CA (Design Pattern Explanation)

Project Training – Computer Architecture

14/06/2024

Areeb Hammad N

Version: 1.0

Created: 14/06/2024

Last Updated: 14/06/2024

Status: DRAFT (The status would change to finalized post the BA, PM and dev team review and sign off)

Task1: Design Pattern Explanation - Prepare a one-page summary explaining the MVC (Model-View-Controller) design pattern and its two variants. Use diagrams to illustrate their structures and briefly discuss when each variant might be more appropriate to use than the others

MVC Design Pattern Summary

Model-View-Controller (MVC) is a design pattern used for developing user interfaces by dividing an application into three interconnected components: Model, View, and Controller. This separation of concerns helps manage complex applications and enhances scalability, maintainability, and flexibility.

MVC Components

Model:

Role: Represents the data and the business logic of the application. It manages data, responds to requests for information, and updates itself based on changes.

Responsibilities: Data retrieval, data storage, business rules, and logic.

View:

Role: Handles the display and presentation of data. It renders the model's data into a format suitable for user interaction.

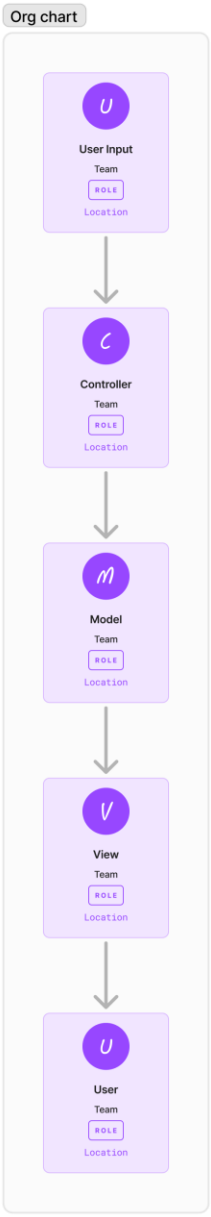
Responsibilities: User interface (UI) rendering, data presentation, and input handling.

Controller:

Role: Acts as an intermediary between the Model and View. It processes user inputs, interacts with the Model to update data, and updates the View accordingly.

Responsibilities: Handling user inputs, updating the Model, and selecting the appropriate View for display.

MVC Structure Diagram



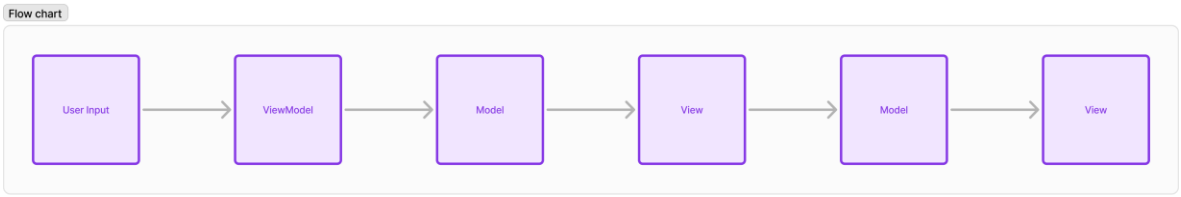
MVC Variants

1. Classic MVC:

- **Structure**: In the classic MVC model, the View is updated directly by the Controller. The Controller receives user input, processes it, updates the Model, and then updates the View accordingly.
- **When to Use**:
    - **Simplicity**: When the application has a straightforward structure with minimal real-time data updates.
    - **Direct User Interactions**: When the application has a clear separation of concerns, with simple UI interactions and minimal data-binding requirements.

2. Model-View-ViewModel (MVVM):

- **Structure**: In MVVM, the ViewModel acts as an intermediary between the Model and the View. The ViewModel provides data binding and commands, which the View can bind to. The ViewModel updates the Model and reacts to changes in the Model to update the View.
- **When to Use**:
    - **Data Binding**: When the application benefits from advanced data-binding capabilities and needs to keep the View updated automatically with changes in the Model.
    - **Complex UIs**: When the application has complex user interactions and dynamic UI updates that benefit from the data-binding features of the View Model.



Task2: Principles in Practice - Draft a one-page scenario where you apply Microservices Architecture and Event-Driven Architecture to a hypothetical e-commerce platform. Outline how SOLID principles could enhance the design. Use bullet points to indicate how DRY and KISS principles can be observed in this context.

Applying Microservices Architecture and Event-Driven Architecture to an E-Commerce Platform

Scenario:

Imagine an e-commerce platform called **ShopSmart**. ShopSmart is designed to handle a high volume of transactions, manage diverse product catalogs, support user accounts, process payments, and handle inventory updates. To achieve scalability, flexibility, and robustness, ShopSmart adopts both **Microservices Architecture** and **Event-Driven Architecture**.

Microservices Architecture:

- **Design**: ShopSmart is divided into multiple microservices, each responsible for a specific domain or function:
    - **User Service**: Manages user accounts, authentication, and profile information.
    - **Product Service**: Handles product listings, search functionality, and product details.
    - **Order Service**: Processes customer orders, manages order history, and handles order fulfillment.

    - **Payment Service**: Manages payment processing, invoicing, and transaction records.
    - **Inventory Service**: Monitors stock levels, manages inventory updates, and handles reordering.

Event-Driven Architecture:

- **Design**: Each microservice communicates through asynchronous events, allowing for real-time updates and decoupled interactions:
    - **Order Placed Event**: Triggered by the Order Service when a new order is created. This event informs the Inventory Service to update stock levels and the Payment Service to process payment.
    - **Inventory Updated Event**: Triggered by the Inventory Service when stock levels change. This event updates the Product Service to reflect current availability.
    - **Payment Processed Event**: Triggered by the Payment Service when a payment is completed. This event informs the Order Service to finalize the order and generate an invoice.

Applying SOLID Principles:

- **Single Responsibility Principle (SRP)**:
    - Each microservice has a single responsibility, focusing on one domain (e.g., User Service for user management, Payment Service for payments).
- **Open/Closed Principle (OCP)**:

    - Microservices are designed to be open for extension but closed for modification. New features can be added via new microservices or event handlers without altering existing services.
- **Liskov Substitution Principle (LSP)**:
    - Subtypes (e.g., different payment gateways) can be introduced without changing the behavior of the Payment Service, ensuring that derived components conform to the expected behavior.
- **Interface Segregation Principle (ISP)**:
    - Microservices provide specific APIs that clients use, ensuring that each microservice's interface is tailored to its own functionality, avoiding the need for clients to interact with irrelevant methods.
- **Dependency Inversion Principle (DIP)**:
    - High-level modules (e.g., Order Service) depend on abstractions (e.g., event interfaces) rather than concrete implementations (e.g., specific Payment Service). This allows for flexible integration and easier maintenance.

Observing DRY and KISS Principles:

- **Don’t Repeat Yourself (DRY)**:
    - **Centralized Event Management**: Use a centralized event broker (e.g., Kafka) to handle events, ensuring that event-handling logic is not duplicated across services.
    - **Shared Libraries**: Implement common functionalities (e.g., authentication, logging) in shared libraries that can be reused across multiple microservices.

- **Keep It Simple, Stupid (KISS)**:
    - **Service Granularity**: Design each microservice to perform a well-defined, single task to keep the services simple and focused.
    - **Event Processing**: Use straightforward event processing logic to avoid complex event chains that can be difficult to manage and debug.

Task 3: Trends and Cloud Services Overview - Write a three paragraph report covering:

1) the benefits of serverless architecture,

2) the concept of Progressive Web Apps (PWAs), and

3) the role of AI and Machine Learning in software architecture. Then, in one paragraph, describe the cloud computing service models (SaaS, PaaS, IaaS) and their use cases

Trends and Cloud Services Overview

Serverless Architecture Benefits:

Serverless architecture offers numerous advantages for modern application development. By abstracting server management away from developers, it allows them to focus solely on writing code and developing features. This architecture automatically scales applications based on demand, eliminating the need for manual scaling and reducing operational costs.

Additionally, serverless computing follows a pay-as-you-go model, where users are charged only for the actual execution time of their code, leading to cost efficiency. This model also accelerates development cycles by simplifying deployment and management processes, as developers do not need to worry about underlying infrastructure.

Progressive Web Apps (PWAs):

Progressive Web Apps (PWAs) represent a hybrid approach combining the best features of web and mobile applications. PWAs are built using standard web technologies but offer an app-like experience through features such as offline capabilities, push notifications, and faster load times. They provide a responsive and engaging user experience across various devices and platforms without requiring separate codebases for web and native applications. PWAs enhance accessibility and performance, leading to improved user engagement and retention, while also being easier and more cost-effective to develop and maintain compared to traditional native apps.

Role of AI and Machine Learning in Software Architecture:

AI and machine learning (ML) are increasingly integral to modern software architecture, driving innovations in data analysis, automation, and user personalization. AI enhances software by enabling predictive analytics, natural language processing, and intelligent decision-making capabilities. Machine learning algorithms can analyze vast amounts of data to uncover patterns, make recommendations, and improve system performance. This

integration allows applications to provide more tailored and adaptive experiences, automate complex tasks, and support advanced features such as chatbots and image recognition. As a result, software architectures are becoming more dynamic and capable of handling sophisticated tasks with greater efficiency.

Cloud Computing Service Models:

Cloud computing offers three primary service models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). **SaaS** provides fully functional software applications over the internet, eliminating the need for local installation and maintenance. Examples include Google Workspace and Microsoft Office 365. **PaaS** offers a platform allowing developers to build, deploy, and manage applications without managing the underlying infrastructure, such as AWS Elastic Beanstalk and Google App Engine. **IaaS** delivers virtualized computing resources over the internet, providing flexible and scalable infrastructure, such as Amazon EC2 and Microsoft Azure VMs. Each model addresses different needs: SaaS for end-user applications, PaaS for application development and deployment, and IaaS for managing and provisioning computing resources.