Software Architecture Patterns and corresponding UML class diagrams

Software architecture is like the blueprint for building software, showing how different parts fit together and interact. It helps the development team understand how to build the software according to customer requirements. There are many ways to organize these parts, called software architecture patterns. In this document we will analyze client-server, 3-tier and model-view-controller architecture.

Client – Server Architecture:
A client-server architecture pattern is described as a distributed application structure having two main components – a client and a server. This architecture facilitates the communication between the client and the server, which may or may not be under the same network. A client requests specific resources to be fetched from the server, which might be in the form of data, content, services, files, etc. The server identifies the requests made and responds to the client appropriately by sending over the requested resources.

Clients and servers exchange messages in a request-response messaging pattern. The client sends a request, and the server returns a response. This exchange of messages is an example of inter-process communication. To communicate, the computers must have a common language, and they must follow rules so that both the client and the server know what to expect. The language and rules of communication are defined in a communication protocol.

To further clarify “Client-Server” architecture, let's explore a specific example.
Client-Server example:
Application Description: The **Library Management System** is a desktop application for librarians, enabling them to manage book collections, user information, and borrowing transactions. Operating with a thick client architecture, the application handles all business logic locally, while a server database stores and retrieves data.
Functional Requirements

1. **Book Management**: Add, update, and delete book records, including details such as title, author, ISBN, and availability status.
2. **User Management**: Register new users, update user information, and manage user records.
3. **Transaction Management**: Facilitate book borrowing and returning, tracking user and book details for each transaction.
4. **Data Storage**: Store and retrieve all book, user, and transaction data in a centralized server database.

Class diagram in plantUML of the application with client-server architecture pattern:

```
@startuml
package "Client" {
  class LibraryApp {
    - bookManager: BookManager
    - userManager: UserManager
    - transactionManager: TransactionManager
    + run()
  }

class BookManager {
    + addBook(title: String, author: String, isbn: String)
    + updateBook(bookId: Int)
    + deleteBook(bookId: Int)
  }
```

```
    class UserManager {
      + addUser(name: String, contactInfo: String)
      + updateUser(userId: Int)
      + deleteUser(userId: Int)
    }

    class TransactionManager {
      + borrowBook(userId: Int, bookId: Int)
      + returnBook(transactionId: Int)
    }

    LibraryApp --> BookManager
    LibraryApp --> UserManager
    LibraryApp --> TransactionManager
}

package "Server" {
    class Book {
      - bookId: Int
      - title: String
      - author: String
      - isbn: String
      - availabilityStatus: Boolean
    }

    class User {
      - userId: Int
      - name: String
      - contactInfo: String
    }

    class Transaction {
      - transactionId: Int
      - bookId: Int
      - userId: Int
      - transactionDate: Date
    }

    interface Database {
      + saveData(data: String)
      + fetchData(query: String): String
    }

    Database <|-- Book
    Database <|-- User
    Database <|-- Transaction
}

BookManager --> Server.Database
UserManager --> Server.Database
TransactionManager --> Server.Database

@enduml
```

Three – tier architecture:
Three-tier architecture is a well-established software application architecture that organizes applications into three logical and physical computing tiers: the presentation tier, or user interface; the application tier, where data is processed; and the data tier, where application data is stored and managed.

- *Presentation tier*

  The presentation tier is the user interface and communication layer of the application, where the end user interacts with the application. Its main purpose is to display information to and collect information from the user. This top-level tier can run on a web browser, as desktop application, or a graphical user interface (GUI), for example.

- *Application tier*

  The application tier, also known as the logic tier or middle tier, is the heart of the application. In this tier, information that is collected in the presentation tier is processed - sometimes against other information in the data tier - using business logic, a specific set of business rules.

- *Data tier*

  The data tier, sometimes called database tier, data access tier or back-end, is where the information that is processed by the application is stored and managed. This can be a relational database system or in a NoSQL Database server.

  Important info: In a three-tier application, all communication goes through the application tier.

- The **Presentation Layer** should never directly access the **Data Layer**. All requests go through the **Application Layer**.
- **Application Logic** should act as a middleman, containing all business rules.

To further clarify "Three-tier" architecture, let's explore a specific example.
Three-tier architecture example:
Requirements:
E-commerce websites

- *Presentation Layer: The online storefront with product catalogs, shopping carts, and checkout interfaces.*
- *Logic Layer: Handles searching, order processing, inventory management, interfacing with 3rd-party payment vendors, and business rules like discounts and promotions.*
- *Data Layer: Stores product information, customer data, order history, and financial transactions in a database.*

Class Diagram in PlantUML for above problem with Three-tier architecture:

```
@startuml
class Storefront {
        - ProductCatalog : List<Product>
        - ShoppingCart : ShoppingCart
        - CheckoutInterface : Checkout
        + viewProducts() : void
        + addToCart(product: Product) : void
        + proceedToCheckout() : void
}
package "Logic Layer" {
        class OrderProcessor {
        - paymentProcessor : PaymentProcessor
        - inventoryManager : InventoryManager
```

```
        + processOrder(order: Order) : void
        + applyDiscount(order: Order, discountCode: String) : float
        }

        class InventoryManager {
        + checkStock(product: Product) : boolean
        + updateStock(product: Product, quantity: int) : void
        }

        class PaymentProcessor {
        + processPayment(paymentInfo: PaymentInfo) : boolean
        + connectTo3rdParty(paymentInfo: PaymentInfo) : boolean
        }
}

class Database {
        + storeProductData(product: Product) : void
        + retrieveProductData(productId: int) : Product
        + storeCustomerData(customer: Customer) : void
        + storeOrderHistory(order: Order) : void
        + retrieveOrderHistory(customerId: int) : List<Order>
}


Storefront --> OrderProcessor : Uses
OrderProcessor --> PaymentProcessor : Uses
OrderProcessor --> InventoryManager : Uses
OrderProcessor --> Database : Accesses
InventoryManager --> Database : Accesses
PaymentProcessor --> Database : Accesses
@enduml
```

Model – View – Controller (MVC) Architecture:

The MVC framework is a software architecture that separates the user interface (UI), data storage, and business logic into three separate components. This allows developers to work on each component independently without affecting the other two parts of the application.

1. **Model.** The model layer is responsible for the application's data logic and storing and retrieving data from back-end data stores. The model layer might also include mechanisms for validating data and carrying out other data-related tasks. This layer is responsible for maintaining all aspects of the data and ensuring its integrity and accessibility.
2. **View.** The view layer provides the UI necessary to interact with the application. It includes components needed to display the data and enables users to interact with that data. For example, the view layer might include buttons, links, tables, drop-down lists or text boxes.
3. **Controller.** The controller layer contains the application logic necessary to facilitate communications across the application, acting as an interface between the view and model layers. The controller is sometimes viewed as the brains of the application, keeping everything moving and in sync.

To further clarify “Model-View-Controller” architecture, let's explore a specific example.

MVC (Model-View-Controller) architecture example:

Problem Description:

A car rental company wants to develop a software system that will enable customers to rent cars online. Customers should be able to browse available cars, select a car to rent, and make a reservation. The system should keep track of the availability of cars, as well as the reservations made by customers.

Identify Use Cases

- *Browse available cars*
- *Select a car to rent*
- *Make a reservation*

Class diagram in plantUML for above problem with MVC architecture:

```
@startuml
 class CarRentalView {
   +displayCar()
   +displayReservationDetails()
 }

class CarRentalController {
   -model: CarRentalModel
   -view: CarRentalView
   +searchCars()
   +reserveCars()
 }

class CarRentalModel {
   -customers: List<>
   -cars: List<>
   -reservations: List<>
   +fetchCars()
   +updateReservationDetails()
 }
CarRentalView -- CarRentalController: Request/Response
CarRentalController --> CarRentalModel : Request Data
CarRentalModel --> Database : Connection
@enduml
```

*The **Model** includes the **Customer**, **Car**, and **Reservation** entities, along with methods for fetching for cars and making reservations.*

*The **View** is represented by the CarRentalView, which defines methods for displaying cars and reservation details.*

*The **Controller** is represented by the CarRentalController class, which interacts with the Model and View to handle user actions and update the system state.*