Model-View-Presenter (MVP) Pattern: Passive View Variant

Affan Rauf

September 20, 2024

What is MVP?

- **Model-View-Presenter (MVP)** is a design pattern for organizing the presentation logic of an application.
- It consists of three components:
  - **Model**: Manages data and business logic.
  - **View**: Displays data and receives user input (completely passive in this variant).
  - **Presenter**: Acts as an intermediary between the Model and the View, handling all the logic.
- MVP is a more structured variant of MVC, often used in UI-heavy applications.

What is Passive View in MVP?

- **Passive View** is a specific form of the MVP pattern where:
  - The **View** is completely passive and only displays data.
  - The **Presenter** is responsible for handling all logic, including updating the View.
  - There is no direct data-binding or Observer mechanism between the Model and View.
- The **Presenter** directly controls the View and updates it based on changes in the Model.
- The View exposes methods for the Presenter to modify it but does not perform any logic on its own.

Key Characteristics of Passive View

- **Separation of Responsibilities**: The Presenter fully controls the View, keeping the logic and data management isolated from the View.
- **No Observer Pattern**: Unlike MVC, the View is not automatically updated when the Model changes. The Presenter must explicitly update the View.
- **Simplified View**: The View is essentially "dumb" and only acts as a display, with no decision-making or logic.
- **Testability**: Since the Presenter contains all logic, it can be easily unit tested without requiring a UI.

MVP (Passive View) Flow

1. **User interacts with the View** (e.g., button click).
2. **View informs the Presenter** of the user action.
3. **Presenter updates the Model** based on user input.
4. **Model changes** (business logic, data changes).
5. **Presenter retrieves data from Model** and updates the View explicitly.

Example: Passive View in MVP

View Interface:

```
interface View {
    void showData(String data);
    void setLoadingIndicator(boolean isVisible);
}
```

Example: Passive View in MVP

View Interface:

```
interface View {
    void showData(String data);
    void setLoadingIndicator(boolean isVisible);
}
```

Presenter:

```
class Presenter {
    private View view;
    private Model model;

    public Presenter(View view, Model model) {
        this.view = view;
        this.model = model;
    }

    public void onLoadData() {
        view.setLoadingIndicator(true);
        String data = model.getData();
        view.showData(data);
        view.setLoadingIndicator(false);
    }
}
```

Code Explanation

- **View Interface**: Defines methods the View must implement, but the Presenter controls all updates to the View.
- **Presenter**: Handles the logic of retrieving data from the Model and updating the View. The View does not interact with the Model directly.
- **Model**: Contains the business logic and data but has no direct connection to the View.

Benefits of Passive View

- **Decoupling of UI and Logic**: The Presenter handles all the logic, while the View is only responsible for displaying data.
- **Easier Testing**: The Presenter can be tested independently of the View, since all logic resides in the Presenter.
- **Simplified View**: The View only has methods to display or update data, making it easy to implement and change without affecting the logic.
- **Predictable Flow**: All updates to the View are controlled explicitly by the Presenter, reducing complexity.

Drawbacks of Passive View

- **Overburdened Presenter**: The Presenter is responsible for all interactions between the Model and View, which can lead to bloated code.
- **Manual Updates**: Since there is no automatic Observer-like mechanism, the Presenter must always explicitly update the View, which can be error-prone.
- **Limited Flexibility**: Because the View is completely passive, any complex interactions or dynamic updates must be handled entirely by the Presenter.

When to Use Passive View

Totally separate the logic ,for testing purpose.

- **UI Testability**: When you need to separate UI logic for unit testing without depending on the actual View.
- **Simple UI Logic**: For applications where the View's role is simply to display data and receive user input without needing complex interactions.
- **Explicit Control**: When you want the Presenter to have full control over all updates and interactions with the View, making the flow more predictable.