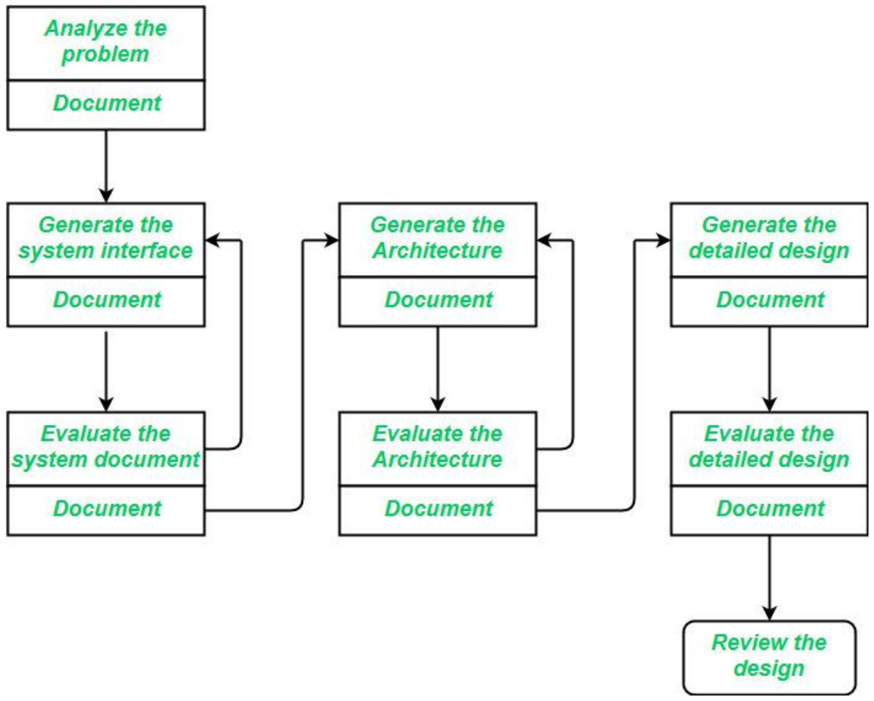
UNIT – III

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



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

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 focussed 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 on 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.

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. Design of the internals of the major components is ignored until the last phase of the design.

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

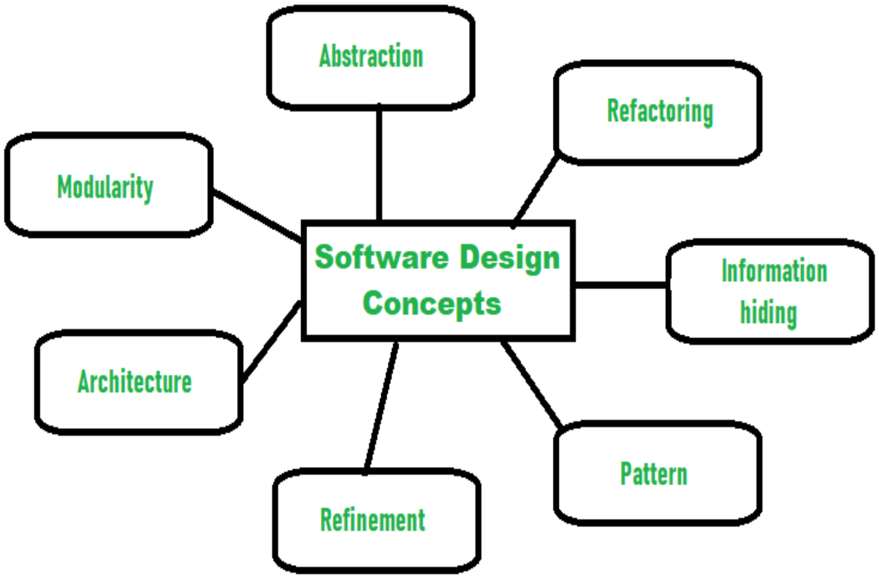
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

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. **Understandability:**
   A good design should be easily understandable, for which it should be modular and all the modules are arranged in layers.
4. **Completeness:**
   The design should have all the components like data structures, modules, and external interfaces, etc.
5. **Maintainability:**
   A good software design should be easily amenable to change whenever a change request is made from the customer side.

Design Concepts:

There are many concepts of software design and some of them are given below:



The following **points should be considered while designing Software:**

1. **Abstraction- hide Irrelevant data**

   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.

2. **Modularity- subdivide the system**

   Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.

3. **Architecture- design a structure of something**

   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.

4. **Refinement- removes impurities**

   Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

5. **Pattern- a repeated form**

   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

6. **Information Hiding- hide the information**

   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.

7. **Refactoring- reconstruct something**

   Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behavior of the design and improves the internal structure".

Design Quality:

According to studies, developers spend 80% of development time reading the code! So, code is read much more often than it is written.

Code changes are due to several reasons like bug fixing, upgrading the tech stack, and enhancements. Most companies have to spend up to 80% of the development budget for maintaining the code.

The design is good and, if we want to change it, we don't have to spend too much time and effort. And, also, we should extend it without fear. A good design should be adaptable to the changes, and the cost of the change should be as minimum as possible. Otherwise, adding new features costs more and more! If the code is maintainable and extendable, you can modify it without fear, and adding new features does not cost too much.

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.

A quality system subsists of the following:

Managerial Structure and Individual Responsibilities: A quality system is the responsibility of the organization as a whole. However, every organization has a sever quality department to perform various quality system activities. The quality system of an arrangement should have the support of the top management. Without help for the quality system at a high level in a company, some members of staff will take the quality system seriously.

Quality System Activities: The quality system activities encompass the following:

Auditing of projects
Review of the quality system
Development of standards, methods, and guidelines, etc.

Production of documents for the top management summarizing the effectiveness of the quality system in the organization.

Design Model:

Design modeling in software engineering represents the features of the software that helps engineer to develop it effectively, the architecture, the user interface, and the component level detail. Design modeling provides a variety of different views of the system like architecture plan for home or building. Different methods like data-driven, pattern-driven, or object-oriented methods are used for constructing the design model. All these methods use set of design principles for designing a model.

Designing a model is an important phase and is a multi-process that represent the data structure, program structure, interface characteristic, and procedural details. It is mainly classified into four categories – Data design, architectural design, interface design, and component-level design.

- **Data design:** It represents the data objects and their interrelationship in an entity-relationship diagram. Entity-relationship consists of information required for each entity or data objects as well as it shows the relationship between these objects. It shows the structure of the data in terms of the tables. It shows three type of relationship – One to one, one to many, and many to many. In one to one relation, one entity is connected to another entity. In one many relation, one Entity is connected to more than one entity. un many to many relations one entity is connected to more than one entity as well as other entity also connected with first entity using more than one entity.

- **Architectural design:** It defines the relationship between major structural elements of the software. It is about decomposing the system into interacting components. It is expressed as a block diagram defining an overview of the system structure – features of the components and how these components communicate with each other to share data. It defines the structure and properties of the component that are involved in the system and also the inter-relationship among these components.

- **User Interfaces design:** It represents how the Software communicates with the user i.e. the behavior of the system. It refers to the product where user interact with controls or displays of the product. For example, Military, vehicles, aircraft, audio equipment, computer peripherals are the areas where user interface design is implemented. UI design becomes efficient only after performing usability testing. This is done to test what works and what does not work as expected. Only after making the repair, the product is said to have an optimized interface.

- **Component level design:** It transforms the structural elements of the software architecture into a procedural description of software components. It is a perfect way to share a large amount of data. Components need not be concerned with how data is managed at a centralized level i.e. components need not worry about issues like backup and security of the data.

Principles of Design Model:

Design must be traceable to the analysis model:

Analysis model represents the information, functions, and behavior of the system. Design model translates all these things into architecture – a set of subsystems that implement major functions and a set of component kevel design that are the realization of Analysis classes. This implies that design model must be traceable to the analysis model.

- **Always consider architecture of the system to be built:**

Software architecture is the skeleton of the system to be built. It affects interfaces, data structures, behavior, program control flow, the manner in which testing is conducted, maintainability of the resultant system, and much more.

- **Focus on the design of the data:**

Data design encompasses the manner in which the data objects are realized within the design. It helps to simplify the program flow, makes the design and implementation of the software components easier, and makes overall processing more efficient.

- **User interfaces should consider the user first:**

The user interface is the main thing of any software. No matter how good its internal functions are or how well designed its architecture is but if the user interface is poor and end-users don’t feel ease to handle the software then it leads to the opinion that the software is bad.

- **Components should be loosely coupled:**

Coupling of different components into one is done in many ways like via a component interface, by messaging, or through global data. As the level of coupling increases, error propagation also increases, and overall maintainability of the software decreases. Therefore, component coupling should be kept as low as possible.

- **Interfaces both user and internal must be designed:**

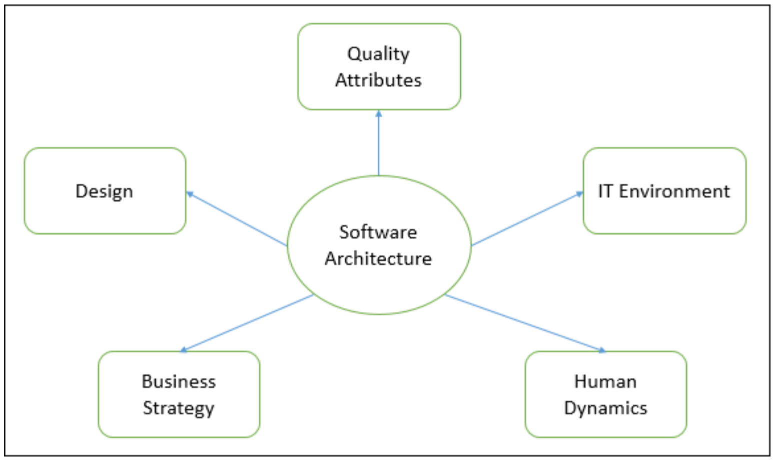
The data flow between components decides the processing efficiency, error flow, and design simplicity. A well-designed interface makes integration easier and tester can validate the component functions more easily.

- **Component level design should exhibit Functional independence:**

It means that functions delivered by component should be cohesive i.e. it should focus on one and only one function or sub-function.

Software Architecture and Design:

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In **Architecture**, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

Software Architecture:

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

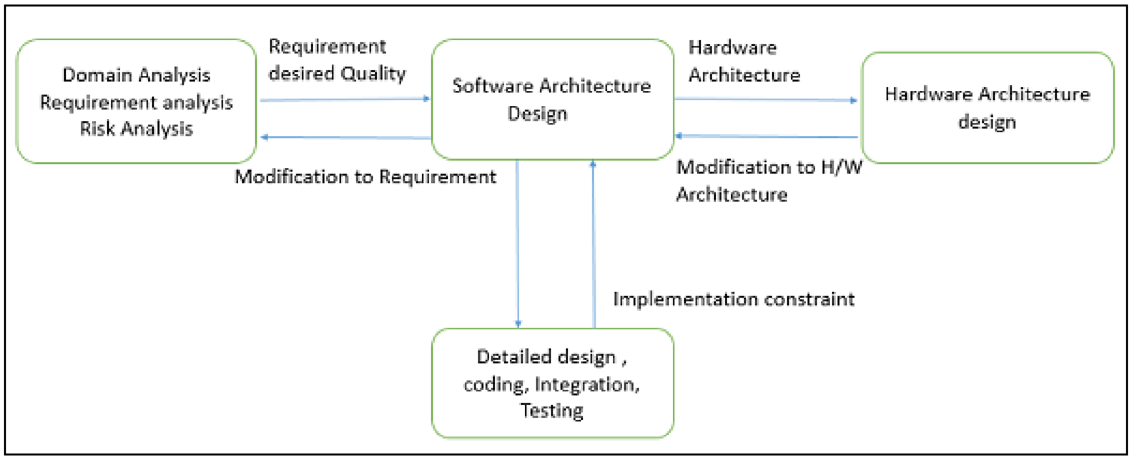
- It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
- Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of −
    - Selection of structural elements and their interfaces by which the system is composed.
    - Behavior as specified in collaborations among those elements.
    - Composition of these structural and behavioral elements into large subsystem.
    - Architectural decisions align with business objectives.
    - Architectural styles guide the organization.

Software Design:

Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows −

- To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.

It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.


Domain Analysis
Requirement analysis
Risk Analysis
Requirement
desired Quality
Modification to Requirement
Software Architecture Design
Hardware Architecture
Modification to H/W Architecture
Hardware Architecture design
Implementation constraint
Detailed design , coding, Integration, Testing

Goals of Architecture:

The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.

Some of the other goals are as follows −

- Expose the structure of the system, but hide its implementation details.
- Realize all the use-cases and scenarios.
- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization’s market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

Limitations:

Software architecture is still an emerging discipline within software engineering. It has the following limitations −

- Lack of tools and standardized ways to represent architecture.
- Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
- Lack of awareness of the importance of architectural design to software development.

- Lack of understanding of the role of software architect and poor communication among stakeholders.
- Lack of understanding of the design process, design experience and evaluation of design.

Role of Software Architect:

A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas −

Design Expertise

- Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.
- Lead the development team and coordinate the development efforts for the integrity of the design.
- Should be able to review design proposals and tradeoff among themselves.

Domain Expertise

- Expert on the system being developed and plan for software evolution.
- Assist in the requirement investigation process, assuring completeness and consistency.
- Coordinate the definition of domain model for the system being developed.

Technology Expertise

- Expert on available technologies that helps in the implementation of the system.
- Coordinate the selection of programming language, framework, platforms, databases, etc.

Methodological Expertise

- Expert on software development methodologies that may be adopted during SDLC (Software Development Life Cycle).
- Choose the appropriate approaches for development that helps the entire team.

Hidden Role of Software Architect

- Facilitates the technical work among team members and reinforcing the trust relationship in the team.
- Information specialist who shares knowledge and has vast experience.
- Protect the team members from external forces that would distract them and bring less value to the project.

Deliverables of the Architect

- A clear, complete, consistent, and achievable set of functional goals
- A functional description of the system, with at least two layers of decomposition
- A concept for the system
- A design in the form of the system, with at least two layers of decomposition
- A notion of the timing, operator attributes, and the implementation and operation plans
- A document or process which ensures functional decomposition is followed, and the form of interfaces is controlled

Quality Attributes:

Quality is a measure of excellence or the state of being free from deficiencies or defects. Quality attributes are the system properties that are separate from the functionality of the system.

Implementing quality attributes makes it easier to differentiate a good system from a bad one. Attributes are overall factors that affect runtime behavior, system design, and user experience.

They can be classified as −

Static Quality Attributes:

Reflect the structure of a system and organization, directly related to architecture, design, and source code. They are invisible to end-user, but affect the development and maintenance cost, e.g.: modularity, testability, maintainability, etc.

Dynamic Quality Attributes:

Reflect the behavior of the system during its execution. They are directly related to system's architecture, design, source code, configuration, deployment parameters, environment, and platform.

They are visible to the end-user and exist at runtime, e.g. throughput, robustness, scalability, etc.

Data Design:

Data design is the first design activity, which results in less complex, modular and efficient program structure. The information domain model developed during analysis phase is transformed into data structures needed for implementing the software. The data objects, attributes, and relationships depicted in entity relationship diagrams and the information stored in data dictionary provide a base for data design activity. During the data design process, data types are specified along with the integrity rules required for the data. For specifying and designing efficient data structures, some principles should be followed. These principles are listed below.

1. The data structures needed for implementing the software as well-as the operations that can be applied on them should be identified.
2. A data dictionary should be developed to depict how different data objects interact with each other and what constraints are to be imposed on the elements of data structure.
3. Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
4. Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
5. A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.
6. Language used for developing the system should support abstract data types.

The structure of data can be viewed at three levels, namely, *program* component level, application level, and business level. At the **program component level,** the design of data structures and the algorithms required to manipulate them is necessary, if high-quality software is desired. At the **application level,** it is crucial to convert the data model into a database so that the specific business objectives of a system could be achieved. At the **business level,** the collection of information stored in different databases should be reorganized into data warehouse, which enables data mining that has an influential impact on the business.

Architectural Style

The architectural style shows how do we organize our code, or how the system will look like from 10000 feet helicopter view to show the highest level of abstraction of our system design. Furthermore, when building the architectural style of our system we focus on layers and modules and how they are communicating with each other.

There are different types of architectural styles, and moreover, we can mix them and produce a hybrid style that consists of a mix between two and even more architectural styles. Below is a list of architectural styles and examples for each category:

- **Structure architectural styles:** such as layered, pipes and filters and component-based styles.
- **Messaging styles:** such as Implicit invocation, asynchronous messaging and publish-subscribe style.
- **Distributed systems:** such as service-oriented, peer to peer style, object request broker, and cloud computing styles.
- **Shared memory styles:** such as role-based, blackboard, database-centric styles.
- **Adaptive system styles:** such as microkernel style, reflection, domain-specific language styles.

Design Patterns

Design patterns are accumulative best practices and experiences that software professionals used over the years to solve the general problem by – trial and error – they faced during software development. The Gang of Four (GOF, refers to Eric Gamma, Richard Helm, Ralf Johnson, and John Vlissides) wrote a book in 1994 titled with “Design Pattern – Elements of reusable object-oriented software” in which they suggested that design patterns are based on two main principles of object-oriented design:

- Develop to an interface, not to an implementation.
- Favor object composition over inheritance.

Also, they presented that the design patterns set contains 23 patterns and categorized into three main sets:

1. Creational design patterns:

Provide a way to create objects while hiding the creation logic. Thus, the object creation is to be done without instantiating objects directly with the “New” keyword to gives the flexibility to decide which objects need to be created for a given use case. The creational design patterns are:

- **Abstract factory pattern:** provide an interface to create objects without specifying the classes.
- **Singleton pattern:** provide only a single instance of the calls and global access to this instance.
- **Builder Pattern:** Separate the construction from representation and allows the same construction to create multiple representations.
- **Prototype pattern:** creating duplicate without affecting the performance and memory. So the duplicate object is built from the skeleton of an existing object.

2. Structural patterns:

Concerned with class and object composition. The Structural design patterns are:

- **Adapter pattern:** it works as a bridge between two incompatible interfaces and compines their capabilities.
- **Bridge pattern:** provide a way to decouple the abstraction from its implementation.
- **Filter pattern:** Also known as criteria pattern, it provides a way to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations.
- **Composite pattern:** provide a way to treat a group of objects in a similar way as a single object. It composes objects in term of a tree structure to represent part as well as a whole hierarchy
- **Decorator pattern:** allows adding new functionality to an existing object without altering its structure.
- **Façade pattern:** provide a unified interface to a set of interfaces.it hides the complexities of the system and provides an interface to the client using which the client can access the system.
- **Flyweight pattern:** reduce the number of objects created and to decrease memory footprint and increase performance. It helps in reusing already existing similar kind objects by storing them and creates a new object when no matching object is found.
- **Proxy pattern:** provides a place holder to another object to control access to it. The object has an original object to interface its functionality to the outer world.

3. Behavioral patterns:

Behavioral patterns are concerned with communications between objects. The following is the list of behavioral patterns:

- **Responsibility pattern:** creates a chain of receiver objects for a request. This pattern decouples the sender and receiver of a request based on the type of request.
- **Command pattern:** it's a data-driven pattern in which A request is wrapped under an object as command and passed to an invoker object.
- **Interpreter pattern:** provides a way to evaluate language grammar or expression. It involves implementing an expression interface that tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine, etc.
- **Iterator pattern:** provides a way to access the elements of a collection object in a sequential manner without any need to know its underlying representation.
- **Mediator pattern:** used to reduce communication complexity between multiple objects or classes. It provides a mediator class that normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling.
- **Memento pattern:** used to restore the state of an object to a previous state.
- **Observer pattern:** used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.
- **State pattern:** is used to change the class behavior based on its state.
- **Null object pattern:** helps to avoid null references by having a default object.
- **Strategy pattern:** provides a way to change class behavior or its algorithm at run time.
- **Template pattern:** an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.

- **Visitor pattern:** used to change the executing algorithm of an element class.

There are two more subsets of design pattern can be added to the 3 categories of design pattern:

It shows the system design at the highest level of abstraction. It also shows the high-level module of the application and how these modules are interacting. On the other hand, architectural patterns have a huge impact on system implementation horizontally and vertically. Finally, the design patterns are used to solve localized issues during the implementation of the software. Also, it has a lower impact on the code than the architectural patterns since the design pattern is more concerned with a specific portion of code implementation such as initializing objects and communication between objects.

Software Engineering | Architectural Design:

Introduction:

The software needs the architectural design to represents the design of software. 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.” The software that is built for computer-based systems can exhibit one of these many architectural styles.

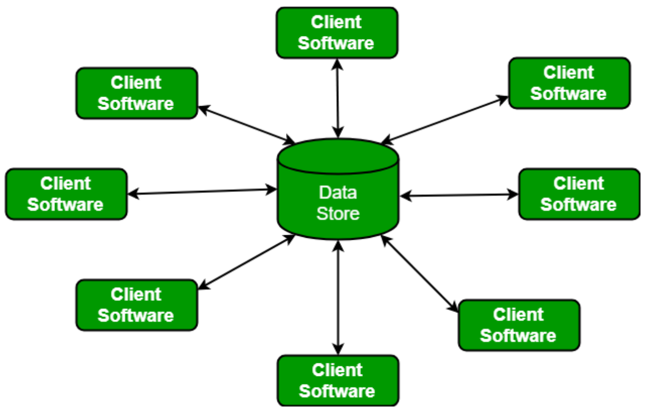
Each style will describe a system category that consists of :

- A set of components(eg: a database, computational modules) that will perform a function required by the system.
- The set of connectors will help in coordination, communication, and cooperation between the components.
- Conditions that how components can be integrated to form the system.
- Semantic models that help the designer to understand the overall properties of the system.

The use of architectural styles is to establish a structure for all the components of the system.

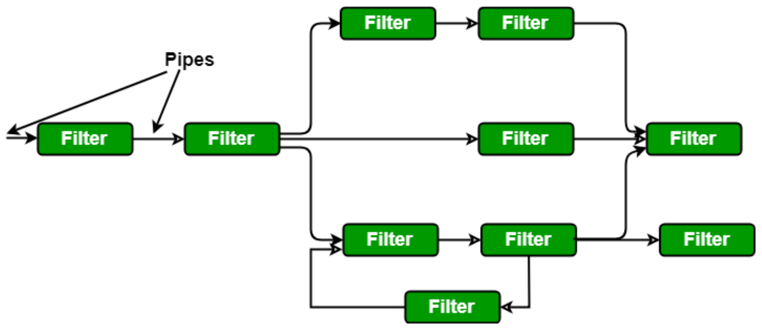
1. **Data centred architectures:**

   - A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
   - The figure illustrates a typical data centered style. The client software access a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
   - This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
   - Data can be passed among clients using blackboard mechanism.



Data flow architectures:

- This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.
- Pipes are used to transmit data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

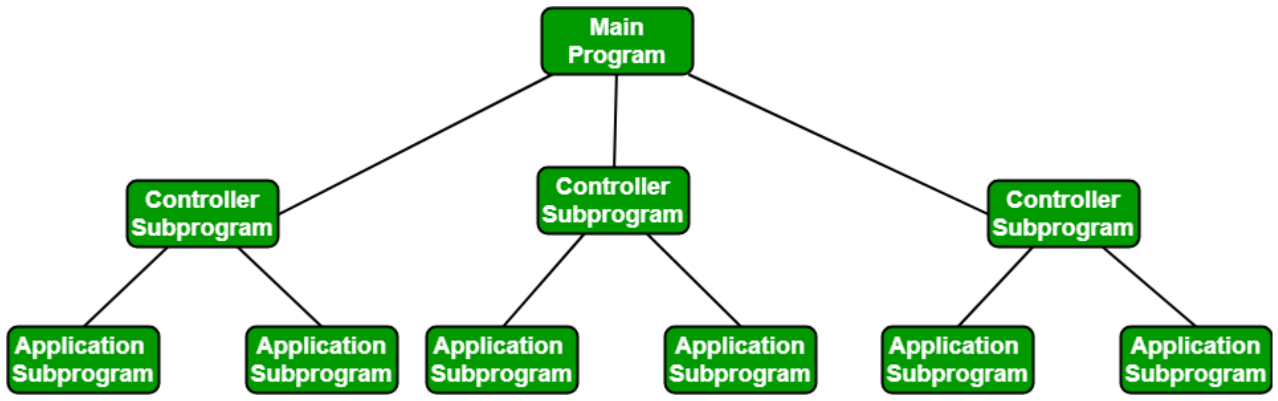


Call and Return architectures:

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Remote procedure call architecture:** This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control

hierarchy. Main program contains number of subprograms that can invoke other components.

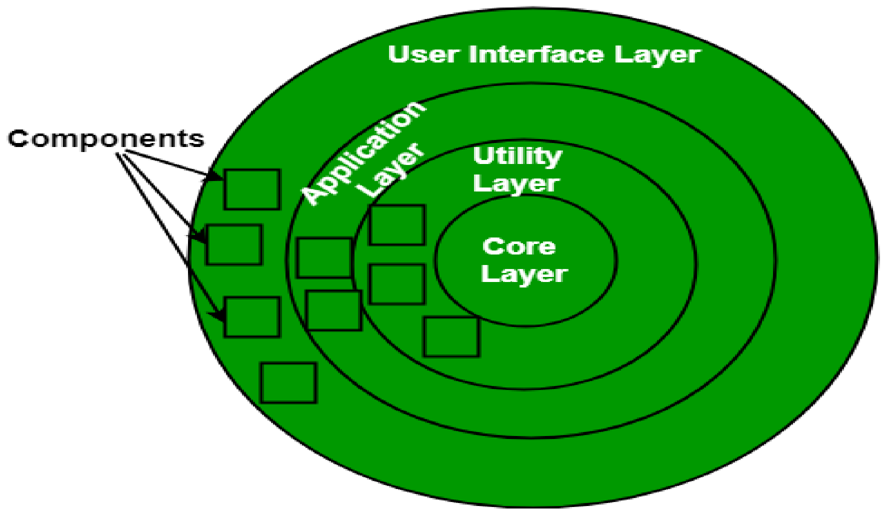


1. **Object Oriented architecture:**

   The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

2. **Layered architecture:**

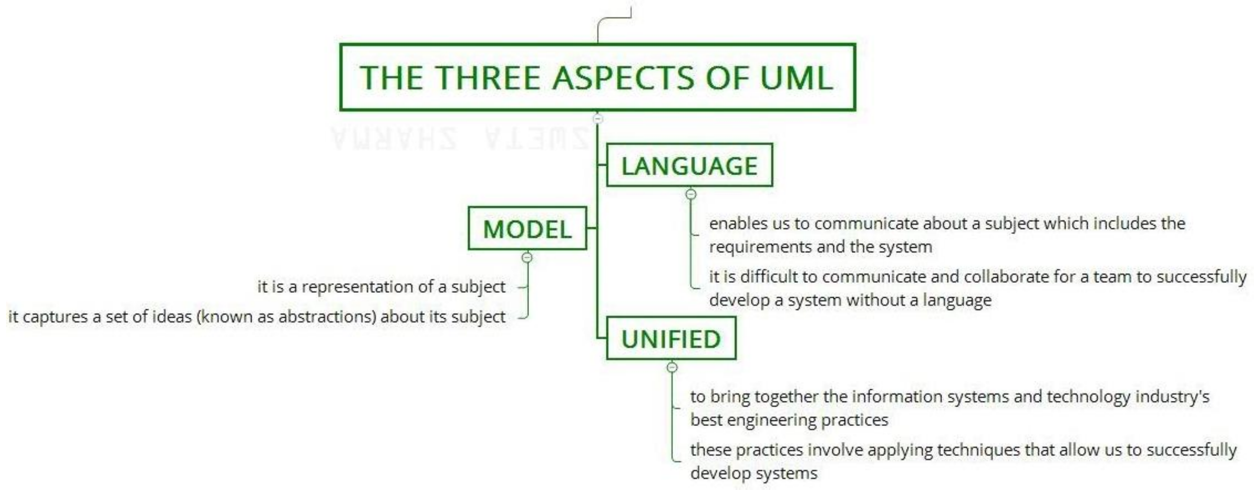
   - A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
   - At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)
   - Intermediate layers to utility services and application software functions.



Conceptual Model of Unified Modeling Language (UML):

The Unified Modeling Language (UML) is a standard visual language for describing and modelling software blueprints. The UML is more than just a graphical language. Stated formally, the UML is for: Visualizing, Specifying, Constructing, and Documenting.

Three Aspects of UML:


THE THREE ASPECTS OF UML
LANGUAGE
enables us to communicate about a subject which includes the requirements and the system
it is difficult to communicate and collaborate for a team to successfully develop a system without a language
MODEL
it is a representation of a subject
it captures a set of ideas (known as abstractions) about its subject
UNIFIED
to bring together the information systems and technology industry's best engineering practices
these practices involve applying techniques that allow us to successfully develop systems

1. Language:

- It enables us to communicate about a subject which includes the requirements and the system.
- It is difficult to communicate and collaborate for a team to successfully develop a system without a language.

2. Model:

- It is a representation of a subject.
- It captures a set of ideas *(known as abstractions)* about its subject.

3. Unified:

- It is to bring together the information systems and technology industry’s best engineering practices.
- These practices involve applying techniques that allow us to successfully develop systems.

A Conceptual Model:

A conceptual model of the language underlines the three major elements:

• The Building Blocks
• The Rules
• Some Common Mechanisms

Once you understand these elements, you will be able to read and recognize the models as well as create some of them.

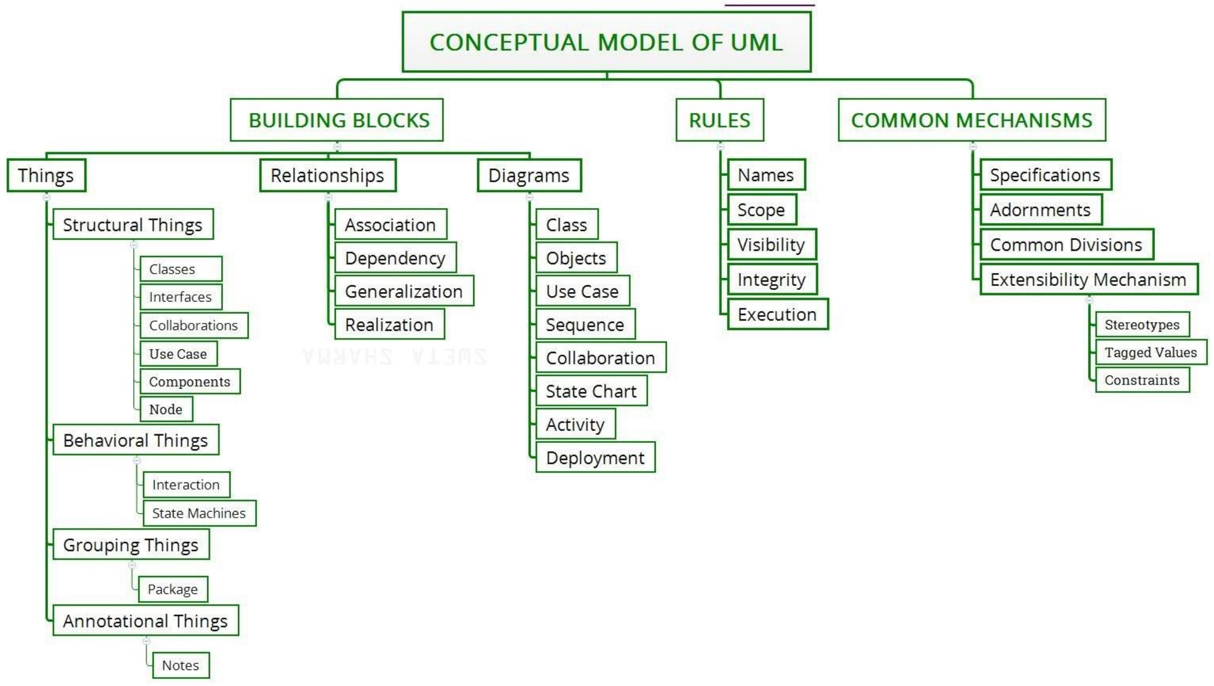


Figure – Conceptual Model of the UML

Building Blocks:

The vocabulary of the UML encompasses three kinds of building blocks:

1. **Things:**
   Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

   There are 4 kinds of things in the UML:

1. Structural things	**2.** Behavioral things
3. Grouping things	**4.** Annotational things

These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

2. **Relationships:**

3. There are 4 kinds of relationships in the UML:

   1. Dependency
   2. Association
   3. Generalization
   4. Realization

   These relationships are the basic relational building blocks of the UML.

4. **Diagrams:**

   It is the graphical presentation of a set of elements. It is rendered as a connected graph of vertices (things) and arcs (relationships).

   1. Class diagram
   2. Object diagram
   3. Use case diagram
   4. Sequence diagram
   5. Collaboration diagram
   6. Statechart diagram
   7. Activity diagram
   8. Component diagram
   9. Deployment diagram

Rules:

The UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistent and in harmony with all its related models.

The UML has semantic rules for:

1. **Names –** What you can call things, relationships, and diagrams.
2. **Scope –** The context that gives specific meaning to a name.
3. **Visibility –** How those names can be seen and used by others.
4. **Integrity –** How things properly and consistently relate to one another.
5. **Execution –** What it means to run or simulate a dynamic model.

Common Mechanisms:

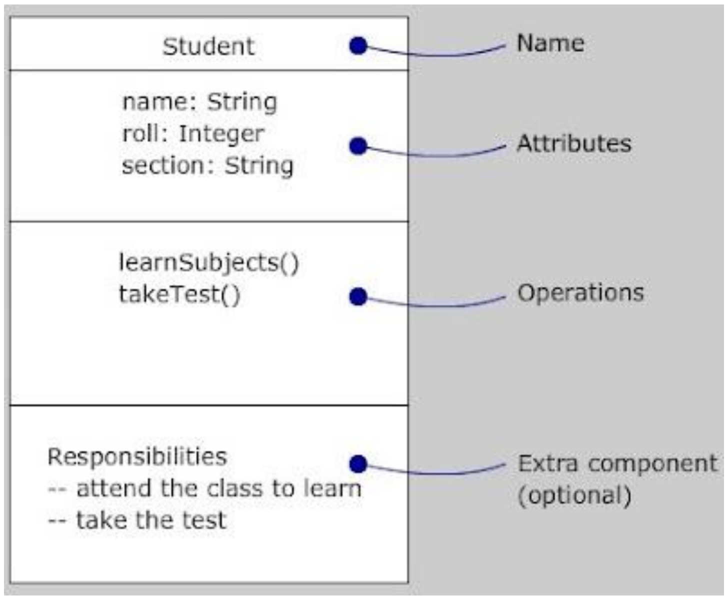
The UML is made simpler by the four common mechanisms. They are as follows:

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

BASIC STRUCTURAL MODELING:

Classes

- In object-oriented programming, programs are organized as cooperative collections of objects
- All objects are instances of a class
- A class is a description of a set of objects that share the same attributes, behaviors, relationships and semantics
- Hence, class becomes the important building block of any object-oriented system
- A class provides a blueprint for creating objects
- It can represent a software, a hardware or even a conceptual thing
- A class usually collaborates with other classes
- It is shown as a rectangle with three compartments that includes its name, attributes and operations

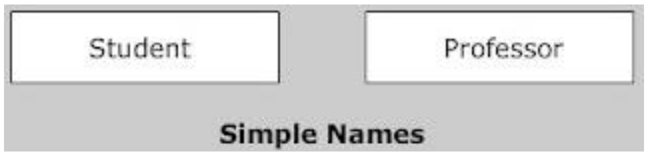


A Class typically has:

	-Names -Attributes -Operations -Responsibilities

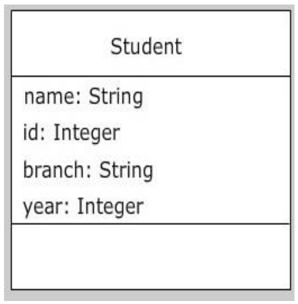
Names:

- Extracted from the problem domain (statement)
- Nouns or noun phrases
- Concise and descriptive



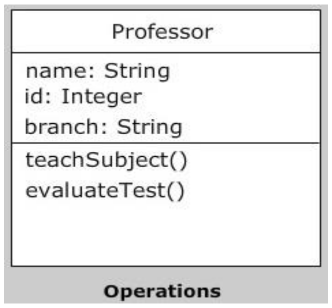
Attributes:

- Describe common characteristics of the objects in the class
- The values of attributes determine the state of an object
- For example, the value of the year attribute of a student indicates which year the student is studying
- They are usually nouns extracted from the problem statement



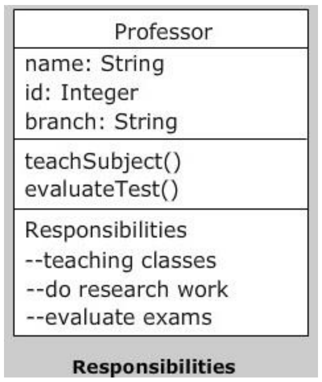
Operations:

- Operations indicate the services provided by the objects of the class
- They are usually verbs or verb phrases from the problem statement



Responsibilities:

- A Contract or obligations of a class to the users of the class means that an object of the class has to fulfill these responsibilities when requested by other objects



Responsibilities

What is UML?
It is the general-purpose modeling language used to visualize the system. It is a graphical language that is standard to the software industry for specifying, visualizing, constructing, and documenting the artifacts of the software systems, as well as for business modeling.
Benefits of UML:

- Simplifies complex software design, can also implement OOPs like a concept that is widely used.
- It reduces thousands of words of explanation in a few graphical diagrams that may reduce time consumption to understand.
- It makes communication more clear and more real.
- It helps to acquire the entire system in a view.
- It becomes very much easy for the software programmer to implement the actual demand once they have a clear picture of the problem.

Types of UML: The UML diagrams are divided into two parts: Structural UML diagrams and Behavioral UML diagrams which are listed below:

Structural UML diagrams	**Behavioral UML diagrams**
Class diagram	Activity diagram
Package diagram	Sequence diagram
Object diagram	Use case diagram
Component diagram	State diagram
Composite structure diagram	Communication diagram
Deployment diagram	Interaction overview diagram
	Timing diagram

CLASS DIAGRAM

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of objectoriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

How to Draw a Class Diagram?

Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the drawing procedure of class diagram.

Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

The following points should be remembered while drawing a class diagram −

- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.

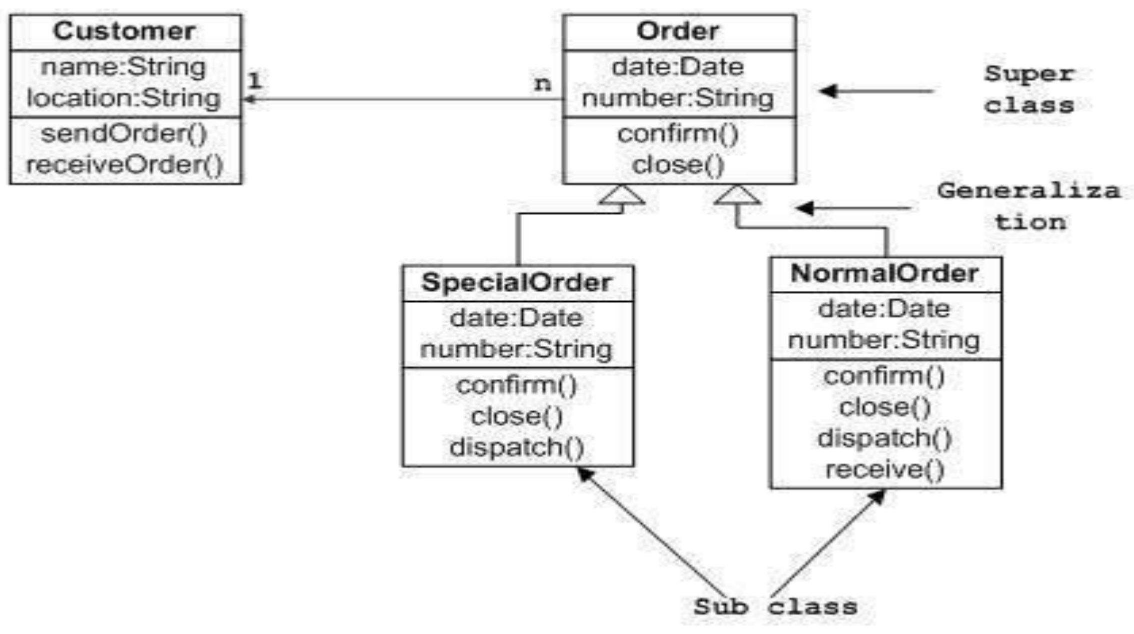
- Responsibility (attributes and methods) of each class should be clearly identified
- For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
- Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
- Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

The following diagram is an example of an Order System of an application. It describes a particular aspect of the entire application.

- First of all, Order and Customer are identified as the two elements of the system. They have a one-to-many relationship because a customer can have multiple orders.
- Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
- The two inherited classes have all the properties as the Order class. In addition, they have additional functions like dispatch () and receive ().

The following class diagram has been drawn considering all the points mentioned above.

Sample Class Diagram



Where to Use Class Diagrams

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

SEQUENCE DIAGRAM

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

We have two types of interaction diagrams in UML. One is the sequence diagram and the other is the collaboration diagram. The sequence diagram captures the time sequence of the message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

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.

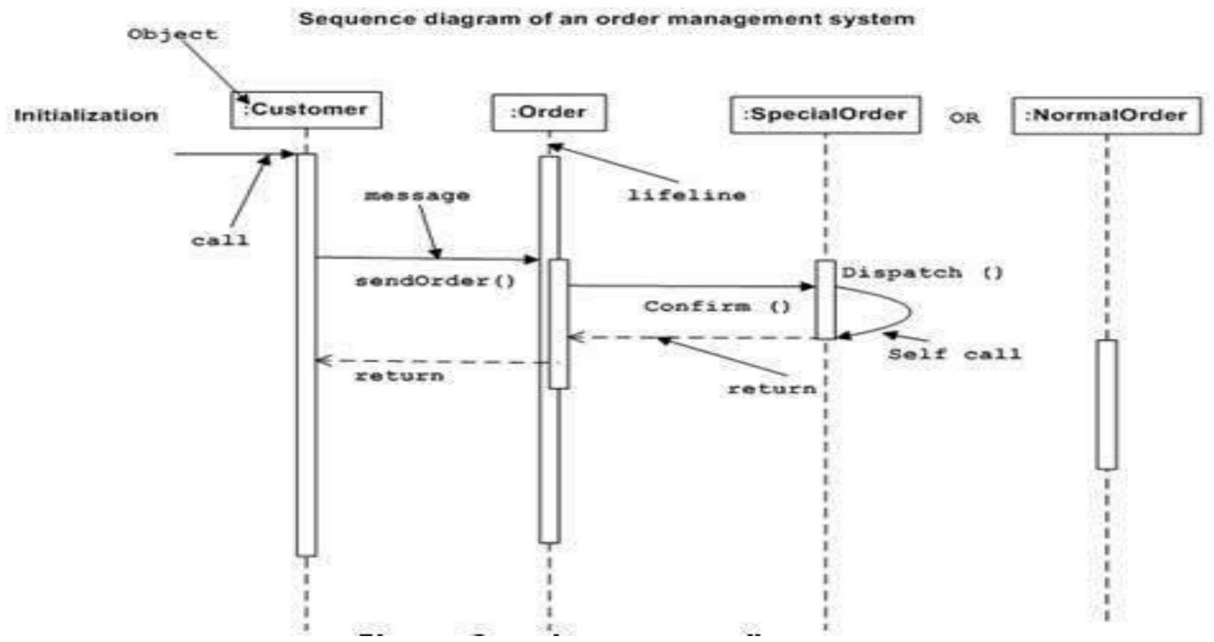
Following are two interaction diagrams modeling the order management system. The first diagram is a sequence diagram and the second is a collaboration diagram

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.

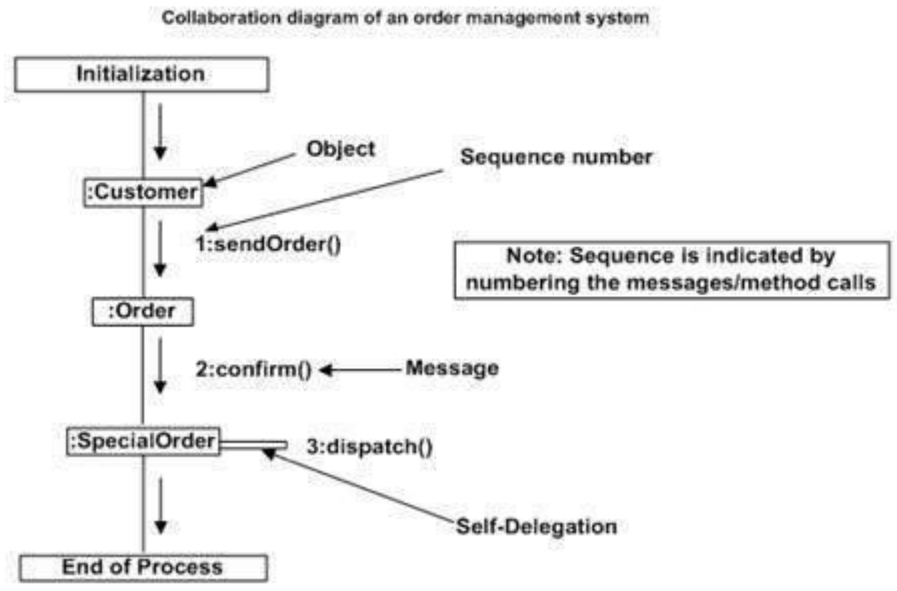


The Collaboration Diagram

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

USE CASE DIAGRAM

To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running/operating.

Only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML, there are five diagrams available to model the dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

Hence to model the entire system, a number of use case diagrams are used.

Purpose of Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence, collaboration, and Statechart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

How to Draw a Use Case Diagram

Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases.

We can say that use cases are nothing but the system functionalities written in an organized manner. The second thing which is relevant to use cases are the actors. Actors can be defined as something that interacts with the system.

Actors can be a human user, some internal applications, or may be some external applications. When we are planning to draw a use case diagram, we should have the following items identified.

- Functionalities to be represented as use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram

- The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.
- Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
- Use notes whenever required to clarify some important points.

Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases **(Order, SpecialOrder, and NormalOrder)** and one actor which is the customer.

The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship. Another important point is to identify the system boundary, which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.

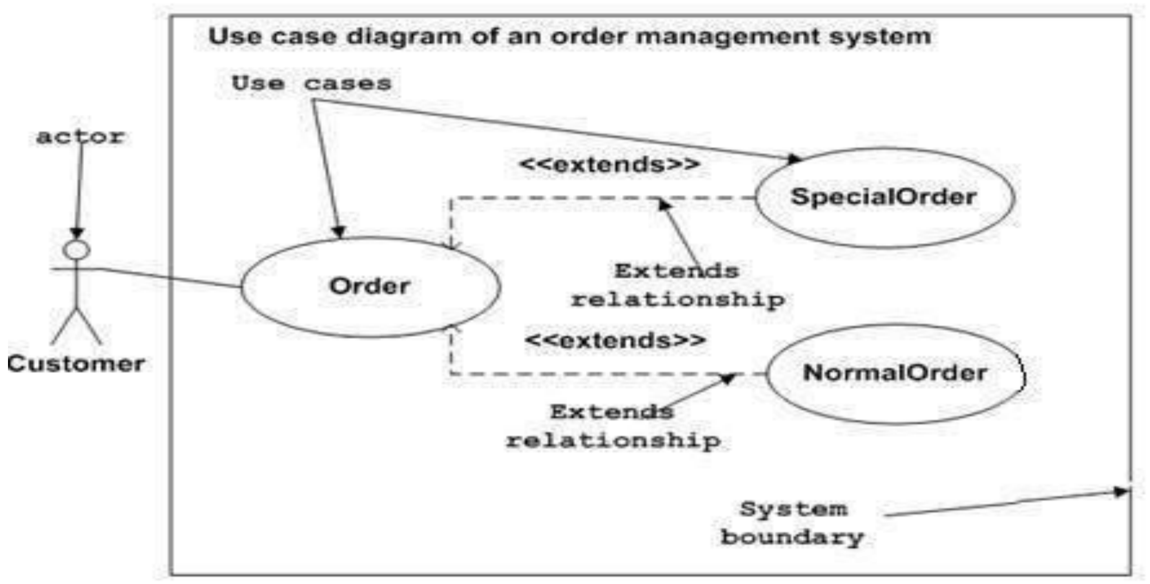


Figure: Sample Use Case diagram

Where to Use a Use Case Diagram

As we have already discussed there are five diagrams in UML to model the dynamic view of a system. Now each and every model has some specific purpose to use. Actually these specific purposes are different angles of a running system.

To understand the dynamics of a system, we need to use different types of diagrams. Use case diagram is one of them and its specific purpose is to gather system requirements and actors.

Use case diagrams specify the events of a system and their flows. But use case diagram never describes how they are implemented. Use case diagram can be imagined as a black box where only the input, output, and the function of the black box is known.

These diagrams are used at a very high level of design. This high level design is refined again and again to get a complete and practical picture of the system. A well-structured use case also describes the pre-condition, post condition, and exceptions. These extra elements are used to make test cases when performing the testing.

Although use case is not a good candidate for forward and reverse engineering, still they are used in a slightly different way to make forward and reverse engineering. The same is true for reverse engineering. Use case diagram is used differently to make it suitable for reverse engineering.

In forward engineering, use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.

Use case diagrams can be used for −

- Requirement analysis and high level design.
- Model the context of a system.
- Reverse engineering.
- Forward engineering.

COMPONENT DIAGRAM

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

How to Draw a Component Diagram

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

Before drawing a component diagram, the following artifacts are to be identified clearly −

- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

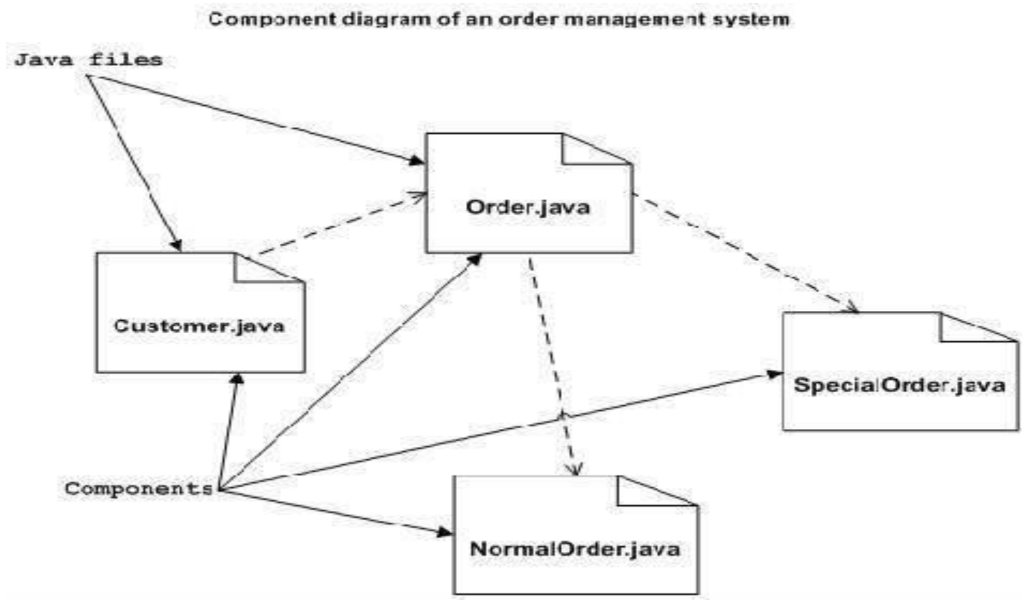
After identifying the artifacts, the following points need to be kept in mind.

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing the using tools.
- Use notes for clarifying important points.

Following is a component diagram for order management system. Here, the artifacts are files. The diagram shows the files in the application and their relationships. In actual, the component diagram also contains dlls, libraries, folders, etc.

In the following diagram, four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far as it is drawn for completely different purpose.

The following component diagram has been drawn considering all the points mentioned above.



Where to Use Component Diagrams

We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.

Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details

Component diagrams can be used to −

- Model the components of a system.
- Model the database schema.
- Model the executables of an application.
- Model the system's source code.