The University of Azad Jammu and Kashmir

Lab Task # 06

Course Instructor: Engr. Saba Rafique

Semester: Fall-2024

Session: 2022-2026

Submission Date: Oct 21, 2024

Submitted By: Syeda Urwa Ajmal

Roll No: 2022-SE-16

Course Name: SC&D

Code: SE-3102

Model View Control Architecture (MVC)

Contents

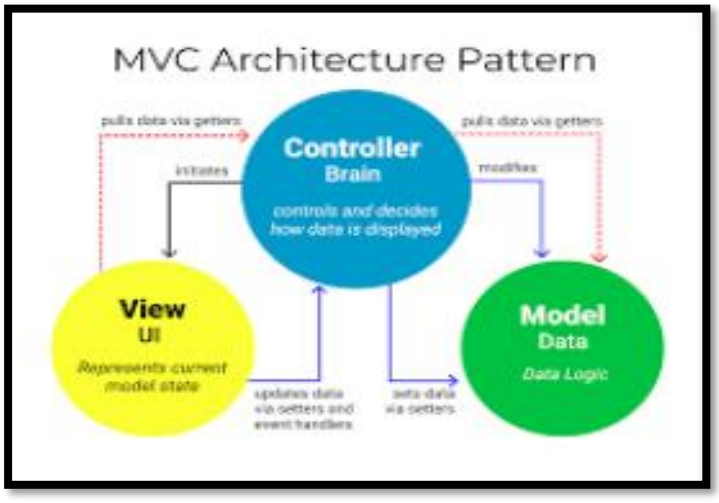
Variants of MVC Architecture

Understanding MVC Architecture

MVC (Model-View-Controller) is a fundamental software design pattern that helps in structuring applications by separating concerns. It enhances **code organization, reusability, and maintainability**, making it a widely used approach in both web and desktop applications.

Components of MVC

1. **Model (Data Layer)**
    - Represents the **business logic** and **application data**.
    - Responsible for data storage, retrieval, validation, and manipulation.
    - It interacts with the **database** and external APIs for data processing.
    - **Example:** A Product class storing product details in an e-commerce application.
2. **View (Presentation Layer)**
    - Manages the **user interface (UI)** and is responsible for displaying data.
    - Retrieves information from the **Model** but does not perform logic or calculations.
    - **Example:** A webpage displaying a list of products fetched from the database.
3. **Controller (Logic Layer)**
    - Acts as a **bridge** between the Model and the View.
    - Handles **user input**, processes requests, and updates the Model accordingly.
    - **Example:** A shopping cart system where the Controller adds selected products to the cart.



How MVC Works (Step-by-Step Execution)

1. **User Interaction**: The user interacts with the **View** (e.g., clicks a button or submits a form).
2. **Controller Processes Input**: The **Controller** receives the user request, processes it, and interacts with the **Model** if necessary.
3. **Model Updates Data**: The **Model** executes logic, retrieves or updates data, and sends it back to the Controller.
4. **View Updates UI**: The Controller passes the updated data to the **View**, which reflects the changes on the UI.

Key Features of MVC

1. Separation of Concerns

- **Model**: Manages data and logic.
- **View**: Handles UI presentation.
- **Controller**: Processes input and updates the Model/View.
- **Benefit:** This makes the application easier to manage, debug, and extend.

2. Modularity

- Components can be **developed and tested independently**.
- UI can be redesigned without modifying the data logic.
- **Example:** A website's front-end (View) can be redesigned without affecting how data is stored (Model).

3. Code Reusability

- The Model can be reused across different Views.
- Developers can modify a single part without affecting others.

4. Scalability

- The MVC structure allows adding new features without disrupting the existing code.
- Ideal for growing applications.

Variants of MVC Architecture

There are different variations of MVC used in different applications:

1. MVVM (Model-View-ViewModel)

- **Description:** The ViewModel acts as an intermediary between the View and Model.
- **Use Case:** Used in **Angular, React (with hooks), and WPF applications**.
- **Strength:** Supports **two-way data binding**, making UI updates more efficient.

2. MVP (Model-View-Presenter)

- **Description:** The Presenter handles interactions between the View and Model, while the View is passive.
- **Variants:**
  - **Passive View**: The View has no logic.
  - **Supervising Controller**: The View has minimal logic for data binding.
- **Use Case:** Used in **Android applications** before MVVM became popular.
- **Strength: Improved testability**, as the Presenter can be tested separately.

3. HMVC (Hierarchical MVC)

- **Description:** Extends MVC by using a hierarchy of MVC components, where each module has its own MVC structure.
- **Use Case: Large-scale enterprise applications** requiring modular development.
- **Strength:** Enhances **scalability and modularity**.

4. PAC (Presentation-Abstraction-Control)

- **Description:** Each component is structured as an agent with its own MVC-like layers.
- **Use Case:** Used in **distributed systems** where multiple components operate independently.
- **Strength:** Highly **modular** and **decentralized**.

5. MVT (Model-View-Template)

- **Description:** A Django-specific variation of MVC where Views handle business logic, and Templates manage UI rendering.
- **Use Case:** Used in **Django-based web applications**.
- **Strength:** Simplifies **web development with reusable templates**.

6. MVU (Model-View-Update)

- **Description:** Uses an immutable Model where the View updates the Model, and changes flow through an update function.
- **Use Case:** Used in **Elm and F# frameworks**.
- **Strength:** Simplifies **state management and debugging**.

7. MVI (Model-View-Intent)

- **Description:** Follows a **reactive programming** approach with **unidirectional data flow**.
- **Use Case:** Used in **mobile and web apps** requiring complex state management.
- **Strength:** Ensures **predictable state transitions**.

Real-Life Example of MVC Architecture – Online Shopping Cart System

The following Python implementation demonstrates an **MVC-based Shopping Cart System**:

Model (Data & Logic Layer)

```
class Product:
    def __init__(self, id, name, price):
        self.id = id
        self.name = name
        self.price = price

class Cart:
    def __init__(self):
        self.items = []

    def add_item(self, product):
        self.items.append(product)

    def total_price(self):
        return sum(item.price for item in self.items)

    def display_cart(self):
```

```
        if not self.items:
            return "Your cart is empty."
        cart_summary = "\n".join([f"{item.name}: ${item.price}" for item in self.items])
        cart_summary += f"\nTotal: ${self.total_price()}"
        return cart_summary
```

Explanation:

- Product class stores **product details** (ID, name, price).
- Cart class manages **adding products, calculating totals, and displaying contents**.

View (Presentation Layer)

```
class CartView:
    def display_products(self, products):
        print("Available Products:")
        for product in products:
            print(f"{product.id}. {product.name} - ${product.price}")

    def display_cart(self, cart):
        print("\nYour Cart:")
        print(cart.display_cart())

    def show_message(self, message):
        print(message)
```

Explanation:

- Displays **product list** and **cart contents**.
- Provides **messages** (e.g., "Product added to cart").

Controller (Action Layer)

```
class ShoppingCartController:
    def __init__(self, products, cart, view):
        self.products = products
        self.cart = cart
        self.view = view
```

```
    def run(self):
        while True:
            self.view.display_products(self.products)
            user_input = input("Enter the product number to add to cart (or 'q' to quit): ")
            if user_input.lower() == 'q':
                break
            try:
                product_id = int(user_input)
                selected_product = self.get_product_by_id(product_id)
                if selected_product:
                    self.cart.add_item(selected_product)
                    self.view.show_message(f"{selected_product.name} has been added to your cart!")
                else:
                    self.view.show_message("Product not found. Please try again.")
            except ValueError:
                self.view.show_message("Invalid input. Please enter a valid product number.")
            self.view.display_cart(self.cart)

    def get_product_by_id(self, product_id):
        return next((product for product in self.products if product.id == product_id), None)
```

Explanation:

- Processes **user input**, manages the **cart**, and updates the **View** accordingly.

Main Execution

```
def main():
    products = [Product(1, "Laptop", 999.99), Product(2, "Smartphone", 599.99), Product(3, "Headphones", 199.99)]
    cart = Cart()
    view = CartView()
    controller = ShoppingCartController(products, cart, view)
```

```
    controller.run()

if __name__ == "__main__":
    main()
```
