SOFTWARE ENGINEERING

UNIT-3

Design Process - Design Principles - Design Concepts - Software Architecture Architectural Style, Design and Mapping - User Interface Design. Class diagram, state diagram.

Software Design

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

The following items are designed and documented during the design phase:

- ✓ Different modules required.
- ✓ Control relationships among modules.
- ✓ Interface among different modules.
- ✓ Data structure among the different modules.
- ✓ Algorithms required to implement among the individual modules.

Objectives of Software Design:

1. **Correctness:**
   A good design should be correct i.e. it should correctly implement all the functionalities of the system.
2. **Efficiency:**
   A good software design should address the resources, time, and cost optimization issues.
3. **Flexibility:**
   A good software design should have the ability to adapt and accommodate changes easily. It includes designing the software in a way, that allows for modifications, enhancements, and scalability without requiring significant rework or causing major disruptions to the existing functionality.
4. **Understandability:**
   A good design should be easily understandable, for which it should be modular and all the modules are arranged in layers.
5. **Completeness:**
   The design should have all the components like data structures, modules, and external interfaces, etc.
6. **Maintainability:**
   A good software design aims to create a system that is easy to understand, modify, and maintain over time. This involves using modular and well-structured design principles. Maintainability in software Design also enables developers to fix bugs, enhance features, and adapt the software to changing requirements without excessive effort or introducing new issues.

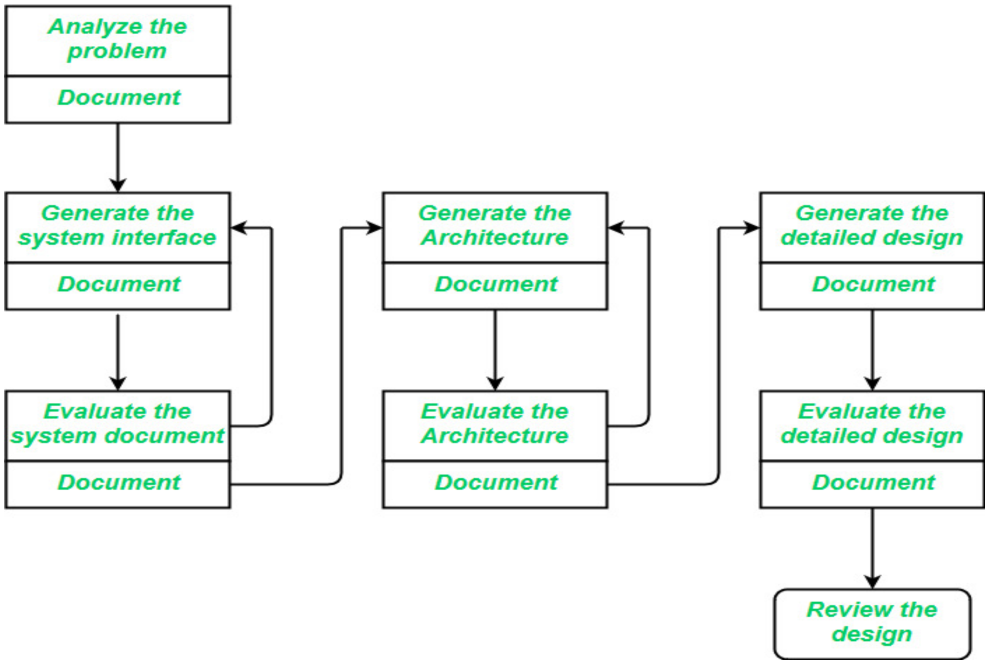
SOFWARE DESIGN PROCESS:

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language. The software design process can be divided into the following three levels of phases of design:

1. Interface Design
2. Architectural Design
3. Detailed Design

Elements of a System:

1. **Architecture –** This is the conceptual model that defines the structure, behavior, and views of a system. We can use flowcharts to represent and illustrate the architecture.
2. **Modules –** These are components that handle one specific task in a system. A combination of the modules makes up the system.
3. **Components –** This provides a particular function or group of related functions. They are made up of modules.
4. **Interfaces –** This is the shared boundary across which the components of a system exchange information and relate.
5. **Data –** This is the management of the information and data flow.


Analyze the problem
Document
Generate the system interface
Document
Evaluate the system document
Document
Generate the Architecture
Document
Evaluate the Architecture
Document
Generate the detailed design
Document
Evaluate the detailed design
Document
Review the design

1. Interface Design:

Interface design is the specification of the interaction between a system and its environment. this phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box.

Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts.

The design problem statement produced during the problem analysis step should **identify the people, other systems, and devices** which are collectively called ***agents***.

Interface design should include the following details:

- ✓ Precise description of events in the environment, or messages from agents to which the system must respond.
- ✓ Precise description of the events or messages that the system must produce.
- ✓ Specification of the data, and the formats of the data coming into and going out of the system.
- ✓ Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

2. Architectural Design:

Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them.

In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.

Issues in architectural design includes:

- ✓ Gross *decomposition of the systems* into major components.
- ✓ *Allocation of functional* responsibilities to components.
- ✓ Component Interfaces, Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- ✓ *Communication and interaction between components*.

The architectural design adds important details ignored during the interface design.

3. Detailed Design:

Design is the specification of the internal elements of all major system components, their *properties, relationships, processing, and often their algorithms and the data structures*.

The detailed design may include:

- ✓ *Decomposition of major system components* into program units.
- ✓ Allocation of functional responsibilities to units.
- ✓ User interfaces
- ✓ Unit states and state changes
- ✓ *Data and control interaction* between units
- ✓ Data packaging and implementation, including issues of scope and visibility of program elements
- ✓ Algorithms and data structures.

Introduction to design process

- ✓ The main aim of design engineering is to generate a model which shows firmness, delight and commodity.
- ✓ Software design is an iterative process through which requirements are translated into the blueprint for building the software.

Software quality guidelines

- A design is generated using the recognizable architectural styles and compose a good design characteristic of components and it is implemented in evolutionary manner for testing.
- A design of the software must be modular i.e the software must be logically partitioned into elements.
- In design, the representation of data , architecture, interface and components should be distinct.
- A design must carry appropriate data structure and recognizable data patterns.
- Design components must show the independent functional characteristic.
- A design creates an interface that reduce the complexity of connections between the components.
- A design must be derived using the repeatable method.
- The notations should be use in design which can effectively communicates its meaning.

Quality attributes

The attributes of design name as 'FURPS' are as follows:

1. Functionality:

It evaluates the feature set and capabilities of the program.

2. Usability:

It is accessed by considering the factors such as human factor, overall aesthetics, consistency and documentation.

3. Reliability:

It is evaluated by measuring parameters like frequency and security of failure, output result accuracy, the mean-time-to-failure(MTTF), recovery from failure and the program predictability.

4. Performance:

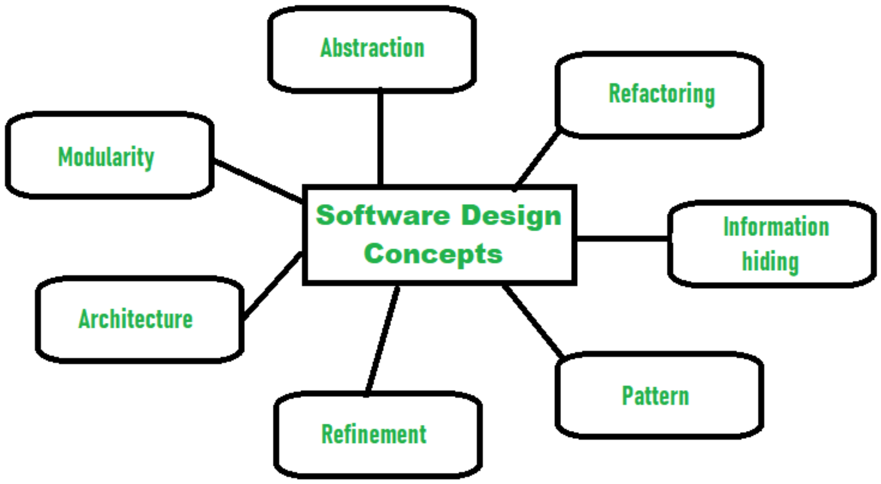
It is measured by considering the factors such as processing speed, response time, resource consumption, throughput and efficiency.

5. Supportability:

- It combines the ability to extend the program, adaptability, service ability. These three terms defines the maintainability.
- Testability, compatibility and configurability are the terms using which a system can be easily installed and found the problem easily.
- Supportability also consists of more attributes such as compatibility, extensibility, fault tolerance, modularity, reusability, robustness, security, portability, scalability.

Design concepts

The set of fundamental software design concepts are as follows:



1. Abstraction

- A solution is stated in large terms using the language of the problem environment at the **highest level abstraction.**
- The **lower level of abstraction** provides a more detail description of the solution.
- A **sequence of instruction** that contain a specific and limited function refers in a procedural abstraction.
- A **collection of data** that describes a data object is a data abstraction.

2. Architecture

- The complete structure of the software is known as software architecture.
- Structure provides conceptual integrity for a system in a number of ways.
- The architecture is the structure of program modules where they interact with each other in a specialized way.
- The components use the structure of data.
- The aim of the software design is to obtain an architectural framework of a system.
- The more detailed design activities are conducted from the framework.

3. Patterns

A design pattern describes a design structure and that structure solves a particular design problem in a specified content.

4. Modularity

- Software is separately divided into **name and addressable components**. Sometime they are called as **modules** which integrate to satisfy the problem requirements.
- Modularity is the single attribute of software that permits a program to be managed easily.

5. Information hiding

Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules not requiring that information.

6. Functional independence

- The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.
- The functional independence is accessed using two criteria i.e **Cohesion and coupling.**

Cohesion

- Cohesion is an **extension of the information hiding concept.**
- A cohesive module **performs a single task** and it requires a **small interaction with the other components** in other parts of the program.

Coupling

Coupling is an indication of interconnection between modules in a structure of software.

7. Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement is reached.

8. Refactoring

- It is a reorganization technique which simplifies the design of components without changing its function behavior.
- Refactoring is the process of changing the software system in a way that it does not change the external behavior of the code still improves its internal structure.

9. Design classes

- The model of software is defined as a set of design classes.
- Every class describes the elements of problem domain and that focus on features of the problem which are user visible.

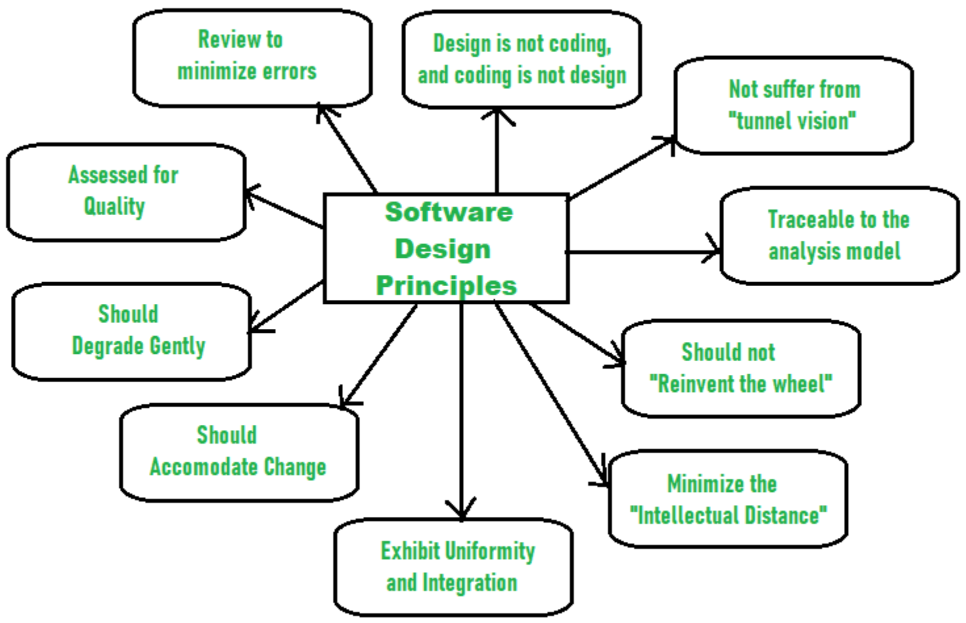
SOFTWARE DESIGN PRINCIPLES:

Design means to draw or plan something to show the look, functions and working of it.

Software Design is also a process to plan or convert the software requirements into a step that are needed to be carried out to develop a software system.

There are several principles that are used to organize and arrange the structural components of Software design. Software Designs in which these principles are applied affect the content and the working process of the software from the beginning.

These principles are stated below :


Review to minimize errors
Design is not coding, and coding is not design
Not suffer from "tunnel vision"
Assessed for Quality
Software Design Principles
Traceable to the analysis model
Should Degrade Gently
Should not "Reinvent the wheel"
Should Accomodate Change
Minimize the "Intellectual Distance"
Exhibit Uniformity and Integration

Principles Of Software Design :

1. **Should not suffer from "Tunnel Vision" –**
   While designing the process, it should not suffer from "tunnel vision" which means that is should not only focus on completing or achieving the aim but on other effects also.
2. **Traceable to analysis model –**
   The design process should be traceable to the analysis model which means it should satisfy all the requirements that software requires to develop a high-quality product.
3. **Should not "Reinvent The Wheel" –**
   The design process should not reinvent the wheel that means it should not waste time or effort in creating things that already exist. Due to this, the overall development will get increased.
4. **Minimize Intellectual distance –**
   The design process should reduce the gap between real-world problems and software solutions for that problem meaning it should simply minimize intellectual distance.
5. **Exhibit uniformity and integration –**
   The design should display uniformity which means it should be uniform throughout the process without any change. Integration means it should mix or combine all parts of software i.e. subsystems into one system.
6. **Accommodate change –**
   The software should be designed in such a way that it accommodates the change implying that the software should adjust to the change that is required to be done as per the user's need.
7. **Degrade gently –**
   The software should be designed in such a way that it degrades gracefully which means it should work properly even if an error occurs during the execution.

8. **Assessed or quality –**
   The design should be assessed or evaluated for the quality meaning that during the evaluation, the quality of the design needs to be checked and focused on.
9. **Review to discover errors –**
   The design should be reviewed which means that the overall evaluation should be done to check if there is any error present or if it can be minimized.
10. **Design is not coding and coding is not design –**
    Design means describing the logic of the program to solve any problem and coding is a type of language that is used for the implementation of a design

Architectural Design in Software Engineering

Requirements of the software should be transformed into an architecture that describes the software's top-level structure and identifies its components.

This is accomplished through architectural design (also called **system design),** which acts as a preliminary 'blueprint' from which software can be developed.

IEEE defines architectural design as 'the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.' This framework is established by examining the software requirements document and designing a model for providing implementation details.

These details are used to specify the components of the system along with their inputs, outputs, functions, and the interaction between them.

An architectural design performs the following functions.

1. It defines an abstraction level at which the designers can specify the functional and performance behavior of the system.
2. It acts as a guideline for enhancing the system (whenever required) by describing those features of the system that can be modified easily without affecting the system integrity.
3. It evaluates all top-level designs.
4. It develops and documents top-level design for the external and internal interfaces.
5. It develops preliminary versions of user documentation.
6. It defines and documents preliminary test requirements and the schedule for software integration.
7. The sources of architectural design are listed below.
8. Information regarding the application domain for the software to be developed
9. Using data-flow diagrams
10. Availability of architectural patterns and architectural styles.

Architectural design is of crucial importance in software engineering during which the essential requirements like reliability, cost, and performance are dealt with.

Though the architectural design is the responsibility of developers, some other people like user representatives, systems engineers, hardware engineers, and operations personnel are also involved.

All these stakeholders must also be consulted while reviewing the architectural design in order to minimize the risks and errors.

Architectural Design Representation

Architectural design can be represented using the following models.

- **Structural model:** Illustrates architecture as an ordered collection of program components
- **Dynamic model:** Specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment
- **Process model:** Focuses on the design of the business or technical process, which must be implemented in the system
- **Functional model:** Represents the functional hierarchy of a system
- **Framework model:** Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

Architectural Design Output

The architectural design process results in an **Architectural Design Document (ADD).** This document consists of a number of graphical representations that comprises software models along with associated descriptive text.

The software models include static model, interface model, relationship model, and dynamic process model. They show how the system is organized into a process at run-time.

Architectural design document gives the developers a solution to the problem stated in the Software Requirements Specification (SRS).

Note that it considers only those requirements in detail that affect the program structure. In addition to ADD, other outputs of the architectural design are listed below.

- Various reports including audit report, progress report, and configuration status accounts report
- Various plans for detailed design phase, which include the following
- Software verification and validation plan
- Software configuration management plan
- Software quality assurance plan
- Software project management plan.

Architectural Styles

Architectural styles define a group of interlinked systems that share structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system.

If an existing architecture is to be re-engineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes re-assignment of the functionality performed by the components.

A computer-based system (software is part of this system) exhibits one of the many available architectural styles. Every architectural style describes a system category that includes the following.

- Computational components such as clients, server, filter, and database to execute the desired system function
- A set of *connectors* such as procedure call, events broadcast, database protocols, and pipes to provide communication among the computational components
- Constraints to define integration of components to form a system
- A *semantic* model, which enable the software designer to identify the characteristics of the system as a whole by studying the characteristics of its components.

Some of the commonly used architectural styles are

1. data-flow architecture,
2. object oriented architecture,
3. layered system architecture,
4. data-centered architecture, and
5. call and return architecture.

Note that the use of an appropriate architectural style promotes design reuse, leads to code reuse, and supports interoperability.

1. Data-flow Architecture

Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations.

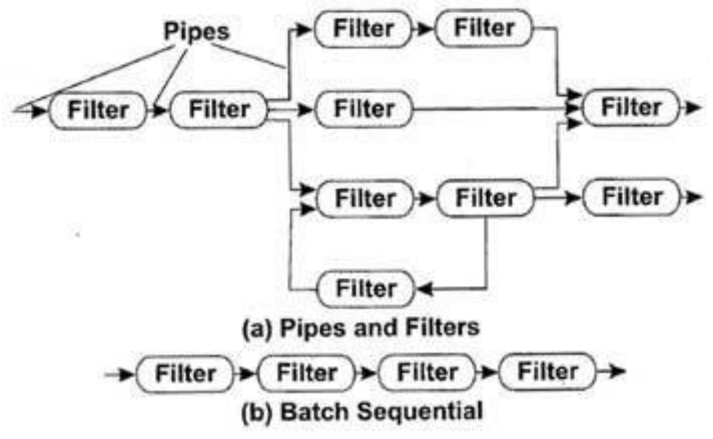
Each component, known as **filter,** transforms the data and sends this transformed data to other filters for further processing using the connector, known as **pipe**.

Each filter works as an independent entity, that is, it is not concerned with the filter which is producing or consuming the data.

A pipe is a unidirectional channel which transports the data received on one end to the other end. It does not change the data in anyway; it merely supplies the data to the filter on the receiver end.

Most of the times, the data-flow architecture degenerates a batch sequential system. In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data. One common example of this architecture is UNIX shell programs.

In these programs, UNIX processes act as filters and the file system through which UNIX processes interact, act as pipes. Other well-known examples of this architecture are compilers, signal processing systems, parallel programming, functional programming, and distributed systems.



Data-flow Architecture

Advantages:

- It supports reusability.
- It is maintainable and modifiable.
- It supports concurrent execution.
- Some disadvantages associated with the data-flow architecture are listed below.
- It often degenerates to batch sequential system.
- It does not provide enough support for applications requires user interaction.
- It is difficult to synchronize two different but related streams.

2.Object-oriented Architecture

In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data.

In this style, components are represented as *objects* and they interact with each other through methods (connectors).

This architectural style has two important characteristics,.

- Objects maintain the integrity of the system.
- An object is not aware of the representation of other objects.

Some of the advantages associated with the object-oriented architecture are listed below.

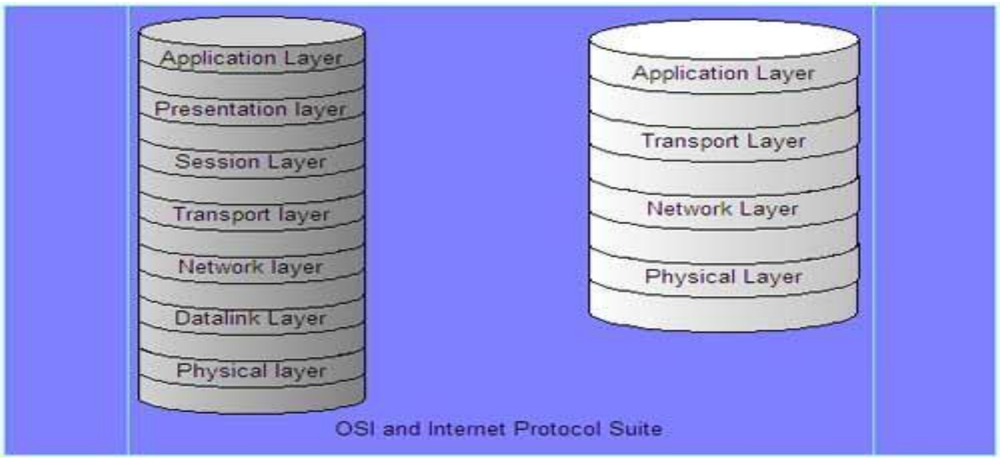
- It allows designers to decompose a problem into a collection of independent objects.
- The implementation detail of objects is hidden from each other and hence, they can be changed without affecting other objects.

3.Layered Architecture

In layered architecture, several layers (components) are defined with each layer performing a well-defined set of operations.

These layers are arranged in a hierarchical manner, each one built upon the one below it. Each layer provides a set of services to the layer above it and acts as a client to the layer below it.

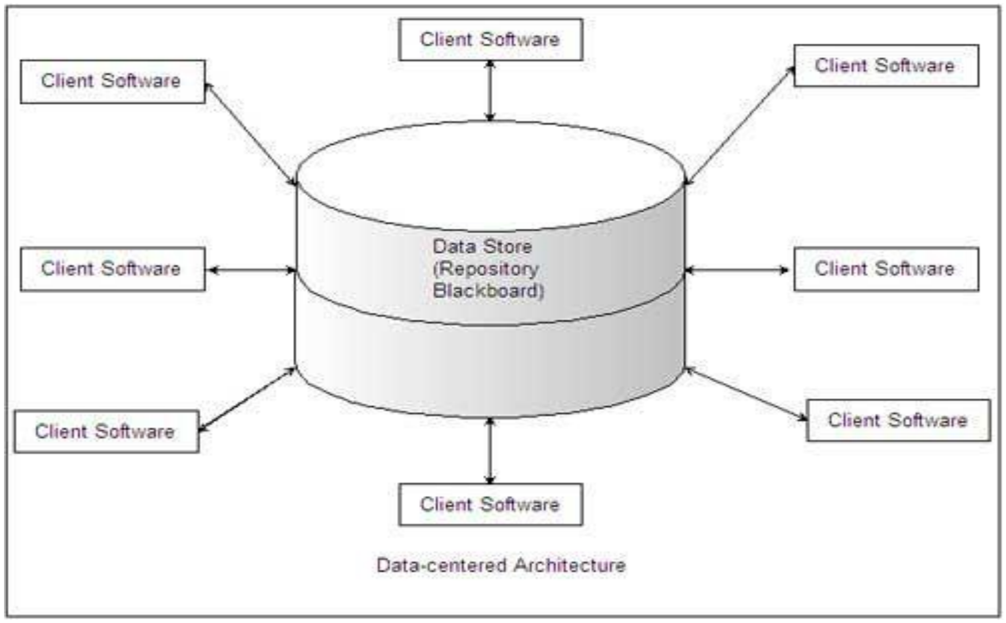
The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during interaction. One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system.



4.Data-centered Architecture

A data-centered architecture has two distinct components: a **central data structure** or data store (central repository) and a **collection of client software.**

The datastore (for example, a database or a file) represents the current state of the data and the client software performs several operations like add, delete, update, etc., on the data stored in the data store.



In this architectural style, new components corresponding to clients can be added and existing components can be modified easily without taking into account other clients. This is because client components operate independently of one another.

Some advantages of the data-centered architecture are listed below.

1. Clients operate independently of one another.
2. Data repository is independent of the clients.
3. It adds scalability (that is, new clients can be added easily).
4. It supports modifiability.
5. It achieves data integration in component-based development using blackboard.

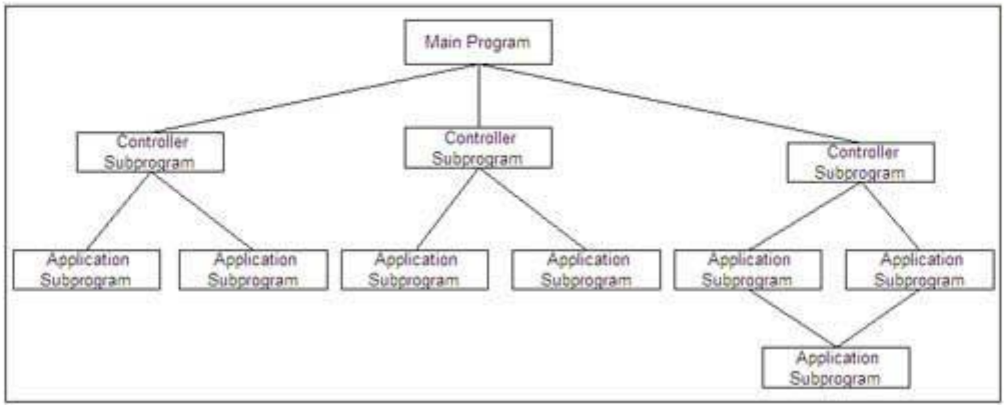
5.Call and Return Architecture

A call and return architecture enables software designers to achieve a program structure, which can be easily modified.

This style consists of the following two substyles.

- **Main program/subprogram architecture:** In this, function is decomposed into a control hierarchy where the main program invokes a number of program components, which in turn may invoke other components.

    - **Remote procedure call architecture:** In this, components of the main or subprogram architecture are distributed over a network across multiple computers.



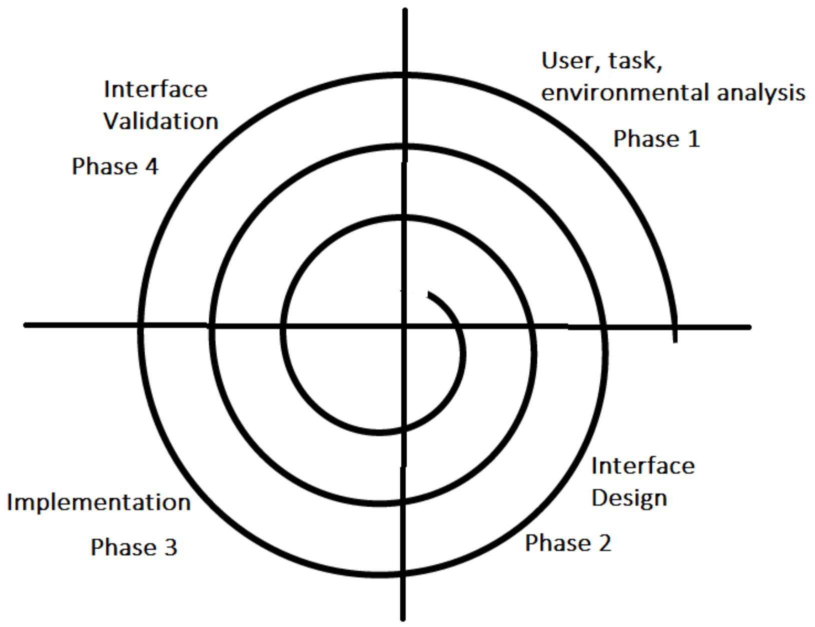
User interface is the front-end application view to which user interacts in order to use the software. The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interface screens

There are two types of User Interface:

1. **Command Line Interface:**
   - ➢ Command Line Interface provides a command prompt, where the user types the command and feeds to the system.
   - ➢ The user needs to remember the syntax of the command and its use.
2. **Graphical User Interface:**
   - ➢ Graphical User Interface provides the simple interactive interface to interact with the system.
   - ➢ GUI can be a combination of both hardware and software. Using GUI, user interprets the software.

User Interface Design Process:



The analysis and design process of a user interface is iterative and can be represented by a spiral model. The analysis and design process of user interface consists of **four framework activities**.

1. **User, task, environmental analysis, and modeling:**
    - ➢ Initially, the focus is based on the profile of users who will interact with the system, i.e. understanding, skill and knowledge, type of user, etc, based on the user's profile users are made into categories.
    - ➢ From each category requirements are gathered. Based on the requirements developer understand how to develop the interface.
    - ➢ Once all the requirements are gathered a detailed analysis is conducted.
    - ➢ In the analysis part, the tasks that the user performs to establish the goals of the system are identified, described and elaborated.
    - ➢ The analysis of the user environment focuses on the physical work environment.

Among the questions to be asked are:

- ✓ Where will the interface be located physically?
- ✓ Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- ✓ Does the interface hardware accommodate space, light, or noise constraints?
- ✓ Are there special human factors considerations driven by environmental factors?

2. **Interface Design:**
    - ➢ The goal of this phase is to define the set of interface objects and actions i.e. Control mechanisms that enable the user to perform desired tasks. Indicate how these control mechanisms affect the system.
    - ➢ Specify the action sequence of tasks and subtasks, also called a user scenario.
    - ➢ Indicate the state of the system when the user performs a particular task.

Always follow the three golden rules stated by Theo Mandel.

- ✓ Design issues such as **response time, command and action structure, error handling,** and help facilities are considered as the design model is refined.
- ✓ This phase serves as the foundation for the implementation phase.

3. **Interface construction and implementation:**
    - ➢ The implementation activity begins with the creation of prototype (model) that enables usage scenarios to be evaluated.
    - ➢ As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment can be used for completing the construction of an interface.
4. **Interface Validation:**
    - ➢ This phase focuses on testing the interface.
    - ➢ The interface should be in such a way that it should be able to perform tasks correctly and it should be able to handle a variety of tasks.
    - ➢ It should achieve all the user's requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.

Golden Rules:

The following are the golden rules stated by Theo Mandel that must be followed during the design of the interface.

Place the user in control:

- **Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions:** The user should be able to easily enter and exit the mode with little or no effort.
- **Provide for flexible interaction:** Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc, and hence all interaction mechanisms should be provided.
- **Allow user interaction to be interruptible and undoable:** When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had been done. The user should also be able to do undo operation.
- **Streamline interaction as skill level advances and allow the interaction to be customized:** Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn't feel bored while using the same interaction mechanism.
- **Hide technical internals from casual users:** The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.

- **Design for direct interaction with objects that appear on screen:** The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.

Reduce the user's memory load:

- **Reduce demand on short-term memory:** When users are involved in some complex tasks the demand on short-term memory is significant. So the interface should be designed in such a way to reduce the remembering of previously done actions, given inputs and results.
- **Establish meaningful defaults:** Always initial set of defaults should be provided to the average user, if a user needs to add some new features then he should be able to add the required features.
- **Define shortcuts that are intuitive:** Mnemonics should be used by the user. Mnemonics means the keyboard shortcuts to do some action on the screen.
- **The visual layout of the interface should be based on a real-world metaphor:** Anything you represent on a screen if it is a metaphor for real-world entity then users would easily understand.
- **Disclose information in a progressive fashion:** The interface should be organized hierarchically i.e. on the main screen the information about the task, an object or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

There are several key principles that software engineers should follow when designing user interfaces:

1. **User-centered design:** User interface design should be focused on the needs and preferences of the user. This involves understanding the user's goals, tasks, and context of use, and designing interfaces that meet their needs and expectations.

2. **Consistency:** Consistency is important in user interface design, as it helps users to understand and learn how to use an application. Consistent design elements such as icons, color schemes, and navigation menus should be used throughout the application.

3. **Simplicity**: User interfaces should be designed to be simple and easy to use, with clear and concise language and intuitive navigation. Users should be able to accomplish their tasks without being overwhelmed by unnecessary complexity.

4. **Feedback:** Feedback is important in user interface design, as it helps users to understand the results of their actions and confirms that they are making progress towards their goals. Feedback can take the form of visual cues, messages, or sounds.

5. **Accessibility:** User interfaces should be designed to be accessible to all users, regardless of their abilities. This involves considering factors such as color contrast, font size, and assistive technologies such as screen readers.

6. **Flexibility:** User interfaces should be designed to be flexible and customizable, allowing users to tailor the interface to their own preferences and needs.

Class Diagram

The class diagram depicts a **static view of an application**. It represents the types of objects residing in the system and the relationships between them.

A class consists of its objects, and also it may inherit from other classes. A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.

It shows the **attributes, classes, functions, and relationships** to give an overview of the software system. It constitutes class names, attributes, and functions in a **separate compartment** that helps in software development. Since it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a **structural diagram**.

Purpose of Class Diagrams

It is the only diagram that is widely used for construction, and it can be mapped with object-oriented languages. It is one of the most popular UML diagrams.

Following are the purpose of class diagrams given below:

1. It analyses and designs a static view of an application.
2. It describes the major responsibilities of a system.
3. It is a base for component and deployment diagrams.
4. It incorporates forward and reverse engineering.

Benefits of Class Diagrams

1. It can represent the object model for complex systems.
2. It reduces the maintenance time by providing an overview of how an application is structured before coding.
3. It provides a general schematic of an application for better understanding.
4. It represents a detailed chart by highlighting the desired code, which is to be programmed.
5. It is helpful for the stakeholders and the developers.

Vital components of a Class Diagram

The class diagram is made up of three sections:

- **Upper Section:** The upper section encompasses the **name of the class**.

  A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics.

  Rules:

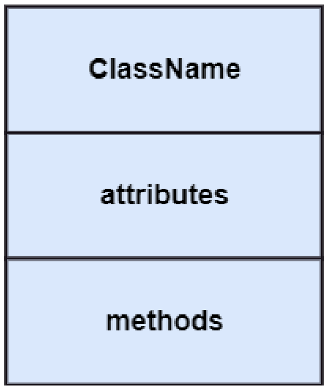
  - Capitalize the initial letter of the class name.
  - Place the class name in the center of the upper section.

        - A class name must be written in bold format.
        - The name of the abstract class should be written in italics format.
- **Middle Section:** The middle section **constitutes the attributes**, which describe the quality of the class.

    The attributes have the following characteristics:

- The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).
    a. The accessibility of an attribute class is illustrated by the visibility factors.
    b. A meaningful name should be assigned to the attribute, which will explain its usage inside the class.
- **Lower Section:** The lower sections contain **methods or operations**.

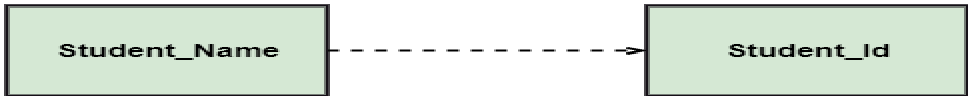
    The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.



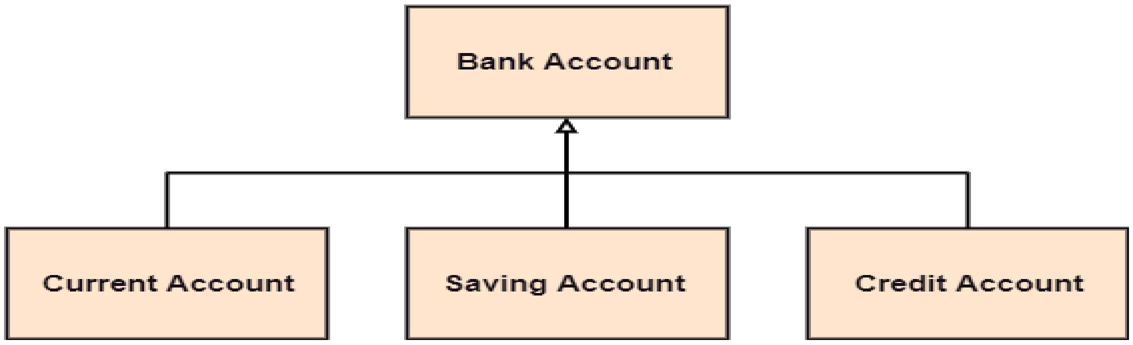
Relationships

In UML, relationships are of **three types:**

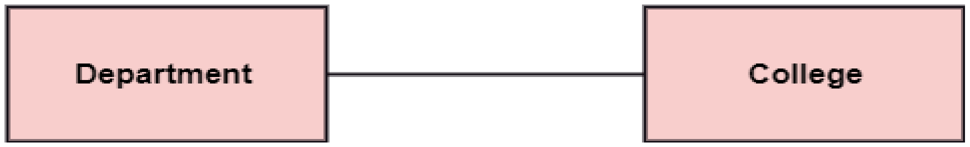
- **Dependency:** A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship. In the following example, Student_Name is dependent on the Student_Id.



- **Generalization:** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class. For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.

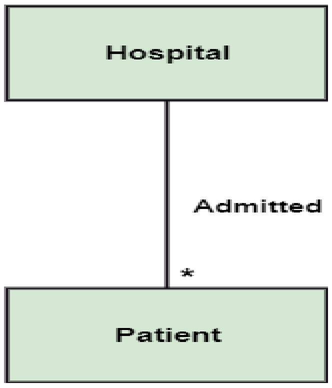


- **Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship.
  For example, a department is associated with the college.



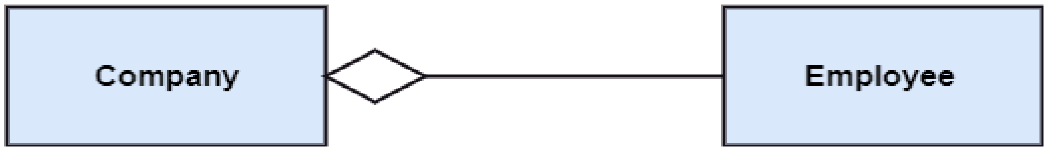
- **Multiplicity:** It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity.

For example, multiple patients are admitted to one hospital.



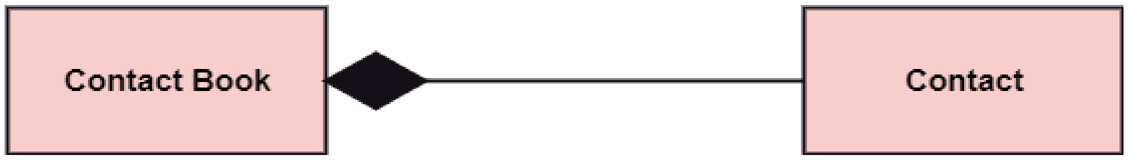
- **Aggregation:** An aggregation is a subset of association, which represents has a relationship. It is more specific then association. It defines a **part-whole or part-of relationship.** In this kind of relationship, the child class can exist independently of its parent class.

The company encompasses a number of employees, and even if one employee resigns, the company still exists.



- **Composition:** The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a **whole-part relationship.**

A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.



How to draw a Class Diagram?

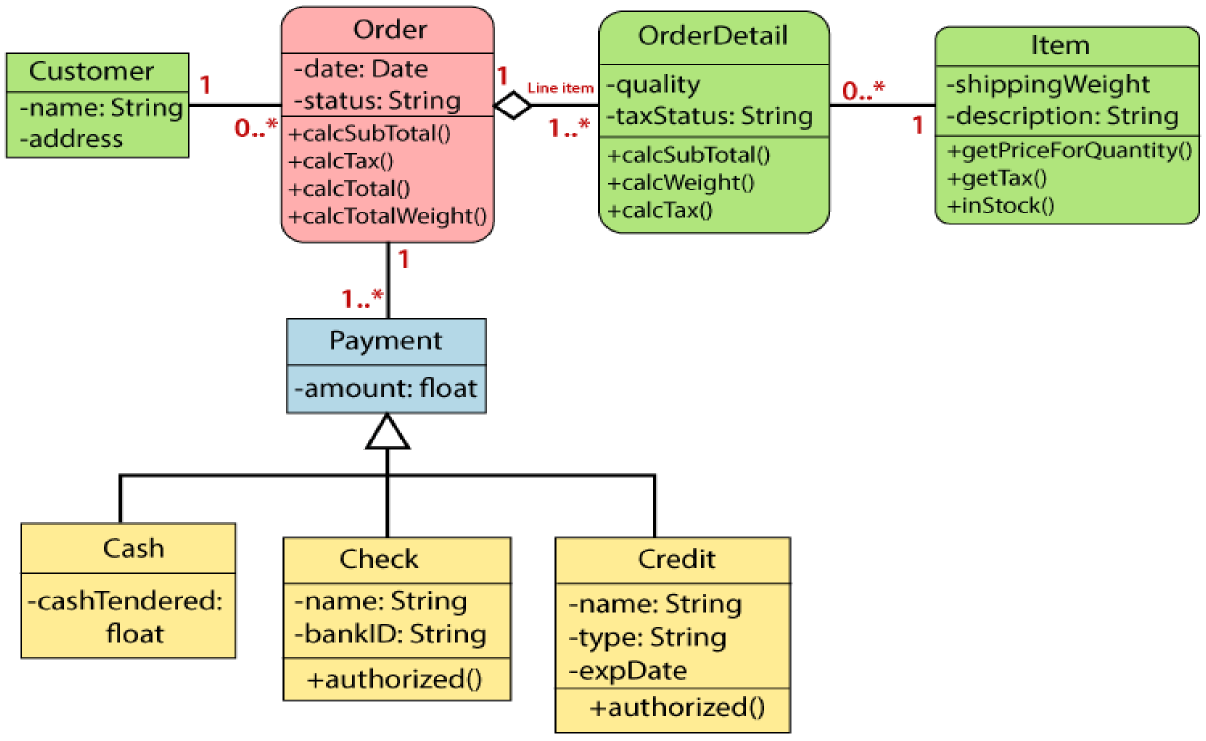
The class diagram is used most widely to **construct software applications**. It not only represents a static view of the system but also all the major aspects of an application. A collection of class diagrams as a whole represents a system.

Some key points that are needed to keep in mind while drawing a class diagram are given below:

1. To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
2. The objects and their relationships should be acknowledged in advance.
3. The attributes and methods (responsibilities) of each class must be known.
4. A minimum number of desired properties should be specified as more number of the unwanted property will lead to a complex diagram.
5. Notes can be used as and when required by the developer to describe the aspects of a diagram.
6. The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.

Class Diagram Example

A class diagram describing the sales order system is given below.



Usage of Class diagrams

Class diagrams can be used for the following purposes:

1. To describe the static view of a system.
2. To show the collaboration among every instance in the static view.
3. To describe the functionalities performed by the system.
4. To construct the software application using object-oriented languages.

State Diagram

The state machine diagram is also called the **State chart or State Transition diagram**, which shows the order of states *underwent by an object within the system*. It captures the software system's behavior. It models the behavior of a class, a subsystem, a package, and a complete system.

It tends out to be an efficient way of modeling the interactions and collaborations in the external entities and the system. It models event-based systems to handle the state of an object. It also defines several distinct states of a component within the system. Each object/component has a specific state.

Following are the types of a state machine diagram that are given below:

1. **Behavioral state machine**
   The behavioral state machine diagram records the behavior of an object within the system. It depicts an implementation of a particular entity. It models the behavior of the system.
2. **Protocol state machine**
   It captures the behavior of the protocol. The protocol state machine depicts the change in the state of the protocol and parallel changes within the system. But it does not portray the implementation of a particular component.

Why State Machine Diagram?

Since it records the dynamic view of a system, it portrays the behavior of a software application. During a lifespan, an object underwent several states, such that the lifespan exist until the program is executing. Each state depicts some useful information about the object.

It blueprints an interactive system that response back to either the internal events or the external ones. The execution flow from one state to another is represented by a state machine diagram. It visualizes an object state from its creation to its termination.

The main purpose is to depict each state of an individual object. It represents an interactive system and the entities inside the system. It records the dynamic behavior of the system.

Basic components of a statechart diagram –

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

Figure – initial state notation

2. **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.

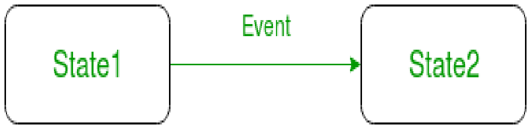


Figure – transition

3. **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.

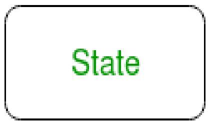


Figure – state notation

4. **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.

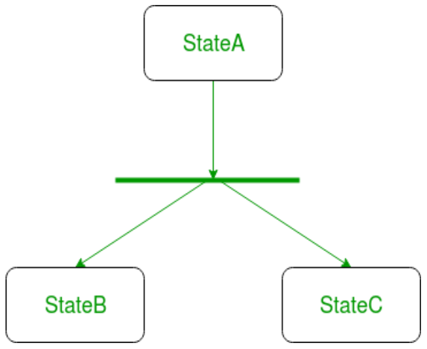


Figure – a diagram using the fork notation

5. **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.

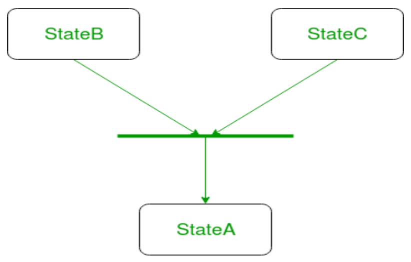


Figure – a diagram using join notation

6. **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.

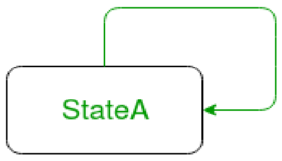


Figure – self transition notation

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

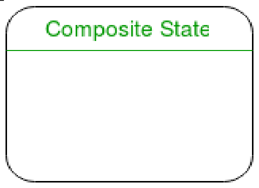


Figure – a state with internal activities

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

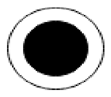


Figure – final state notation

Types of State

The UML consist of three states:

1. **Simple state:** It does not constitute any substructure.
2. **Composite state:** It consists of nested states (sub states), such that it does not contain more than one initial state and one final state. It can be nested to any level.
3. **Submachine state:** The submachine state is semantically identical to the composite state, but it can be reused.

How to Draw a State Machine Diagram?

The state machine diagram is used to portray various states underwent by an object. The change in one state to another is due to the occurrence of some event. All of the possible states of a particular component must be identified before drawing a state machine diagram.
Following are the steps that are to be incorporated while drawing a state machine diagram:

1. Identify the initial state and the final terminating states.
2. Identify the possible states in which the object can exist (boundary values corresponding to different attributes guide us in identifying different states).
3. Label the events which trigger these transitions.

When to use a State Machine Diagram?

It portrays the changes underwent by an object from the start to the end. It basically envisions how triggering an event can cause a change within the system.
State machine diagram is used for:

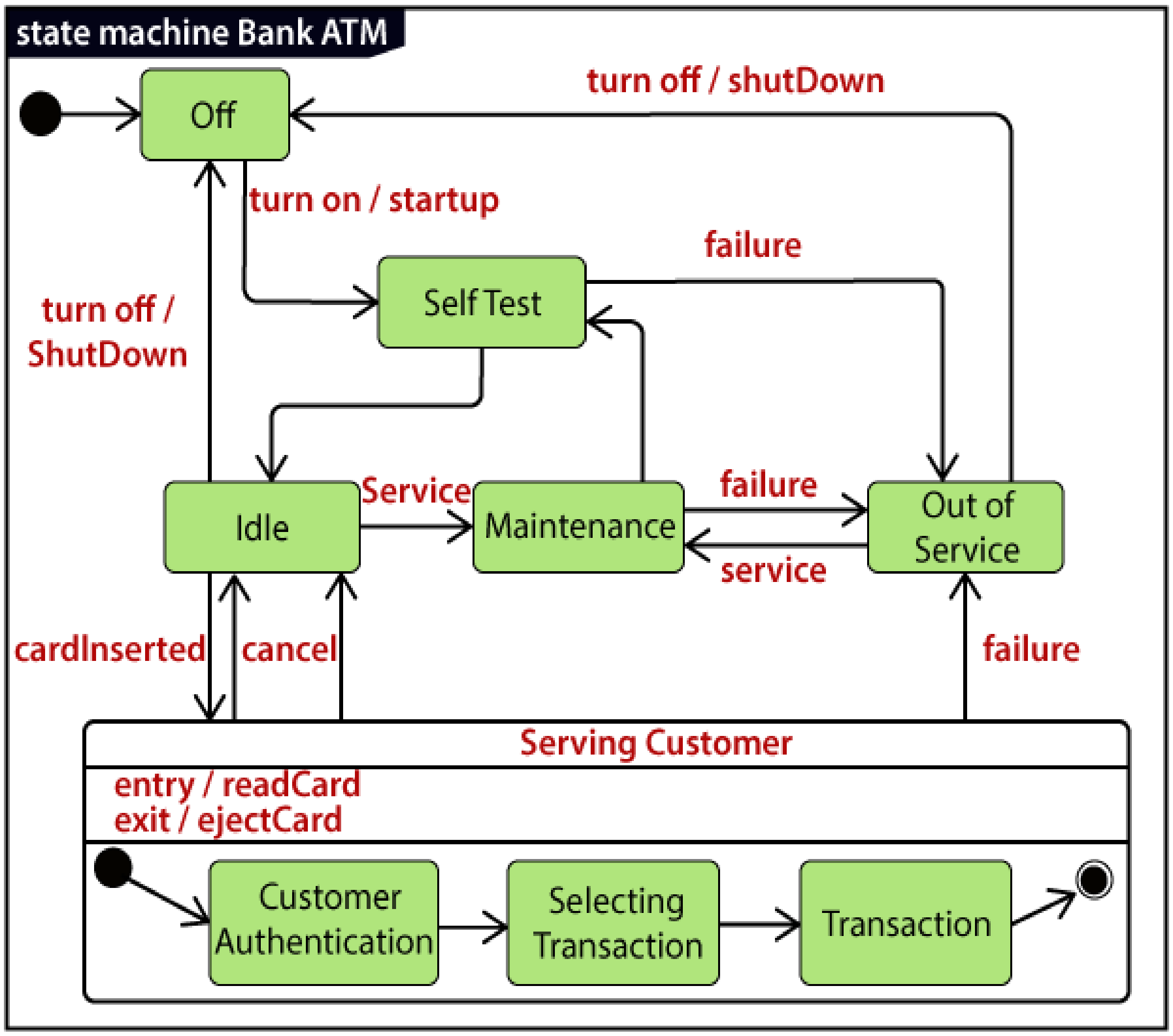
1. For modeling the object states of a system.
2. For modeling the reactive system as it consists of reactive objects.
3. For pinpointing the events responsible for state transitions.
4. For implementing forward and reverse engineering.

Example of a State Machine Diagram

An example of a top-level state machine diagram showing Bank Automated Teller Machine (ATM) is given below.

Initially, the ATM is turned off. After the power supply is turned on, the ATM starts performing the startup action and enters into the **Self-Test** state. If the test fails, the ATM will enter into the **Out Of Service** state, or it will undergo **a trigger less transition** to the **Idle** state. This is the state where the customer waits for the interaction.

Whenever the customer inserts the bank or credit card in the ATM's card reader, the ATM state changes from **Idle** to **Serving Customer**, the entry action **readCard** is performed after entering into **Serving Customer** state. Since the customer can cancel the transaction at any instant, so the transition from **Serving Customer** state back to the **Idle** state could be triggered by **cancel** event.



Here the **Serving Customer** is a composite state with sequential substates that are **Customer Authentication, Selecting Transaction,** and **Transaction**.

Customer Authentication and **Transaction** are the composite states itself is displayed by a hidden decomposition indication icon. After the transaction is finished, the **Serving Customer** encompasses a triggerless transition back to the **Idle** state. On leaving the state, it undergoes the exit action **ejectCard** that discharges the customer card.