Model-View-Controller Pattern

Overview of MVC

- The **Model-View-Controller (MVC)** design pattern is a widely used architectural pattern for building user interfaces (UIs) in software applications.
- Its main objective is to separate the concerns of application **state management** (Model), **user interface (UI) generation** (View), and **user interaction handling** (Controller).
- By decoupling these components, MVC ensures modular, testable, and maintainable code.

Core Components of MVC

1. Model

- **Role**: The Model is responsible for managing the application's data, state, and business logic.
- **Key Features/Responsibilities**:
    - **Encapsulation**: It encapsulates the application's state, often stored in private fields.
    - **Access Methods**: Provides getter methods to retrieve state and setter methods to update the state.
    - **Observers/Notifications**: Notifies observers (like the View or Controller) whenever the state changes.
        - The rest of the app needs to know when the state is changed!
- **Example:**

```
public class Model {
    private List<Player> players;
    private int turn;

    public Player getActivePlayer() {
        return players.get(turn);
    }

    public void endTurn() {
        turn = (turn + 1) % players.size();
        notifyObservers();
    }

    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(this);
        }
    }
}
```

2. View

- **Role**: The View is responsible for rendering the UI by observing the Model's state and generating visual elements for the user.
- **Key Features/Responsibilities**:
  - **UI Representation**: Uses the Model's data to render the UI dynamically.
  - **Observability**: Observes the Model for changes to refresh the UI when the state is updated.
  - **User Interaction Handling**: Captures user interactions (e.g., button clicks) and forwards them to the Controller.
- **Example:**

```
public class AppView implements FXComponent {
    private Controller controller;

    public AppView(Controller controller) {
        this.controller = controller;
    }

    @Override
    public Parent render() {
        VBox layout = new VBox();
        Button button = new Button("Click Me");
        button.setOnAction(event -> controller.handleClick()
        layout.getChildren().add(button);
        return layout;
    }
}
```

- **Compound Views**: Views can be composed of smaller views:

```
public class CompoundView implements FXComponent {
    private Controller controller;
    private FXComponent leftPanel;
    private FXComponent rightPanel;

    public CompoundView(Controller controller) {
        this.controller = controller;
        this.leftPanel = new LeftPanel(controller);
        this.rightPanel = new RightPanel(controller);
    }

    @Override
    public Parent render() {
        HBox layout = new HBox();
        layout.getChildren().add(leftPanel.render());
        layout.getChildren().add(rightPanel.render());
        return layout;
    }
}
```

3. Controller

- **Role**: The Controller is the intermediary between the View and the Model. It processes user input, updates the Model, and may notify the View to refresh.
- **Key Features/Responsibilities**:
    - **Event Handlers**: Defines methods to handle user interactions (e.g., button clicks, swipes) - i.e. consists of methods that translate user interaction events into commands for the model.
    - **Model Updates**: Translates user input into Model changes by calling setter methods.
    - Usually needs to encapsulate a reference to the model.
- **Example**:

```
public class Controller {
    private Model model;

    public Controller(Model model) {
        this.model = model;
    }

    public void handleSwipe(Direction dir) {
        switch (dir) {
            case UP:
                model.swipeUp();
                break;
            case DOWN:
                model.swipeDown();
                break;
            case LEFT:
                model.swipeLeft();
                break;
            case RIGHT:
                model.swipeRight();
                break;
        }
    }
}
```

Interaction Between Components

- MVC components interact in structured ways depending on the variation of the MVC implementation.

1. **Classic MVC**

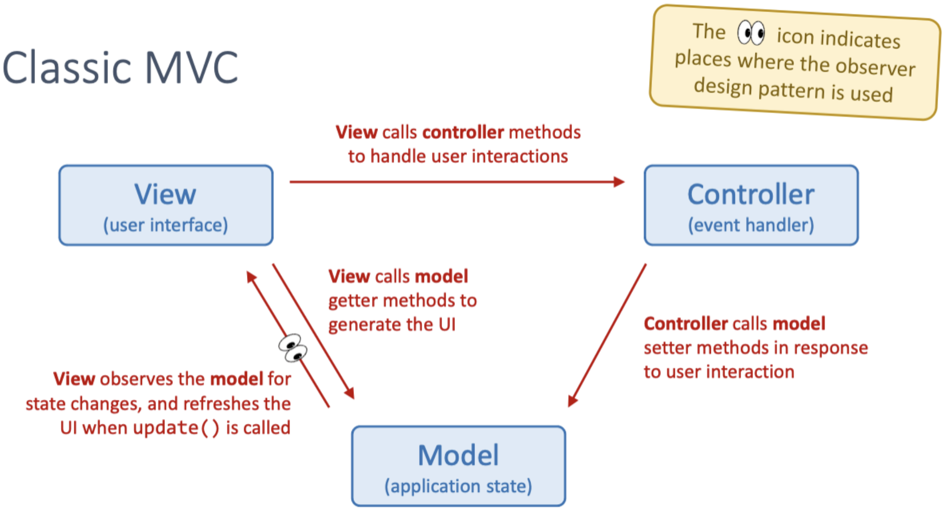
    - **Flow**:

        1. The View observes the Model for changes and calls `update()` to refresh the UI.
        2. The View forwards user interactions to the Controller.
        3. The Controller updates the Model based on the user interactions.
        4. The Model notifies the View of any state changes.

    - **Setup**:

        ```
        Model model = new Model();
        Controller controller = new Controller(model);
        View view = new View(controller, model);
        model.addObserver(view);
        ```

    - **Illustration:**

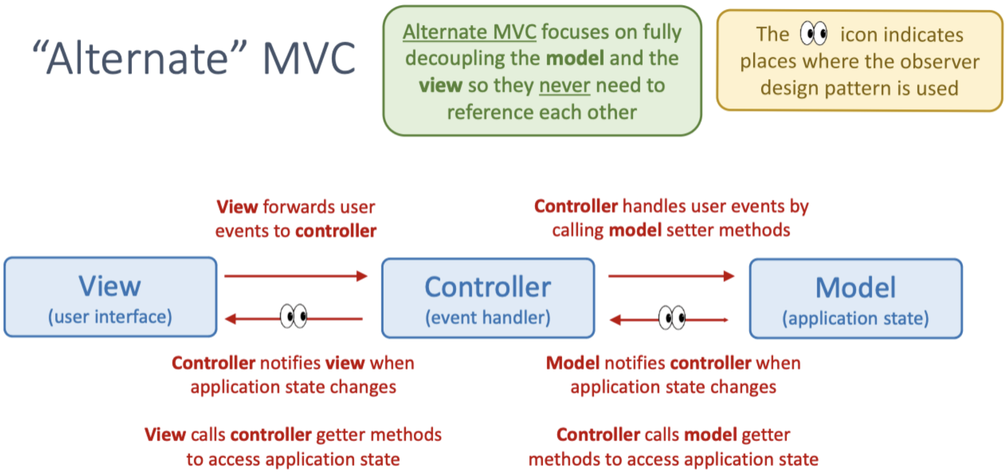

Classic MVC
The 👀 icon indicates places where the observer design pattern is used
View calls controller methods to handle user interactions
View (user interface)
Controller (event handler)
View calls model getter methods to generate the UI
Controller calls model setter methods in response to user interaction
View observes the model for state changes, and refreshes the UI when update() is called
Model (application state)

2. **Alternate MVC**

- Focuses on **decoupling** the View and Model so that they never directly interact. Any interactions are handled and must be through the **Controller.**
- **Flow**:
  1. The Model notifies the Controller when its state changes.
  2. The Controller retrieves the updated state from the Model and forwards it to the View.
  3. The View captures user events and forwards them to the Controller.
  4. The Controller updates the Model based on user interactions.
- **Setup**:

```
Model model = new Model();
Controller controller = new Controller(model);
model.addObserver(controller);
View view = new View(controller);
controller.addObserver(view);
```

- **Illustration:**



“Alternate” MVC

Alternate MVC focuses on fully decoupling the **model** and the **view** so they never need to reference each other

The 👀 icon indicates places where the observer design pattern is used

View forwards user events to **controller**

Controller handles user events by calling **model** setter methods

Controller notifies **view** when application state changes

Model notifies **controller** when application state changes

View calls **controller** getter methods to access application state

Controller calls **model** getter methods to access application state

Applying MVC: Example with 2048 Game

- **Model Responsibilities**:
  - Encapsulate game state such as the grid, score, and win/loss condition.
  - Expose methods to get the current grid state, score, etc.
  - Provide methods like `swipeUp()`, `swipeDown()`, etc., to modify the state.
- **View Responsibilities**:
  - Render the game grid and score dynamically based on the Model's data.
  - Refresh the UI whenever the grid or score updates.
  - Forward user inputs (e.g., swipes) to the Controller.
- **Controller Responsibilities**:
  - Handle swipe inputs by calling appropriate Model methods.
  - Notify the View when the Model state changes to ensure UI updates.