Software Architecture Patterns and corresponding UML class diagrams

The big picture

Architectural design is a process for identifying the sub-systems making up a system and the framework for sub-system control and communication. The output of this design process is a description of the software architecture. Architectural design is an early stage of the system design process. It represents the link between specification and design processes and is often carried out in parallel with some specification activities. It involves identifying major system components and their communications.

Software architectures can be designed at **two levels of abstraction**:

- **Architecture in the small** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- **Architecture in the large** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Three **advantages** of explicitly designing and documenting software architecture:

- **Stakeholder communication**: Architecture may be used as a focus of discussion by system stakeholders.
- **System analysis**: Well-documented architecture enables the analysis of whether the system can meet its non-functional requirements.
- **Large-scale reuse**: The architecture may be reusable across a range of systems or entire lines of products.

Software architecture is most often represented using simple, informal **block diagrams** showing entities and relationships. **Pros**: simple, useful for communication with stakeholders, great for project planning. **Cons**: lack of semantics, types of relationships between entities, visible properties of entities in the architecture.

Uses of architectural models:

As a way of facilitating discussion about the system design

A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.

As a way of documenting an architecture that has been designed

The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural design decisions

Architectural design is a **creative process** so the process differs depending on the type of system being developed. However, a number of **common decisions** span all design processes and these decisions affect the non-functional characteristics of the system:

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Systems in the same domain often have **similar architectures** that reflect domain concepts. Application product lines are built around a core architecture with variants that satisfy particular customer requirements. The architecture of a system may be designed around one of more **architectural patterns/styles**, which capture the essence of an architecture and can be instantiated in different ways.

The particular architectural style should depend on the **non-functional system requirements**:

- **Performance**: localize critical operations and minimize communications. Use large rather than fine-grain components.
- **Security**: use a layered architecture with critical assets in the inner layers.
- **Safety**: localize safety-critical features in a small number of sub-systems.
- **Availability**: include redundant components and mechanisms for fault tolerance.
- **Maintainability**: use fine-grain, replaceable components.

Architectural views

Each architectural model only shows one view or perspective of the system. It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

4+1 view model of software architecture:

- A **logical** view, which shows the key abstractions in the system as objects or object classes.
- A **process** view, which shows how, at run-time, the system is composed of interacting processes.
- A **development** view, which shows how the software is decomposed for development.
- A **physical** view, which shows the system hardware and how software components are distributed across the processors in the system.
- Related using **use cases** or scenarios (+1).

Architectural patterns

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

Model-View-Controller

- Serves as a basis of interaction management in many web-based systems.
- Decouples three major interconnected components:
    - The model is the central component of the pattern that directly manages the data, logic and rules of the application. It is the application's dynamic data structure, independent of the user interface.
    - A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible.
    - The controller accepts input and converts it to commands for the model or view.
- Supported by most language frameworks.

Controller: Maps user actions to model updates. Selects View (Controller --> Model :State Change, Controller --> View : View Selection)

View: Renders model, Requests model updates, Sends user events to controller (View --> Controller : User events, View --> Model : State query)

Model: Encapsulates application state, Notifies view of state changes. (Model --> View : Change Notification)

Overview of Model-View-Controller (MVC) pattern:

Description: Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.

Problem description: The display presented to the user frequently changes over time in response to input or computation. Different users have different needs for how they want to view the program.s information. The system needs to reflect data changes to all users in the way that they want to view them, while making it easy to make changes to the user interface.

Solution description: This involves separating the data being manipulated from the manipulation logic and the details of display using three components: Model (a problem-domain component with data and operations, independent of the user interface), View (a data display component), and Controller (a component that receives and acts on user input).

Consequences: Advantages: views and controllers can be easily be added, removed, or changed; views can be added or changed during execution; user interface components can be changed, even at runtime. Disadvantages: views and controller are often hard to separate; frequent updates may slow data display and degrade user interface performance; the MVC style makes user interface components (views, controllers) highly dependent on model components.

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*

UML class diagram 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.*

Layered architecture

- Used to model the interfacing of sub-systems.
- Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

Layer 3: User Interface

Layer 2: User interface management, Authentication and authorization

Layer 1: Core business logic/application functionality, System utilities

Layer 0: System Support (OS, database, etc)

Overview of Layered architecture pattern:

Description: Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.

When used: Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.

Advantages: Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.

Disadvantages: In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Let's now focus on a particular type of Layered architecture: the Three-tier or Three-layered architecture pattern.

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
```

Client-server architecture

- Distributed system model which shows how data and processing is distributed across a range of components, but can also be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

A system that follows the client–server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major com- ponents of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file man- agement services, and a compile server, which offers programming language compilation services.
2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
3. A network that allows the clients to access these services. Most client–server systems are implemented as distributed systems, connected using Internet protocols.

Client–server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system.

Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

Overview of Client-Server architecture pattern:

Description: In a client-server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.

When used: Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.

Advantages: The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.

Disadvantages: Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

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
```