Unit-5 Architectural Design

Architectural design is the initial design process of identifying sub system is establishing, a framework for sub system control and communication is called architectural design. It is the first stage in the design process and presents a critical link between the design and requirement engineering. Architectural design concerned with establishing a basic structural framework that identifies the major components of a system and the communication between these components. It is the link between design and requirement engineering and it identifies the main structural components in a system and the relationships between them. The output of architectural design process is an architectural model that describes how the system is organized as a set of communicating components. Architectural design has following three advantages:

- **Stakeholders Communication**: The architectural design of software system is a high level presentation of the system that may be used as medium of discussion by a range of different stakeholders to understand the system in more detail and to identify the coherent picture of application.
- **System Analysis:** Architectural design of system can be used as medium of requirement analysis. It has a profound effect on whether or not the system can meet critical requirement such as performance, reliability and maintainability.
- **Large scale reuse**: The system architecture is often same for systems with similar requirements and so can support large scale software reuse. It can be possible to development of architecture where the same architecture is reused across a range of related systems.

Architectural Design Decisions

In architectural design, the architects have to make number of decisions that mainly affect the system and its development process. The architects have to focus on different issues such as:

- Is there a generic application architecture that can be used?
- What approach will be used to structure the system?
- What architectural styles are appropriate?
- How will the system be distributed?

- What control strategy should be used?
- How should the architecture be documented?
- How will the system be decomposed into modules?
- How will the architectural design be evaluated?

The architectural design of the system affects the performance, dependability, maintainability etc of the system. The particular architectural design and structure chosen for an application depends on the non-functional system requirements.

- **Maintainability**: While designing architectural design, if maintainability is the critical non functional requirement the system architecture should be designed using self contained components that may be readily be changed and shared data structures should be avoided.
- **Performance**: In architectural design if performance is the critical requirement it should be designed to localize critical operations with in small number of sub systems with little communication as possible between sub systems.
- **Availability**: If availability is the critical requirements, this suggests that the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system.
- Security: In architectural design, if security is the critical requirement then layered architecture should be used with the most critical assets protected in innermost layer and high level of validation is performed in that layer.
- **Safety**: If safety is the critical requirement then architecture should be designed so that safety related operations are all located either in single subsystem or small number of sun system.

Architectural Views

Software architecture involves the high level structure of software system abstraction, by using decomposition and composition, with architectural style and quality attributes. A software architecture design must conform to the major functionality and performance requirements of the system, as well as satisfy the non-functional requirements such as reliability, scalability, portability, and availability. A view is a representation of an entire system from the perspective

of a related set of concerns. It is used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers. It provides four essential views:

- **Physical View**: It describes the mapping of software onto hardware and reflects its distributed aspect. It focuses on how system components distributed across the processors in the system and useful for planning deployment of the system.
- **Logical View**: It describes the object model of the design. This view provides the abstraction in the system in terms of object or object classes.
- **Process View**: It describes the activities of the system, captures the concurrency and synchronization aspects of the design. It shows how the system is composed of interacting processes at run time. It is used to make judgment about non functional requirements of the system.
- **Development View**: It describes the static organization or structure of the software in its development of environment. It focuses on how the software system is decomposed into components that can be developed by individual developer.

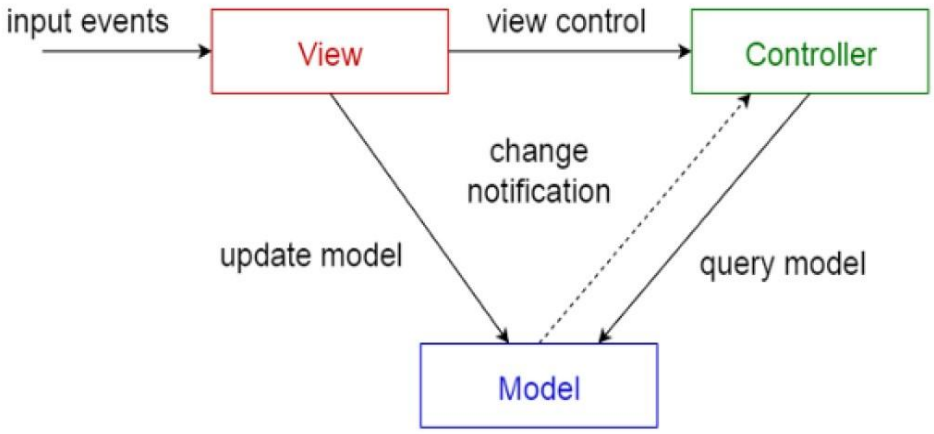
Architectural Patterns

Patterns are a means of representing, sharing and reusing knowledge. An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments. Patterns should include information about when they are and when they are not useful. Patterns may be represented using tabular and graphical descriptions.

There are many different types of software **architecture patterns**, Some of them are explores below and how they are integral to software development:

MVC Design Pattern

The **Model View Controller (MVC**) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects. MVC is more of an architectural pattern, but not for complete application. MVC mostly relates to the UI / interaction layer of an application.

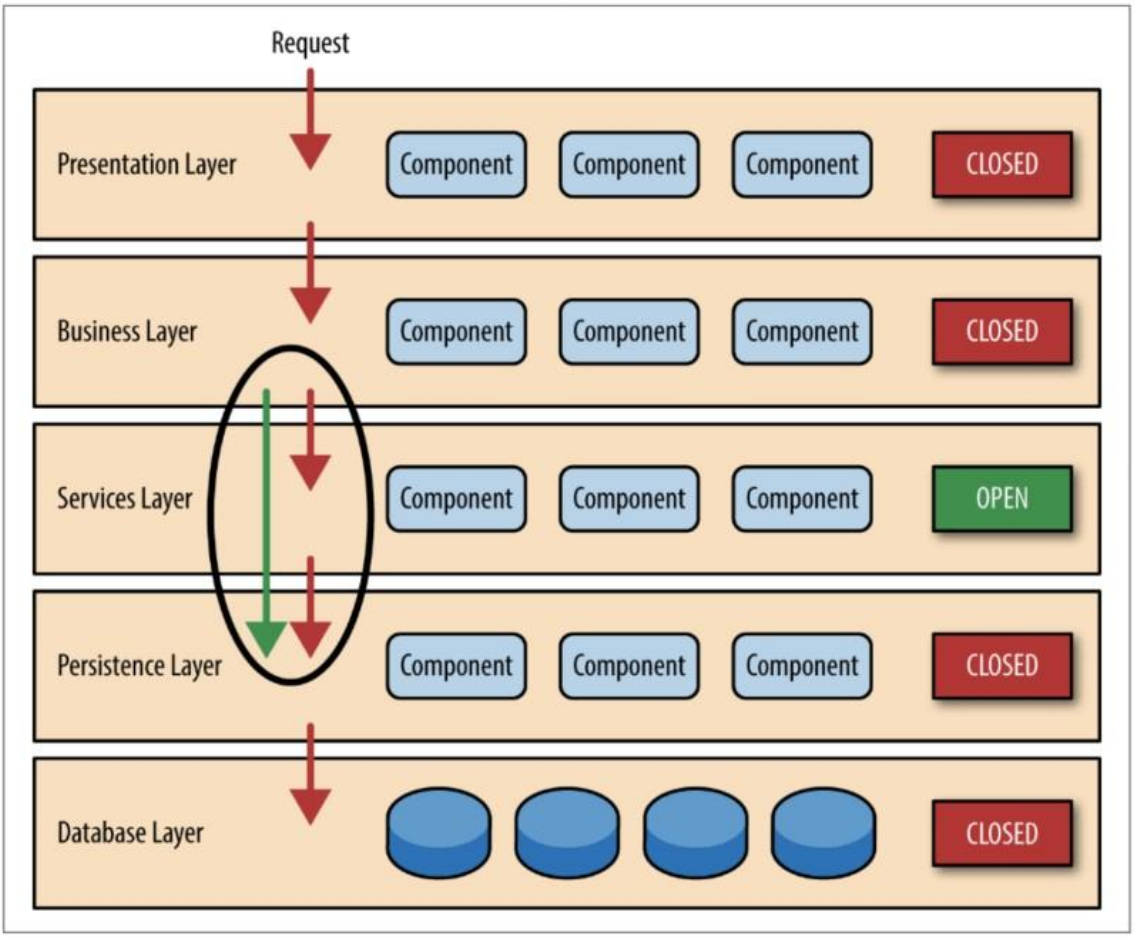


- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user. (Its just a data that is shipped across the application like for example from back-end server view and from front-end view to the database. In java programming, Model can be represented by the use of POJO (Plain-old-java-object) which is a simple java class.
- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it. View just represent, displays the application's data on screen. View page are generally in the format of .html or .jsp in java programming (which is flexible).
- The **Controller** exists between the view and the model. It is where the actual business logic is written. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.

Layered Architecture Patterns

Layered architecture, also known as the n-tier architecture pattern, is a software design approach that separates an application into distinct layers, each with a specific purpose and set of responsibilities. This pattern is widely used in enterprise-level software development, and it has

proven to be a powerful tool for creating large, complex systems that are easy to maintain and evolve over time.



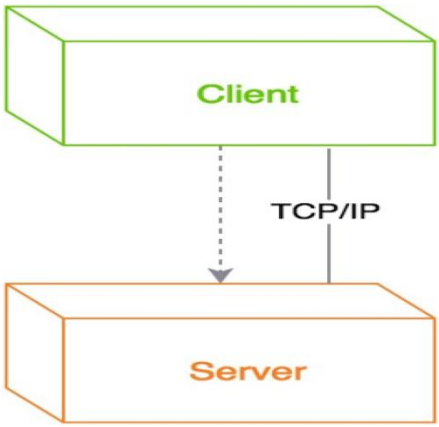
The main advantage of using a layered architecture is that it promotes separation of concerns and modularity in the code. By separating the application into distinct layers, developers can focus on specific areas of the system without worrying about the implementation details of other layers. This makes the code more reusable and easier to understand, and it also facilitates testing and maintenance.

Another advantage of layered architecture is that it allows for flexibility and scalability in the system. Because the different layers are decoupled from each other, it is easy to make changes or add new functionality to one layer without affecting the others. Additionally, different layers can be scaled independently, allowing the system to handle a high volume of traffic or data without performance issues.

Client-server pattern

In the client-server architecture patterns, there are two main components: The client, who is the service *requester*, and the server, which is the service *provider*. Although both client and server may be located within the same system, they often communicate over a network on separate hardware.

The client component initiates certain interactions with the server to generate the services needed. While the client components have ports that describe the needed services, the servers have ports that describe the services they provide. Both components are linked by request/reply connectors. A classic example of this architecture pattern is the World Wide Web. The client-server pattern is also used for online applications such as file sharing and email.



A simple example is online banking services. When a bank customer accesses online banking services using a web browser, the client initiates a request to the bank's web server. In this case, the web browser is the client, accessing the bank's web server for data using the customer's login details. The application server interprets this data using the bank's business logic and then provides the appropriate output to the web server.

Repository Architecture Pattern

The Repository design pattern is a architectural pattern in software development, particularly in the context of data access and persistence. It provides a way to separate the business logic of an application from the details of how data is stored and retrieved.

The primary goal of the Repository pattern is to create an abstraction layer between the application and the data persistence mechanism, such as a database, file system, or external

service. It encapsulates the logic for querying and manipulating data, providing a clean and consistent interface for the rest of the application to interact with the data storage.

Key components of the Repository pattern are:

Repository: The Repository is an interface or an abstract class that defines the contract for accessing and manipulating data. It declares methods for common operations like creating, reading, updating, and deleting entities. The repository provides a higher-level, domain-specific API that shields the application from the low-level details of data storage.

Concrete Repository: The Concrete Repository classes implement the Repository interface and provide the specific implementation details for interacting with a particular data storage mechanism. They handle the low-level details of data access, such as constructing queries, executing database operations

Entity: An Entity represents a domain object or a data model within the application. It typically corresponds to a table in a database or a data structure in memory. Entities are the objects that are stored, retrieved, and manipulated through the repository.

By using the Repository pattern, the application can work with the abstraction provided by the repository interface, without being tightly coupled to specific data access technologies or implementation details. This separation of concerns improves code maintainability, testability, and allows for easier switching or extension of data storage mechanisms.