UNIT III

DESIGN ENGINEERING

DESIGN PROCESS

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.
- The blueprint depicts a holistic view of software. That is, the design represented at a high level of abstraction- a level that can directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.

❑ McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

1. The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
2. The design must be readable, understandable guide for those who generate code and for those who test and sequentially support the software.
3. The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

DESIGN QUALITY

QUALITY GUIDELINES

• Uses recognizable architectural styles or patterns

• Modular; that is logically partitioned into elements or subsystems

• Distinct representation of data, architecture, interfaces and components

• Appropriate data structures for the classes to be implemented

• Independent functional characteristics for components

• Interfaces that reduces complexity of connection

• Repeatable method

QUALITY ATTRIBUTES

❖ FURPS quality attributes

1. Functionality
   - Feature set and capabilities of programs
   - Security of the overall system
2. Usability
   - user-friendliness
   - Aesthetics
   - Consistency
   - Documentation
3. Reliability
   - Evaluated by measuring the frequency and severity of failure
   - Mean-time-to-failure(MTTF)
   - Recover from failure
4. Performance
   - Speed, response time, resource consumption, throughput, efficiency.
5. Supportability
   - Extensibility
   - Adaptability
   - Serviceability
   - maintainability

DESIGN CONCEPTS

- A set of fundamental software design concepts has evolved over the history of software engineering.
- Although the degree of interest in each concept has varied over the years, each has stood the test of time.
- Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

Design concepts are:

1. Abstractions

NRCM your roots to success...

2. Architecture
3. Patterns
4. Modularity
5. Information Hiding
6. Functional Independence
7. Refinement
8. Re-factoring
9. Design Classes

1. **ABSTRACTION**

Many levels of abstraction.

- **Highest level of abstraction:** Solution is slated in broad terms using the language of the problem environment
- **Lower levels of abstraction:** More detailed description of the solution is provided
- **Procedural abstraction:** Refers to a sequence of instructions that a specific and limited function
- **Data abstraction:** Named collection of data that describe a data object

2. ARCHITECTURE

Structure organization of program components (modules) and their interconnection Architecture Models

- **Structural Models**-- An organized collection of program components
- **Framework Models**-- Represents the design in more abstract way
- **Dynamic Models**-- Represents the behavioral aspects indicating changes as a function of external events
- **Process Models--** Focus on the design of the business or technical process

3. PATTERNS

Provides a description to enables a designer to determine the followings:

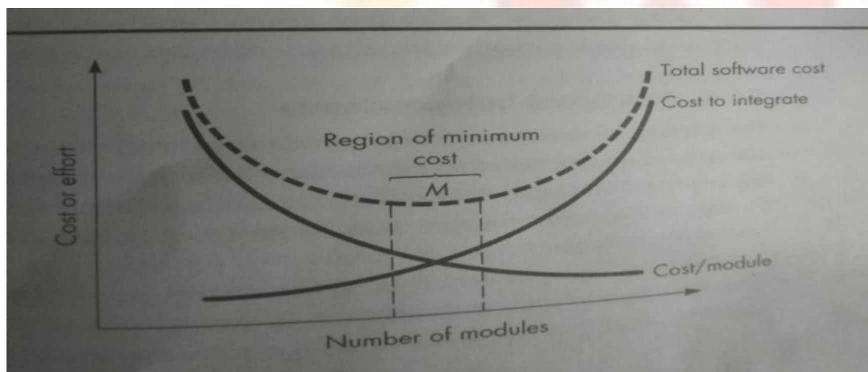
a) whether the pattern is applicable to the current work

b) Whether the pattern can be reused

c) Whether the pattern can serve as a guide for developing a similar but functionally or structurally different pattern

4. **MODULARITY**

- Divides software into separately named and addressable components, sometimes called modules. Modules are integrated to satisfy problem requirements. Consider two problems p1 and p2. If the complexity of p1 is cp1 and of p2 is cp2 then effort to solve p1=cp1 and effort to solve p2=cp2 If cp1>cp2 then ep1>ep2
- The complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately.
- Based on Divide and Conquer strategy: it is easier to solve a complex problem when broken into sub-modules



5. INFORMATION HIDING

Information contained within a module is inaccessible to other modules who do not need such information. Achieved by defining a set of Independent modules that communicate with one another only that information necessary to achieve S/W function. Provides the greatest benefits when modifications are required during testing and later. Errors introduced during modification are less likely to propagate to other location within the S/W.

6. FUNCTIONAL INDEPENDENCE

A direct outgrowth of Modularity. abstraction and information hiding. Achieved by developing a module with single minded function and an aversion to excessive interaction with other modules. Easier to develop and have simple interface. Easier to maintain because secondary effects caused b design or code modification are limited, error propagation is reduced and reusable modules are possible. Independence is assessed by two quantitative criteria:

- ✓ Cohesion
- ✓ Coupling

Cohesion -- Performs a single task requiring little interaction with other components

Coupling--Measure of interconnection among modules. Coupling should be low and cohesion should be high for good design.

7. REFINEMENT

Process of elaboration from high level abstraction to the lowest level abstraction. High level abstraction begins with a statement of functions. Refinement causes the designer to elaborate providing more and more details at successive level of abstractions Abstraction and refinement are complementary concepts.

8. REFACTORING

Organization technique that simplifies the design of a component without changing its function or behavior. Examines for redundancy, unused design elements and inefficient or unnecessary algorithms.

9. DESIGN CLASSES

Class represents a different layer of design architecture. Five types of Design Classes

- **User interface class** -- Defines all abstractions that are necessary for human computer interaction
- **Business domain class** -- Refinement of the analysis classes that identity attributes and services to implement some of business domain
- **Process class** -- implements lower level business abstractions required to fully manage the business domain classes
- **Persistent class** -- Represent data stores that will persist beyond the execution of the software

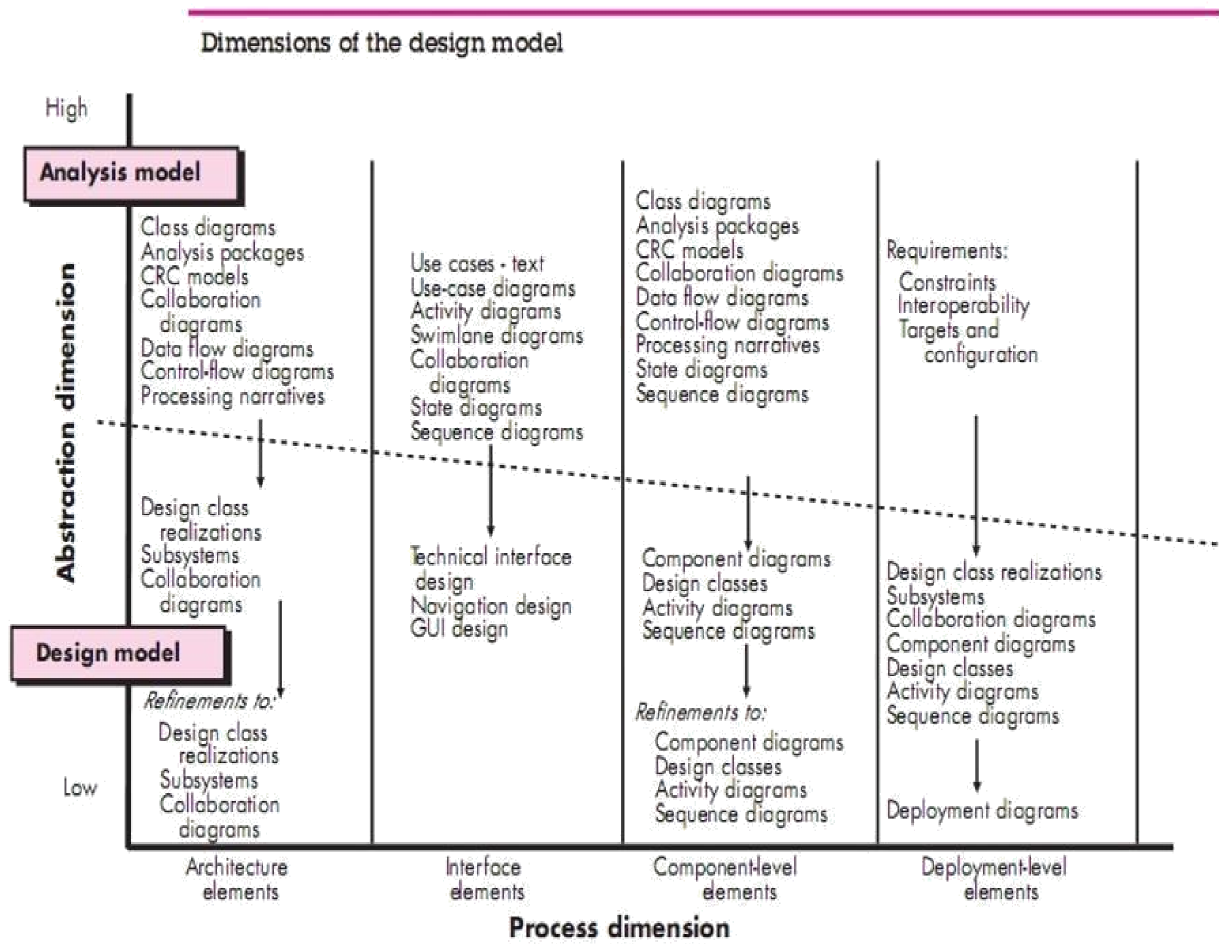
System class -- Implements management and control functions to operate and communicate within the computer environment and with the outside world.

THE DESIGN MODEL

Introduction of Design Model

- The design model can be viewed in two different dimensions.
  - (Horizontally) The process dimension
    - It indicates the evolution of the parts of the design model as each design task is executed.
  - (Vertically) The abstraction dimension

- It represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined.
- The elements of the design model use many of the same UML diagrams that were used in the analysis model.
- The difference is that these diagrams are
  - Refined and elaborated as part of design;
  - More implementation-specific detail is provided,
  - Architectural structure and style, components that reside within the architecture,
  - Interfaces between the components and with the outside world are all emphasized.



1. **Data Design Elements**

- **Customer's/ User's View:**
  - Data Architecting (Creates a model of data that is represented at a high level of abstraction). (Build Architecture of Data)
- **Program Component Level:** The design of Data structure & algorithms.

- **Application Level:** Translate Data Model into a database.
- **Business Level:** Data warehouse(Reporting & Analysis of DB) & Data mining(Analysis).
- At last it means creation of Data Dictionary.

2. **Architectural Design Elements:**

Provides an overall view of the software product(Similar like Floor Plan of house)

- The architectural model [Sha96] is derived from three sources:

  (1) Information about the application domain for the software to be built;

  (2) Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand;

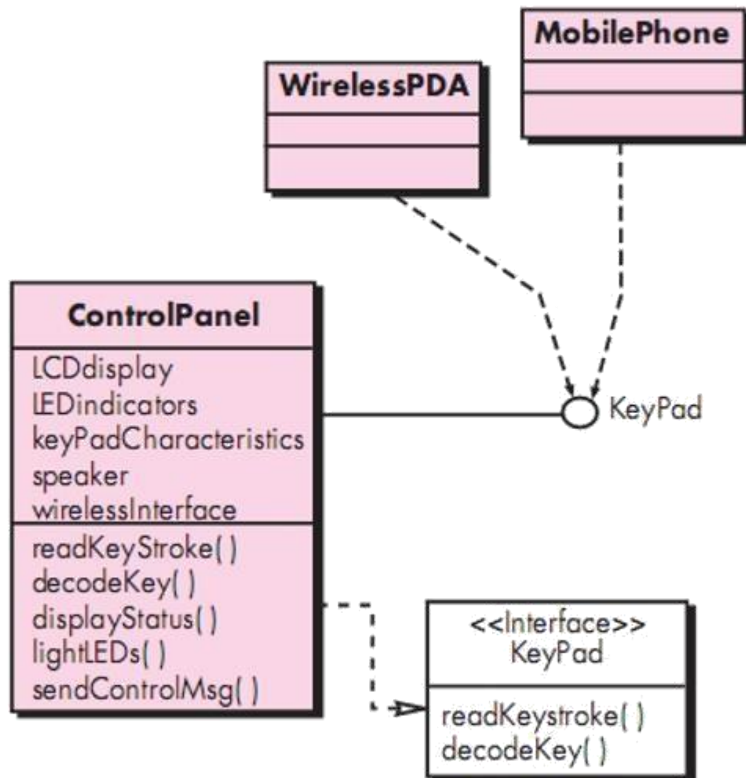
  (3) The availability of architectural styles and patterns

- Difference: An architectural style is a conceptual way of how the system will be created / will work.
- An architectural pattern describes a solution for implementing a style at the level of subsystems or modules and their relationships.

3. **Interface Design Elements**

- The interface design elements for software represent information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
- For example : A set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. These drawings describe the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan.
- There are three important elements of interface design:
  - (1) The user interface (UI);
  - (2) External interfaces to other systems, devices, networks, or other producers or consumers of information;
  - (3) Internal interfaces between various design components.
- **UI design** (increasingly called usability design) is a major software engineering action
- Usability design incorporates
  - Visual elements (e.g., layout, color, graphics, interaction mechanisms),
  - Ergonomic elements (e.g., information layout and placement, metaphors, UI navigation),

- Technical elements (e.g., UI patterns, reusable components).
- In general, the UI is a unique subsystem within the overall application architecture.



4. Component-Level Design Elements

- The component-level design for software fully describes the internal detail of each software component.
- Component elements (detailed drawing of each room, wiring, place of switches…)
  - Internal details of each software component
    - Data structures,
    - algorithmic details,
    - interface to access component operation (behavior).

5. **Deployment-Level Design Elements**

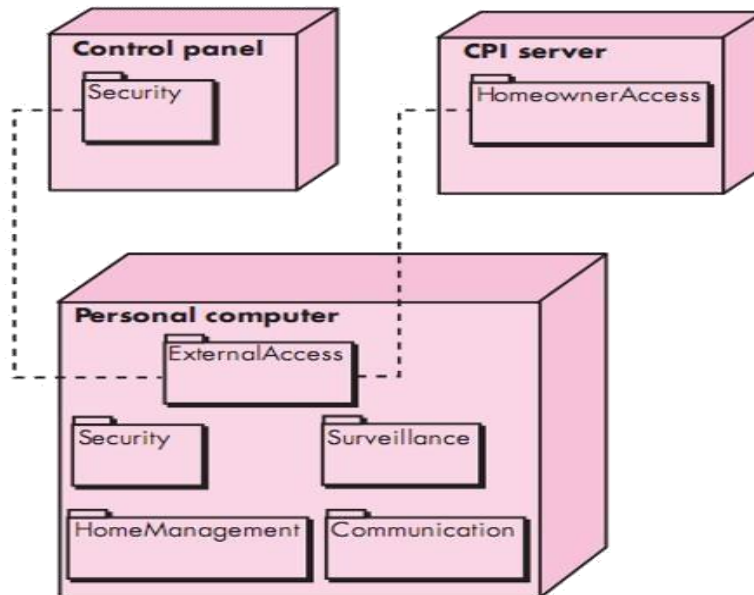
- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

- For example, the elements of the SafeHome product are configured to operate within three primary computing environments

  – A home-based PC,

  – The SafeHome control panel,

  – Server housed at CPI Corp. (providing Internet-based access to the system).



Software architecture:

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.

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.

Data design

Here data design is described at both the architectural and component levels. At the architecture level, data design is the process of creating a model of the information represented at a high level of abstraction (using the customer's view of data)

1.Data Design at the Architectural Level

- The challenge is extract useful information from the data environment, particularly when the information desired is cross-functional.
- To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in databases (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information
- However, the existence of multiple databases, their different structures, and the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment
- An alternative solution, called a data warehouse, adds on additional layer to the data architecture
- A data warehouse is a separate data environment that is not directly integrated with day-to-day applications that encompasses all data used by a business

2.Data Design at the Component Level

At the component level, data design focuses on specific data structures required to realize the data objects to be manipulated by a component.

- Refine data objects and develop a set of data abstractions
- Implement data object attributes as one or more data structures
- Review data structures to ensure that appropriate relationships have been established Set of principles for data specification:

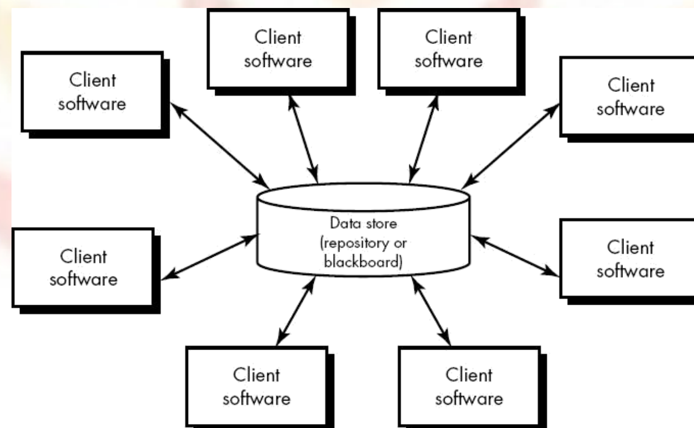
1. The systematic analysis principles applied to function and behavior should also be applied to data
2. All data structures and the operations to be performed on each should be identified
3. A data dictionary should be established and used to define both data and program design
4. Low level data design decisions should be deferred until late in the design process
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure
6. A library of useful data structures and the operations that may be applied to them should be developed
7. A software design and programming language should support the specification and realization of abstract data types

Architectural styles and patterns

Architectural styles:

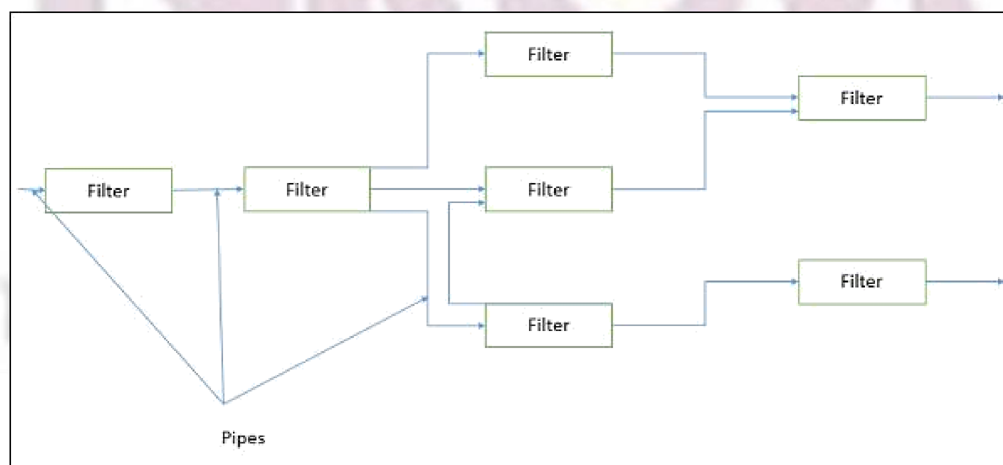
1. Data-centered architecture

- The data store in the file or database is occupying at the center of the architecture.
- Store data is access continuously by the other components like an update, delete, add, modify from the data store.
- Data-centered architecture helps integrity.
- Pass data between clients using the blackboard mechanism.
- The processes are independently executed by the client components.



2. Data-flow architecture

- This architecture is applied when the input data is converted into a series of manipulative components into output data.
- A pipe and filter pattern is a set of components called as filters.
- Filters are connected through pipes and transfer data from one component to the next component.
- The flow of data degenerates into a single line of transform then it is known as batch sequential.



3. Call and return architectures

This architecture style allows to achieve a program structure which is easy to modify.
Following are the sub styles exist in this category:

a) Main program or subprogram architecture

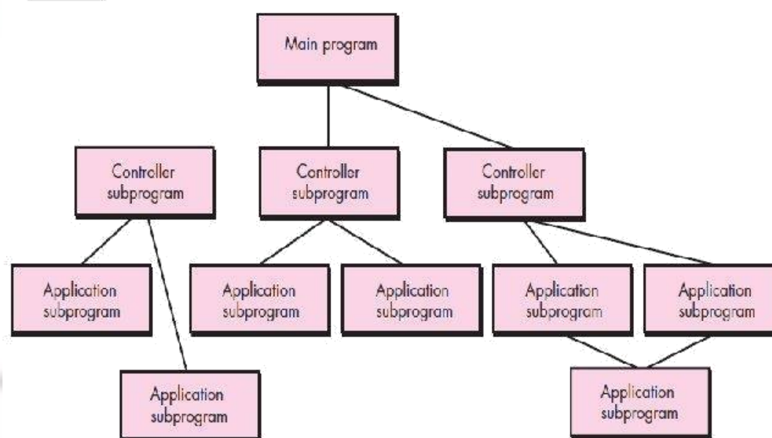
The program is divided into smaller pieces hierarchically.

The main program invokes many of program components in the hierarchy that program components are divided into subprogram.

b) Remote procedure call architecture

The main program or subprogram components are distributed in network of multiple computers.

The main aim is to increase the performance.



4. Object-oriented architectures

- This architecture is the latest version of call-and-return architecture.
- It consist of the bundling of data and methods.

5. Layered architectures

- The different layers are defined in the architecture. It consists of outer and inner layer.
- The components of outer layer manage the user interface operations.
- Components execute the operating system interfacing at the inner layer.
- The inner layers are application layer, utility layer and the core layer.
- In many cases, It is possible that more than one pattern is suitable and the alternate architectural style can be designed and evaluated.

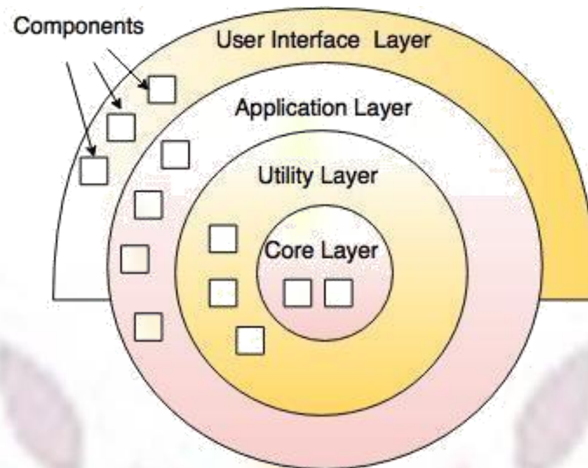


Fig.- Layered Architecture

Architectural patterns:

Different Software Architecture Patterns :

1. Layered Pattern
2. Client-Server Pattern
3. Event-Driven Pattern
4. Microkernel Pattern
5. Microservices Pattern

Let's see one by one in detail.

1. LayeredPattern :

As the name suggests, components(code) in this pattern are separated into layers of subtasks and they are arranged one above another.
Each layer has unique tasks to do and all the layers are independent of one another. Since each layer is independent, one can modify the code inside a layer without affecting others.

It is the most commonly used pattern for designing the majority of software. This layer is also known as 'N-tier architecture'. Basically, this pattern has 4 layers.

1. Presentation layer (The user interface layer where we see and enter data into an application.)
2. Business layer (this layer is responsible for executing business logic as per the request.)
3. Application layer (this layer acts as a medium for communication between the 'presentation layer' and 'data layer'.
4. Data layer (this layer has a database for managing data.)

Ideal for:

E-commerce web applications development like Amazon.

2. Client-ServerPattern :

The client-server pattern has two major entities. They are a server and multiple clients.
Here the server has resources(data, files or services) and a client requests the server for a particular resource. Then the server processes the request and responds back accordingly.

Examples of software developed in this pattern:

- Email.
- WWW.
- File sharing apps.
- Banking, etc…

So this pattern is suitable for developing the kind of software listed in the examples.

3. Event-DrivenPattern **:**

Event-Driven Architecture is an agile approach in which services (operations) of the software are triggered by events.

Well, what does an event mean?

When a user takes action in the application built using the EDA approach, a state change happens and a reaction is generated that is called an event.

Eg: A new user fills the signup form and clicks the signup button on Facebook and then a FB account is created for him, which is an event.

Ideal for:

Building websites with JavaScript and e-commerce websites in general.

4. MicrokernelPattern **:**

Microkernel pattern has two major components. They are a core system and plug-in modules.

- The core system handles the fundamental and minimal operations of the application.
- The plug-in modules handle the extended functionalities (like extra features) and customized processing.

5. MicroservicesPattern **:**

The collection of small services that are combined to form the actual application is the concept of microservices pattern. Instead of building a bigger application, small programs are built for every service (function) of an application independently. And those small programs are bundled together to be a full-fledged application.

So adding new features and modifying existing microservices without affecting other microservices are no longer a challenge when an application is built in a microservices pattern.

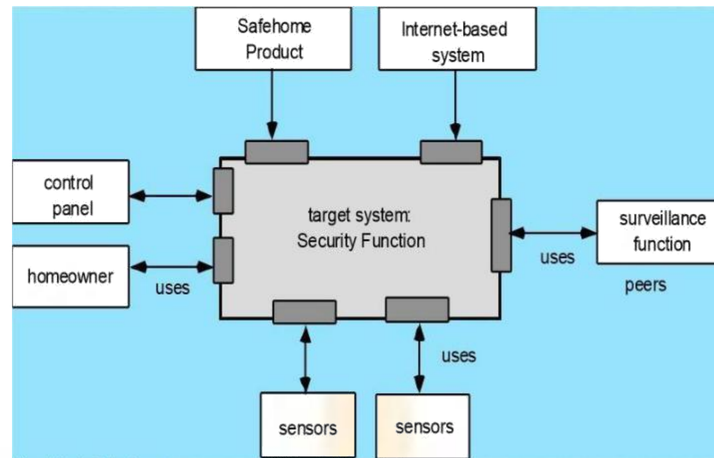
Modules in the application of microservices patterns are loosely coupled. So they are easily understandable, modifiable and scalable.

Architectural design

- **As architectural design begins**, the software to be developed must be put into context
- That is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.

Representing the System in Context

- At the architectural design level, a software architect uses an **architectural context diagram (ACD)** to model the manner in which software interacts with entities external to its boundaries.
- The generic structure of the architectural context diagram is illustrated in Figure.



In figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

- **Super ordinate systems : those** systems that use the target system as part of some higher-level processing scheme.
- **Subordinate systems—those** systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- **Peer-level systems—those** systems that interact on a peer-to- peer basis (i.e., information is either produced or consumed by the peers and the target system.
- **Actors—entities** *(people, devices)* that interact with the target system by producing or consuming information.
- Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

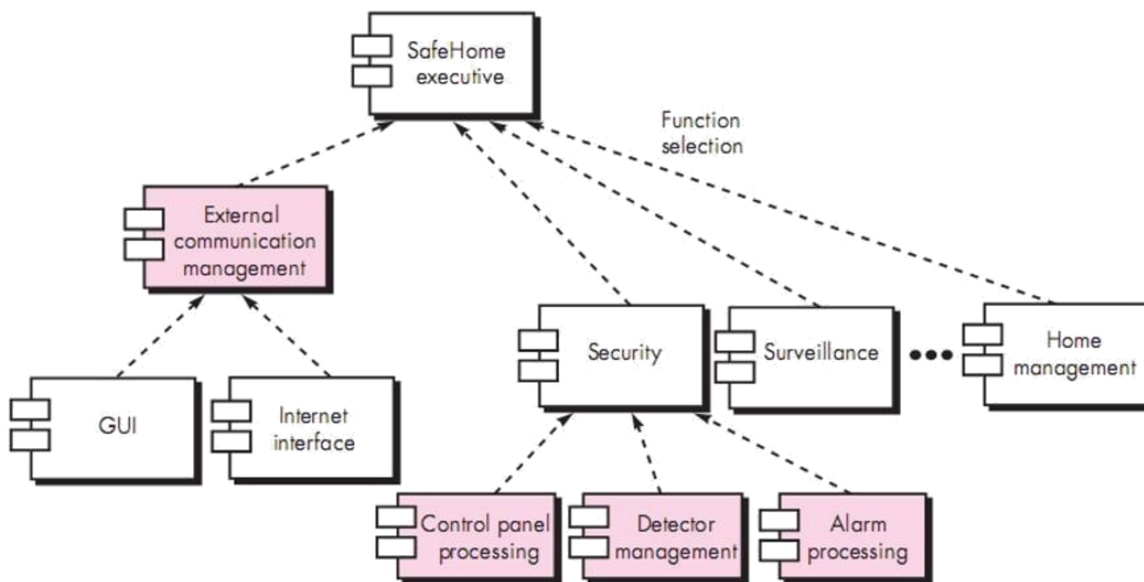
Defining Archetypes

- *An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.*
- In general, a relatively small set of archetypes is required to design even relatively complex systems.
- *Archetypes are the abstract building blocks of an architectural design.*
- In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model.
- An archetype is a generic, idealized model of a person, object, or concept from which similar instances are derived, copied, patterned, or emulated.
- The SafeHome home security function, you might define the following archetypes :
  - **Node** : Represents a cohesive collection of input and output elements of the home security function.
  - For example a node might be included of (1) various sensors and (2) a variety of alarm (output) indicators.
  - **Detector** : An abstraction that covers all sensing equipment that feeds information into the target system.
  - **Indicator**. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

- **Controller**. An abstraction that describes the mechanism that allows the arming (Supporting) or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

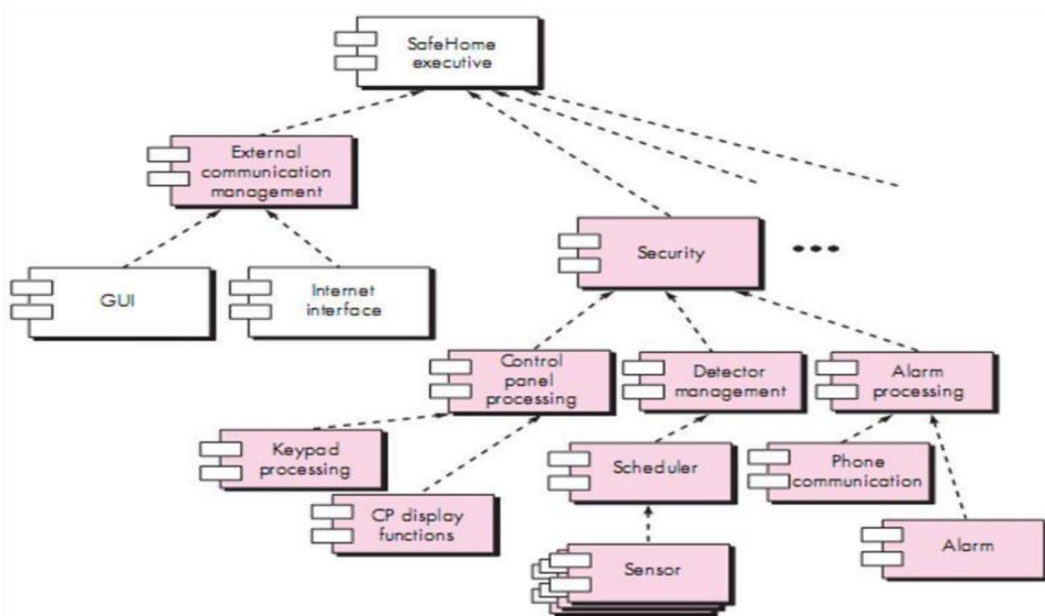
Refining the Architecture into Components

- As the software architecture is refined into components.
- Analysis classes represent entities within the application (business) domain that must be addressed within the software architecture.
- In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.
- For Example : The SafeHome home security function example, you might define the set of top-level components that address the following functionality:
- **External communication management — coordinates** communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- **Control panel processing—** manages all control panel functionality.
- **Detector management** — coordinates access to all detectors attached to the system.
- **Alarm processing** — verifies and acts on all alarm conditions
- The overall architectural structure (represented as a **UML component diagram**) is in the following Figure.



Describing Instantiations of the System

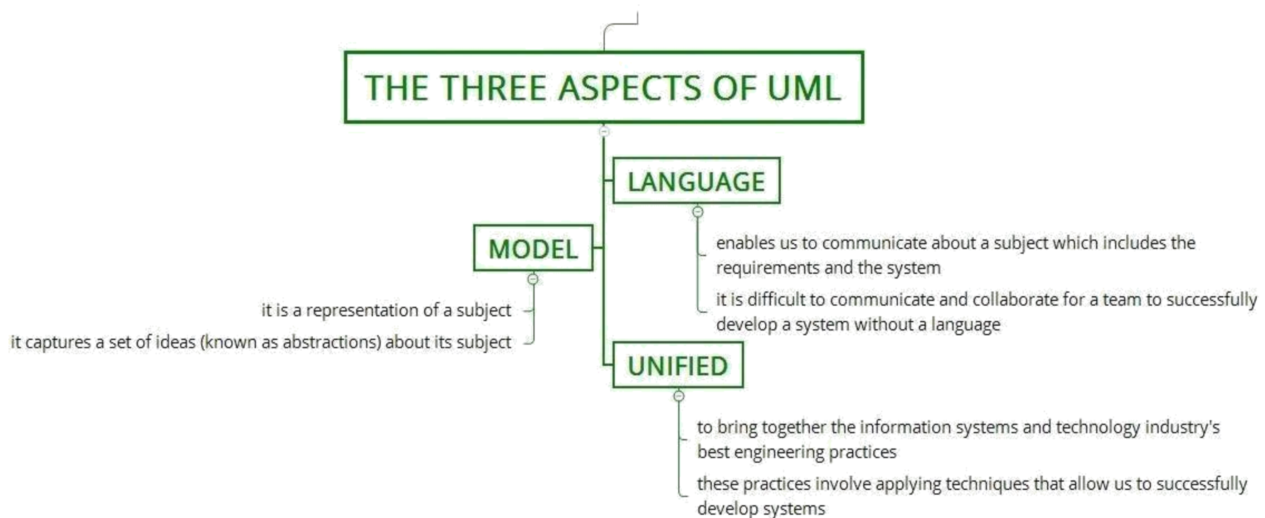
- The architectural design that has been modeled to this point is still relatively high level.
- The context of the system has been represented
- Archetypes that indicate the important abstractions within the problem domain have been defined,
- The overall structure of the system is apparent, and the major software components have been identified.
- However, further refinement is still necessary.
- To accomplish this, an actual instantiation of the architecture is developed.It means, again it simplify by more details.
- The figure demonstrates this concept.



Conceptual model of 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. The artifacts of a software-intensive system (particularly systems built using the object-oriented style).

Three Aspects of UML:



- **Figure –** Three Aspects of UML
- **Note –** Language, Model, and Unified are the important aspect of UML as described in the map above.

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.

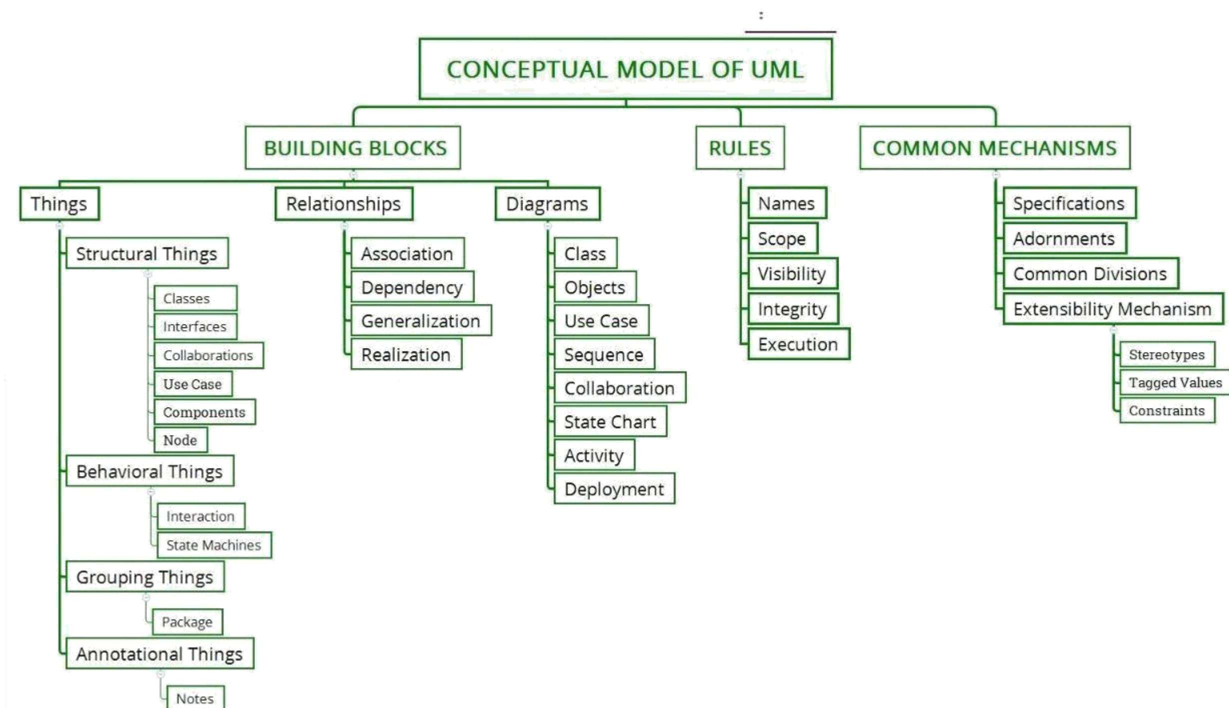
3. Unified:

- 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



BASIC STRUCTURAL MODELING

Contents:

1. Classes
2. Relationships
3. Common Mechanisms
4. Diagrams

1.Classes:

- Names
- Attributes
- Operations

2.Relationships:

- Dependencies
- Generalizations
- Associations
- Aggregation

3.Common Mechanisms:

- Notes
- Other Adornments
- Stereotypes
- Tagged Values
- Constraints

4. Diagrams:

Structural Diagrams

The UML's structural diagrams exist to visualize, specify, construct, and document the static aspects of a system. You can think of the static aspects of a system as representing its relatively stable skeleton and scaffolding. Just as the static aspects of a house encompass the existence and placement of such things as walls, doors, windows, pipes, wires, and vents, so too do the static aspects of a software system encompass the existence and placement of such things as classes, interfaces, collaborations, components, and nodes.

The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

1.Class diagram	Classes, interfaces, and collaborations
2.Component diagram	Components
3.Object diagram	Objects
4.Deployment diagram	Nodes

Behavioral Diagrams

The UML's behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system. You can think of the dynamic aspects of a system as representing its changing parts. Just as the dynamic aspects of a house encompass airflow and traffic through the rooms of a house, so too do the dynamic aspects of a software system encompass such things as the flow of messages over time and the physical movement of components across a network.

The UML's behavioral diagrams are roughly organized around the major ways you can modelthe dynamics of a system.

1.Use case diagram	Organizes the behaviors of the system
2.Sequence diagram	Focuses on the time ordering of messages
3.Collaboration diagram	Focuses on the structural organization of objects that send and receive messages
4.State diagram	Focuses on the changing state of a system driven by events
5.Activity diagram	Focuses on the flow of control from activity to activity

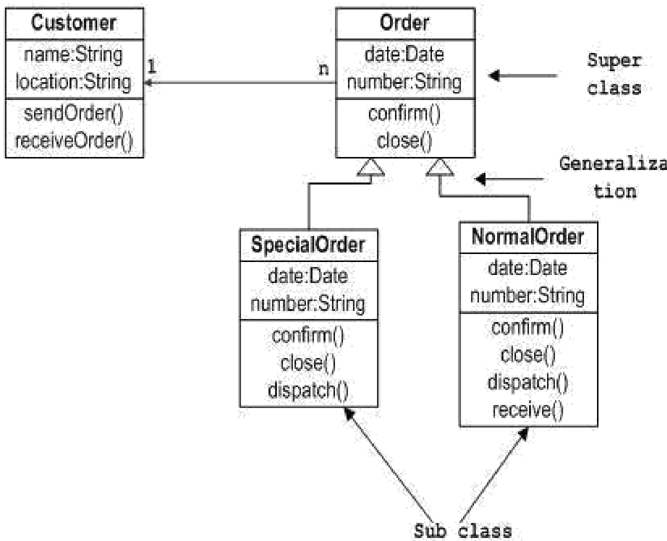
Class diagram:

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.



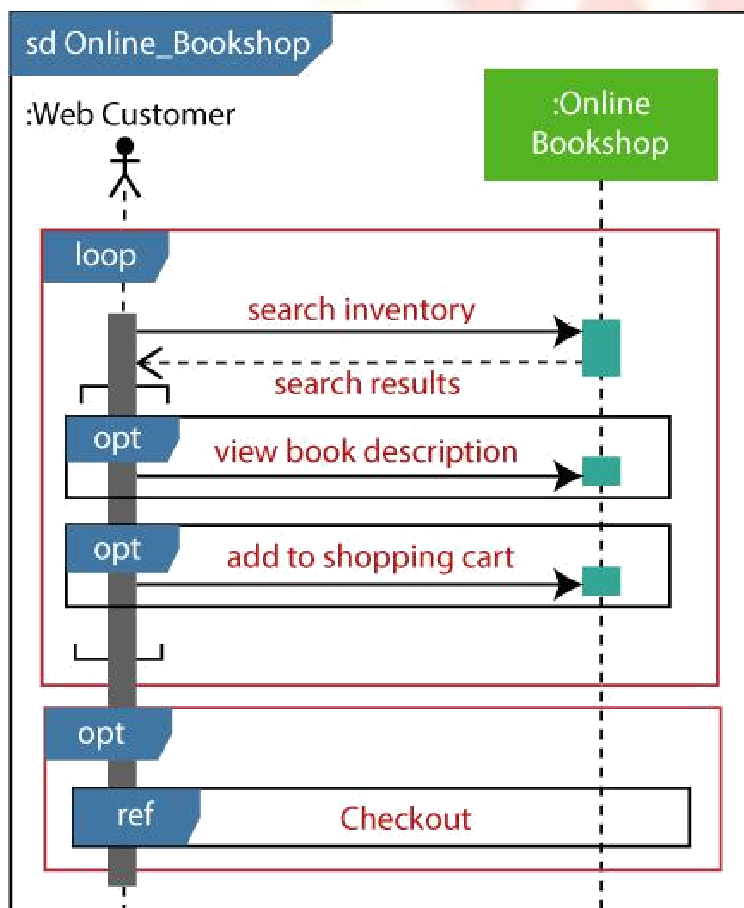
Sequence Diagram:

1. To model high-level interaction among active objects within a system.
2. To model interaction among objects inside a collaboration realizing a use case.
3. It either models generic interactions or some certain instances of interaction.

Example of a Sequence Diagram

An example of a high-level sequence diagram for online bookshop is given below.

Any online customer can search for a book catalog, view a description of a particular book, add a book to its shopping cart, and do checkout.

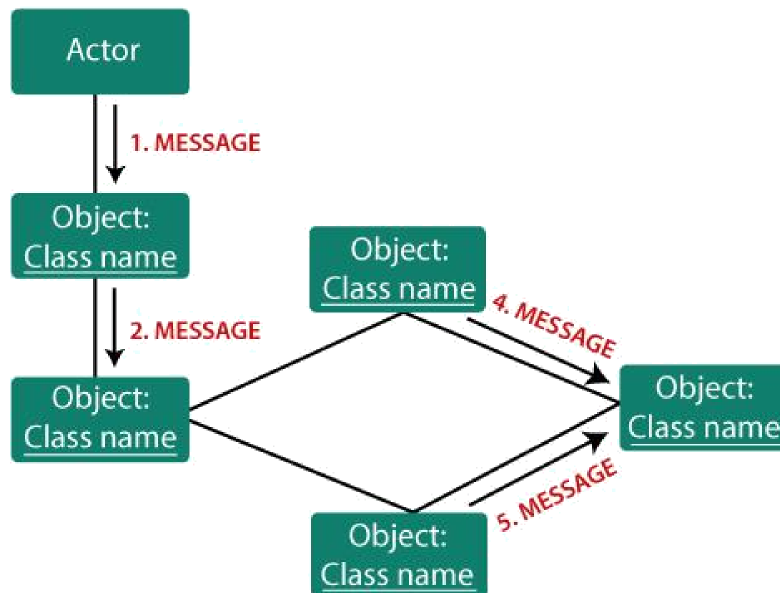


Collaboration diagram:

Notations of a Collaboration Diagram

- Objects: The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.

- Actors: In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
- Links: The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
- Message: It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.



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.

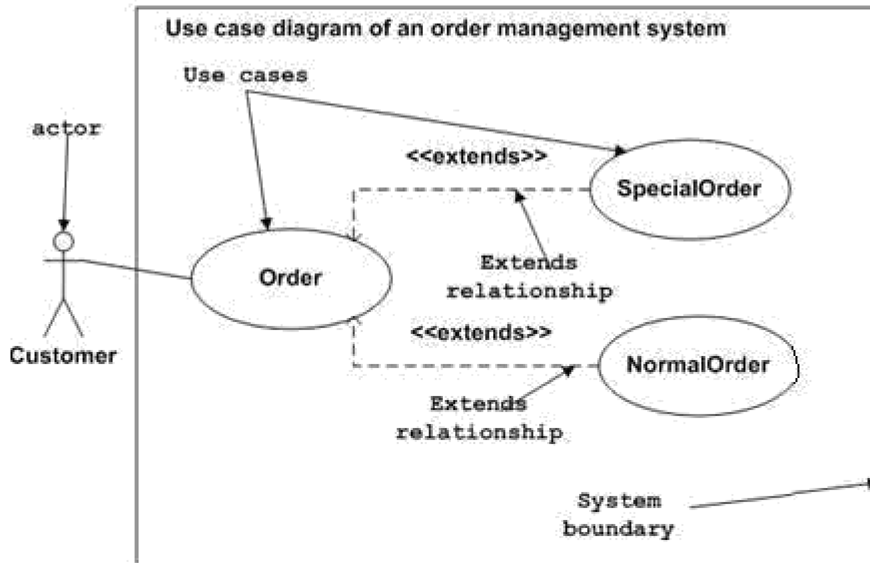


Figure: Sample Use Case diagram

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.

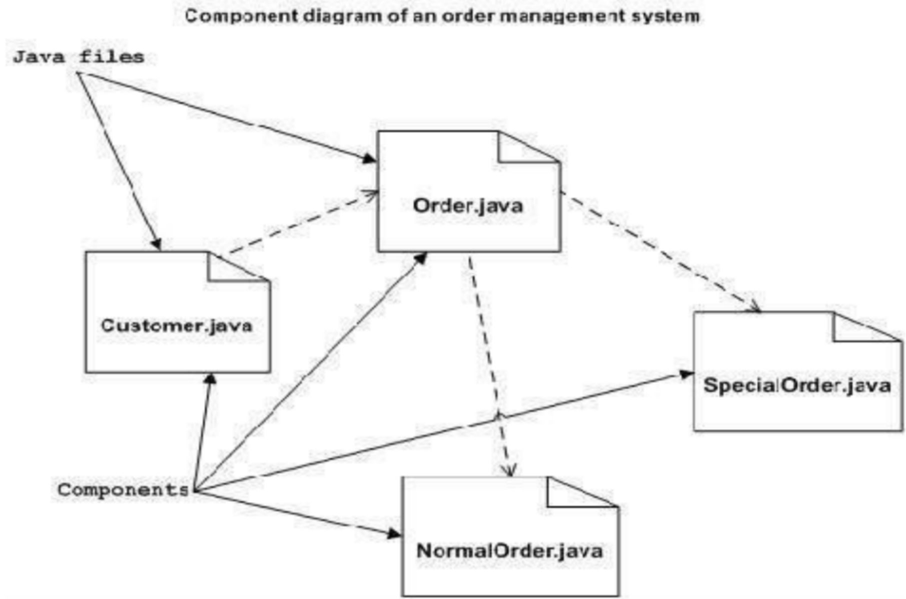
Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

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.


Component diagram of an order management system
Java files
Order.java
Customer.java
SpecialOrder.java
Components
NormalOrder.java