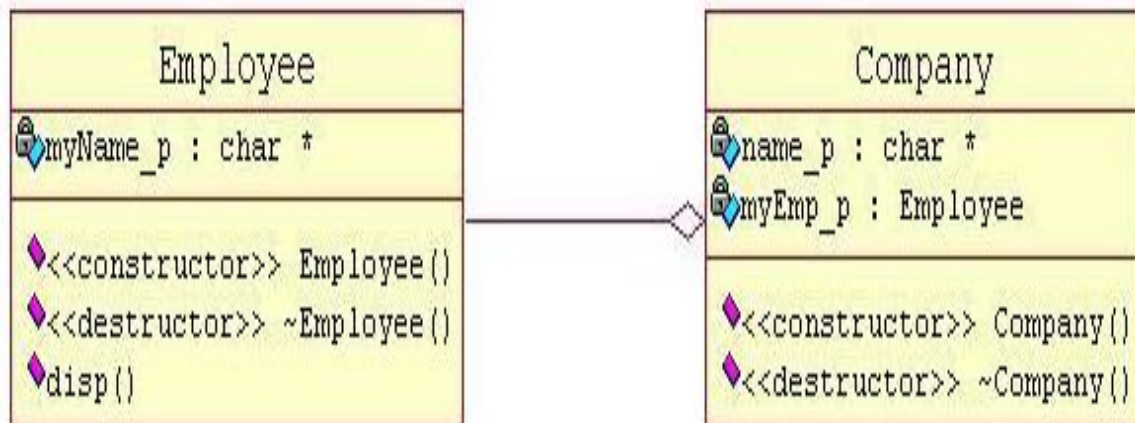
Lets take an example of Employee and Company.

A single Employee can not belong to multiple Companies (legally!!), but if we delete the Company, Employee object will not destroy.

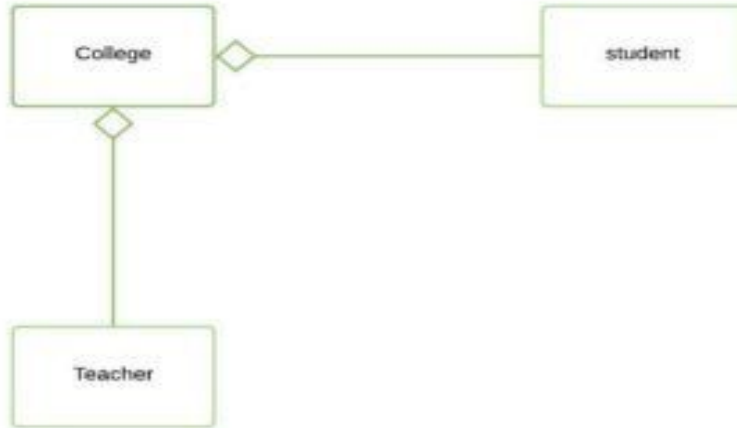
Here is respective Model and Code for the above example.

Aggregation

Employee class has Agregation Relationship with Company class



Aggregation



It is a special form of Association where:

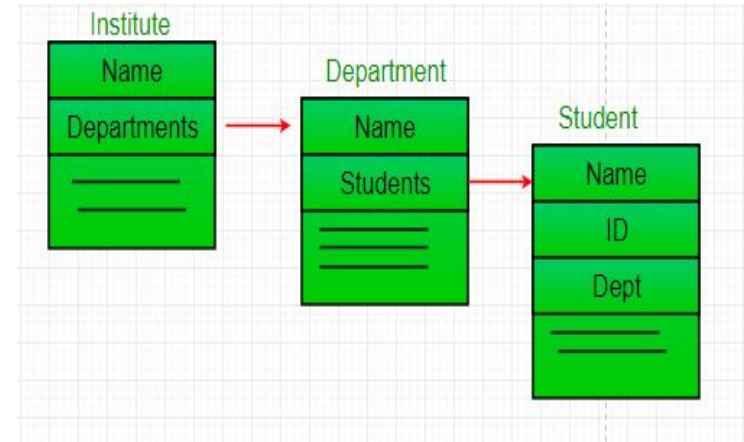
It represents Has-A relationship.

It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.

In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity

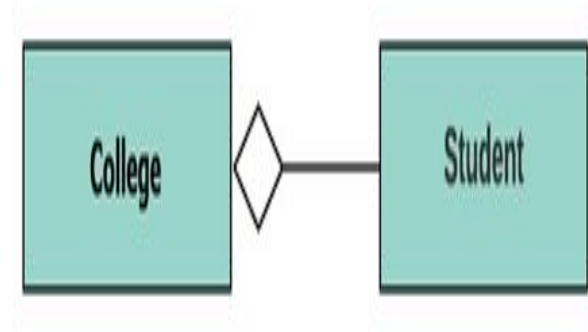
Aggregation

- In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).
- It represents a Has-A relationship.



Aggregation

- Aggregation is a special type of association that models a whole- part relationship between aggregate and its parts.
- For example, the class college is made up of one or more student. In aggregation, the contained classes are never totally dependent on the lifecycle of the container. Here, the college class will remain even if the student is not available.



Composition

Composition is again specialize form of Aggregation. It is a strong type of Aggregation. Here the Parent and Child objects have coincident lifetimes. Child object dose not have it's own lifecycle and if parent object gets deleted, then all of it's child objects will also be deleted.

Implentation details:

1. Typically we use normal member variables
2. Can use pointer values if the composition class automatically handles allocation/deallocation
3. Responsible for creation/destruction of subclasses

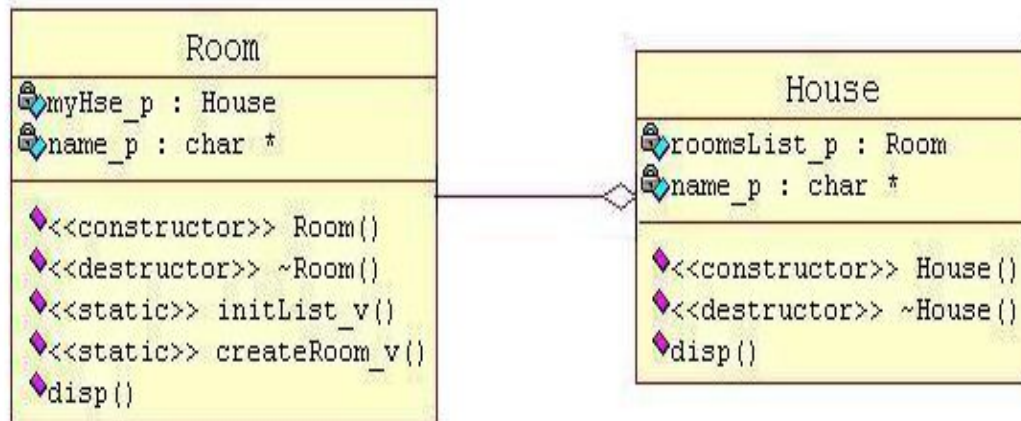
Lets take an example of a relationship between House and it's Rooms.

House can contain multiple rooms there is no independent life for room and any room can not belong to two different house. If we delete the house room will also be automatically deleted.

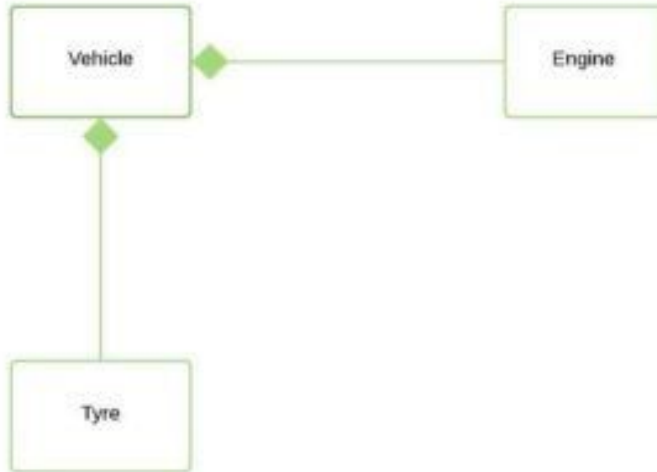
Here is respective Model and Code for the above example.

Composition

Room class has Composition Relationship with House class



Composition



Composition is a restricted form of Aggregation in which two entities are highly dependent on each other. It represents part-of relationship. In composition, both the entities are dependent on each other.

When there is a composition between two entities, the composed object cannot exist without the other entity.

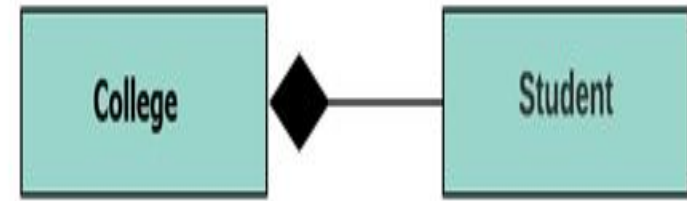
Lets take example of Library.

Composition

- a library can have no. of books on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That's why it is composition.

Composition:

- The composition is a special type of aggregation which denotes strong ownership between two classes when one class is a part of another class.
- For example, if college is composed of classes student. The college could contain many students, while each student belongs to only one college. So, if college is not functioning all the students also removed.



Aggregation vs. Composition

Aggregation

Aggregation indicates a relationship where the child can exist separately from their parent class. Example: Automobile (Parent) and Car (Child). So, If you delete the Automobile, the child Car still exist.

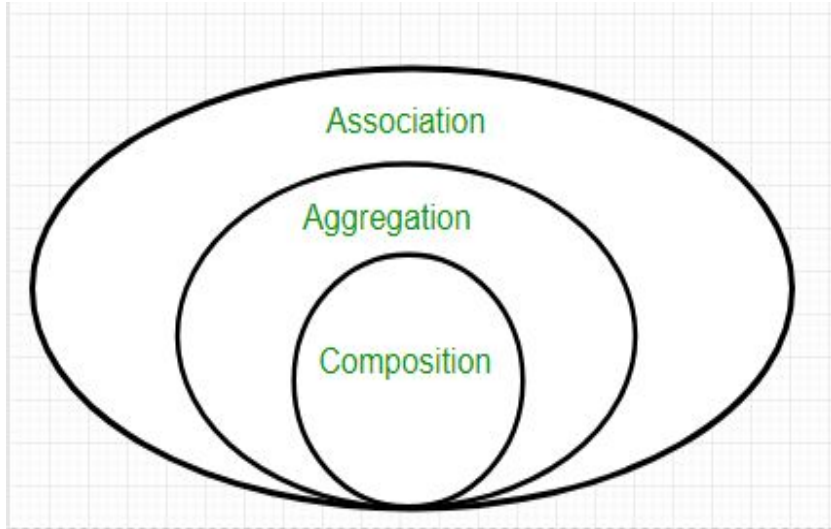
Composition

Composition display relationship where the child will never exist independent of the parent. Example: House (parent) and Room (child). Rooms will never separate into a House.

Aggregation vs Composition

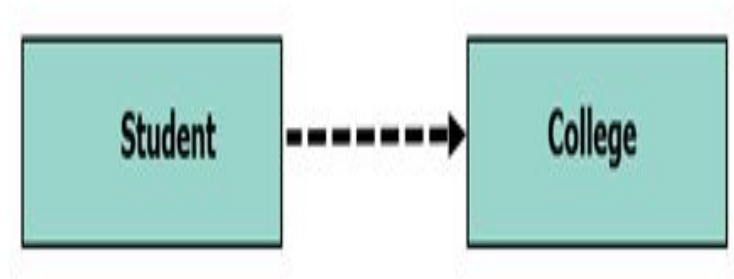
- **Dependency:** Aggregation implies a relationship where the child can exist independently of the parent. For example, **Bank and Employee**, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: **Human and heart**, heart don't exist separate to a Human
- **Type of Relationship:** Aggregation relation is “has-a” and composition is “part-of” relation.
- **Type of association:** Composition is a strong Association whereas Aggregation is a weak Association.

Association, Composition and Aggregation



Dependency

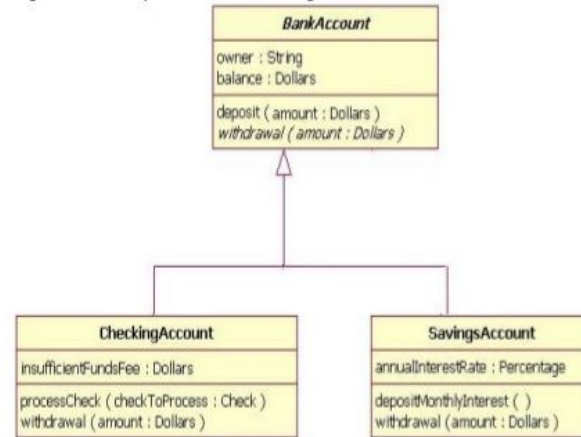
A dependency means the relation between two or more classes in which a change in one may force changes in the other. However, it will always create a weaker relationship. Dependency indicates that one class depends on another.



Generalization(Inheritance)

- It is also called a parent-child relationship. In generalization, one element is a specialization of another general component. It may be substituted for it. It is mostly used to represent inheritance.

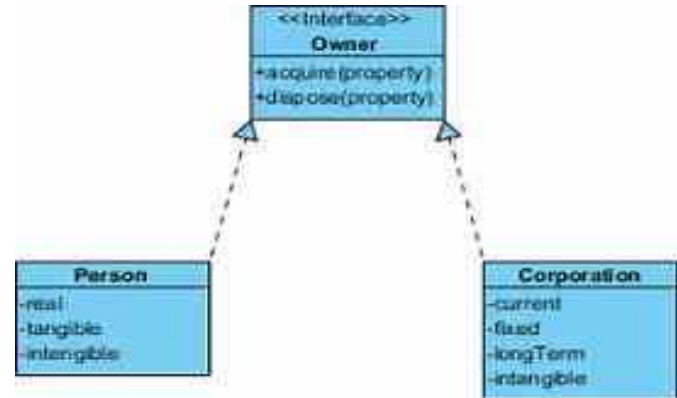
Generalization Relationships



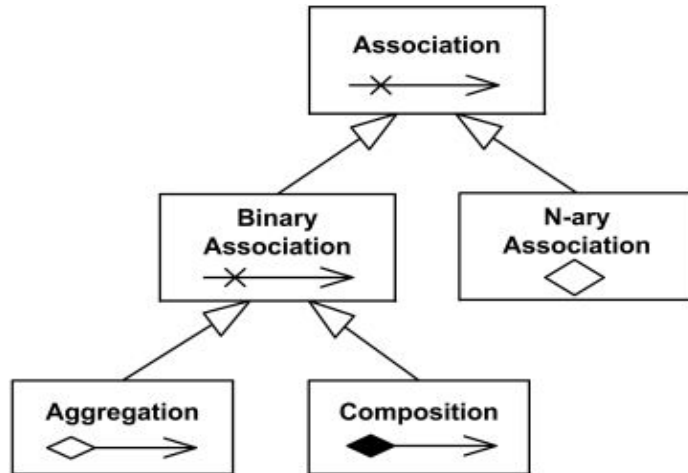
An example of inheritance using tree notation

Realization

In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of interfaces.



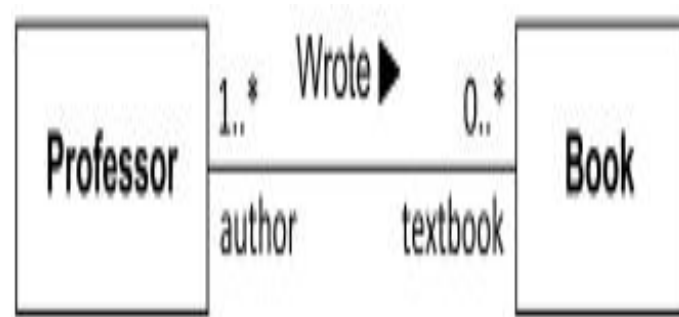
Association relationship overview diagram



An association is usually drawn as a solid line connecting two classifiers or a single classifier to itself. Name of the association can be shown somewhere near the middle of the association line but not too close to any of the ends of the line. Each end of the line could be decorated with the name of the association end

Association End

Association end is a connection between the line depicting an association and the icon depicting the connected classifier. Name of the association end may be placed near the end of the line. The association end name is commonly referred to as role name (but it is not defined as such in the UML 2.4 standard). The role name is optional and suppressible.



Professor "playing the role" of author is associated with textbook end typed as Book.

Association End

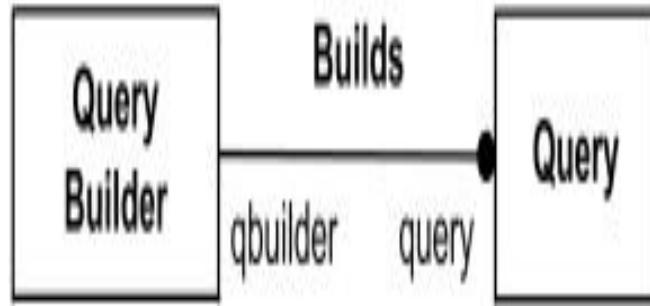
The idea of the role is that the same classifier can play the same or different roles in other associations. For example, Professor could be an author of some Books or an editor.

Association end could be owned either by

- end classifier, or
- association itself

Association ends of associations with more than two ends must be owned by the association. Ownership of association ends by an associated classifier may be indicated graphically by a small filled circle (aka dot). The dot is drawn at the point where line meets the classifier. It could be interpreted as showing that the model includes a property of the type represented by the classifier touched by the dot. This property is owned by the classifier at the other end.

Association End



Association end query is owned by classifier QueryBuilder
and association end qbuilder is owned by association Builds itself

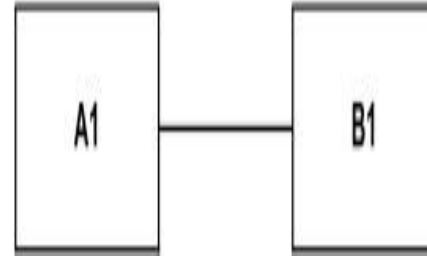
The "ownership" dot may be used in combination with the other graphic line-path notations for properties of associations and association ends. These include aggregation type and navigability.

Navigability

Notation: navigable end is indicated by an open arrowhead on the end of an association

not navigable end is indicated with a small x on the end of an association

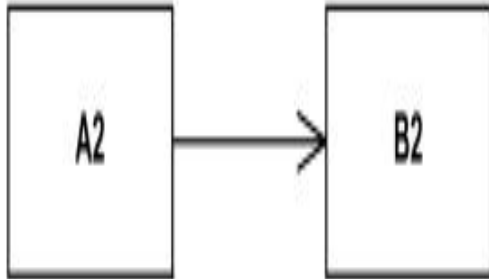
no adornment on the end of an association means unspecified navigability



Both ends of association have unspecified navigability.

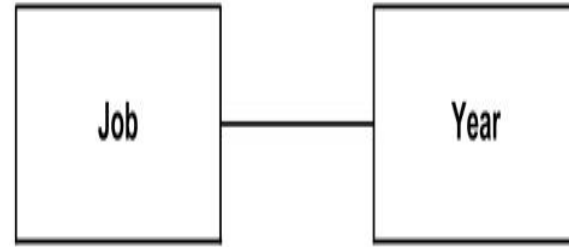
Navigability

- A2 has unspecified navigability while B2 is navigable from A2.



Arity

- Binary association relates two typed instances. It is normally rendered as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct). The line may consist of one or more connected segments.



Job and Year classifiers are associated

Arity

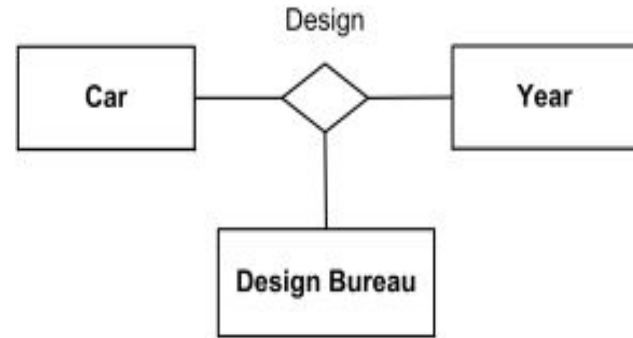
- A small solid triangle could be placed next to or in place of the name of binary association (drawn as a solid line) to show the order of the ends of the association. The arrow points along the line in the direction of the last end in the order of the association ends. This notation also indicates that the association is to be read from the first end to the last end.



Order of the ends and reading: Car - was designed in - Year

N-ary Association

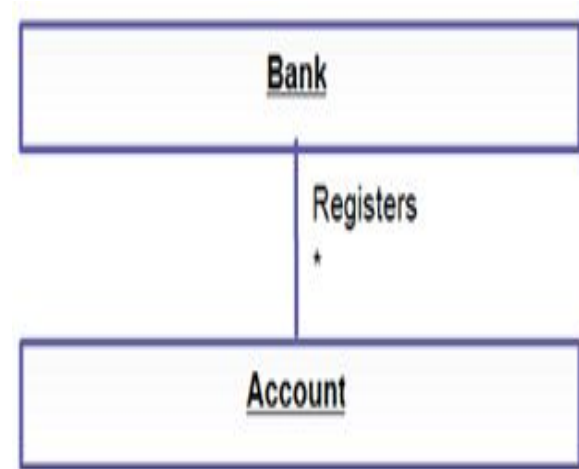
- Any association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end's type. N-ary association with more than two ends can only be drawn this way.



Ternary association Design relates three classifiers

Multiplicity

- The number of elements or cardinality could be defined by multiplicity. It is one of the most misunderstood relationships which describes the number of instances allowed for a particular element by providing an inclusive non-negative integers interval. It has both lower and upper bound. For example, a bank would have many accounts registered to it. Thus near the account class, a star sign is present.

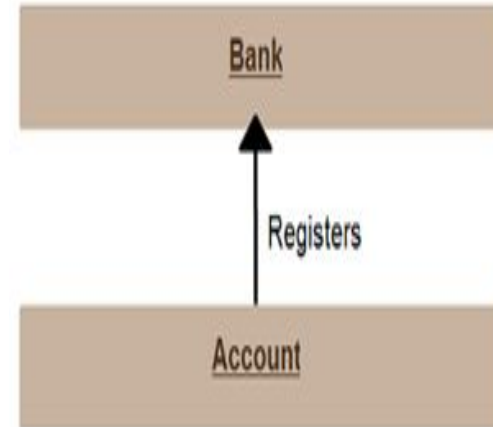


MULTIPLICITY

0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
7...7	Seven only
0..1	Zero or One
4..7	Four to seven

Directed Association(NAVIGABILITY)

- This is a one-directional relationship in a class diagram which ensures the flow of control from one to another classifier. The navigability is specified by one of the association ends. The relationship between two classifiers could be described by naming any association. The direction of navigation is indicated by an arrow. Below example shows an arrowhead relationship between the container and the contained.

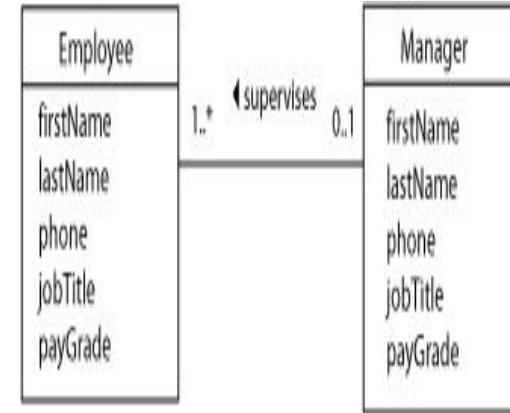


recursive associations

A recursive association connects a single class type (serving in one role) to itself (serving in another role).

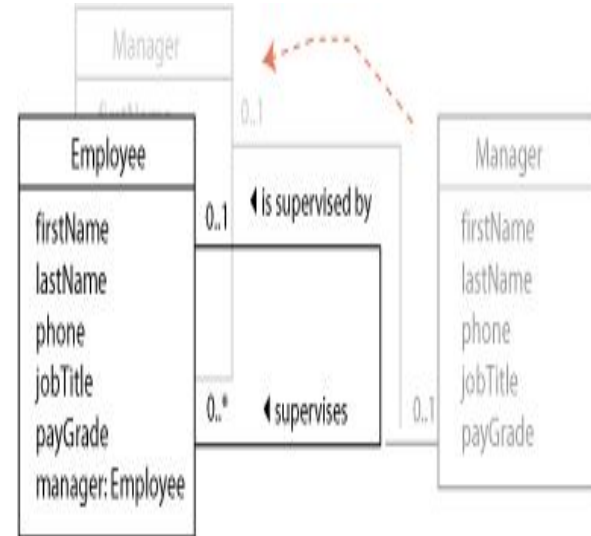
Example: In most companies, each employee (except the CEO) is supervised by one manager. Of course, not all employees are managers. This example is used in almost every database textbook, since the association between employees and managers is relatively easy to understand. Perhaps the best way to visualize it is to start with two class types:

Incorrect model



correct model

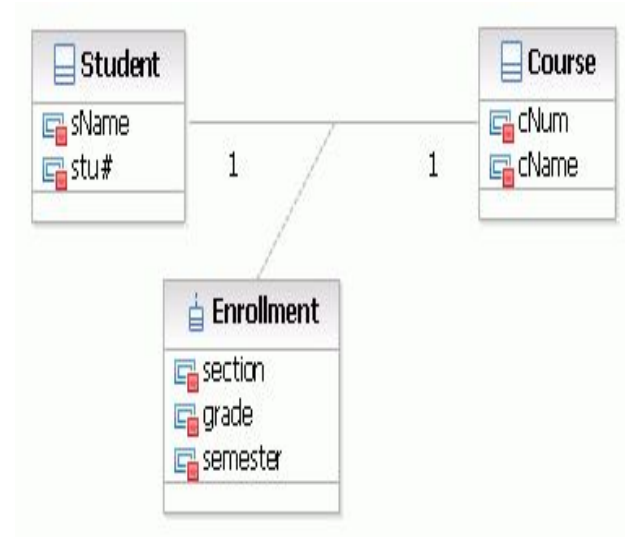
- The problem with this model is that each manager is also an employee. Building the second (manager) table not only duplicates information from the employee table, it virtually guarantees that there will be mistakes and conflicts in the data. We can fix the problem by eliminating the redundant class and re-drawing the association line.



Association Class

An association class is a class that is part of an association relationship between two other classes. You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations.

For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course. An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.



5. Use Case Approach:

This technique combines text and pictures to provide a better understanding of the requirements.

The use cases describe the 'what', of a system and not 'how'. Hence they only give a functional view of the system.

The components of the use case design includes three major things – Actor, Use cases, use case diagram.

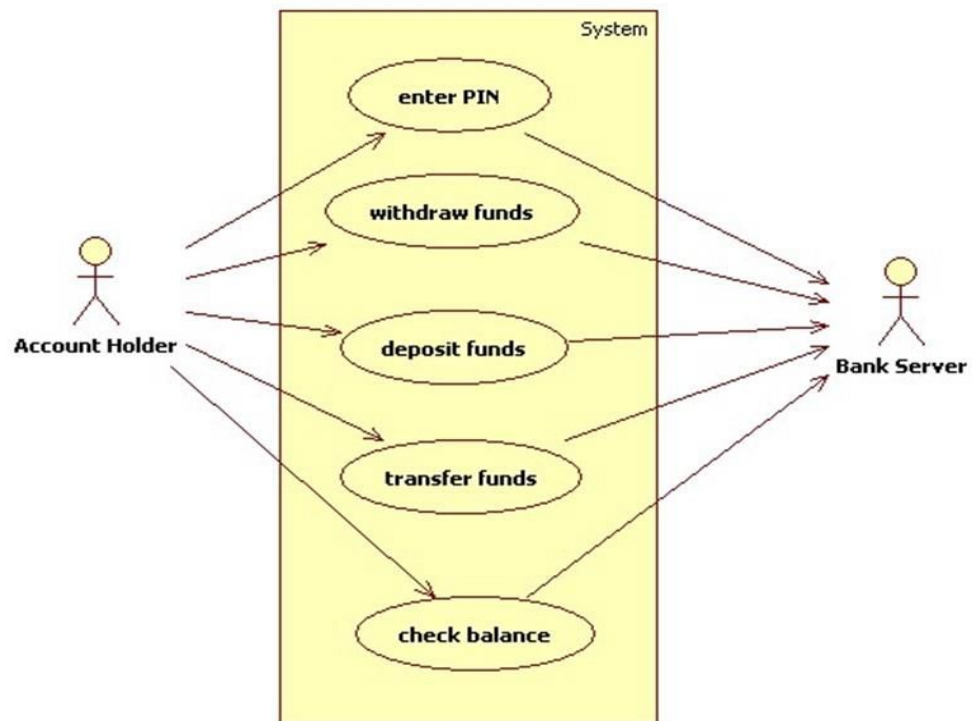
Actor – It is the external agent that lies outside the system but interacts with it in some way. An actor maybe a person, machine etc. It is represented as a stick figure. Actors can be primary actors or secondary actors.

- Primary actors – It requires assistance from the system to achieve a goal.
- Secondary actor – It is an actor from which the system needs assistance.

Use Case Approach:

1. **Use cases** – They describe the sequence of interactions between actors and the system. They capture who(actors) do what(interaction) with the system. A complete set of use cases specifies all possible ways to use the system.
2. **Use case diagram** – A use case diagram graphically represents what happens when an actor interacts with a system. It captures the functional aspect of the system.
 - A stick figure is used to represent an actor.
 - An oval is used to represent a use case.
 - A line is used to represent a relationship between an actor and a use case.

Use case diagram



USE CASE DIAGRAM

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system.

Purpose of Use Case Diagrams

The main purpose of a use case diagram is to portray the dynamic aspect of a system. It accumulates the system's requirement, which includes both internal as well as external influences. It invokes persons, use cases, and several things that invoke the actors and elements accountable for the implementation of use case diagrams. It represents how an entity from the external environment can interact with a part of the system.

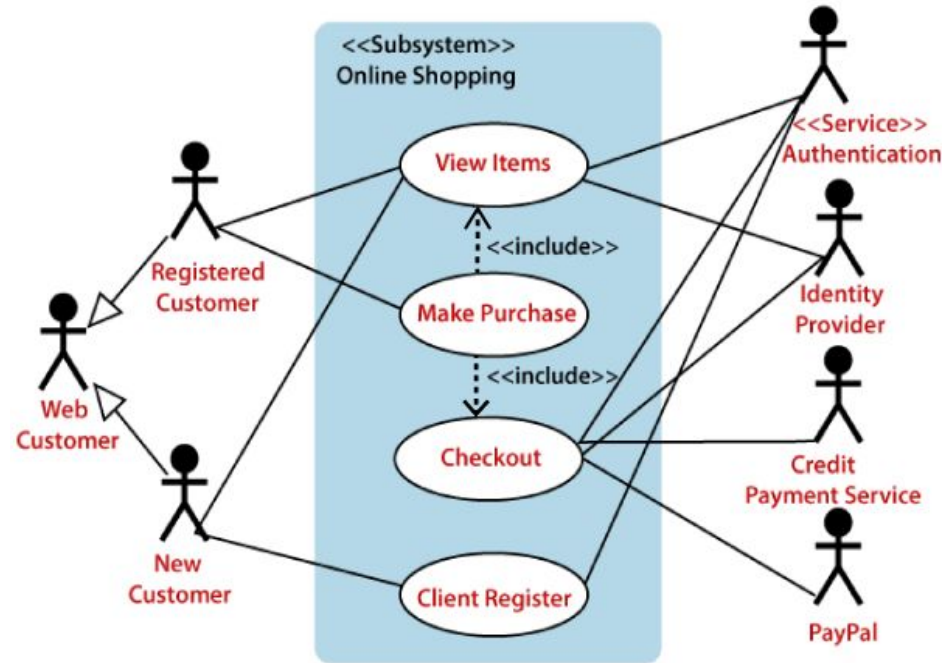
Following are the purposes of a use case diagram given below:

- It gathers the system's needs.
- It depicts the external view of the system.
- It recognizes the internal as well as external factors that influence the system.
- It represents the interaction between the actors.

Following are some rules that must be followed while drawing a use case diagram:

- A pertinent and meaningful name should be assigned to the actor or a use case of a system.
- The communication of an actor with a use case must be defined in an understandable way.
- Specified notations to be used as and when required.
- The most significant interactions should be represented among the multiple no of interactions between the use case and actors.

Example of a Use Case Diagram

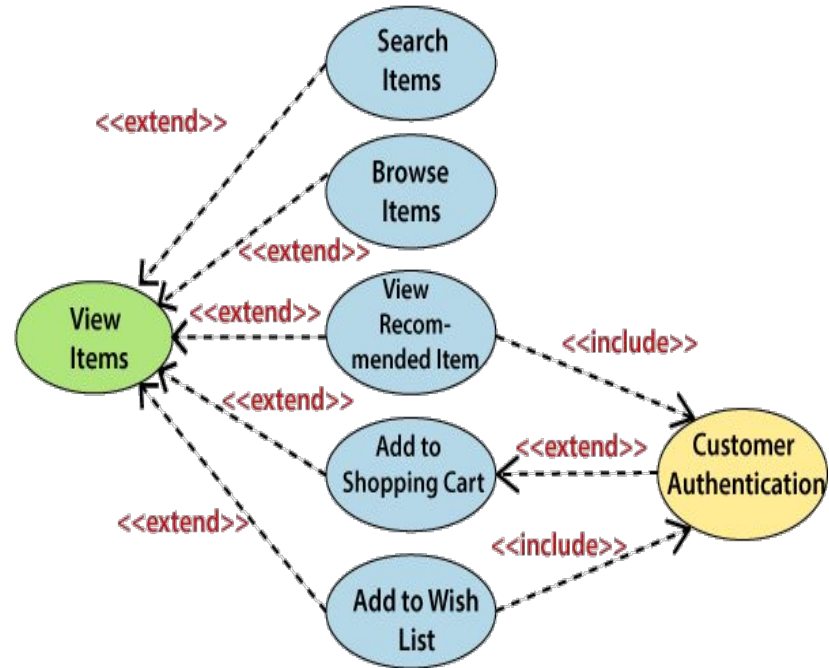


Here the Web Customer actor makes use of any online shopping website to purchase online. The top-level uses are as follows; View Items, Make Purchase, Checkout, Client Register. The View Items use case is utilized by the customer who searches and view products. The Client Register use case allows the customer to register itself with the website for availing gift vouchers, coupons, or getting a private sale invitation. It is to be noted that the Checkout is an included use case, which is part of Making Purchase, and it is not available by itself.

Example of a Use Case Diagram

The View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item. The View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item.

Both View Recommended Item and Add to Wish List include the Customer Authentication use case, as they necessitate authenticated customers, and simultaneously item can be added to the shopping cart without any user authentication.



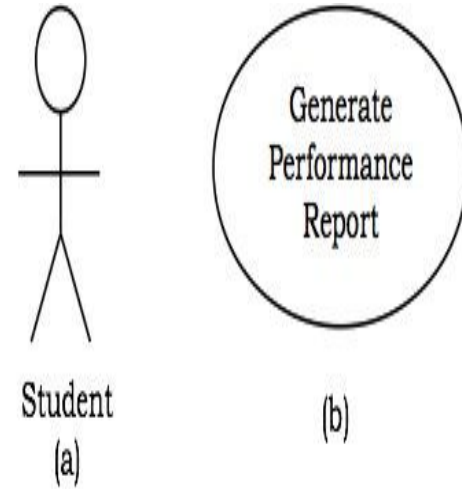
Use Case Model

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 –

- To 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.

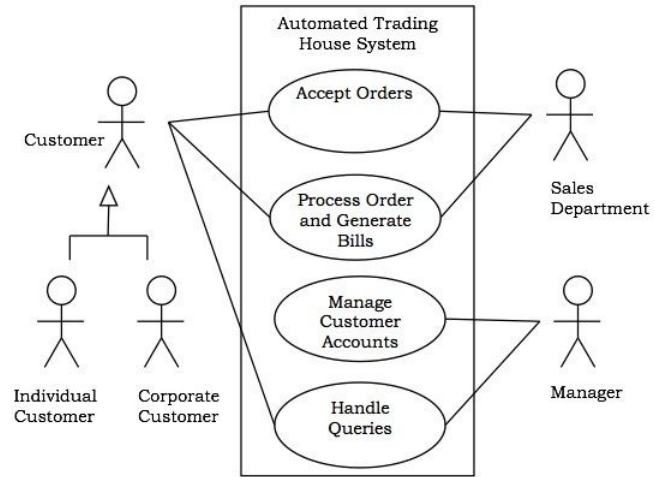
Example-

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.



Functional Requirements vs Non Functional Requirements

- What is a Functional Requirement?

In software engineering, a functional requirement defines a system or its component. It describes the functions a software must perform. A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform. Functional software requirements help you to capture the intended behaviour

Functional software requirements help you to capture the intended behavior of the system. This behavior may be expressed as functions, services or tasks or which system is required to perform.

What is Non-Functional Requirement?

- A non-functional requirement defines the quality attribute of a software system. They represent a set of standards used to judge the specific operation of a system. Example, how fast does the website load?
- A non-functional requirement is essential to ensure the usability and effectiveness of the entire software system. Failing to meet non-functional requirements can result in systems that fail to satisfy user needs.
- Non-functional Requirements allows you to impose constraints or restrictions on the design of the system across the various agile backlogs. Example, the site should load in 3 seconds when the number of simultaneous users are > 10000. Description of non-functional requirements is just as critical as a functional requirement.

KEY DIFFERENCE

- A functional requirement defines a system or its component whereas a non-functional requirement defines the performance attribute of a software system.
- Functional requirements along with requirement analysis help identify missing requirements while the advantage of Non-functional requirement is that it helps you to ensure good user experience and ease of operating the software.
- Functional Requirement is a verb while Non-Functional Requirement is an attribute
- Types of Non-functional requirement are Scalability Capacity, Availability, Reliability, Recoverability, Data Integrity, etc. whereas transaction corrections, adjustments, and cancellations, Business Rules, Certification Requirements, Reporting Requirements, Administrative functions, Authorization levels, Audit Tracking, External Interfaces, Historical Data management, Legal or Regulatory Requirements are various types of functional requirements.

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.

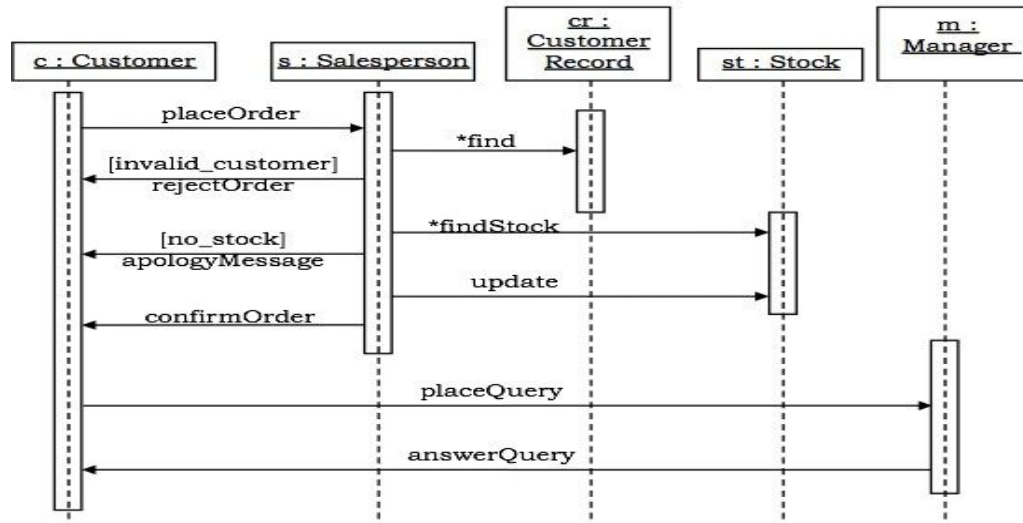
Sequence Diagrams

- 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.

What is a Sequence Diagram?

- Sequence diagrams, commonly used by developers, model the interactions between objects in a single use case. They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed.
- In simpler words, a sequence diagram shows different parts of a system work in a 'sequence' to get something done.

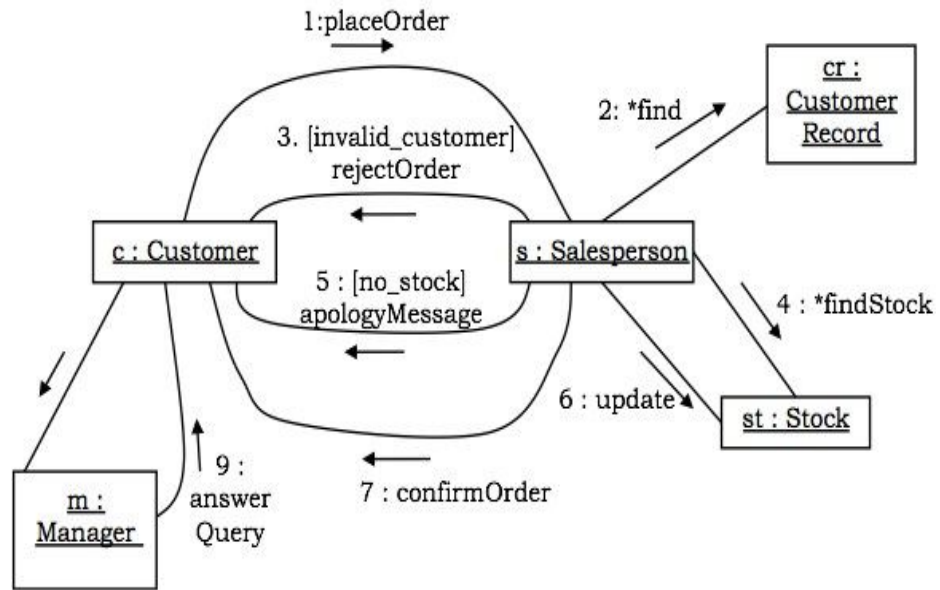
Sequence Diagrams



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.

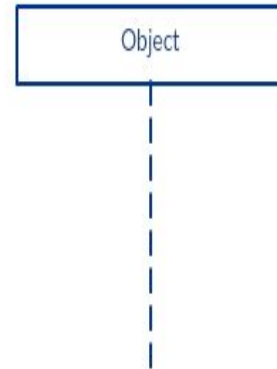
Collaboration Diagrams



Sequence Diagram Notations

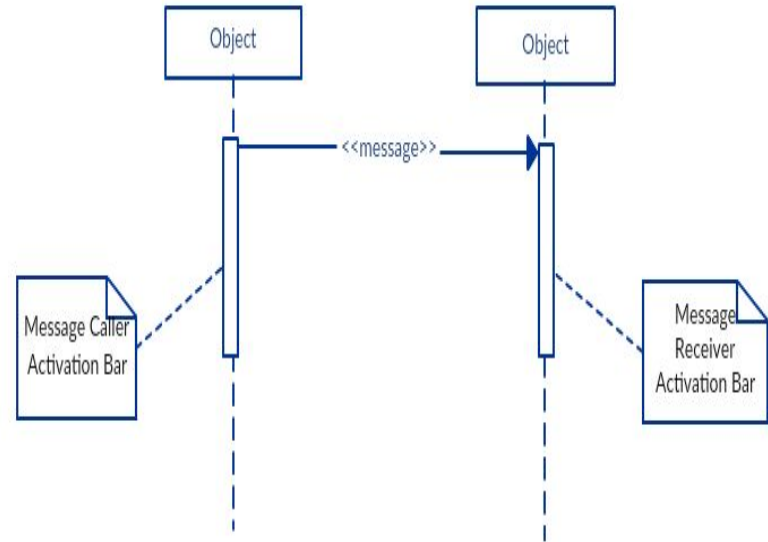
- A sequence diagram is structured in such a way that it represents a timeline which begins at the top and descends gradually to mark the sequence of interactions. Each object has a column and the messages exchanged between them are represented by arrows.
- A Quick Overview of the Various Parts of a Sequence Diagram

Lifeline Notation



Activation Bars

- Activation bar is the box placed on the lifeline. It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.



Message Arrows

- An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram. A message can flow in any direction; from left to right, right to left or back to the Message Caller itself. While you can describe the message being sent from one object to the other on the arrow, with different arrowheads you can indicate the type of message being sent or received.
- The message arrow comes with a description, which is known as a message signature, on it. The format for this message signature is below. All parts except the message_name are optional.
- attribute = message_name (arguments): return_type

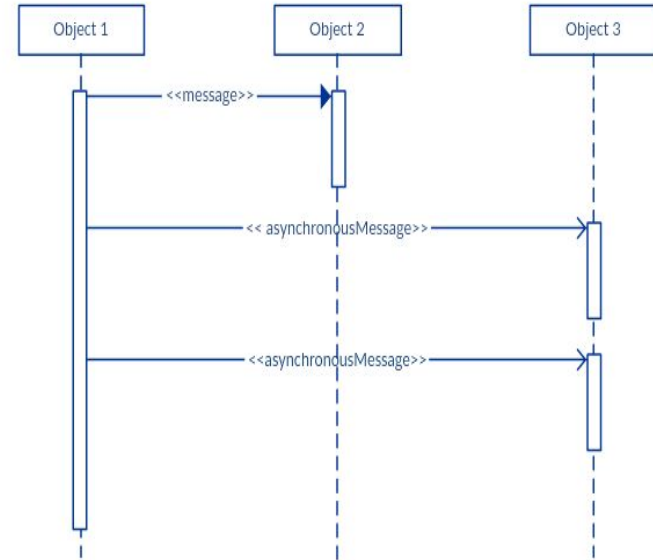
Synchronous message

- As shown in the activation bars example, a synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message. The arrowhead used to indicate this type of message is a solid one, like the one below.



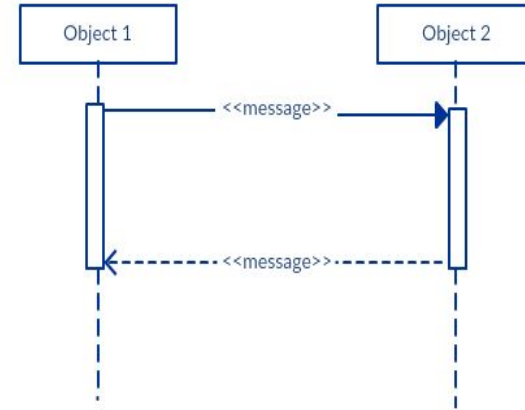
Asynchronous message

- An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system. The arrowhead used to show this type of message is a line arrow like shown in the example below.



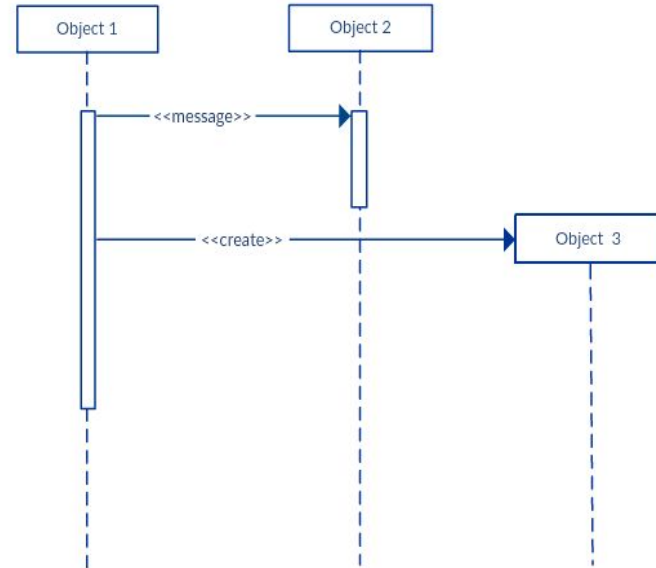
Return message

- A return message is used to indicate that the message receiver is done processing the message and is returning control over to the message caller. Return messages are optional notation pieces, for an activation bar that is triggered by a synchronous message always implies a return message.



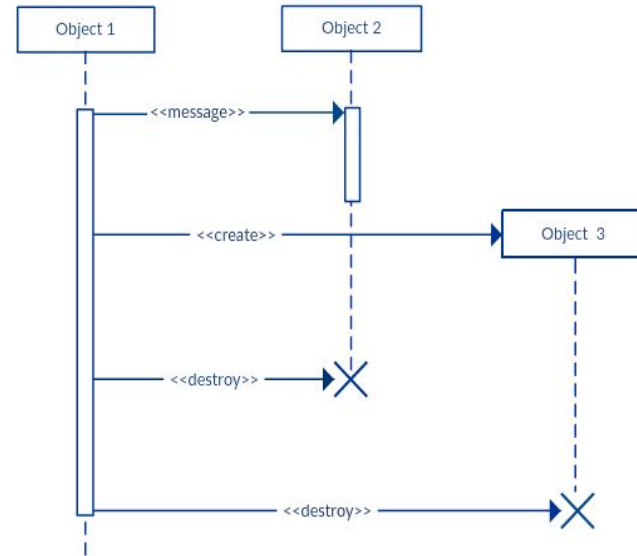
Participant creation message

- Objects do not necessarily live for the entire duration of the sequence of events. Objects or participants can be created according to the message that is being sent.
- The dropped participant box notation can be used when you need to show that the particular participant did not exist until the create call was sent. If the created participant does something immediately after its creation, you should add an activation box right below the participant box.



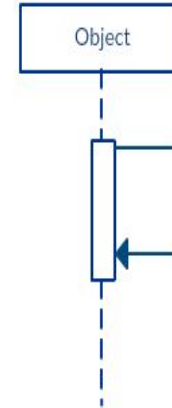
Participant destruction message

- Likewise, participants when no longer needed can also be deleted from a sequence diagram. This is done by adding an 'X' at the end of the lifeline of the said participant.



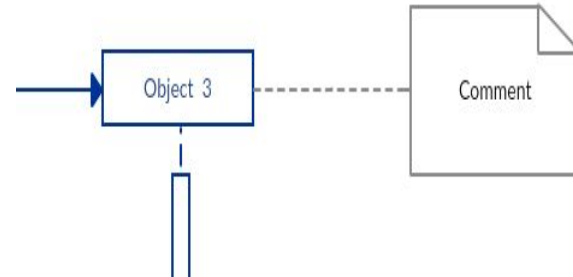
Reflexive message

- When an object sends a message to itself, it is called a reflexive message. It is indicated with a message arrow that starts and ends at the same lifeline as shown in the example below.



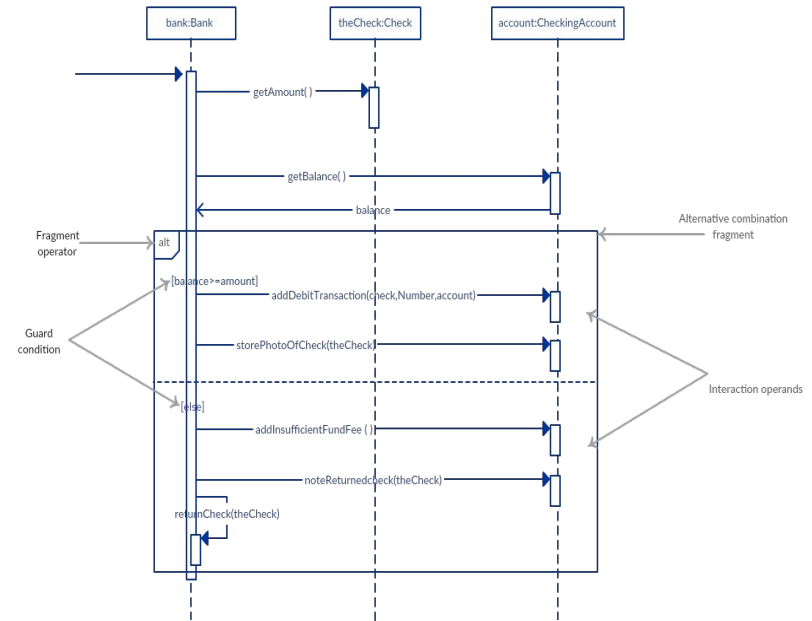
Comment

- UML diagrams generally permit the annotation of comments in all UML diagram types. The comment object is a rectangle with a folded-over corner as shown below. The comment can be linked to the related object with a dashed line.



Alternatives

- The alternative combination fragment is used when a choice needs to be made between two or more message sequences. It models the “if then else” logic.
- The alternative fragment is represented by a large rectangle or a frame; it is specified by mentioning ‘alt’ inside the frame’s name box (a.k.a. fragment operator).
- To show two or more alternatives, the larger rectangle is then divided into what is called interaction operands using a dashed line, as shown in the sequence diagram example above. Each operand has a guard to test against and it is placed at the top left corner of the operand.



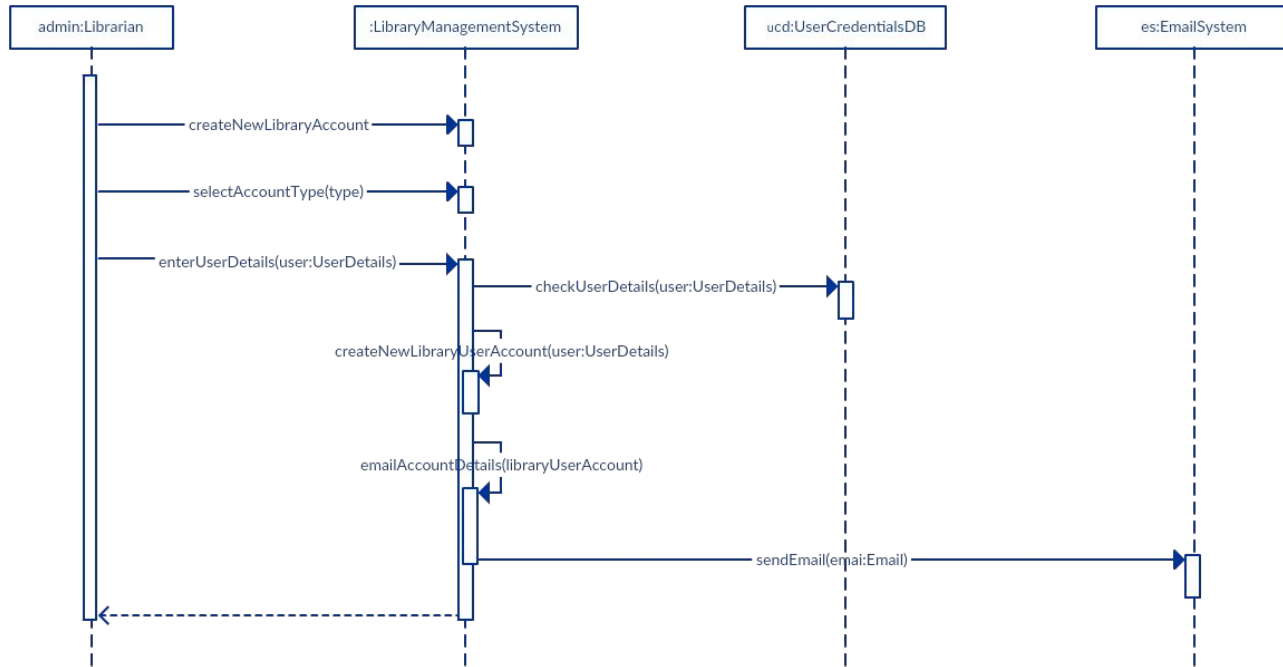
Options

- The option combination fragment is used to indicate a sequence that will only occur under a certain condition, otherwise, the sequence won't occur. It models the "if then" statement.
- Similar to the alternative fragment, the option fragment is also represented with a rectangular frame where 'opt' is placed inside the name box.
- Unlike the alternative fragment, an option fragment is not divided into two or more operands. Option's guard is placed at the top left corner.
- (Find an example sequence diagram with an option fragment in the [Sequence Diagram Templates and Examples](#) section).

Loops

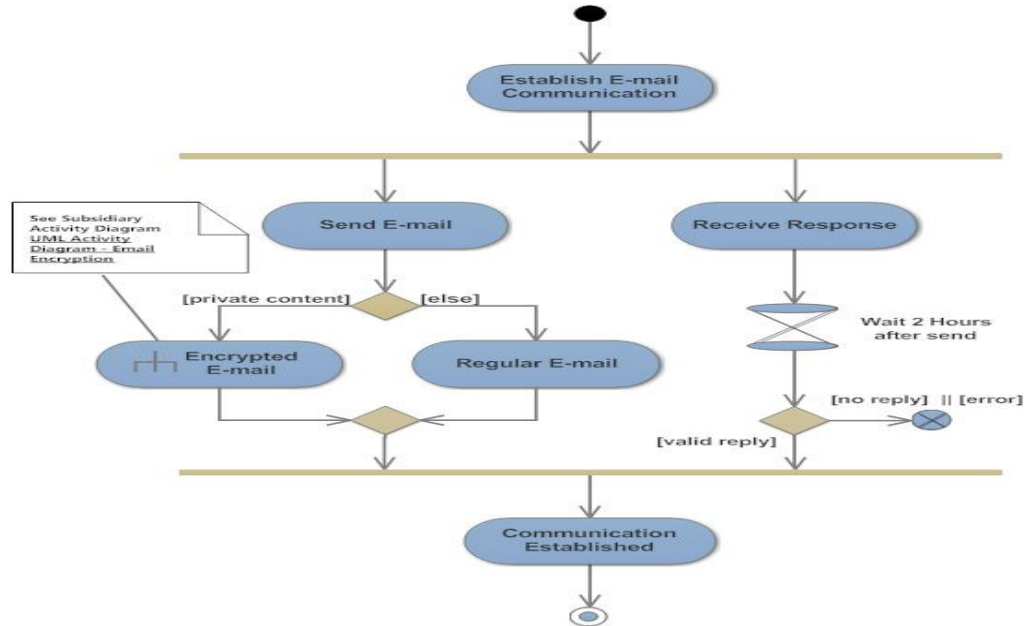
- Loop fragment is used to represent a repetitive sequence. Place the words 'loop' in the name box and the guard condition near the top left corner of the frame.
- In addition to the Boolean test, the guard in a loop fragment can have two other special conditions tested against. These are minimum iterations (written as `minint = [the number]` and maximum iterations (written as `maxint = [the number]`).
- If it is a minimum iterations guard, the loop must execute not less than the number mentioned, and if it is a maximum iterations guard, the loop mustn't execute more than the number indicated.

How to Draw a Sequence Diagram



Activity Diagram
Activity diagram example

UML Activity Diagram: Email Connection



What is an Activity Diagram?

- An activity diagram visually presents a series of actions or flow of control in a system similar to a flowchart or a data flow diagram. Activity diagrams are often used in business process modeling. They can also describe the steps in a use case diagram. Activities modeled can be sequential and concurrent. In both cases an activity diagram will have a beginning (an initial state) and an end (a final state).

Basic Activity Diagram Notations and Symbols

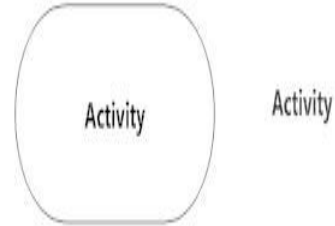
● Initial State or Start Point

A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram. For activity diagram using swimlanes, make sure the start point is placed in the top left corner of the first column.



Activity or Action State

- An action state represents the non-interruptible action of objects. You can draw an action state in SmartDraw using a rectangle with rounded corners.



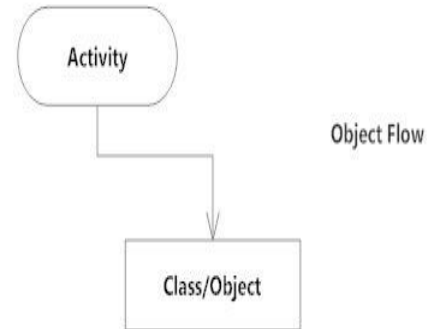
Action Flow

- Action flows, also called edges and paths, illustrate the transitions from one action state to another. They are usually drawn with an arrowed line.



Object Flow

- Object flow refers to the creation and modification of objects by activities. An object flow arrow from an action to an object means that the action creates or influences the object. An object flow arrow from an object to an action indicates that the action state uses the object.



Decisions and Branching

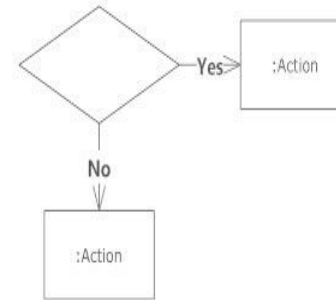
- A diamond represents a decision with alternate paths.
When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities. The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



Decision Symbol

Guards

- In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity. These are not essential, but are useful when a specific answer, such as "Yes, three labels are printed," is needed before moving forward.

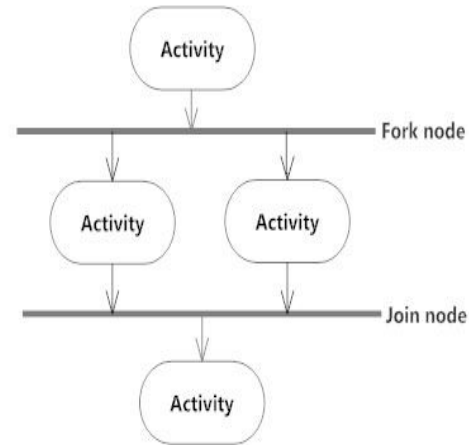


Guard Symbols

Synchronization

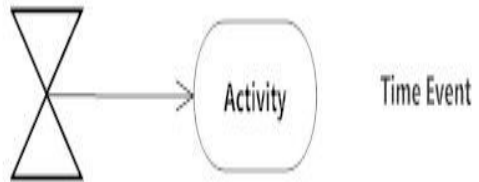
- A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A join node joins multiple concurrent flows back into a single outgoing flow.
- A fork and join mode used together are often referred to as synchronization.

Synchronization



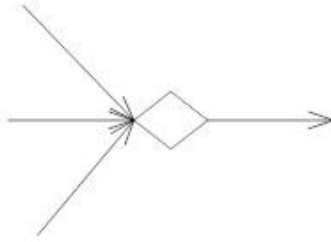
Time Event

- This refers to an event that stops the flow for a time; an hourglass depicts it.



Merge

- A merge brings together multiple flows that are not concurrent.

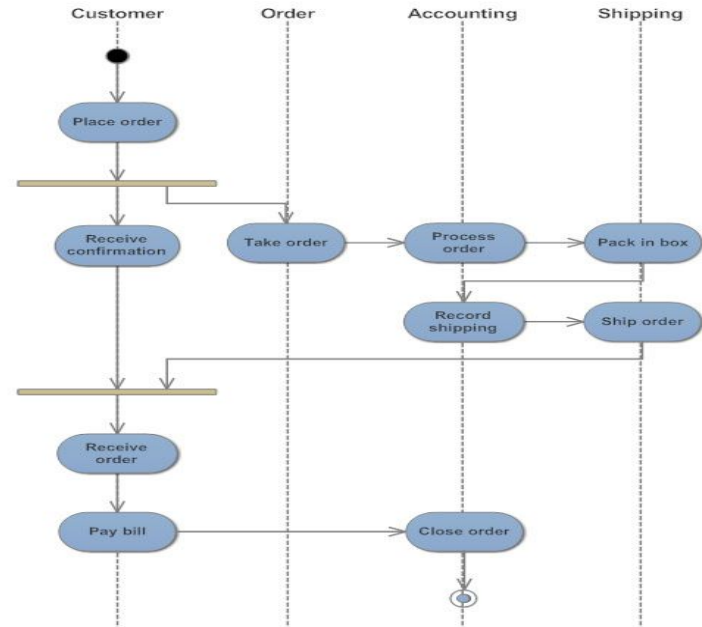


Merge

Swimlanes

- Swimlanes group related activities into one column.

UML Activity Diagram: Order Processing




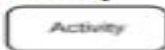
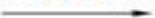






Final State or End Point

- An arrow pointing to a filled circle nested inside another circle represents the final action state.



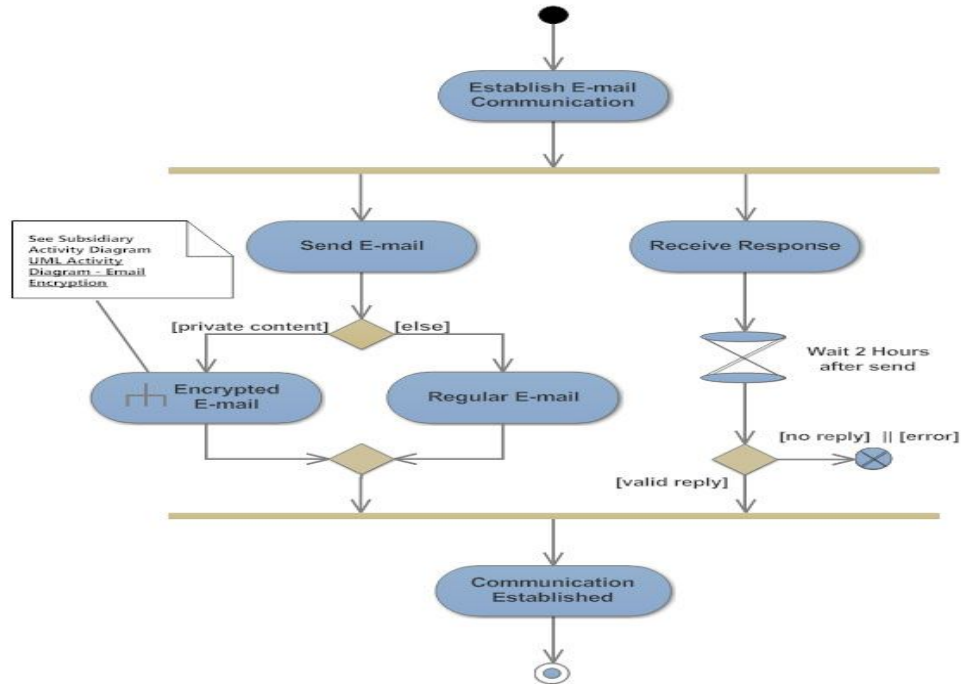
End Point Symbol

Basic Notation of the Activity Diagram

Initial Node 	A black circle is the standard notation for an initial state before an activity takes place. It can either stand alone or you can use a note to further elucidate the starting point.
Activity 	The activity symbols are the basic building blocks of an activity diagram and usually have a short description of the activity they represent.
Control Flow 	Arrows represent the direction flow of the flow chart. The arrow points in the direction of progressing activities.
Branch 	A marker shaped like a diamond is the standard symbol for a decision. There are always at least two paths coming out of a decision and the condition text lets you know which options are mutually exclusive.
Fork 	A fork splits one activity flow into two concurrent activities
Join 	A join combines two concurrent activities back into a flow where only one activity is happening at a time.
	The final flow marker shows the ending point for a process in a flow. The difference between a final flow node and the end state node is that the latter represents the end of all flows in an activity.
 Complete Activity Flow	The black circle that looks like a selected radio button is the UML symbol for the end state of an activity. As shown in two examples above, notes can also be used to explain an end state.
Notes 	The shape used for notes.

Activity Diagram Examples

UML Activity Diagram: Email Connection



UML Activity Diagram

In UML, the activity diagram is used to demonstrate the flow of control within the system rather than the implementation. It models the concurrent and sequential activities.

The activity diagram helps in envisioning the workflow from one activity to another. It put emphasis on the condition of flow and the order in which it occurs. The flow can be sequential, branched, or concurrent, and to deal with such kinds of flows, the activity diagram has come up with a fork, join, etc.

It is also termed as an object-oriented flowchart. It encompasses activities composed of a set of actions or operations that are applied to model the behavioral diagram.

Components of an Activity Diagram

Activities

The categorization of behavior into one or more actions is termed as an activity. In other words, it can be said that an activity is a network of nodes that are connected by edges. The edges depict the flow of execution. It may contain action nodes, control nodes, or object nodes. The control flow of activity is represented by control nodes and object nodes that illustrates the objects used within an activity. The activities are initiated at the initial node and are terminated at the final node.



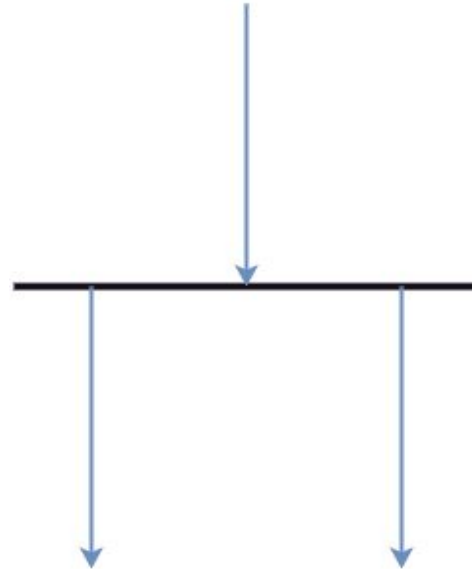
Activity partition /swimlane

The swimlane is used to cluster all the related activities in one column or one row. It can be either vertical or horizontal. It used to add modularity to the activity diagram. It is not necessary to incorporate swimlane in the activity diagram. But it is used to add more transparency to the activity diagram.



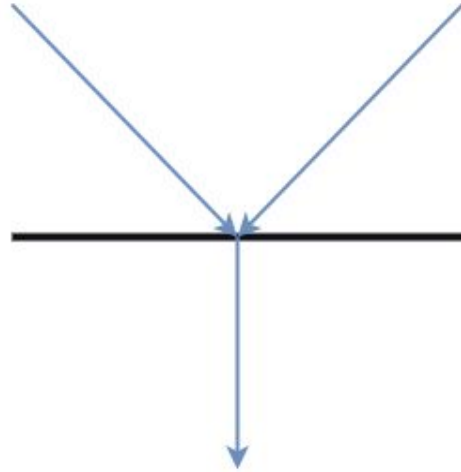
Forks

- Forks and join nodes generate the concurrent flow inside the activity. A fork node consists of one inward edge and several outward edges. It is the same as that of various decision parameters. Whenever a data is received at an inward edge, it gets copied and split crossways various outward edges. It split a single inward flow into multiple parallel flows.



Join Nodes

Join nodes are the opposite of fork nodes. A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.



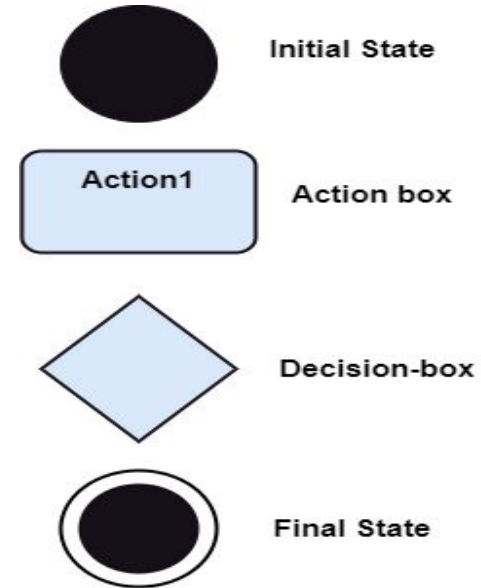
Pins

- It is a small rectangle, which is attached to the action rectangle. It clears out all the messy and complicated thing to manage the execution flow of activities. It is an object node that precisely represents one input to or output from the action.

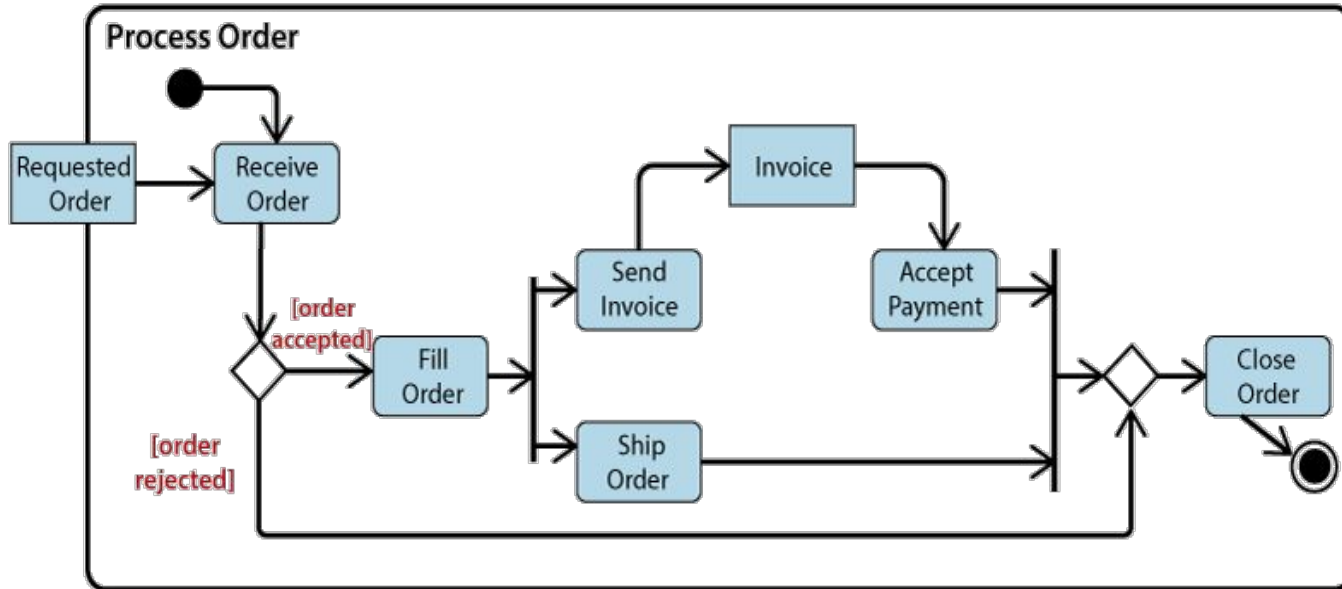
Notation of an Activity diagram

Activity diagram constitutes following notations:

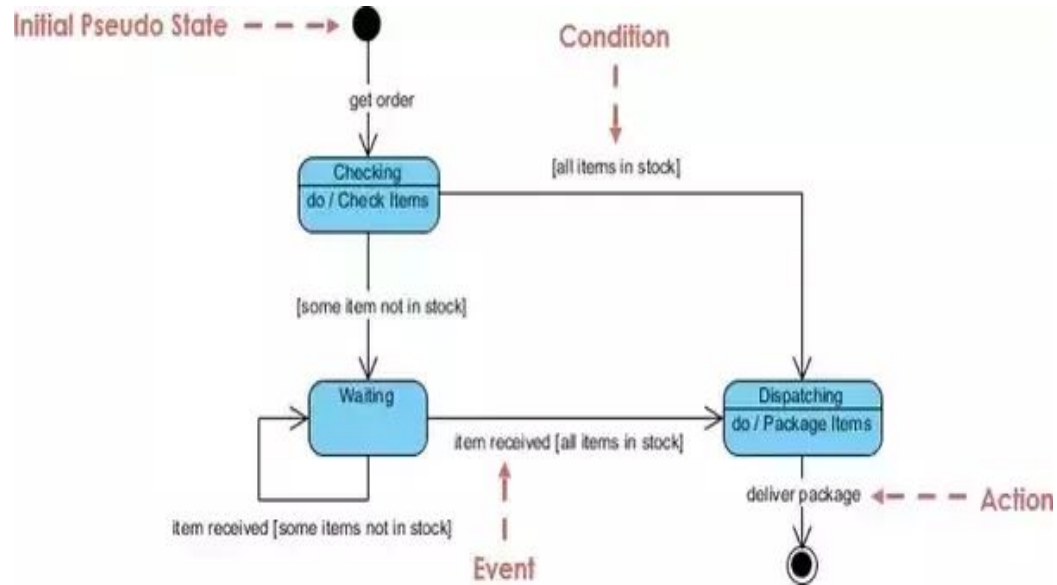
- **Initial State:** It depicts the initial stage or beginning of the set of actions.
- **Final State:** It is the stage where all the control flows and object flows end.
- **Decision Box:** It makes sure that the control flow or object flow will follow only one path.
- **Action Box:** It represents the set of actions that are to be performed.



Example of an Activity Diagram

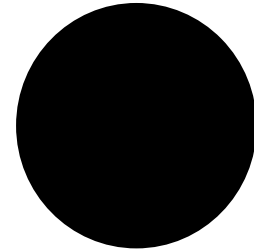


Statechart Diagram



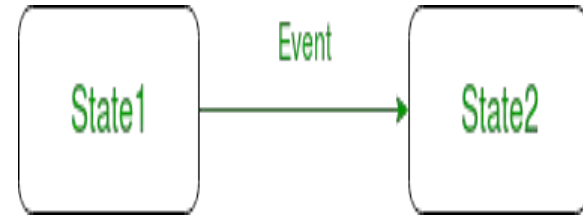
Basic components of a statechart diagram –

- **Initial state** – We use a black filled circle represent the initial state of a System or a class.



components

- **Transition** – We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.



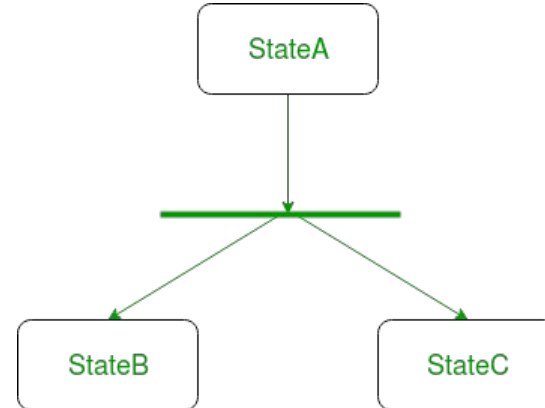
components

- State – We use a rounded rectangle to represent a state.
A state represents the conditions or circumstances of an object of a class at an instant of time.



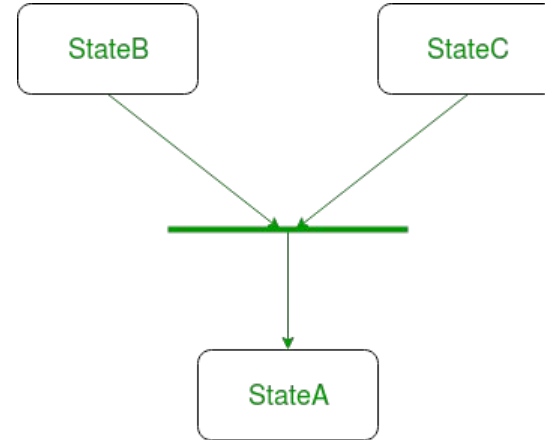
components

- **Fork** – We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



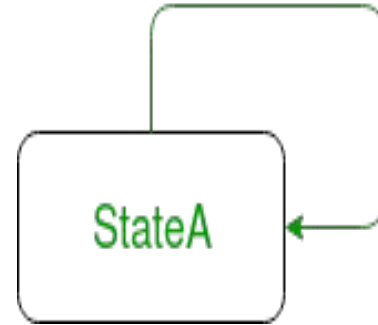
components

- Join – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.



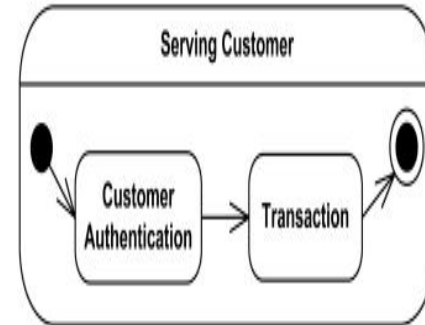
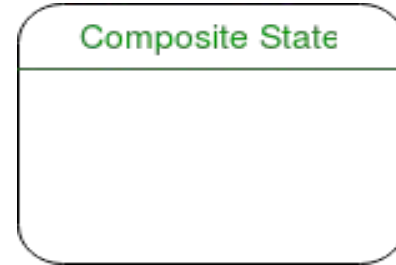
components

- **Self transition** – We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.



components

- **Composite state** – We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.



components

- **Final state** – We use a filled circle within a circle notation to represent the final state in a state machine diagram.



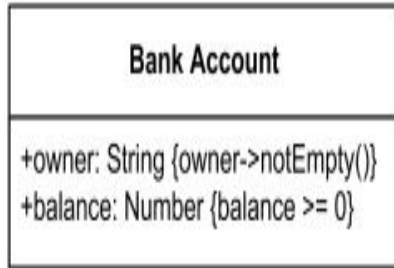
UML Constraint

- A constraint is a packageable element which represents some condition, restriction or assertion related to some element (that owns the constraint) or several elements. Constraint is usually specified by a Boolean expression which must evaluate to a true or false. Constraint must be satisfied (i.e. evaluated to true) by a correct design of the system. Constraints are commonly used for various elements on class diagrams.
- In general there are many possible kinds of owners for a constraint. Owning element must have access to the constrained elements to verify constraint. The owner of the constraint will determine when the constraint is evaluated. For example, operation can have pre-condition and/or a post-condition constraints.
- Constraint could have an optional name, though usually it is anonymous. A constraint is shown as a text string in curly braces according to the following syntax:
- `constraint ::= '{' [name ':'] boolean-expression '}'`

UML specification does not restrict languages which could be used to express constraint. Some examples of constraint languages are:

- OCL(Object Constraint Language)
- Java
- some machine readable language
- natural language
- OCL is a constraint language predefined in UML but if some UML tool is used to draw diagrams, any constraint language supported by that tool could be applied.
- For an element whose notation is a text string (such as a class attribute), the constraint string may follow the element text string in curly braces.

UML Constraint

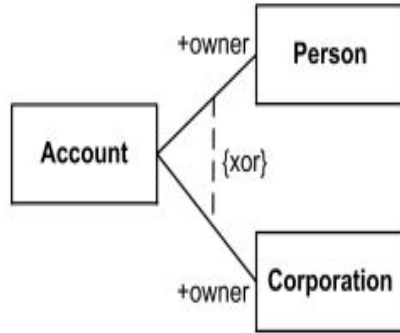


Bank account attribute constraints
- non empty owner and positive balance.

For a constraint that applies to a single element (such as a class or an association path), the constraint string may be placed near the symbol for the element, preferably near the name, if any. A UML tool must make it possible to determine the constrained element.

For a constraint that applies to two elements (such as two classes or two associations), the constraint may be shown as a dashed line between the elements labeled by the constraint string in curly braces.

UML Constraint



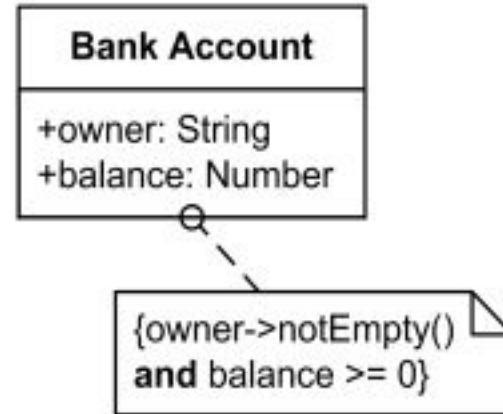
Account owner is either Person or Corporation,
{xor} is predefined UML constraint.

If the constraint is shown as a dashed line between two elements, then an arrowhead may be placed on one end. The direction of the arrow is relevant information within the constraint. The element at the tail of the arrow is mapped to the first position and the element at the head of the arrow is mapped to the second position in the `constrainedElements` collection.

For three or more paths of the same kind (such as generalization paths or association paths), the constraint may be attached to a dashed line crossing all of the paths.

UML Constraint

- The constraint string may be placed in a note symbol (same as used for comments) and attached to each of the symbols for the constrained elements by a dashed line.



Unit 3

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

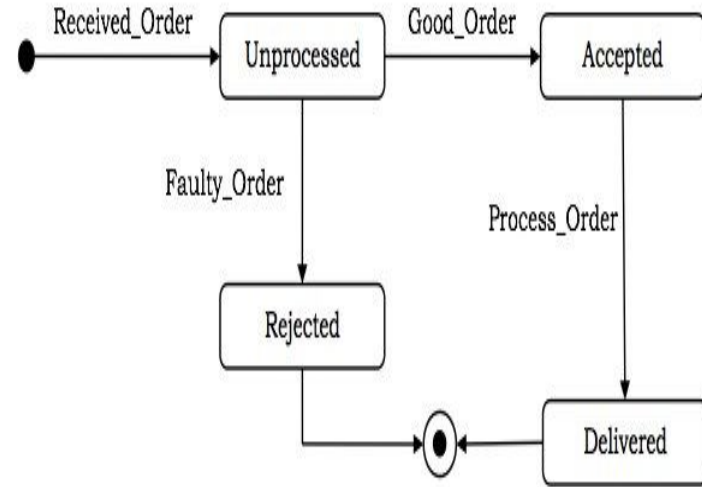
Events causing transitions

Actions due to the events

State-Chart Diagrams

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.



What is a Statechart Diagram?

- Statechart diagrams provide us an efficient way to model the interactions or communication that occur within the external entities and a system. These diagrams are used to model the event-based system. A state of an object is controlled with the help of an event.
- Statechart diagrams are used to describe various states of an entity within the application system.

There are a total of two types of state machine diagram in UML:

Behavioral state machine

- It captures the behavior of an entity present in the system.
- It is used to represent the specific implementation of an element.
- The behavior of a system can be modelled using behavioral state machine diagram in OOAD.

Protocol state machine

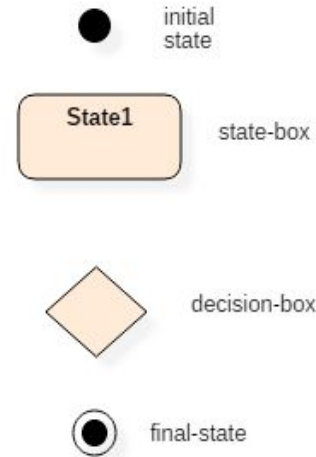
- These diagrams are used to capture the behavior of a protocol.
- It represents how the state of protocol changes concerning the event. It also represents corresponding changes in the system.
- They do not represent the specific implementation of an element.

Why State Machine Diagram?

- Statechart diagram is used to capture the dynamic aspect of a system. State machine diagrams are used to represent the behavior of an application. An object goes through various states during its lifespan. The lifespan of an object remains until the program is terminated. The object goes from multiple states depending upon the event that occurs within the object. Each state represents some unique information about the object.
- Statechart diagrams are used to design interactive systems that respond to either internal or external event. Statechart diagram in UML visualizes the flow of execution from one state to another state of an object.
- It represents the state of an object from the creation of an object until the object is destroyed or terminated.
- The primary purpose of a statechart diagram is to model interactive systems and define each and every state of an object. Statechart diagrams are designed to capture the dynamic behavior of an application system. These diagrams are used to represent various states of a system and entities within the system.

Notation and Symbol for State Machine

- Following are the various notations that are used throughout the state chart diagram. All these notations, when combined, make up a single diagram.



UML state diagram notations

- **Initial state**

The initial state symbol is used to indicate the beginning of a state machine diagram.

- **Final state**

This symbol is used to indicate the end of a state machine diagram.

- **Decision box**

It contains a condition. Depending upon the result of an evaluated guard condition, a new path is taken for program execution.

- **Transition**

A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of an object.

- **State box**

It is a specific moment in the lifespan of an object. It is defined using some condition or a statement within the classifier body. It is used to represent any static as well as dynamic situations.

State box

- It is denoted using a rectangle with round corners. The name of a state is written inside the rounded rectangle.
- The name of a state can also be placed outside the rectangle. This can be done in case of composite or submachine states. One can either place the name of a state within the rectangle or outside the rectangle in a tabular box. One cannot perform both at the same time.
- A state can be either active or inactive. When a state is in the working mode, it is active, as soon as it stops executing and transits into another state, the previous state becomes inactive, and the current state becomes active.

Types of State

- Unified Modeling Language defines three types of states:

- **Simple state**

They do not have any substrate.

- **Composite state**

These types of states can have one or more than one substrate.

A composite state with two or more substates is called an orthogonal state.

- **Submachine state**

These states are semantically equal to the composite states.

Unlike the composite state, we can reuse the submachine states.

How to draw a Statechart diagram?

- Statechart diagrams are used to describe the various state that an object passes through. A transition between one state into another state occurs because of some triggered event. To draw a state diagram in UML, one must identify all the possible states of any particular entity.
- The purpose of these UML diagrams is to represent states of a system. States plays a vital role in state transition diagrams. All the essential object, states, and the events that cause changes within the states must be analyzed first before implementing the diagram.

Following rules must be considered while drawing a state chart diagram:

- The name of a state transition must be unique.
- The name of a state must be easily understandable and describe the behavior of a state.
- If there are multiple objects, then only essential objects should be implemented.
- Proper names for each transition and an event must be given.

When to use State Diagrams?

State diagrams are used to implement real-life working models and object-oriented systems in depth. These diagrams are used to compare the dynamic and static nature of a system by capturing the dynamic behavior of a system.

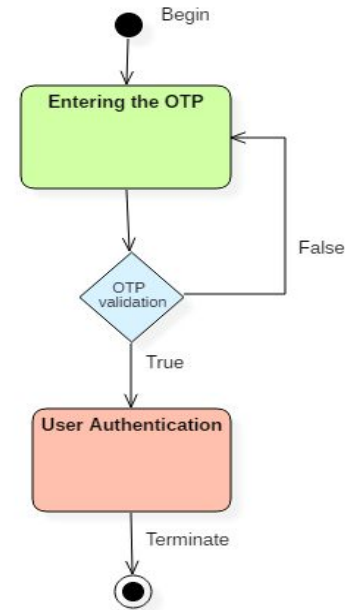
Statechart diagrams are used to capture the changes in various entities of the system from start to end. They are used to analyze how an event can trigger change within multiple states of a system.

State char diagrams are used,

- To model objects of a system.
- To model and implement interactive systems.
- To display events that trigger changes within the states.

Example of State Machine

- Following state diagram example chart represents the user authentication process.
- There are a total of two states, and the first state indicates that the OTP has to be entered first. After that, OTP is checked in the decision box, if it is correct, then only state transition will occur, and the user will be validated. If OTP is incorrect, then the transition will not take place, and it will again go back to the beginning state until the user enters the correct OTP as shown in the above state machine diagram example.



State machine vs. Flowchart

Statemachine	FlowChart
It represents various states of a system.	The Flowchart illustrates the program execution flow.
The state machine has a WAIT concept, i.e., wait for an action or an event.	The Flowchart does not deal with waiting for a concept.
State machines are used for a live running system.	Flowchart visualizes branching sequences of a system.
The state machine is a modeling diagram.	A flowchart is a sequence flow or a DFD diagram.
The state machine can explore various states of a system.	Flowchart deal with paths and control flow.

Statechart diagrams are also called as state machine diagrams.

These diagrams are used to model the event-based system.

A state of an entity is controlled with the help of an event.

There is a total of two types of state machine diagrams: 1) Behavioral 2) State machine 3) Protocol state machine

Statechart diagram is used to capture the dynamic aspect of a system.

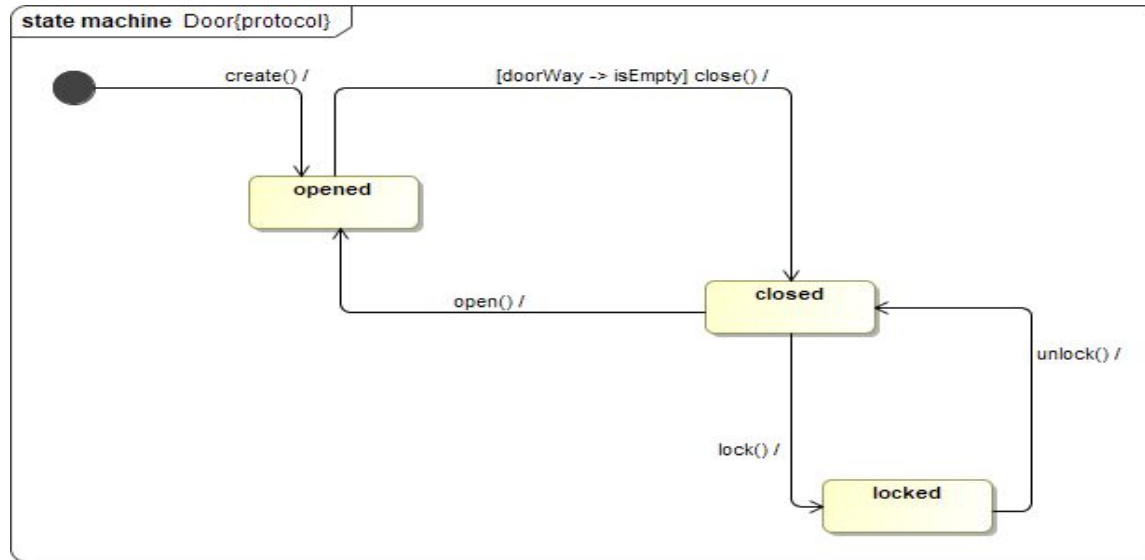
A state is a specific moment in the lifespan of an object.

Behavioral State

- State in behavioral state machines models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some behavior (i.e., the model element under consideration enters the state when the behavior commences and leaves it as soon as the behavior is completed).
- Inherited states are drawn with dashed lines or gray-toned lines.
- The UML defines the following kinds of states:
 - simple state,
 - composite state,
 - submachine state.

Link-<https://www.uml-diagrams.org/state-machine-diagrams-reference.html>

Protocol State Diagram



The UML allows to define so-called Protocol State Machines (PSM) to define the correct usage of the operations defined by a classifier. Using PSMs, the valid call-sequences of operations can be specified using not only states and transitions, but also using guards and post-conditions.

What is an Activity Diagram in UML?

- ACTIVITY DIAGRAM is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system. The basic purpose of activity diagrams is to capture the dynamic behavior of the system.. It is also called object-oriented flowchart.
- This UML diagram focuses on the execution and flow of the behavior of a system instead of implementation. Activity diagrams consist of activities that are made up of actions that apply to behavioral modeling technology.

Components of Activity Diagram

● Activities

It is a behavior that is divided into one or more actions. Activities are a network of nodes connected by edges. There can be action nodes, control nodes, or object nodes. Action nodes represent some action. Control nodes represent the control flow of an activity. Object nodes are used to describe objects used inside an activity. Edges are used to show a path or a flow of execution. Activities start at an initial node and terminate at a final node.

Activity partition/swimlane

An activity partition or a swimlane is a high-level grouping of a set of related actions. A single partition can refer to many things, such as classes, use cases, components, or interfaces.

If a partition cannot be shown clearly, then the name of a partition is written on top of the name of an activity.

Components of Activity Diagram

● Fork and Join nodes

Using a fork and join nodes, concurrent flows within an activity can be generated. A fork node has one incoming edge and numerous outgoing edges. It is similar to one too many decision parameters. When data arrives at an incoming edge, it is duplicated and split across numerous outgoing edges simultaneously. A single incoming flow is divided into multiple parallel flows.

A join node is opposite of a fork node as It has many incoming edges and a single outgoing edge. It performs logical AND operation on all the incoming edges. This helps you to synchronize the input flow across a single output edge.

Components of Activity Diagram

● Pins

An activity diagram that has a lot of flows gets very complicated and messy.

Pins are used to clearing up the things. It provides a way to manage the execution flow of activity by sorting all the flows and cleaning up messy thins. It is an object node that represents one input to or an output from an action.

Both input and output pins have precisely one edge. pin is an object node for inputs and outputs to actions.

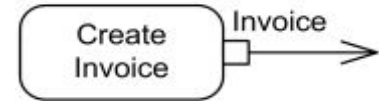
Pin is usually shown as a small rectangle attached to the action rectangle. The name of the pin can be displayed near the pin.

Pins

Item is input pin to the Add to Shopping Cart action.



Invoice is output pin from the Create Invoice action.



Why use Activity Diagrams?

Activity diagram in UML allows you to create an event as an activity which contains a collection of nodes joined by edges. An activity can be attached to any modeling element to model its behavior. Activity diagrams are used to model,

- Use cases
- Classes
- Interfaces
- Components
- Collaborations

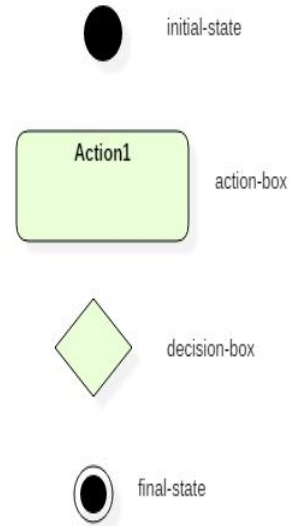
Activity diagrams are used to model processes and workflows. The essence of a useful activity diagram is focused on communicating a specific aspect of a system's dynamic behavior. Activity diagrams capture the dynamic elements of a system.

Activity diagram is similar to a flowchart that visualizes flow from one activity to another activity. Activity diagram is identical to the flowchart, but it is not a flowchart. The flow of activity can be controlled using various control elements in the UML flow diagram. In simple words, an activity diagram is used to activity diagrams that describe the flow of execution between multiple activities.

Activity Diagram Notations

Activity diagrams symbols can be generated by using the following notations:

- **Initial states:** The starting stage before an activity takes place is depicted as the initial state
- **Final states:** The state which the system reaches when a specific process ends is known as a Final State
- State or an activity box:
- **Decision box:** It is a diamond shape box which represents a decision with alternate paths. It represents the flow of control.



How to draw an activity diagram?

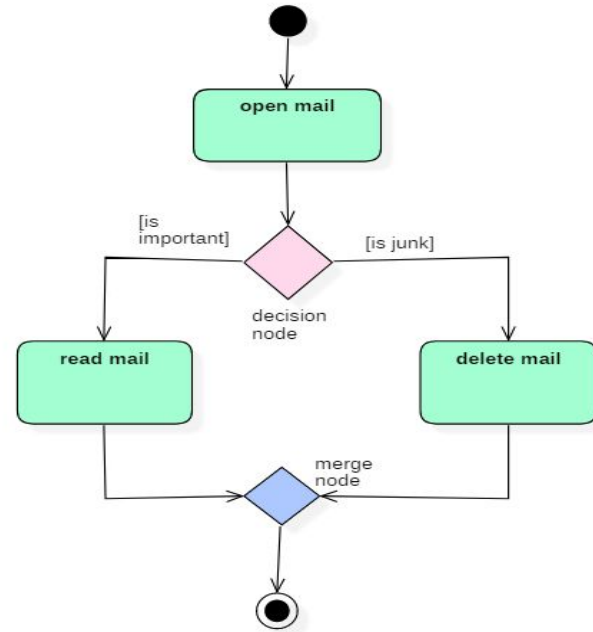
- Activity diagram is a flowchart of activities. It represents the workflow between various system activities. Activity diagrams are similar to the flowcharts, but they are not flowcharts. Activity diagram is an advancement of a flowchart that contains some unique capabilities.
- Activity diagrams include swimlanes, branching, parallel flow, control nodes, expansion nodes, and object nodes. Activity diagram also supports exception handling.
- To draw an activity diagram, one must understand and explore the entire system. All the elements and entities that are going to be used inside the diagram must be known by the user. The central concept which is nothing but an activity must be clear to the user. After analyzing all activities, these activities should be explored to find various constraints that are applied to activities. If there is such a constraint, then it should be noted before developing an activity diagram.

Following rules must be followed while developing an activity diagram,

- All activities in the system should be named.
- Activity names should be meaningful.
- Constraints must be identified.
- Activity associations must be known.

Example of Activity Diagram

- Let us consider mail processing activity as a sample for Activity Diagram. Following diagram represents activity for processing e-mails.
- In the above activity diagram, three activities are specified. When the mail checking process begins user checks if mail is important or junk. Two guard conditions [is essential] and [is junk] decides the flow of execution of a process. After performing the activity, finally, the process is terminated at termination node.



When Use Activity Diagram

- Activity diagram is used to model business processes and workflows. These diagrams are used in software modeling as well as business modeling.
- Most commonly activity diagrams are used to,
- Model the workflow in a graphical way, which is easily understandable.
- Model the execution flow between various entities of a system.
- Model the detailed information about any function or an algorithm which is used inside the system.
- Model business processes and their workflows.
- Capture the dynamic behavior of a system.
- Generate high-level flowcharts to represent the workflow of any application.
- Model high-level view of an object-oriented or a distributed system.

Summary

- Activity diagram is also called as object-oriented flowcharts.
- Activity diagrams consist of activities that are made up of smaller actions.
- Activity is a behavior that is divided into one or more actions.
- It uses action nodes, control nodes and object nodes.
- An activity partition or a swimlane is a high-level grouping of a set of related actions.
- Fork and join nodes are used to generate concurrent flows within an activity.
- Activity diagram is used to model business processes and workflows.

Interaction Diagrams

- From the term Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behavior of the system.
- This interactive behavior is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**. The basic purpose of both the diagrams are similar.
- Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

Purpose of Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

The purpose of interaction diagram is –

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

How to Draw an Interaction Diagram?

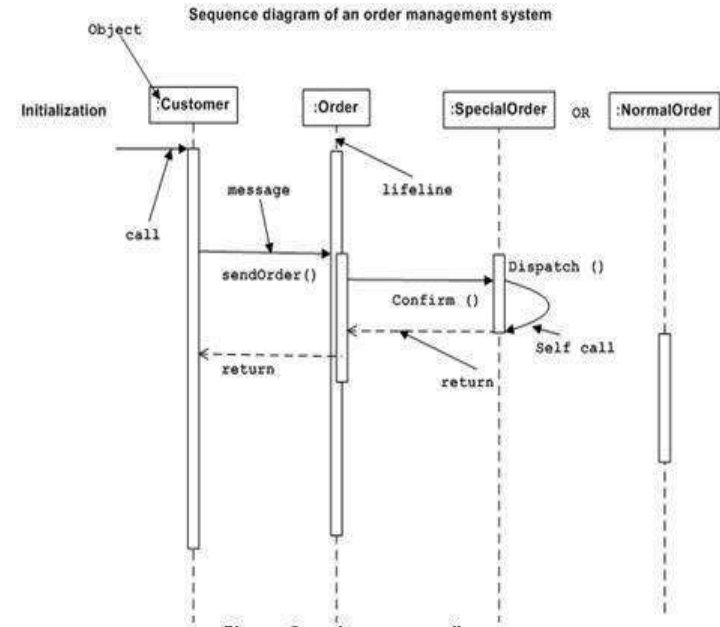
As we have already discussed, the purpose of interaction diagrams is to capture the dynamic aspect of a system. So to capture the dynamic aspect, we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snapshot of the running system at a particular moment

Following things are to be identified clearly before drawing the interaction diagram

- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.
- Following are two interaction diagrams modeling the order management system. The first diagram is a sequence diagram and the second is a collaboration diagram

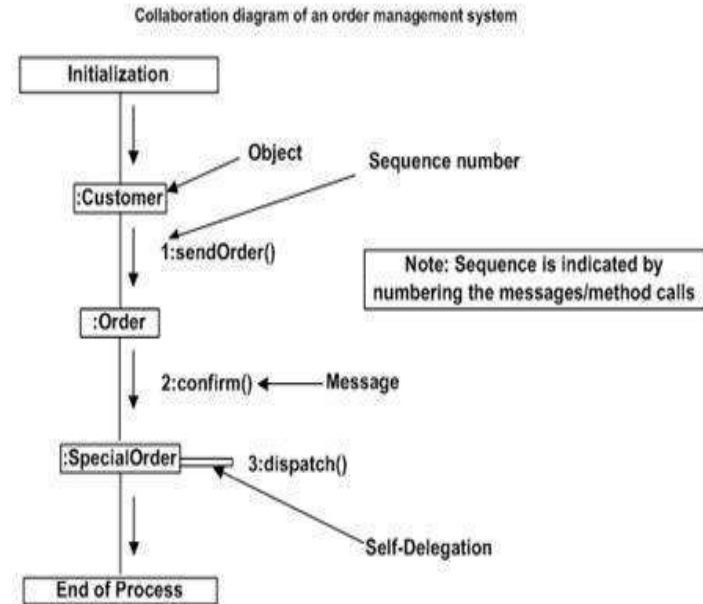
The Sequence Diagram

- The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).
- The following diagram shows the message sequence for SpecialOrder object and the same can be used in case of NormalOrder object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.
- The first call is sendOrder () which is a method of Order object. The next call is confirm () which is a method of SpecialOrder object and the last call is Dispatch () which is a method of SpecialOrder object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.



The Collaboration Diagram

- The second interaction diagram is the collaboration diagram. It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.
- Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.
- To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used



Where to Use Interaction Diagrams?

We have already discussed that interaction diagrams are used to describe the dynamic nature of a system. Now, we will look into the practical scenarios where these diagrams are used. To understand the practical application, we need to understand the basic nature of sequence and collaboration diagram.

The main purpose of both the diagrams are similar as they are used to capture the dynamic behavior of a system. However, the specific purpose is more important to clarify and understand.

Sequence diagrams are used to capture the order of messages flowing from one object to another. Collaboration diagrams are used to describe the structural organization of the objects taking part in the interaction. A single diagram is not sufficient to describe the dynamic aspect of an entire system, so a set of diagrams are used to capture it as a whole.

Interaction diagrams are used when we want to understand the message flow and the structural organization. Message flow means the sequence of control flow from one object to another. Structural organization means the visual organization of the elements in a system.

Interaction diagrams can be used –

- To model the flow of control by time sequence.
- To model the flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

What is Sequence Diagram?

- Sequence Diagrams are interaction diagrams that detail how operations are carried out. They capture the interaction between objects in the context of a collaboration. Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when.

Sequence Diagrams captures:

- the interaction that takes place in a collaboration that either realizes a use case or an operation (instance diagrams or generic diagrams)
- high-level interactions between user of the system and the system, between the system and other systems, or between subsystems (sometimes known as system sequence diagrams)

Purpose of Sequence Diagram

- Model high-level interaction between active objects in a system
- Model the interaction between object instances within a collaboration that realizes a use case
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of a interaction (showing just one path through the interaction)

Sequence Diagrams at a Glance

- Sequence Diagrams show elements as they interact over time and they are organized according to object (horizontally) and time (vertically):

Object Dimension

- The horizontal axis shows the elements that are involved in the interaction
- Conventionally, the objects involved in the operation are listed from left to right according to when they take part in the message sequence. However, the elements on the horizontal axis may appear in any order

Time Dimension

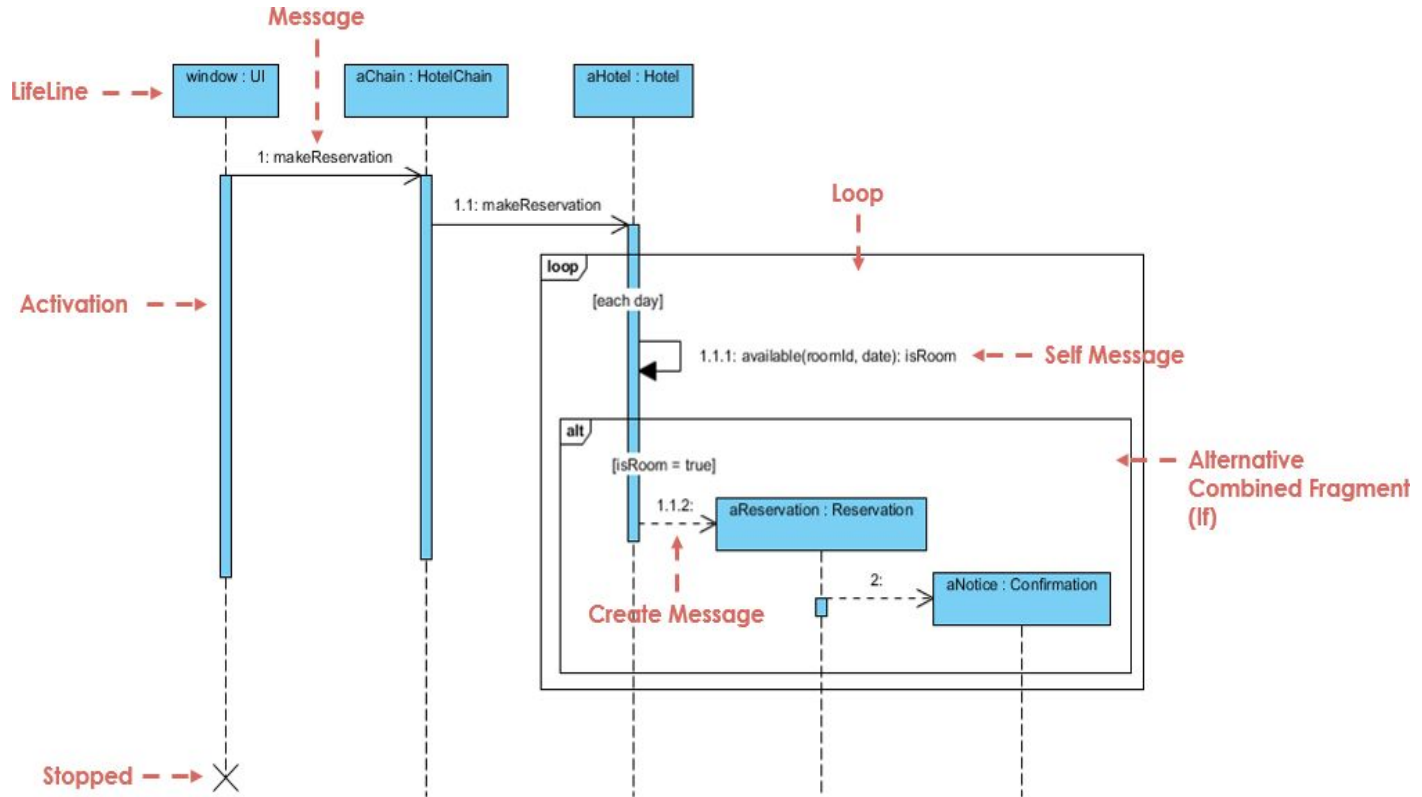
- The vertical axis represents time proceedings (or progressing) down the page.

Note that:


- Time in a sequence diagram is all a about ordering, not duration. The vertical space in an interaction diagram is not relevant for the duration of the interaction.

Sequence Diagram Example: Hotel System


- Sequence Diagram is an interaction diagram that details how operations are carried out -- what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence.
- Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.



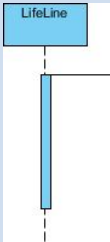
Sequence Diagram Notation

Notation Description	Visual Representation
<p>Actor</p> <p>a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data)</p> <p>external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject).</p> <p>represent roles played by human users, external hardware, or other subjects.</p> <p>Note that:</p> <p>An actor does not necessarily represent a specific physical entity but merely a particular role of some entity</p> <p>A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person.</p>	


Sequence Diagram Notation

Notation Description	Visual Representation
<p>Lifeline</p> <p>A lifeline represents an individual participant in the Interaction.</p>	 <p>The diagram shows a single lifeline. It consists of a light blue rectangular box at the top with the text 'LifeLine' inside. A vertical dashed line extends downwards from the bottom of this box, representing the duration of the participant's existence in the interaction.</p>

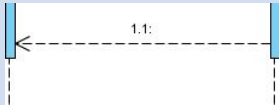
Sequence Diagram Notation

Notation Description	Visual Representation
<p>Activations</p> <p>A thin rectangle on a lifeline) represents the period during which an element is performing an operation.</p> <p>The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively</p>	 <p>The diagram illustrates a single lifeline labeled 'LifeLine'. A thin, light blue vertical rectangle is drawn on the lifeline, representing an activation period. The top and bottom edges of this rectangle are aligned with the start and end of a specific operation on the lifeline.</p>


Sequence Diagram Notation

Notation Description	Visual Representation
<p>Call Message</p> <p>A message defines a particular communication between Lifelines of an Interaction.</p> <p>Call message is a kind of message that represents an invocation of operation of target lifeline.</p>	 <p>The diagram shows a call message between two lifelines. A horizontal arrow points from a small blue rectangle on the left lifeline to a larger blue rectangle on the right lifeline. The arrow is labeled '1: message' above it. The right lifeline is enclosed in a dashed rectangular box, indicating a call to an external component.</p>

Sequence Diagram Notation

Notation Description	Visual Representation
<p>Return Message</p> <p>A message defines a particular communication between Lifelines of an Interaction.</p> <p>Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.</p>	 <p>The diagram illustrates a return message between two lifelines. Two vertical blue bars represent the lifelines. A dashed arrow points from the right lifeline back to the left lifeline. The arrow is labeled '1:1' in the center. The arrow starts at the bottom of the right lifeline and ends at the bottom of the left lifeline.</p>

Sequence Diagram Notation

Notation Description	Visual Representation
<p>Recursive Message</p> <p>A message defines a particular communication between Lifelines of an Interaction.</p> <p>Recursive message is a kind of message that represents the invocation of message of the same lifeline. It's target points to an activation on top of the activation where the message was invoked from.message.</p>	

Sequence Diagram Notation

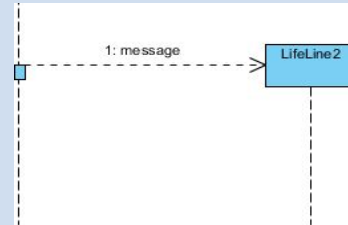
Notation Description

Create Message

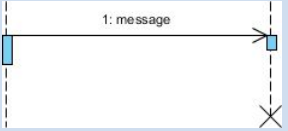
A message defines a particular communication between Lifelines of an Interaction.

Create message is a kind of message that represents the instantiation of (target) lifeline.

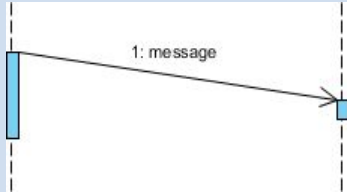
Visual Representation




Sequence Diagram Notation

Notation Description	Visual Representation
<p>Destroy Message</p> <p>A message defines a particular communication between Lifelines of an Interaction.</p> <p>Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.</p>	 <p>The diagram illustrates a destroy message between two lifelines. On the left, a lifeline is represented by a vertical dashed line with a small blue rectangle (activation bar) at the top. On the right, another lifeline is represented by a vertical dashed line with a small blue rectangle at the top and an 'X' at the bottom, indicating its destruction. A horizontal arrow points from the activation bar of the left lifeline to the top of the right lifeline. Above the arrow, the text '1: message' is written. The entire diagram is enclosed in a light blue rectangular frame.</p>

Sequence Diagram Notation

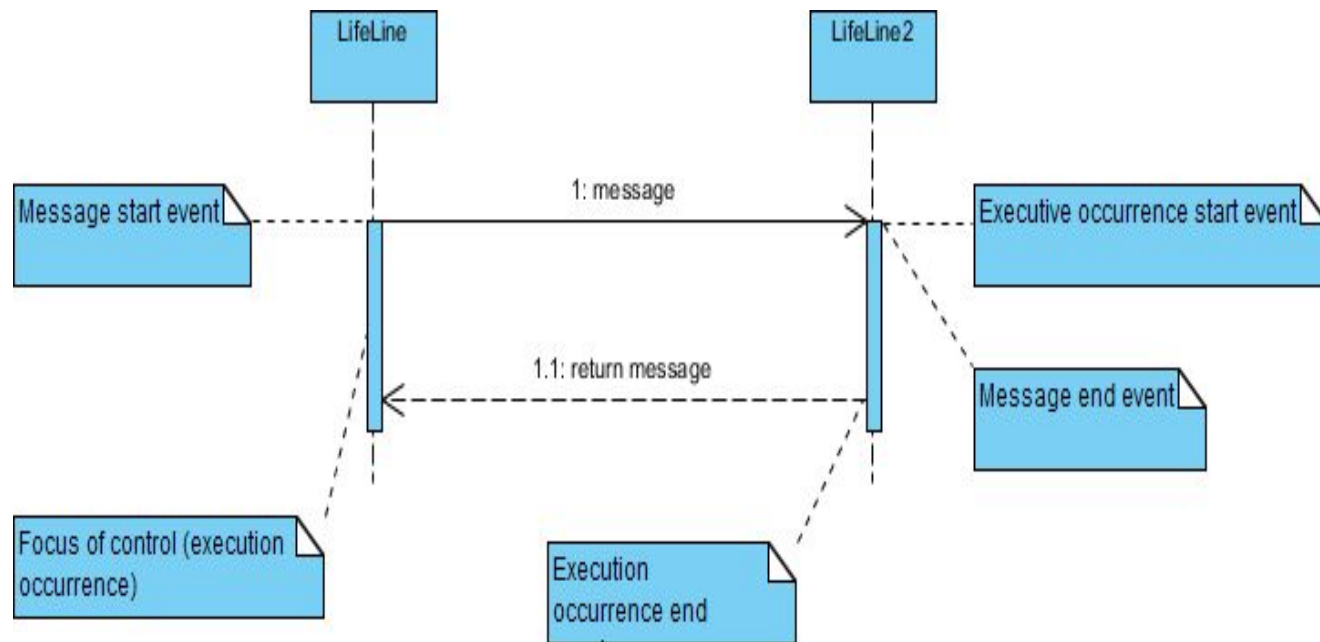
Notation Description	Visual Representation
<p>Duration Message</p> <p>A message defines a particular communication between Lifelines of an Interaction.</p> <p>Duration message shows the distance between two time instants for a message invocation.</p>	 <p>The diagram illustrates a duration message between two lifelines. Two vertical dashed lines represent the lifelines. A solid line with an open arrowhead connects a point on the first lifeline to a point on the second lifeline. The text "1: message" is placed above the arrow. A light blue rectangular box is drawn on the first lifeline, spanning the vertical distance between the start and end points of the message arrow, representing the duration of the message.</p>

Sequence Diagram Notation

Notation Description	Visual Representation
<p>Note</p> <p>A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.</p>	

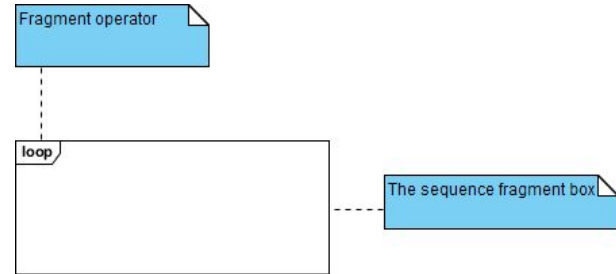
Message and Focus of Control

- An Event is any point in an interaction where something occurs.
- Focus of control: also called execution occurrence, an execution occurrence
- It shows as tall, thin rectangle on a lifeline)
- It represents the period during which an element is performing an operation. The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively.



Sequence Fragments

- introduces sequence (or interaction) fragments. Sequence fragments make it easier to create and maintain accurate sequence diagrams
- A sequence fragment is represented as a box, called a combined fragment, which encloses a portion of the interactions within a sequence diagram
- The fragment operator (in the top left corner) indicates the type of fragment
- Fragment types: ref, assert, loop, break, alt, opt, neg

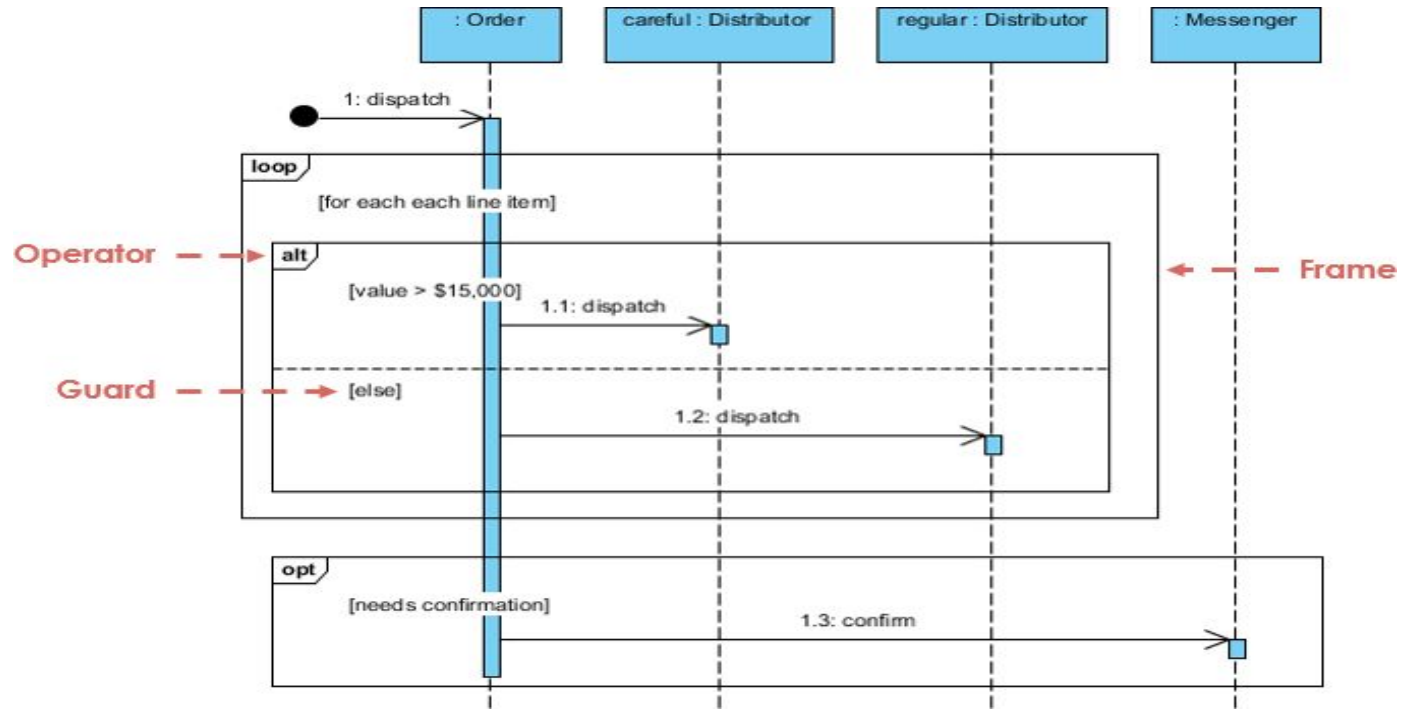


Operator	Fragment Type
alt	Alternative multiple fragments: only the one whose condition is true will execute.
opt	Optional: the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
par	Parallel: each fragment is run in parallel.
loop	Loop: the fragment may execute multiple times, and the guard indicates the basis of iteration.
region	Critical region: the fragment can have only one thread executing it at once.
neg	Negative: the fragment shows an invalid interaction.
ref	Reference: refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram: used to surround an entire sequence diagram.

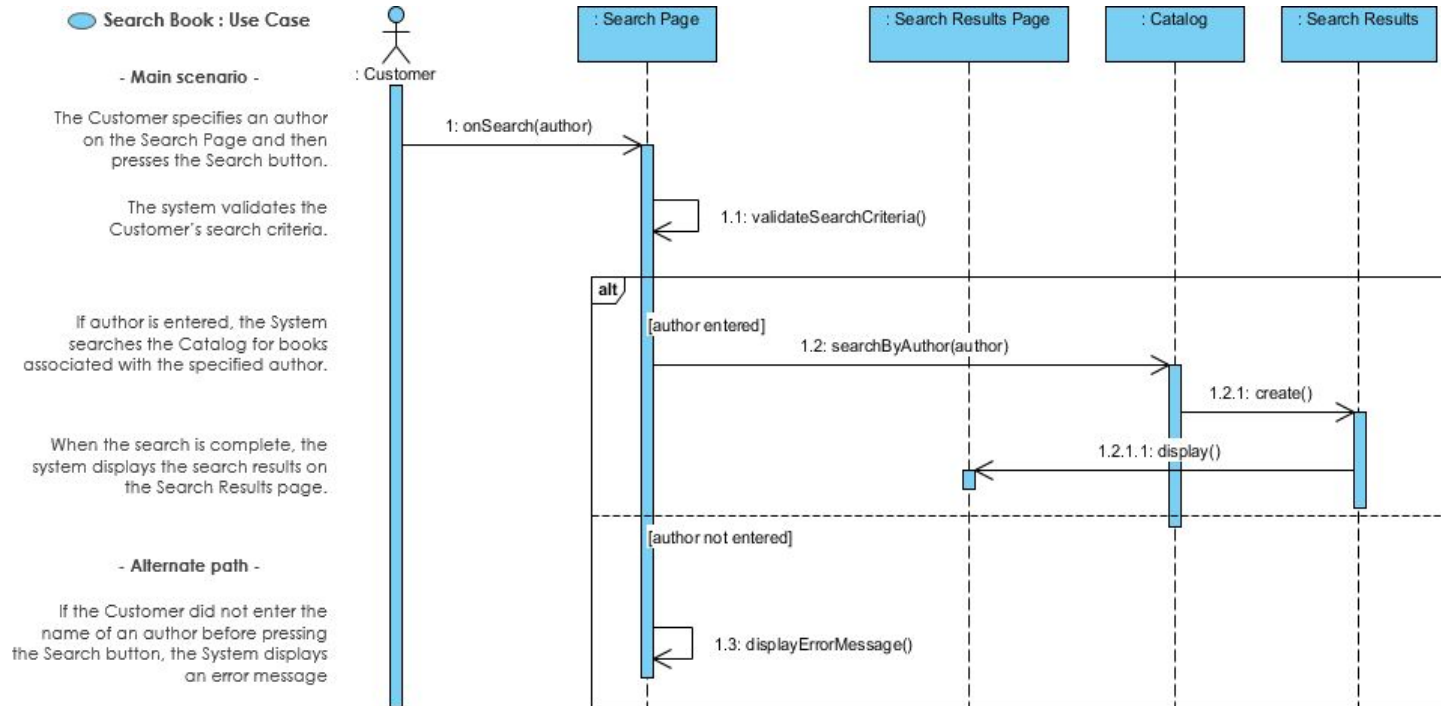
- Note That:

- It is possible to combine frames in order to capture, e.g., loops or branches.
- Combined fragment keywords: alt, opt, break, par, seq, strict, neg, critical, ignore, consider, assert and loop.
- Constraints are usually used to show timing constraints on messages. They can apply to the timing of one message or intervals between messages.

Combined Fragment Example



Sequence Diagram for Modeling Use Case Scenarios



Sequence Diagram - Model before Code

- A good sequence diagram is still a bit above the level of the real code
- Sequence diagrams are language neutral
- Non-coders can do sequence diagrams
- Easier to do sequence diagrams as a team

collaboration diagram

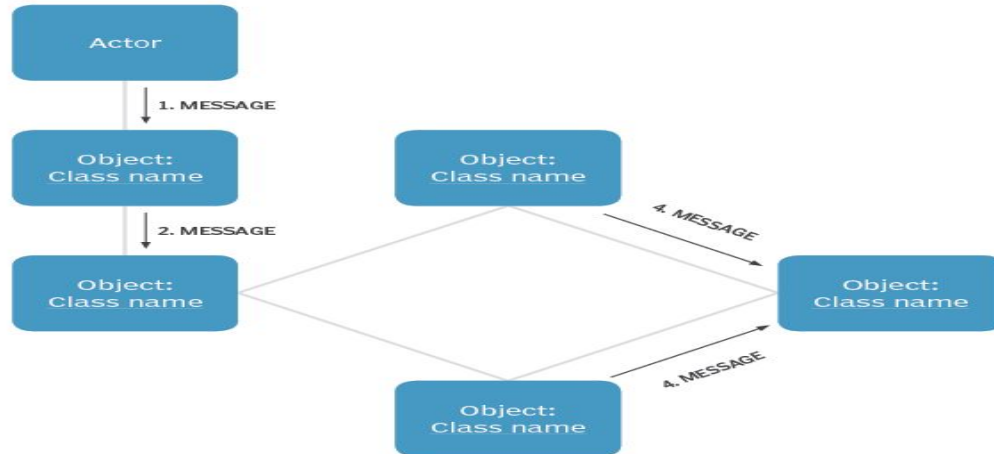
- A collaboration diagram, also known as a communication diagram, is an illustration of the relationships and interactions among software objects in the Unified Modeling Language (UML). These diagrams can be used to portray the dynamic behavior of a particular use case and define the role of each object.
- Collaboration diagrams are created by first identifying the structural elements required to carry out the functionality of an interaction. A model is then built using the relationships between those elements. Several vendors offer software for creating and editing collaboration diagrams.

Notations of a collaboration diagram

- **Objects**- Objects are shown as rectangles with naming labels inside. The naming label follows the convention of object name: class name. If an object has a property or state that specifically influences the collaboration, this should also be noted.
- **Actors**- Actors are instances that invoke the interaction in the diagram. Each actor has a name and a role, with one actor initiating the entire use case.
- **Links**- Links connect objects with actors and are depicted using a solid line between two elements. Each link is an instance where messages can be sent.
- **messages**- Messages between objects are shown as a labeled arrow placed near a link. These messages are communications between objects that convey information about the activity and can include the sequence number.

The most important objects are placed in the center of the diagram, with all other participating objects branching off. After all objects are placed, links and messages should be added in between.

Components of a collaboration diagram



When to use a collaboration diagram

Collaboration diagrams should be used when the relationships among objects are crucial to display. A few examples of instances where collaboration diagrams might be helpful include:

- Modeling collaborations, mechanisms or the structural organization within a system design.
- Providing an overview of collaborating objects within an object-oriented system.
- Exhibiting many alternative scenarios for the same use case.
- Demonstrating forward and reverse engineering.
- Capturing the passage of information between objects.
- Visualizing the complex logic behind an operation.

However, collaboration diagrams are best suited to the portrayal of simple interactions among relatively small numbers of objects. As the number of objects and messages grows, a collaboration diagram can become difficult to read and use efficiently. Additionally, collaboration diagrams typically exclude descriptive information, such as timing.

Collaboration vs sequence diagrams

- In UML, the two types of interaction diagrams are collaboration and sequence diagrams. While both types use similar information, they display them in separate ways. Collaboration diagrams are used to visualize the structural organization of objects and their interactions. Sequence diagrams, on the other hand, focus on the order of messages that flow between objects. However, in most scenarios, a single figure is not sufficient in describing the behavior of a system and both figures are required.

Unit 4

What is Component Diagram-

Component diagrams are used to visualize the organization of system components and the dependency relationships between them. They provide a high-level view of the components within a system.

The components can be a software component such as a database or user interface; or a hardware component such as a circuit, microchip or device; or a business unit such as supplier, payroll or shipping.

Component diagram

- Are used in Component-Based-Development to describe systems with Service-Oriented-Architecture
- Show the structure of the code itself
- Can be used to focus on the relationship between components while hiding specification detail
- Help communicate and explain the functions of the system being built to stakeholders

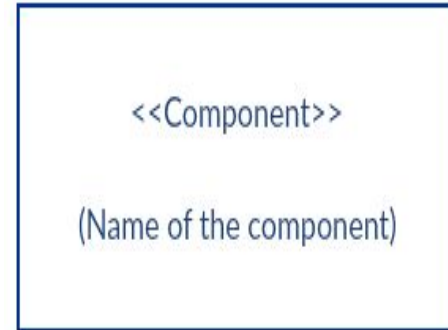
Component Diagram Symbols

- We have explained below the common component diagram notations that are used to draw a component diagram.

Component

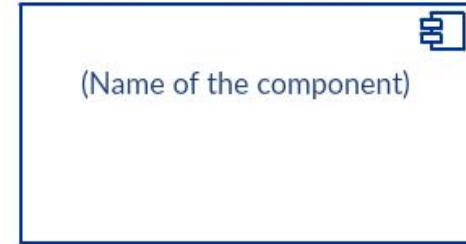
There are three ways the component symbol can be used.

- 1) Rectangle with the component stereotype (the text <<component>>). The component stereotype is usually used above the component name to avoid confusing the shape with a class icon.



Component Diagram Symbols

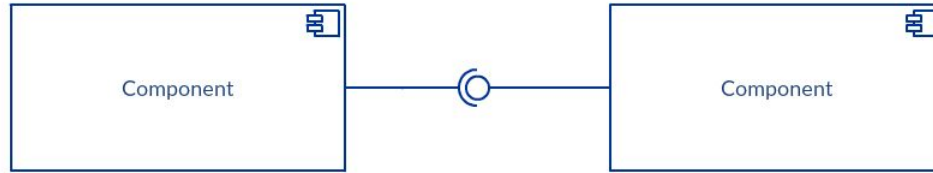
2) Rectangle with the component icon in the top right corner and the name of the component.



3) Rectangle with the component icon and the component stereotype.

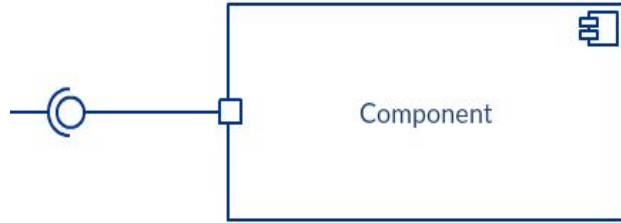


Provided Interface and the Required Interface



Interfaces in component diagrams show how components are wired together and interact with each other. The assembly connector allows linking the component's required interface (represented with a semi-circle and a solid line) with the provided interface (represented with a circle and solid line) of another component. This shows that one component is providing the service that the other is requiring.

Port



Port (represented by the small square at the end of a required interface or provided interface) is used when the component delegates the interfaces to an internal class.

Dependencies



Although you can show more detail about the relationship between two components using the ball-and-socket notation (provided interface and required interface), you can just as well use a dependency arrow to show the relationship between two components.

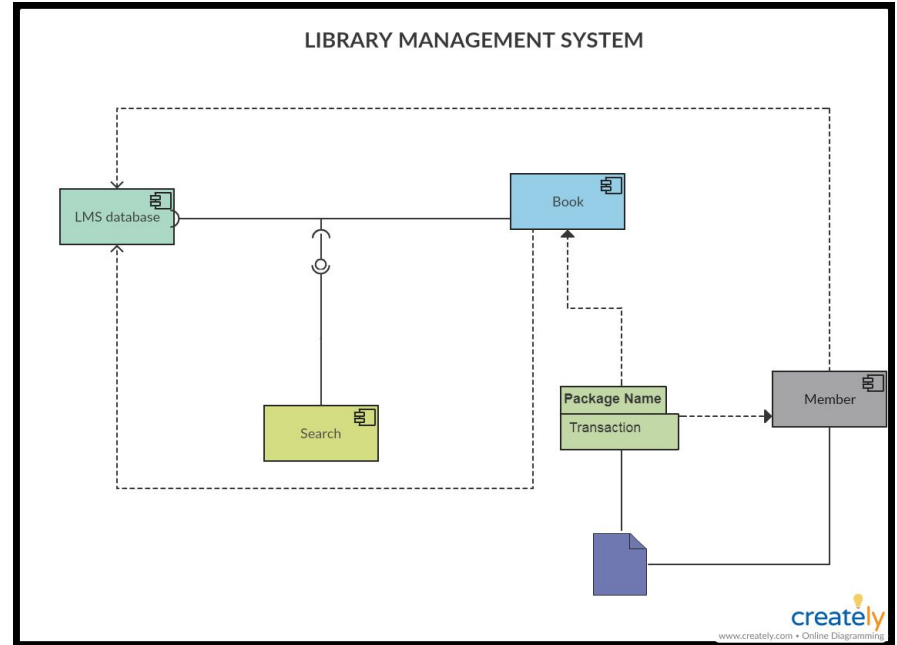
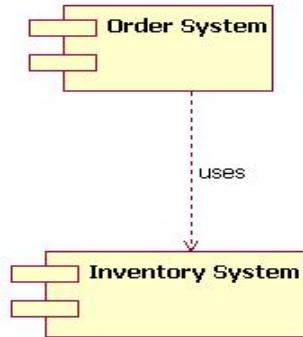
How to Draw a Component Diagram

You can use a component diagram when you want to represent your system as components and want to show their interrelationships through interfaces. It helps you get an idea of the implementation of the system. Following are the steps you can follow when drawing a component diagram.

- Step 1: figure out the purpose of the diagram and identify the artifacts such as the files, documents etc. in your system or application that you need to represent in your diagram.
- Step 2: As you figure out the relationships between the elements you identified earlier, create a mental layout of your component diagram
- Step 3: As you draw the diagram, add components first, grouping them within other components as you see fit
- Step 4: Next step is to add other elements such as interfaces, classes, objects, dependencies etc. to your component diagram and complete it.
- Step 5: You can attach notes on different parts of your component diagram to clarify certain details to others.

Component Diagram Examples

- Component Diagram for Library Management System.



Deployment Diagram

- Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed.
- Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

Purpose of Deployment Diagrams

- The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.
- Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.
- UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components.
- Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.

The purpose of deployment diagrams can be described as –

- Visualize the hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe the runtime processing nodes.

How to Draw a Deployment Diagram?

Deployment diagram represents the deployment view of a system. It is related to the component diagram because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardware used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters –

- Performance
- Scalability
- Maintainability
- Portability

Before drawing a deployment diagram, the following artifacts should be identified –

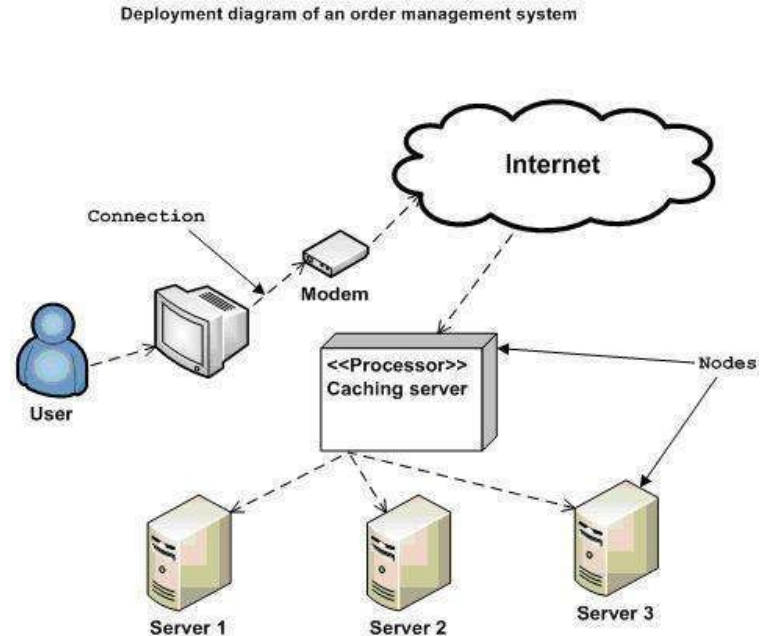
- Nodes
- Relationships among nodes

Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as –

- Monitor
- Modem
- Caching server
- Server

Deployment Diagrams

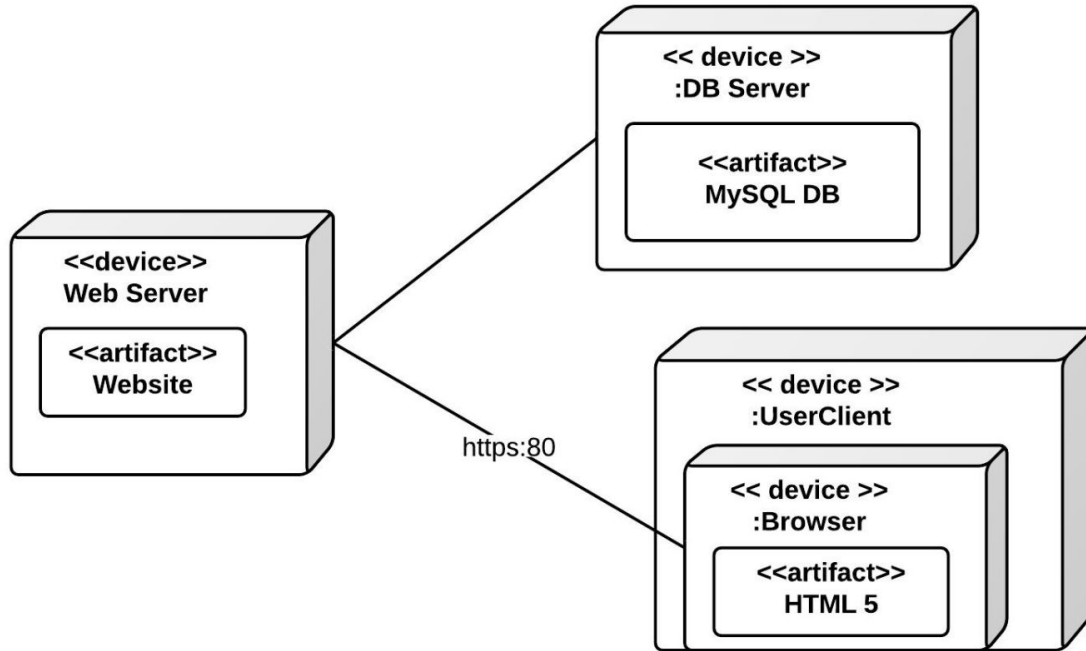
- The application is assumed to be a web-based application, which is deployed in a clustered environment using server 1, server 2, and server 3. The user connects to the application using the Internet. The control flows from the caching server to the clustered environment.
- The following deployment diagram has been drawn considering all the points mentioned above.



Where to Use Deployment Diagrams?

- Deployment diagrams are mainly used by **system engineers**. These diagrams are used to describe the physical components (hardware), their distribution, and association.
- Deployment diagrams can be visualized as the hardware components/nodes on which the software components reside.
- Software applications are developed to model complex business processes. Efficient software applications are not sufficient to meet the business requirements. Business requirements can be described as the need to support the increasing number of users, quick response time, etc.
- To meet these types of requirements, hardware components should be designed efficiently and in a cost-effective way.
- Now-a-days software applications are very complex in nature. Software applications can be standalone, web-based, distributed, mainframe-based and many more. Hence, it is very important to design the hardware components efficiently.

Deployment diagram



Deployment diagrams can be used –

- To model the hardware topology of a system.
- To model the embedded system.
- To model the hardware details for a client/server system.
- To model the hardware details of a distributed application.
- For Forward and Reverse engineering.

What is a package diagram?

- Package diagrams are structural diagrams used to show the organization and arrangement of various model elements in the form of packages. A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages. Each element is nested within the package, which is depicted as a file folder within the diagram, then arranged hierarchically within the diagram. Package diagrams are most commonly used to provide a visual organization of the layered architecture within any UML classifier, such as a software system.

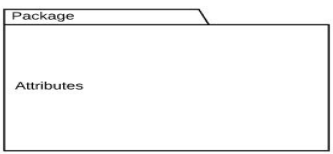

Benefits of a package diagram

A well-designed package diagram provides numerous benefits to those looking to create a visualization of their UML system or project.

- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated as systems and projects evolve

Basic components of a package diagram

- The makeup of a package diagram is relatively simple. Each diagram includes only two symbols:

Symbol Image	Symbol Name	Description
	Package	Groups common elements based on data, behavior, or user interaction
	Dependency	Depicts the relationship between one element (package, named element, etc) and another

These symbols can be used in a variety of ways to represent different iterations of packages, dependencies, and other elements within a system. Here are the basic components you'll find within a package diagram:

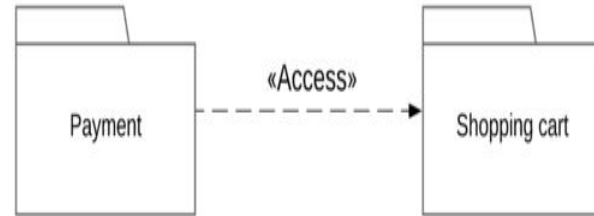
- **Package:** A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.
- **Packageable element:** A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves. Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
- **Dependencies:** A visual representation of how one element (or set of elements) depends on or influences another. Dependencies are divided into two groups: access and import dependencies.

Dependency notations in a package diagram

Package diagrams are used, in part, to depict import and access dependencies between packages, classes, components, and other named elements within your system. Each dependency is rendered as a connecting line with an arrow representing the type of relationship between the two or more elements.

There are two main types of dependencies:

Access: Indicates that one package requires assistance from the functions of another package. Example:



Dependency notations in a package diagram

- Import: Indicates that functionality has been imported from one package to another.
- Example: `X<<import>>Y<<import>>Z`. Import is transitive. So in the above example, X imports from Y and Y imports from Z. When we use `<<import>>` it means that packages 'X' can use elements in package Z.



Dependency notations in a package diagram

- Conversely with the example:

`X<<access>>Y<<access>>Z`

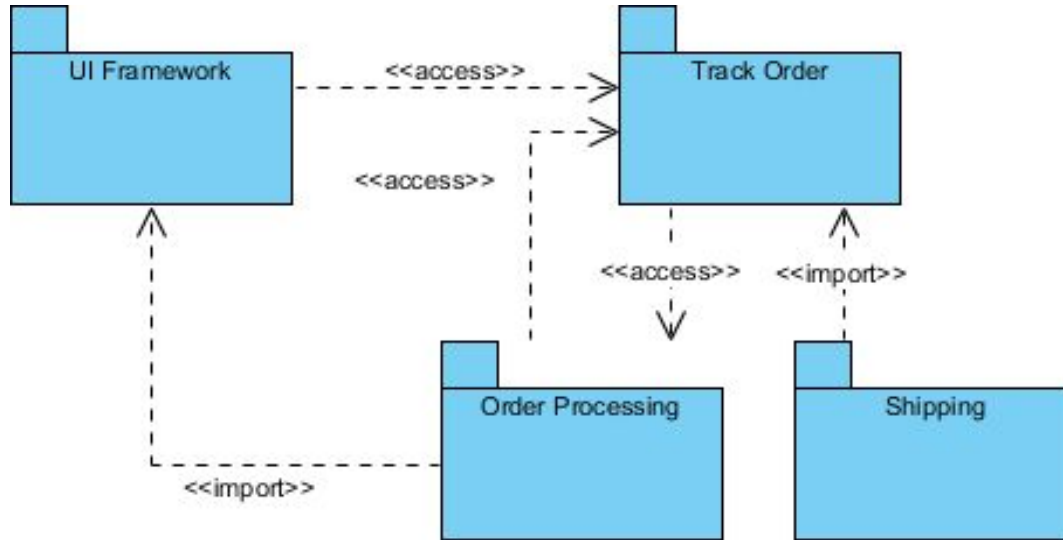
Access is non-transitive. So in the above example, X will not be able to access any of the elements in Z.

Using packages with other UML diagrams

- Packages are UML constructs that can be used to organize the elements within any UML classifier in a variety of UML diagrams. Package diagrams are most commonly found used in:
- **Use-case diagrams:** Each use-case is depicted as an individual package
- **Class diagrams:** Classes are organized into packages

Packages can also be used within other UML model types to organize and arrange elements such as classes, data entities, and use cases. By fusing the package diagram structure with other UML diagrams, you can simplify any model type, making it easier to understand.

Package diagram example



OMT(object modeling techniques)

- The object modeling techniques is an methodology of object oriented analysis, design and implementation that focuses on creating a model of objects from the real world and then to use this model to develop object-oriented software. object modeling technique, OMT was developed by James Rumbaugh. Now-a-days, OMT is one of the most popular object oriented development techniques. It is primarily used by system and software developers to support full life cycle development while targeting object oriented implementations.
- OMT has proven itself easy to understand, to draw and to use. It is very successful in many application domains: telecommunication, transportation, compilers etc. The popular object modeling technique are used in many real world problems. The object-oriented paradigm using the OMT spans the entire development cycle, so there is no need to transform one type of model to another.

Phase of OMT

The OMT methodology covers the full software development life cycle. The methodology has the following phase.

- **Analysis** - Analysis is the first phase of OMT methodology. The aim of analysis phase is to build a model of the real world situation to show its important properties and domain. This phase is concerned with preparation of precise and correct modelling of the real world. The analysis phase starts with defining a problem statement which includes a set of goals. This problem statement is then expanded into three models; an object model, a dynamic model and a functional model. The object model shows the static data structure or skeleton of the real world system and divides the whole application into objects. In others words, this model represents the artifacts of the system. The dynamic model represents the interaction between artifacts above designed represented as events, states and transitions. The functional model represents the methods of the system from the data flow perspective. The analysis phase generates object model diagrams, state diagrams, event flow diagrams and data flow diagrams.

Phase of OMT

System design - The system design phase comes after the analysis phase. System design phase determines the overall system architecture using subsystems, concurrent tasks and data storage. During system design, the high level structure of the system is designed. The decisions made during system design are:

- The system is organized in to sub-systems which are then allocated to processes and tasks, taking into account concurrency and collaboration.
- Persistent data storage is established along with a strategy to manage shared or global information.
- Boundary situations are checked to help guide trade off priorities.

Phase of OMT

Object design - The object design phase comes after the system design phase is over. Here the implementation plan is developed. Object design is concerned with fully classifying the existing and remaining classes, associations, attributes and operations necessary for implementing a solution to the problem. In object design:

- Operations and data structures are fully defined along with any internal objects needed for implementation.
- Class level associations are determined.
- Issues of inheritance, aggregation, association and default values are checked.

Phase of OMT

Implementation - Implementation phase of the OMT is a matter of translating the design in to a programming language constructs. It is important to have good software engineering practice so that the design phase is smoothly translated in to the implementation phase. Thus while selecting programming language all constructs should be kept in mind for following noteworthy points.

- To increase flexibility.
- To make amendments easily.
- For the design traceability.
- To increase efficiency.

Design Patterns

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.

These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

Program to an interface not an implementation

Favor object composition over inheritance

Usage of Design Pattern

Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions

to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way

Types of Design Pattern

- As per the design pattern reference book Design Patterns - Elements of Reusable ObjectOriented Software , there are 23 design patterns. These patterns can be classified in three
- categories: Creational, Structural and behavioral patterns. We'll also discuss another category
- of design patterns: J2EE design patterns.

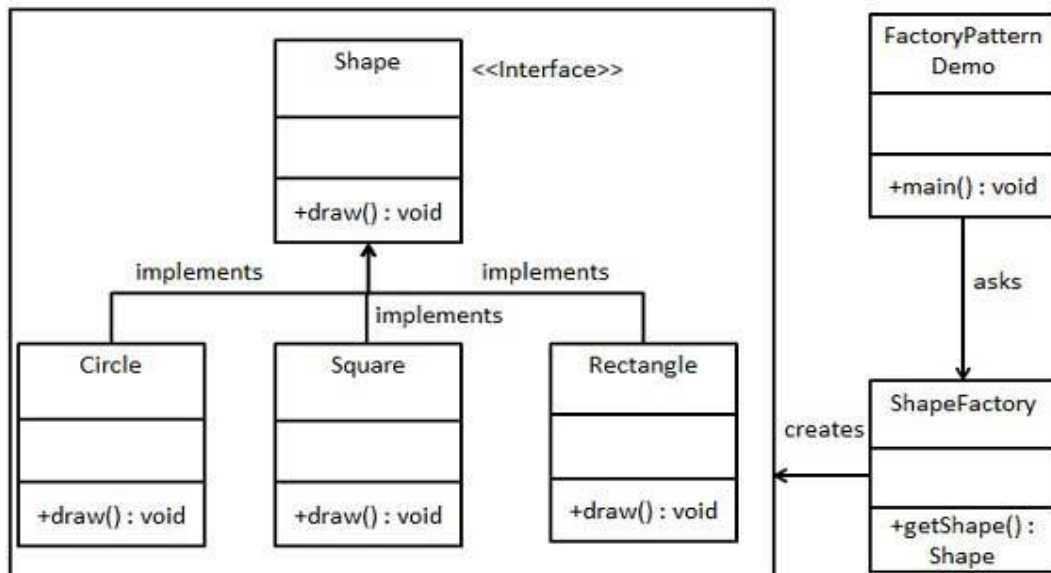
S.N	Pattern & Description
1	Creational Patterns -These design patterns provides way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	Structural Patterns -These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	Behavioral Patterns -These design patterns are specifically concerned with communication between objects.
4	J2EE Patterns -These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

Factory Pattern

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation

We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step. FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.



Unit 5

Software Quality-

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc. for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

- **Portability:** A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.
- **Usability:** A software product has better usability if various categories of users can easily invoke the functions of the product.
- **Reusability:** A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.
- **Correctness:** A software product is correct if various requirements as specified in the SRS document have been correctly implemented.
- **Maintainability:** A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Software quality assurance (SQA)

Software quality assurance (SQA) is a process which assures that all software engineering processes, methods, activities and work items are monitored and comply against the defined standards. These defined standards could be one or a combination of any like ISO 9000, CMMI model, ISO15504, etc.

SQA incorporates all software development processes starting from defining requirements to coding until release. Its prime goal is to ensure quality.

Software Quality Assurance Plan

- Abbreviated as SQAP, the software quality assurance plan comprises of the procedures, techniques, and tools that are employed to make sure that a product or service aligns with the requirements defined in the SRS (software requirement specification).
- The plan identifies the SQA responsibilities of a team, lists the areas that need to be reviewed and audited. It also identifies the SQA work products.



The SQA plan document consists of the below sections:

- Purpose section
- Reference section
- Software configuration management section
- Problem reporting and corrective action section
- Tools, technologies and methodologies section
- Code control section
- Records: Collection, maintenance and retention section
- Testing methodology

SQA Activities

Given below is the list of SQA activities:

#1) Creating an SQA Management Plan: The foremost activity includes laying down a proper plan regarding how the SQA will be carried out in your project.

Along with what SQA approach you are going to follow, what engineering activities will be carried out, and it also includes ensuring that you have a right talent mix in your team.

#2) Setting the Checkpoints- The SQA team sets up different checkpoints according to which it evaluates the quality of the project activities at each checkpoint/project stage. This ensures regular quality inspection and working as per the schedule.

#3) Apply software Engineering Techniques:- Applying some software engineering techniques aids a software designer in achieving high-quality specification. For gathering information, a designer may use techniques such as **interviews and FAST** (Functional Analysis System Technique).

SQA Activities

#4) Executing Formal Technical Reviews: An FTR is done to **evaluate the quality and design of the prototype**. In this process, **a meeting is conducted with the technical staff to discuss regarding the actual quality requirements of the software and the design quality of the prototype**. This activity helps in detecting errors in the early phase of SDLC and reduces rework effort in the later phases.

#5) Having a Multi- Testing Strategy: By multi-testing strategy, we mean that one should not rely on any single testing approach, instead, **multiple types of testing should be performed so that the software product can be tested well from all angles to ensure better quality**.

#6) Enforcing Process Adherence: This activity insists the need for process adherence during the software development process. **The development process should also stick to the defined procedures**.

This activity is a blend of two sub-activities which are explained below in detail:

(i) Product Evaluation: This activity confirms that the **software product is meeting the requirements that were discovered in the project management plan**. It ensures that the set standards for the project are followed correctly.

(ii) Process Monitoring: This activity verifies if the **correct steps were taken during software development**. This is done by matching the actually taken steps against the documented steps.

SQA Activities

#7) Controlling Change: In this activity, we use a **mix of manual procedures and automated tools to have a mechanism for change control.**

By **validating the change requests, evaluating the nature of change and controlling the change effect**, it is ensured that the software quality is maintained during the development and maintenance phases.

#8) Measure Change Impact: If any defect is reported by the QA team, then the concerned team fixes the defect.

After this, the **QA team should determine the impact of the change which is brought by this defect fix. They need to test not only if the change has fixed the defect, but also if the change is compatible with the whole project.**

For this purpose, we use software quality metrics which allows managers and developers to observe the activities and proposed changes from the beginning till the end of SDLC and initiate corrective action wherever required.

SQA Activities

#9) Performing SQA Audits:The SQA audit **inspects the entire actual SDLC process followed by comparing it against the established process.**

It also checks whatever reported by the team in the status reports were actually performed or not. This activity also exposes any non-compliance issues.

#10) Maintaining Records and Reports:It is crucial to keep the necessary documentation related to SQA and share the required SQA information with the stakeholders. **The test results, audit results, review reports, change requests documentation, etc. should be kept for future reference.**

#11) Manage Good Relations:In fact, it is very important to maintain harmony between the QA and the development team.

We often hear that **testers and developers often feel superior to each other. This should be avoided as it can affect the overall project quality.**

Software Quality Assurance (SQA)

- Software Quality Assurance (SQA) is simply a way to assure quality in the software. It is the set of activities which ensure processes, procedures as well as standards suitable for the project and implemented correctly.
- Software Quality Assurance is a process which works parallel to development of a software. It focuses on improving the process of development of software so that problems can be prevented before they become a major issue. Software Quality Assurance is a kind of an Umbrella activity that is applied throughout the software process.

Major Software Quality Assurance Activities:

SQA Management Plan:

Make a plan how you will carry out the sqa through out the project. Think which set of software engineering activities are the best for project. check level of sqa team skills.

Set The Check Points:

SQA team should set checkpoints. Evaluate the performance of the project on the basis of collected data on different check points.

Multi testing Strategy: Do not depend on single testing approach. When you have lot of testing approaches available use them.

Major Software Quality Assurance Activities:

Measure Change Impact:

The changes for making the correction of an error sometimes re introduces more errors keep the measure of impact of change on project. Reset the new change to change check the compatibility of this fix with whole project.

Manage Good Relations:

In the working environment managing the good relation with other teams involved in the project development is mandatory. Bad relation of sqa team with programmers team will impact directly and badly on project. Don't play politics.

Benefits of Software Quality Assurance (SQA):

- SQA produce high quality software.
- High quality application saves time and cost.
- SQA is beneficial for better reliability.
- SQA is beneficial in the condition of no maintenance for long time.
- High quality commercial software increase market share of company.
- Improving the process of creating software.
- Improves the quality of the software.

Disadvantage of SQA:

- There are a number of disadvantages of quality assurance. Some of them include adding more resources, employing more workers to help maintain quality and so much more.

Types of Software Testing

Introduction:-

Testing is a process of executing a program with the aim of finding error. To make our software perform well it should be error free. If testing is done successfully it will remove all the errors from the software.

Principles of Testing:-

- (i) All the test should meet the customer requirements
- (ii) To make our software testing should be performed by third party
- (iii) Exhaustive testing is not possible. As we need the optimal amount of testing based on the risk assessment of the application.
- (iv) All the test to be conducted should be planned before implementing it
- (v) It follows pareto rule(80/20 rule) which states that 80% of errors comes from 20% of program components.
- (vi) Start testing with small parts and extend it to large parts.

Types of Testing:-

● 1. Unit Testing

It focuses on smallest unit of software design. In this we test an individual unit or group of inter related units. It is often done by programmer by using sample input and observing its corresponding outputs.

Example:

- a) In a program we are checking if loop, method or function is working fine
- b) Misunderstood or incorrect, arithmetic precedence.
- c) Incorrect initialization

2. Integration Testing

The objective is to take unit tested components and build a program structure that has been dictated by design. Integration testing is testing in which a group of components are combined to produce output.

- Integration testing is of four types: (i) Top down (ii) Bottom up (iii) Sandwich(top to down+bottom up) (iv) Big-Bang

Example

(a) Black Box testing:- It is used for validation.

In this we ignore internal working mechanism and focus on what is the output?.

(b) White Box testing:- It is used for verification.

In this we focus on internal mechanism i.e.
how the output is achieved?

3. Regression Testing

Every time new module is added leads to changes in program. This type of testing make sure that whole component works properly even after adding components to the complete program.

Example

In school record suppose we have module **staff**, students and finance combining these modules and checking if on integration these module works fine is regression testing

4. Smoke Testing

This test is done to make sure that software under testing is ready or stable for further testing

It is called smoke test as testing initial pass is done to check if it did not catch the fire or smoked in the initial switch on.

Example:

If project has 2 modules so before going to module

make sure that module 1 works properly

5. Alpha Testing

This is a type of validation testing. It is a type of acceptance testing which is done before the product is released to customers. It is typically done by QA people.

Example:

When software testing is performed internally within the organization

6. Beta Testing

The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for the limited number of users for testing in real time environment .

Example:

When software testing is performed for the limited
number of people

7. System Testing

In this software is tested such that it works fine for different operating system. It is covered under the black box testing technique. In this we just focus on required input and output without focusing on internal working.

In this we have security testing, recovery testing, stress testing and performance testing

Example:

- This includes functional as well as non functional testing

Functional Testing

Functional testing is a type of testing which verifies that each function of the software application operates in conformance with the requirement specification. This testing mainly involves black box testing, and it is not concerned about the source code of the application.

Every functionality of the system is tested by providing appropriate input, verifying the output and comparing the actual results with the expected results. This testing involves checking of User Interface, APIs, Database, security, client/ server applications and functionality of the Application Under Test. The testing can be done either manually or using automation

Non-Functional Testing

- Non-functional testing is a type of testing to check non-functional aspects (performance, usability, reliability, etc.) of a software application. It is explicitly designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing.
- A good example of non-functional test would be to check how many people can simultaneously login into a software.

8. Stress Testing

In this we give unfavorable conditions to the system and check how they perform in those conditions.

Example:

- (a) Test cases that require maximum memory or other resources are executed
- (b) Test cases that may cause thrashing in a virtual operating system
- (c) Test cases that may cause excessive disk requirement

9. Performance Testing

It is designed to test the run-time performance of software within the context of an integrated system. It is used to test speed and effectiveness of program. It is also called load testing. In it we check , what is the performance of the system in the given load.

Example:

Checking number of processor cycles.

<https://www.guru99.com/types-of-software-testing.html>

Testing Guidelines

There are certain testing guidelines that should be followed while testing the software:

Development team should avoid testing the software: Testing should always be performed by the testing team. The developer team should never test the software themselves. This is because after spending several hours building the software, it might unconsciously become too proprietary and that might prevent seeing any flaws in the system. The testers should have a destructive approach towards the product. Developers can perform unit testing and integration testing but software testing should be done by the testing team.

Testing Guidelines

- **Software can never be 100% bug-free:** Testing can never prove the software to 100% bug-free. In other words, there is no way to prove that the software is free of errors even after making a number of test cases.
- **Start as early as possible:** Testing should always start parallelly alongside the requirement analysis process. This is crucial in order to avoid the problem of defect migration. It is important to determine the test objects and scope as early as possible.
- **Prioritize sections:** If there are certain critical sections, then it should be ensured that these sections are tested with the highest priority and as early as possible.
- **The time available is limited:** Testing time for software is limited. It must be kept in mind that the time available for testing is not unlimited and that an effective test plan is very crucial before starting the process of testing. There should be some criteria to decide when to terminate the process of testing. This criterion needs to be decided beforehand. For instance, when the system is left with an acceptable level of risk or according to timelines or budget constraints.

Testing Guidelines

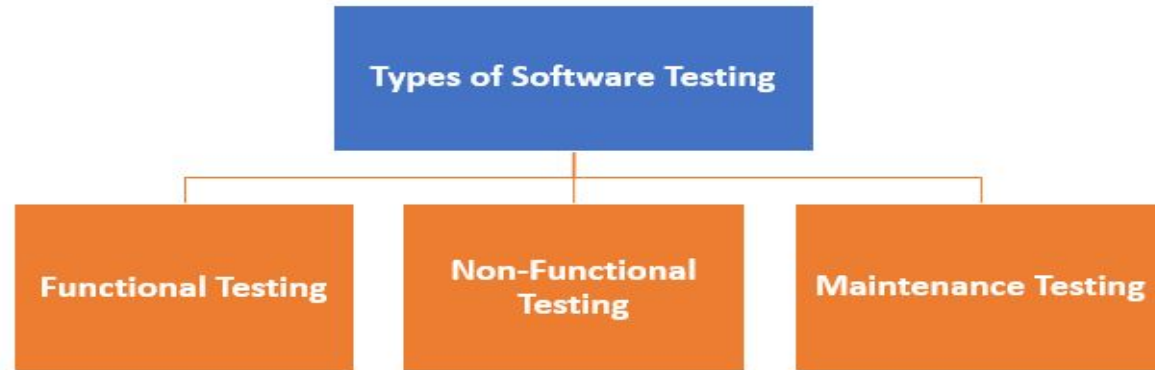
- **Testing must be done with unexpected and negative inputs:** Testing should be done with correct data and test cases as well as with flawed test cases to make sure the system is leak proof. Test cases must be well documented to ensure future reuse for testing at later stages. This means that the test cases must be enlisted with proper definitions and descriptions of inputs passed and respective outputs expected. Testing should be done for functional as well as the non-functional requirements of the software product.
- **Inspecting test results properly:** Quantitative assessment of tests and their results must be done. The documentation should be referred to properly while validating the results of the test cases to ensure proper testing. Testing must be supported by automated tools and techniques as much as possible. Besides ensuring that the system does what all it is supposed to do, testers also need to ensure that the system does not perform operations which it isn't supposed to do.
- **Validating assumptions:** The test cases should never be developed on the basis of assumptions or hypothesis. They must always be validated properly. For instance, assuming that the software product is free from any bugs while designing test cases may result in extremely weak test cases.

What are the benefits of Software Testing?

Here are the benefits of using software testing:

- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

Typically Testing is classified into three categories.



Testing Category	Types of Testing
Functional Testing	UNIT INTEGRATION Smoke UAT (User Acceptance Testing) Localization Globalization Interoperability So on
Non-Functional Testing	Performance Endurance Load Volume Scalability Usability So on
Maintenance	Regression Maintenance

Summary of Software Testing Basics:

- Software testing is defined as an activity to check whether the actual results match the expected results and to ensure that the software system is Defect free.
- Testing is important because software bugs could be expensive or even dangerous.
- The important reasons for using software testing are: cost-effective, security, product quality, and customer satisfaction.
- Typically Testing is classified into three categories functional testing, non-functional testing or performance testing, and maintenance.
- The important strategies in software engineering are: unit testing, integration testing, validation testing, and system testing.

Verification and Validation

Verification and Validation is the process of investigating that a software system satisfies specifications and standards and it fulfills the required purpose. Barry Boehm described verification and validation as the following:

Verification: Are we building the product right? $2+2=4$

Validation: Are we building the right product? $2+2=5$

Verification:

Verification is the process of checking that a software achieves its goal without any bugs. It is the process to ensure whether the product that is developed is right or not. It verifies whether the developed product fulfills the requirements that we have.

Verification is Static Testing.

Activities involved in verification:

Inspections

Reviews

Walkthroughs

Desk-checking

Validation:

Validation is the process of checking whether the software product is up to the mark or in other words product has high level requirements. It is the process of checking the validation of product i.e. it checks what we are developing is the right product. it is validation of actual and expected product.

Validation is the Dynamic Testing

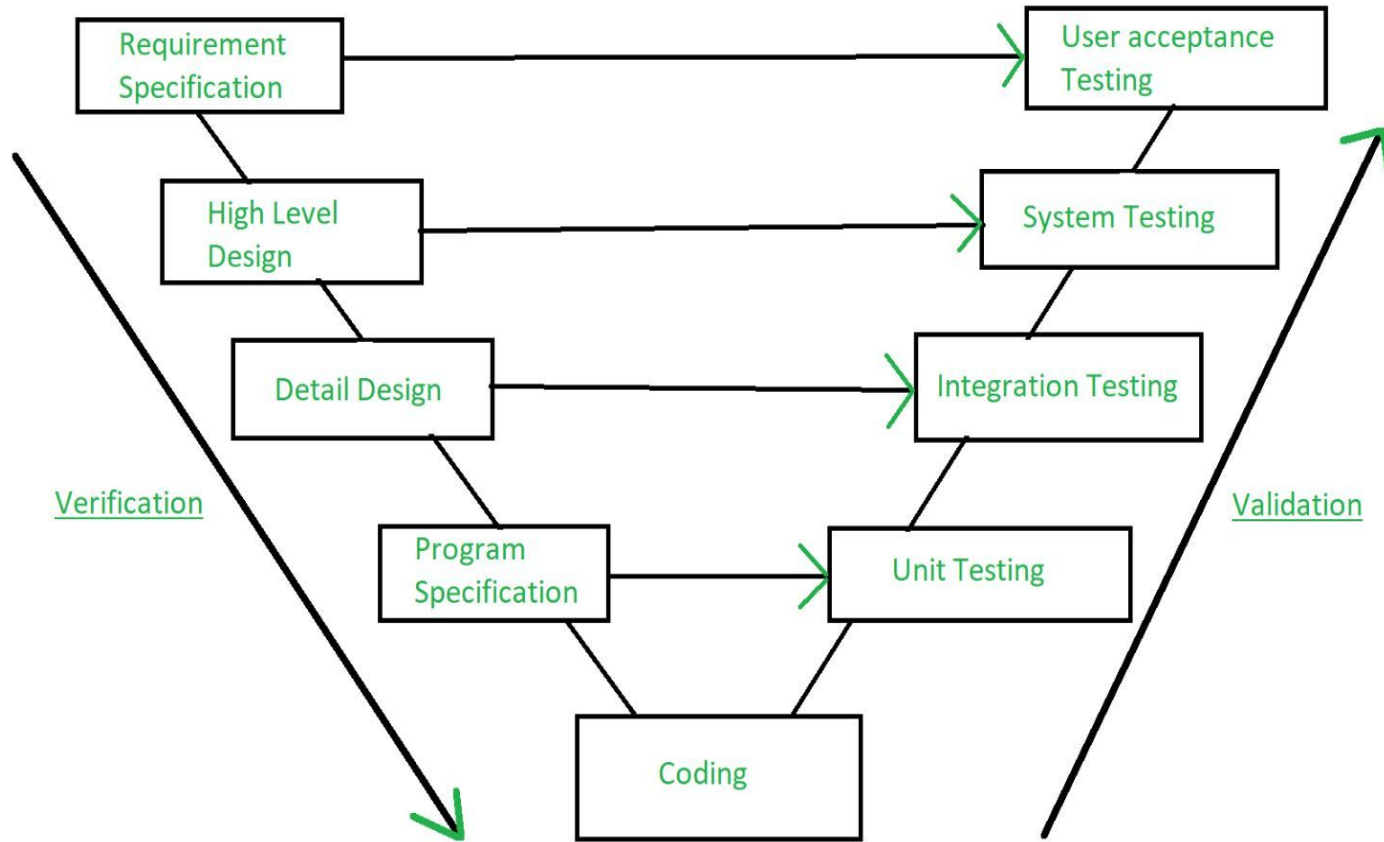
Activities involved in validation:

Black box testing

White box testing

Unit testing

Integration testing



Verification



Validation

Verification	Validation
The verifying process includes checking documents, design, code, and program	It is a dynamic mechanism of testing and validating the actual product
It does not involve executing the code	It always involves executing the code
Verification uses methods like reviews, walkthroughs, inspections, and desk- checking etc.	It uses methods like Black Box Testing, White Box Testing, and non-functional testing
Whether the software conforms to specification is checked	It checks whether the software meets the requirements and expectations of a customer
It finds bugs early in the development cycle	It can find bugs that the verification process can not catch
Target is application and software architecture, specification, complete design, high level, and database design etc.	Target is an actual product
QA team does verification and make sure that the software is as per the requirement in the SRS document.	With the involvement of testing team validation is executed on software code.
It comes before validation	It comes after verification

Example of verification and validation

In Software Engineering, consider the following specification

A clickable button with name Submet

Verification would check the design doc and correcting the spelling mistake.

Otherwise, the development team will create a button like "SUBMIT"

Owing to Validation testing, the development team will make the submit button clickable



What is software reuse?

Software reuse is a term used for developing the software by using the existing software components. Some of the components that can be reuse are as follows;

- Source code
- Design and interfaces
- User manuals
- Software Documentation

What are the advantages of software reuse?

- **Less effort:** Software reuse requires less effort because many components use in the system are ready made components.
- **Time-saving:** Re-using the ready made components is time saving for the software team.
- **Reduce cost:** Less effort, and time saving leads to the overall cost reduction.
- **Increase software productivity:** when you are provided with ready made components, then you can focus on the new components that are not available just like ready made components.
- **Utilize fewer resources:** Software reuse save many sources just like effort, time, money etc.
- **Leads to a better quality software:** Software reuse save our time and we can consume our more time on maintaining software quality and assurance.

What are Commercial-off-the-shelf and Commercial-off-the-shelf components?

- Commercial-off-the-shelf is ready-made software.
- Commercial-off-the-shelf software components are ready-made components that can be reused for a new software.

What is reuse software engineering?

- Reuse software engineering is based on guidelines and principles for reusing the existing software.

What are stages of reuse-oriented software engineering?

Requirement specification:

First of all, specify the requirements. This will help to decide that we have some existing software components for the development of software or not.

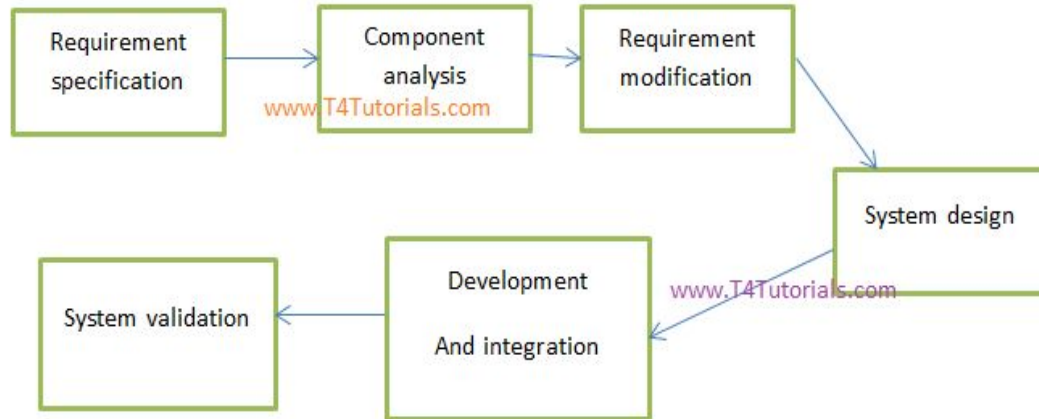
Component analysis

Helps to decide that which component can be reused where.

Stages of reuse-oriented software engineering

- **Requirement updations / modifications.** -If the requirements are changed by the customer, then still existing components are helpful for reuse or not.
- **Reuse System design**-If the requirements are changed by the customer, then still existing system designs are helpful for reuse or not.
- **Development**-Existing components are matching with new software or not.
- **Integration**-Can we integrate the new systems with existing components?
- **System validation**-To validate the system that it can be accepted by the customer or not.

Stages of reuse-oriented software engineering



software reuse success factors

- Capturing Domain Variations
- Easing Integration
- Understanding Design Context
- Effective Teamwork
- Managing Domain Complexity

software reuse

- **How does inheritance promote software re-usability?**

Inheritance helps in the software re-usability by using the existing components of the software to create new component.

In object oriented programming protected data members are accessible in the child and so we can say that yes inheritance promote software re-usability.

- **Reuse of software helps reduce the need for testing?**

This statement is true just for those components that are ready made and are ready to be reuse. New components must be test.

Software Re-Engineering

Software Re-Engineering is the examination and alteration of a system to reconstitute it in a new form. The principles of Re-Engineering when applied to the software development process is called software re-engineering. It affects positively at software cost, quality, service to the customer and speed of delivery. In Software Re-engineering, we are improving the software to make it more efficient and effective.

The need of software Re-engineering:

- Software re-engineering is an economical process for software development and quality enhancement of the product. This process enables us to identify the useless consumption of deployed resources and the constraints that are restricting the development process so that the development process could be made easier and cost-effective (time, financial, direct advantage, optimize the code, indirect benefits, etc.) and maintainable.

The software reengineering is necessary for having-

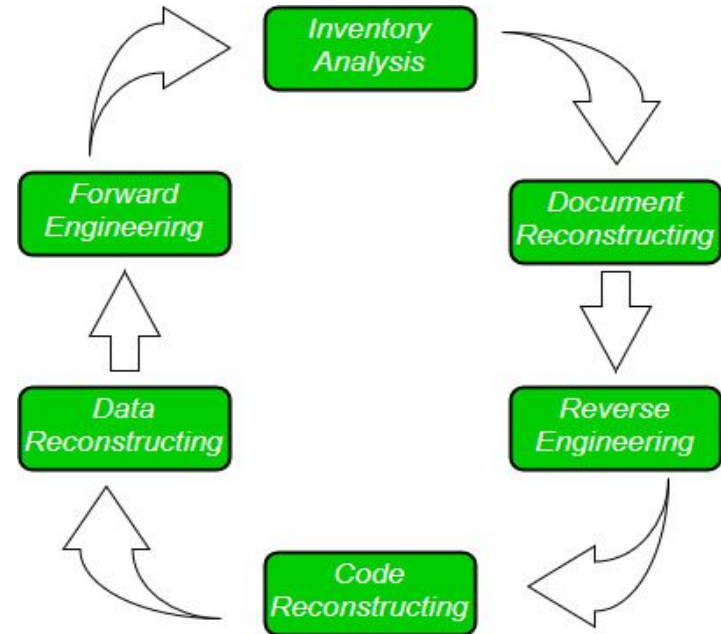
- **a) Boost up productivity:** Software reengineering increase productivity by optimizing the code and database so that processing gets faster.
- **b) Processes in continuity:** The functionality of older software product can be still used while the testing or development of software.
- **c) Improvement opportunity:** Meanwhile the process of software reengineering, not only software qualities, features and functionality but also your skills are refined, new ideas hit in your mind. This makes the developers mind accustomed to capturing new opportunities so that more and more new features can be developed.

The software reengineering is necessary for having-

- d) **Reduction in risks:** Instead of developing the software product from scratch or from the beginning stage here developers develop the product from its existing stage to enhance some specific features that are brought in concern by stakeholders or its users. Such kind of practice reduces the chances of fault fallibility.
- e) **Saves time:** As we stated above here that the product is developed from the existing stage rather than the beginning stage so the time consumes in software engineering is lesser.
- f) **Optimization:** This process refines the system features, functionalities and reduces the complexity of the product by consistent optimization as maximum as possible.

Re-Engineering cost factors:

- The quality of the software to be re-engineered.
- The tool support availability for engineering.
- The extent of the data conversion which is required.
- The availability of expert staff for Re-engineering.



Software Re-Engineering Activities:

1. Inventory Analysis:

- Every software organisation should have an inventory of all the applications.
- Inventory can be nothing more than a spreadsheet model containing information that provides a detailed description of every active application.
- By sorting this information according to business criticality, longevity, current maintainability and other local important criteria, candidates for re-engineering appear.
- The resource can then be allocated to a candidate application for re-engineering work.

Software Re-Engineering Activities:

2. Document reconstructing:

- Documentation of a system either explains how it operates or how to use it.
- Documentation must be updated.
- It may not be necessary to fully document an application.
- The system is business-critical and must be fully re-documented.

3. Reverse Engineering:

- Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural and procedural design information from an existing program.

Software Re-Engineering Activities:

4. Code Reconstructing:

- To accomplish code reconstructing, the source code is analysed using a reconstructing tool. Violations of structured programming construct are noted and code is then reconstructed.
- The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced.

5. Data Restructuring:

- Data restructuring begins with a reverse engineering activity.
- Current data architecture is dissected, and the necessary data models are defined.
- Data objects and attributes are identified, and existing data structure are reviewed for quality.

Software Re-Engineering Activities:

6. Forward Engineering:

- Forward Engineering also called as renovation or reclamation not only for recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.

Difference between Re Engineering and Reverse Engineering

- Re-engineering is commonly, but incorrectly, used in reference to reverse engineering. While both refer to the further investigation or engineering of finished products, the methods of doing so, and the desired outcomes, are vastly different. Reverse engineering attempts to discover how something works, while re-engineering seeks to improve a current design by investigating particular aspects of it.

Re-Engineering

- Re-engineering is the investigation and redesign of individual components. It may also describe the entire overhaul of a device by taking the current design and improving certain aspects of it. The aims of re-engineering may be to improve a particular area of performance or functionality, reduce operational costs or add new elements to a current design. The methods used depend on the device but typically involve engineering drawings of the amendments followed by extensive testing of prototypes before production. The rights to re-engineer a product belong solely to the original owner of the design or relevant patent.

Reverse Engineering

- Unlike re-engineering, reverse engineering takes a finished product with the aim of discovering how it works by testing it. Typically this is done by companies that seek to infiltrate a competitor's market or understand its new product. In doing so they can produce new products while allowing the original creator to pay all the development costs and take all the risks involved with creating a new product. Analysis of a product in this way is done without technical drawings or prior knowledge of how the device works, and the basic method used in reverse engineering begins by identifying the system's components, followed by an investigation into the relationship among these components.

Legal Issues

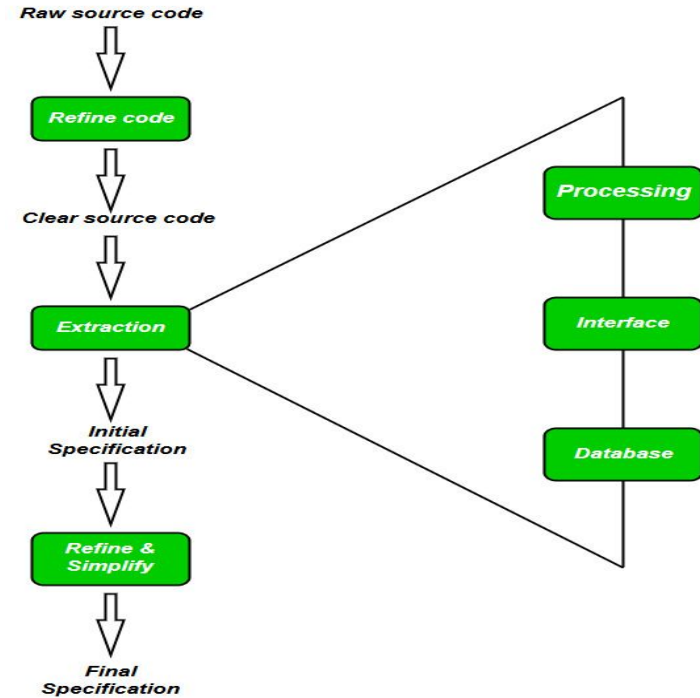
- Reverse engineering is a controversial subject. While the companies performing it may be at a distinct advantage, saving both time and money, the original creator of the design may be severely affected by the increased competition. Although design patents can protect an engineer or company from this kind of activity, the security this can offer is limited. By reverse engineering a product, you can discover original ideas that are not protected; in doing so, you can infringe another's intellectual property rights. It is therefore important that designs are not disclosed to competitors and protection is in place to prevent fraudulent activity.

Software Reverse Engineering

- Software Reverse Engineering is a process of recovering the design, requirement specifications and functions of a product from an analysis of its code. It builds a program database and generates information from this.
- The purpose of reverse engineering is to facilitate the maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

Reverse Engineering Goals:

- Cope with Complexity.
- Recover lost information.
- Detect side effects.
- Synthesise higher abstraction.
- Facilitate Reuse.



Steps of Software Reverse Engineering:

Collection Information:

- This step focuses on collecting all possible information (i.e., source design documents etc.) about the software.

Examining the information:

- The information collected in step-1 is studied so as to get familiar with the system.

Extracting the structure:

- This step concerns with identification of program structure in the form of structure chart where each node corresponds to some routine.

Recording the functionality:

- During this step processing details of each module of the structure, charts are recorded using structured language like decision table, etc.

Steps of Software Reverse Engineering:

Recording data flow:

- From the information extracted in step-3 and step-4, set of data flow diagrams are derived to show the flow of data among the processes.

Recording control flow:

- High level control structure of the software is recorded.

Review extracted design:

- Design document extracted is reviewed several times to ensure consistency and correctness. It also ensures that the design represents the program.

Generate documentation:

- Finally, in this step, the complete documentation including SRS, design document, history, overview, etc. are recorded for future use.

