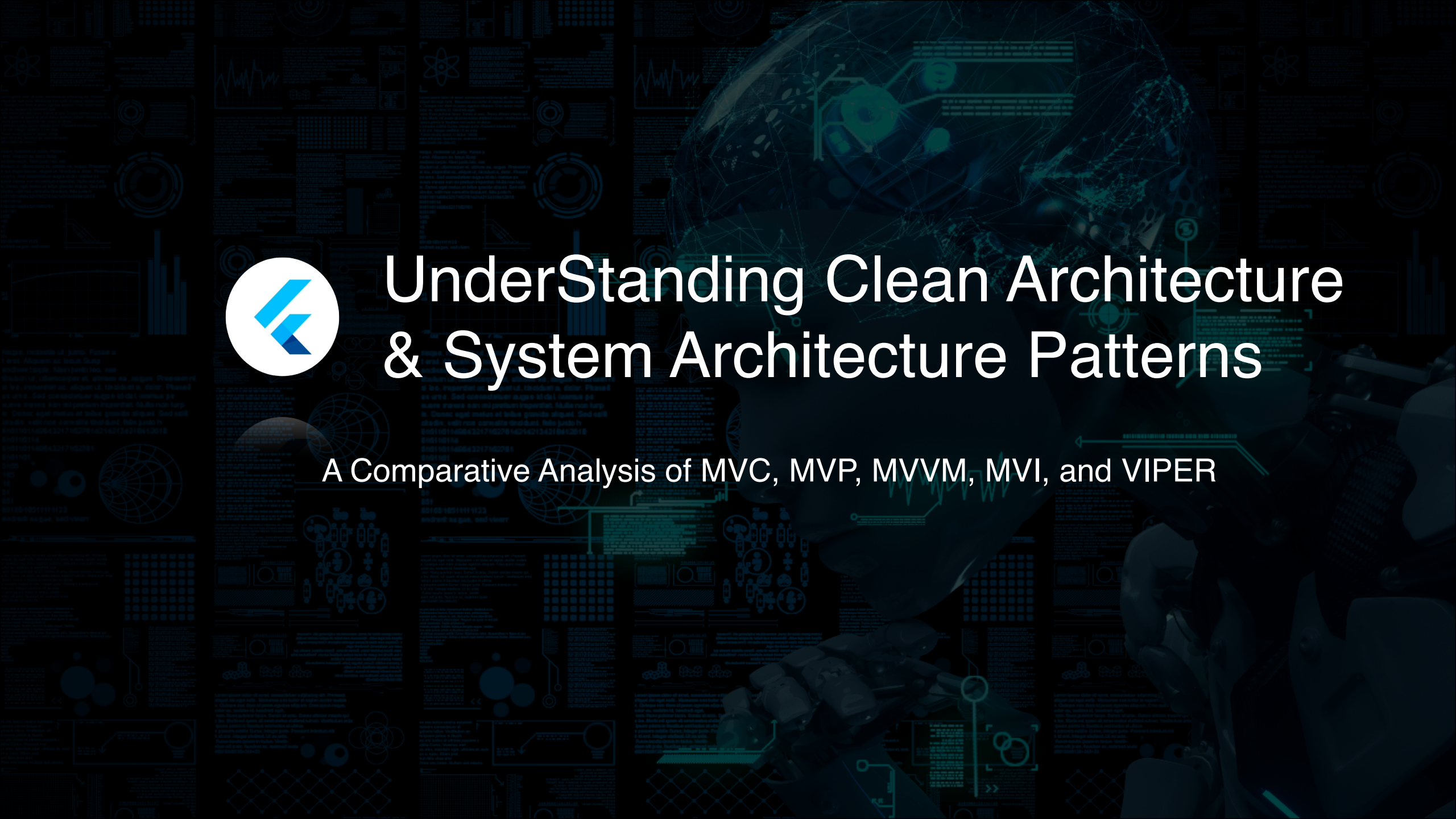


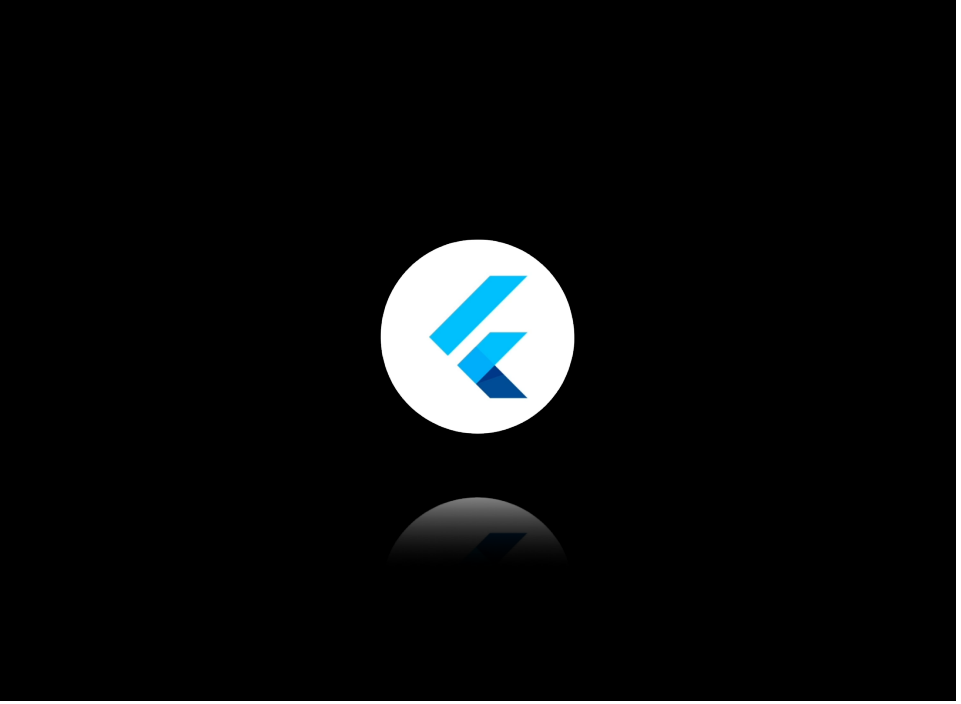
UnderStanding Clean Architecture & System Architecture Patterns

A Comparative Analysis of MVC, MVP, MVVM, MVI, and VIPER

Introduction

- The main purpose of implementing the clean architecture is the separation of concern (SoC).
- Today, we'll explore five popular software patterns:
    1. MVC (Model View Controller)
    2. MVP (Model View Presenter)
    3. MVI (Model View Intent)
    4. MVVM (Model View ViewModel)
    5. VIPER (View Interactor Presenter Entity Router)
- Each pattern has its strengths and weaknesses, making it essential to select the one that aligns with your project's unique requirements.

Building Scalable Apps

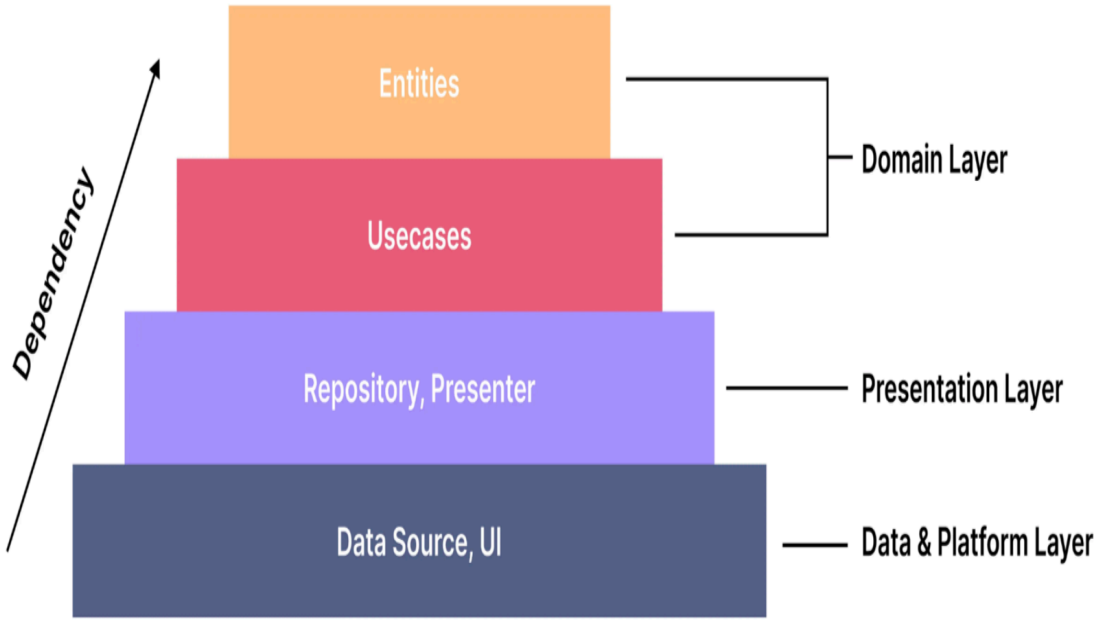


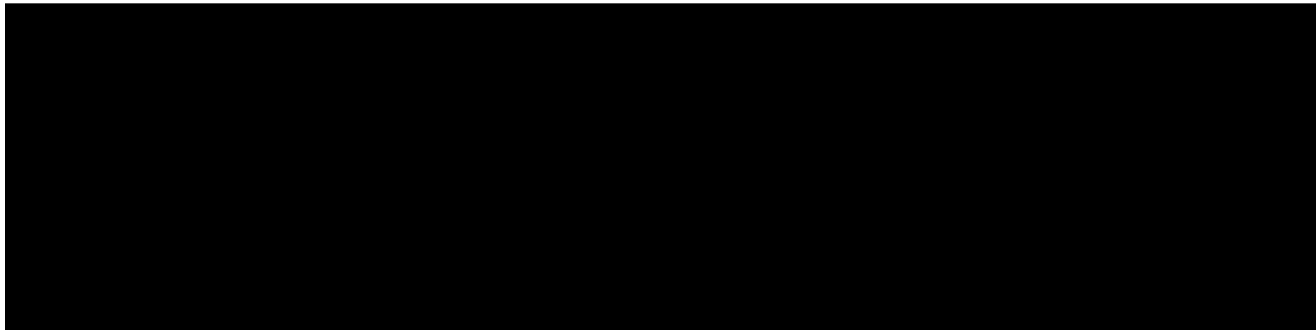
Clean Architecture

Architecture is very important in developing an application. Architecture can be likened to a floor plan that describes how the flow in an application project. The main purpose of implementing the architecture is the separation of concern (SoC). So, it will be easier if we can work by focusing on one thing at a time.

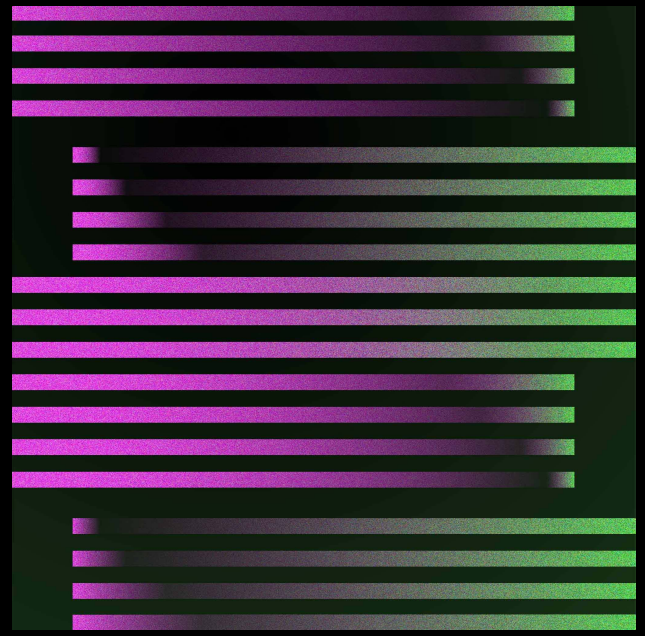
In the context of Flutter, clean architecture will help us to separate code for business logic with code related to platforms such as UI, state management, and external data sources. In addition, the code that we write can be easier to test (testable) independently.

https://pub.dev/packages/flutter_clean_architecture





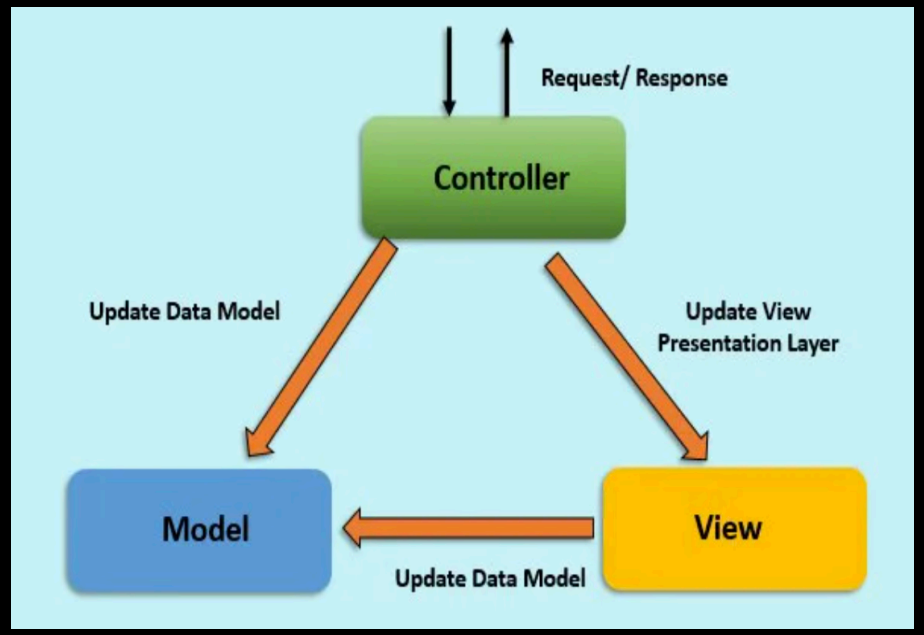
Features of Clean Architecture



- **Separation of concerns:** This principle means that different entities should be separated in the code. It helps with reusing those entities, systematising development, easy debugging, and isolating frequently changing parts so they don’t affect others.
- **Testability:** With appropriate architecture, it is way easier. QA’s will find it helpful to write test cases and be able to test functionalities separately. You will avoid finding issues in runtime, which usually means a week-long fix.
- **Reliability:** A well-chosen architecture is crucial to creating stable apps without major inconsistencies because it defines how parts of code interact with each other.
- **Scalability:** Your app should have a solid basis for the buildup of new features and changes according to business needs. It also should be suitable for future renovations (e. g., new libraries, operating systems) in programming technologies.
- **Maintainability and ease of use:** A good architecture simplifies code for both writing and reading. It can also result in a low maintenance cost.

MVC Architecture Pattern

- Model: Represents the data and business logic.
- View: Displays the user interface.
- Controller: Handles user input and updates the model and view.



1. Use Case

MVC can be a good choice for small to moderately complex applications where the user interface is straightforward.

2. Strengths

Simplicity, well-known pattern, easy to understand for small teams, and works well when the UI is closely tied to platform-specific components.

3. Weaknesses

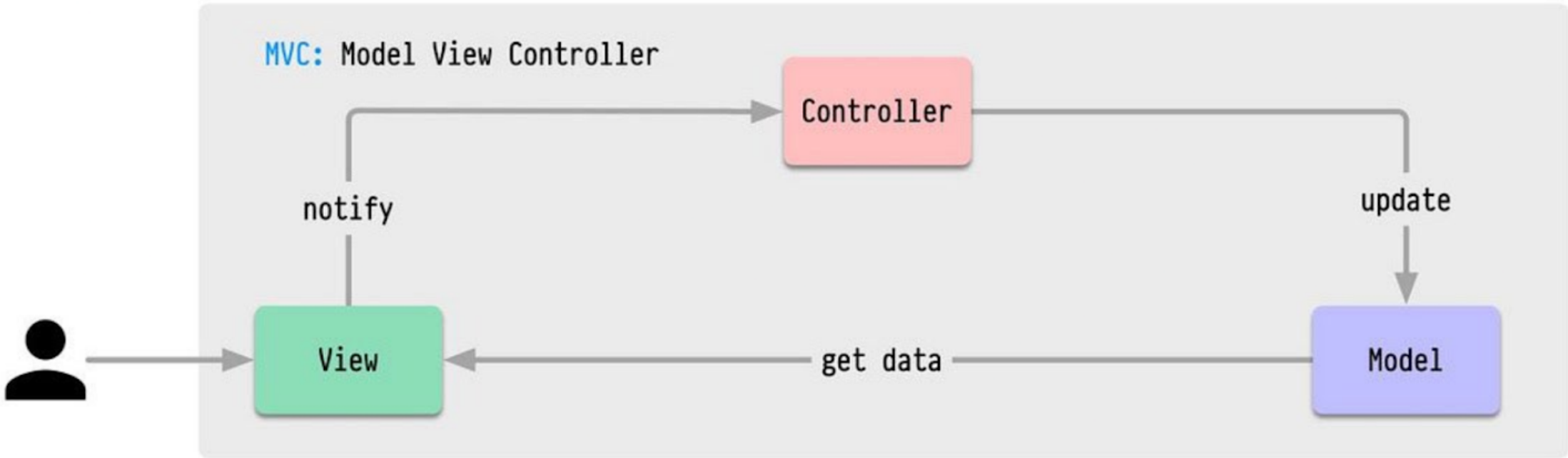
Can lead to massive view controllers in iOS development, making code harder to maintain in large applications.

4. Example

A simple to-do list app is a good example of where MVC can be used effectively.

MVC Architecture Pattern Example

- The model would store the list of to-do items.
- The view would display the list of to-do items.
- The controller would handle user actions such as adding, editing, and deleting to-do items.

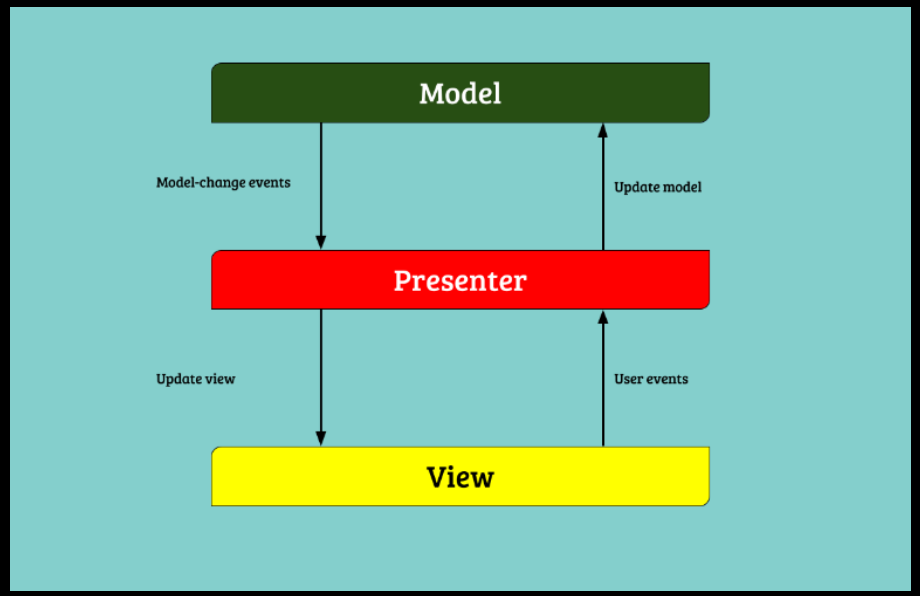


To-do List using the MVC Pattern

- In this example, the Todo class represents the model. It stores the title and completion status of a to-do item.
- The TodoView class represents the view. It displays the list of to-do items and allows the user to add new to-do items and toggle the completion status of existing to-do items.
- The TodoController class represents the controller. It handles user actions such as adding, editing, and deleting to-do items. It also maintains the state of the to-do list, which is represented by the todos variable.
- To use the app, the user would first create a TodoController object. Then, they would pass the TodoController object to the TodoView class. The TodoView class would then display the list of to-do items and allow the user to interact with them.
- When the user adds a new to-do item, the TodoView class would call the addTodo() method on the TodoController object. The TodoController object would then add a new Todo object to the todos variable. The TodoView class would then update the UI to reflect the new to-do item.
- When the user toggles the completion status of a to-do item, the TodoView class would call the toggleTodo() method on the TodoController object.
- The TodoController object would then toggle the completion status of the corresponding Todo object in the todos variable. The TodoView class would then update the UI to reflect the new completion status of the to-do item.

MVP Architecture Pattern

- Model: Represents the data and business logic.
- View: Displays the user interface and sends user input to the presenter.
- Presenter: Handles user input, updates the model, and updates the view.



1. Use Case

MVP is suitable for applications that require better separation of concerns than MVC provides, making it easier to unit test and maintain.

2. Strengths

Separation of concerns, improved testability due to the presentation logic residing in the presenter, and better suited for large and complex UIs.

3. Weaknesses

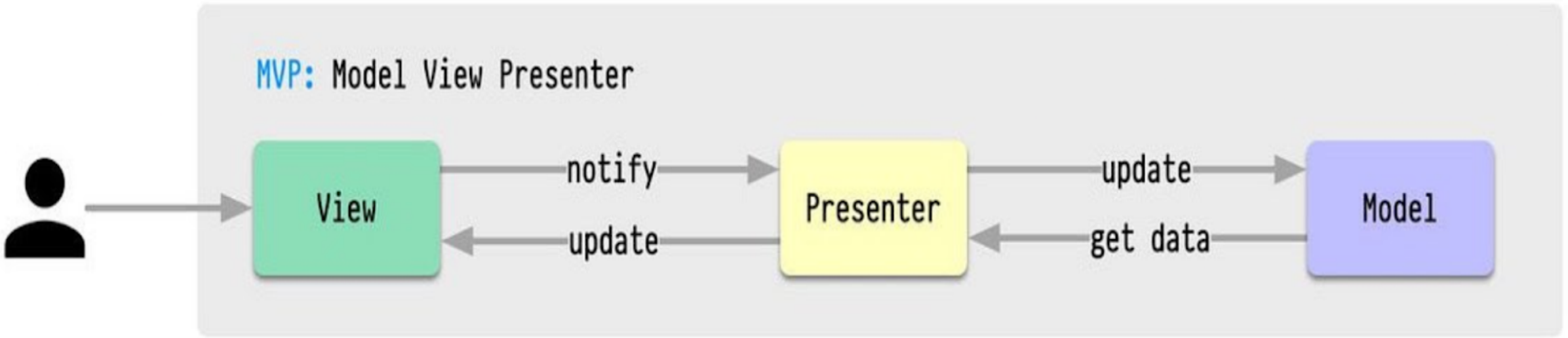
Still requires careful management of the contract between the view and presenter, which can become complex in large projects.

4. Example

A more complex app, such as a social media app, could benefit from using MVP.

MVP Architecture Pattern Example

- The model would store the user profile data.
- The view would display the user profile data.
- The presenter would handle user actions such as updating the user profile and posting new content.

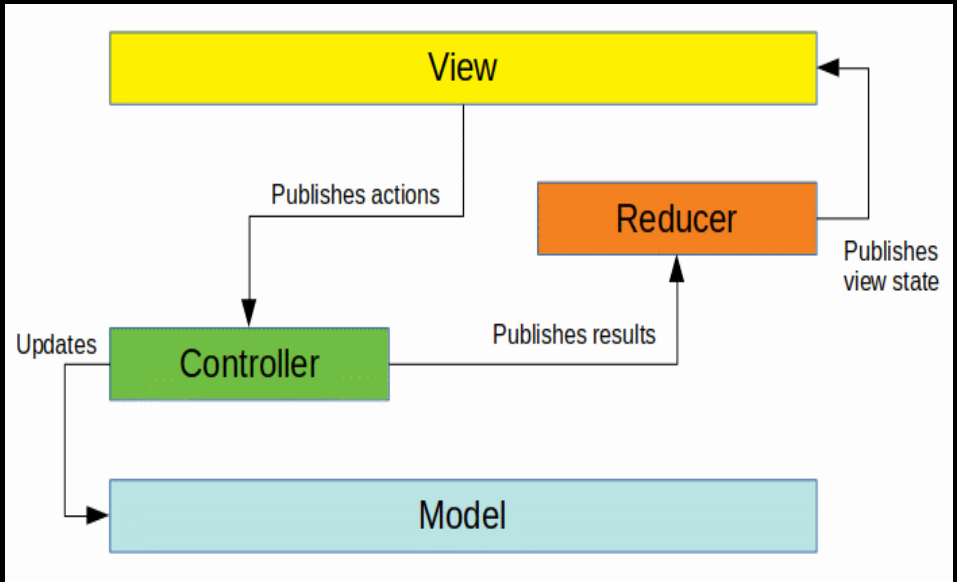


- In this example, the User class represents the model. It stores the user profile data, such as the user's name, profile picture URL, and bio.
- The UserProfileView class represents the view. It displays the user's profile
- The UserPresenter class represents the presenter. It handles user actions such as getting the user profile and updating the user profile. It also gets and updates the user profile data from the UserRepository.
- To use the app, the user would first create a UserPresenter object. Then, they would pass the UserPresenter object to the UserProfileView class. The UserProfileView class would then display the user's profile data.
- When the user wants to get their profile, the UserProfileView class would call the getUserProfile() method on the UserPresenter object. The UserPresenter object would then call the getUserProfile() method on the UserRepository object. The UserRepository object would then get the user profile data from a remote API or database and return it to the UserPresenter object. The UserPresenter object would then update the view with the user profile data.
- When the user wants to update their profile, the UserProfileView class would call the updateUserProfile() method on the UserPresenter object. The UserPresenter object would then call the updateUserProfile() method on the UserRepository object. The UserRepository object would then update the user profile data in a remote API or database and return to the UserPresenter object. The UserPresenter object would then update the view with the new user profile data.
- The MVP architecture is a good choice for this application because it helps to separate the business logic from the UI. This makes the code more modular, reusable, and testable.

Social media app List using the MVP

MVI Architecture Pattern

- Model: Represents the data and business logic.
- View: Displays the user interface.
- Controller: Handles user input and updates the model and view.



1. Use Case

MVI is well-suited for applications with complex and highly interactive user interfaces, where you need a clear, unidirectional data flow.

2. Strengths

Enforces a unidirectional data flow, immutability, and easy state management, making it excellent for reactive UIs and apps with real-time updates.

3. Weaknesses

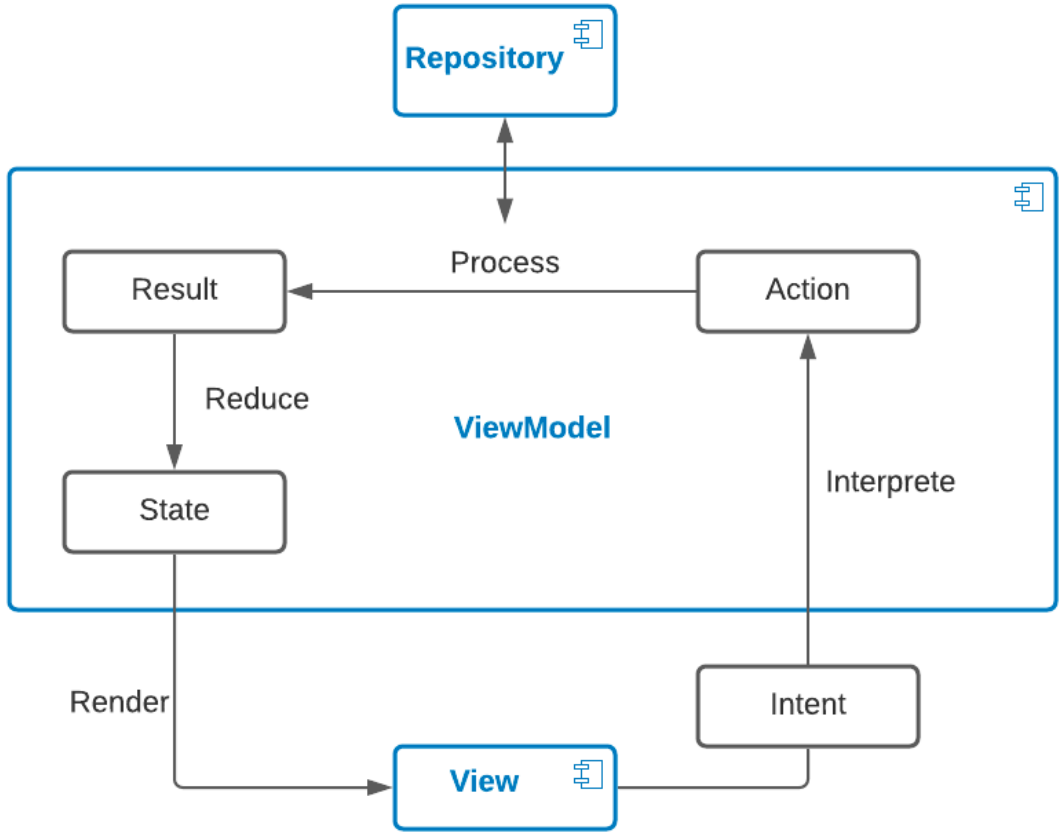
Can be overkill for simple applications and may require a learning curve for developers unfamiliar with reactive programming concepts.

4. Example

A real-time chat app is a good example of where MVI can be used effectively.

- The model would store the chat history.
- The view would display the chat history.
- The intent would handle user actions such as sending new messages.

MVI Architecture Pattern Example

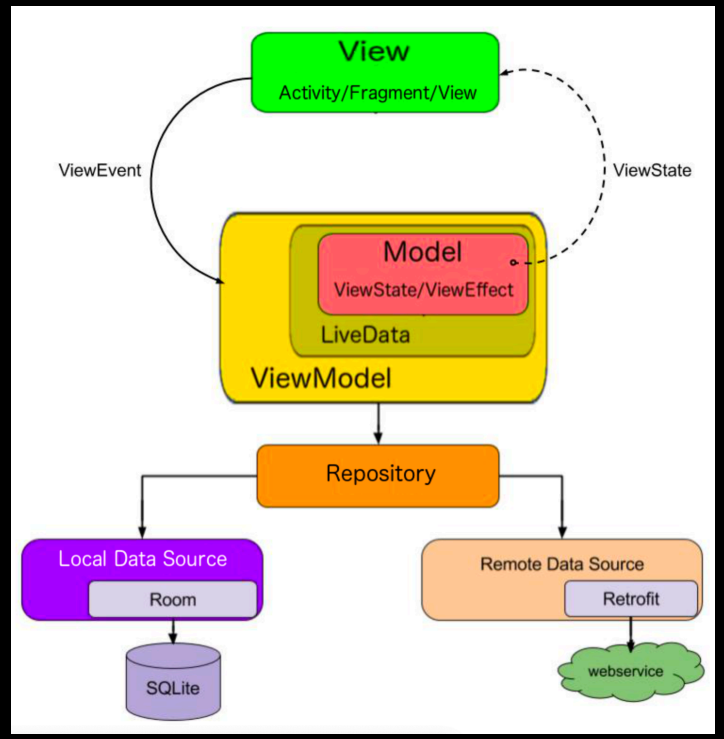


- In this example, the ChatState class represents the model. It stores the chat history.
- The ChatView class represents the view. It displays the chat history and allows the user to send new messages.
- The SendChatMessageIntent class represents the intent. It handles the user action of sending a new message.
- The reducer() function is used to update the model in response to intents.
- The sendChatMessageEpic() epic is used to perform side effects, such as sending the chat message to a remote server.
- The ChatStore class is responsible for managing the state of the application. It provides a stream of the current state and a method to dispatch intents.
- To use the app, the user would first create a ChatStore object. Then, they would pass the ChatStore object to the ChatView class. The ChatView class would then display the chat history and allow the user to send new messages.
- When the user sends a new message, the ChatView class would dispatch a SendChatMessageIntent object. The ChatStore object would then call the reducer() function to update the model. The ChatView class would then update the UI to reflect the new state.
- The MVI architecture is a good choice for this application because it provides a unidirectional data flow. This makes the code easier to reason about and debug. It also makes it easier to test the code, as the state can be easily controlled.

Real-time chat app List using the MVI

MVVM Architecture Pattern

- Model: Represents the data and business logic.
- View: Displays the user interface.
- Controller: Handles user input and updates the model and view.



1. Use Case

MVVM is ideal for applications where data-binding and a clear separation of UI and business logic are essential, such as data-driven apps.

2. Strengths

Promotes data-binding, testability, and maintainability, making it well-suited for apps with complex UIs and data-heavy requirements.

3. Weaknesses

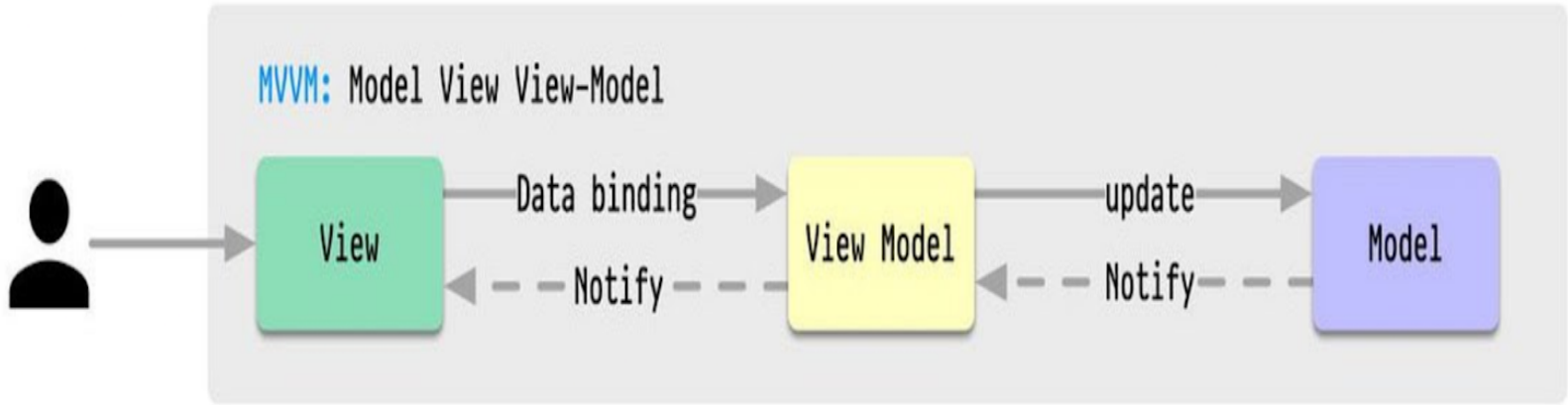
May require additional setup and libraries (e.g., data-binding libraries) in some development environments.

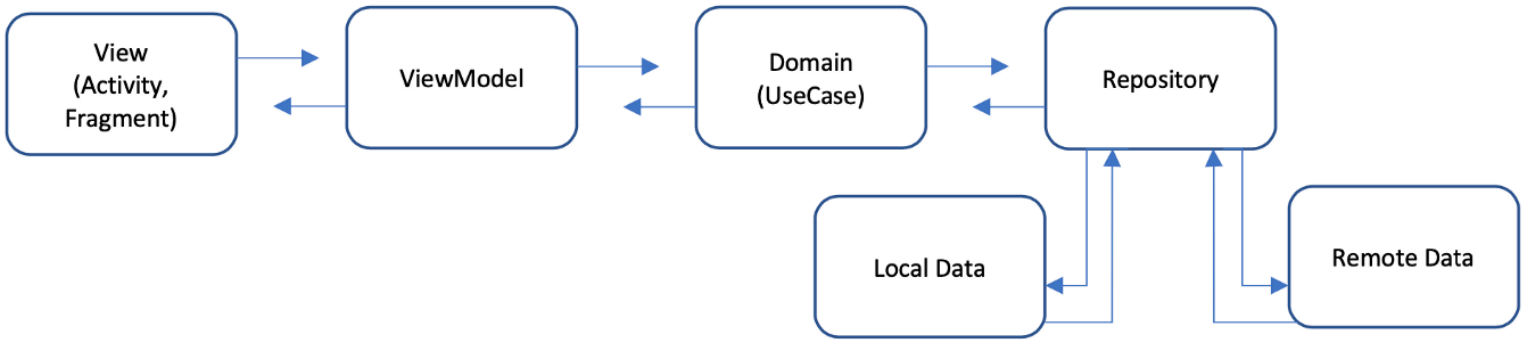
4. Example

A data-driven app, such as a stock trading app, could benefit from using MVVM.

MVVM Architecture Pattern Example

- The Model would store the stock data.
- The view would display the stock data.
- The view model would handle user interactions such as buying and selling stocks.





- In this example, the Stock class represents the model. It stores the stock data, such as the stock name and price.
- The StockViewModel class represents the view model. It exposes the stock data to the view and handles user interactions such as buying and selling stocks.
- The StockView class represents the view. It displays the stock data and allows the user to buy and sell stocks.
- To use the app, the user would first create a StockViewModel object. Then, they would pass the StockViewModel object to the StockView class. The StockView class would then display the stock data and allow the user to buy and sell stocks.
- When the user wants to buy a stock, the StockView class would call the buyStock() method on the StockViewModel object. The StockViewModel object would then implement this method to buy the stock.
- When the user wants to sell a stock, the StockView class would call the sellStock() method on the StockViewModel object. The StockViewModel object would then implement this method to sell the stock.
- The MVVM architecture is a good choice for this application because it makes it easy to bind data to the UI. This simplifies development, as the developer does not have to worry about how to update the UI when the data changes.
- The MVVM architecture also makes the code more modular and reusable. The view model can be easily reused in different views, and the view can be easily reused with different view models.

Stock trading app using the MVVM Pattern

VIPER Architecture Pattern

- View: Displays the user interface and sends user input to the presenter.
- Interactor: Contains business logic and interacts with entities.
- Presenter: Handles user input, updates the view, and communicates with the interactor.
- Entity: Represents data objects.
- Router: Handles navigation between screens.

1. Use Case

VIPER is suitable for large, complex, and modular applications that require strict separation of concerns and maintainability.

2. Strengths

Strong separation of components, facilitates unit testing, and allows for easy scalability as the project grows, making it a good fit for enterprise-level applications.

3. Weaknesses

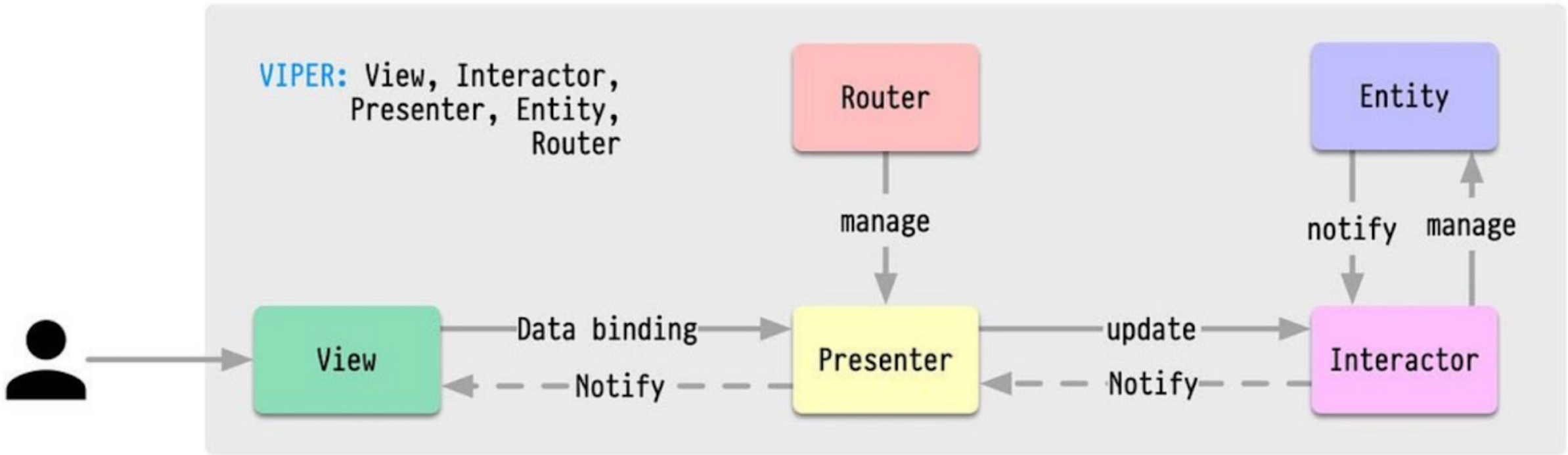
Can introduce significant initial complexity, which might not be justified for smaller projects or prototypes.

4. Example

A large and complex enterprise app, such as an e-commerce app, could benefit from using VIPER.

- The view would display the product catalog.
- The interactor would handle the business logic of adding items to the cart.
- The presenter will handle the view of the app.
- The entity is basically model oh the product.
- The router would navigate between different screens.

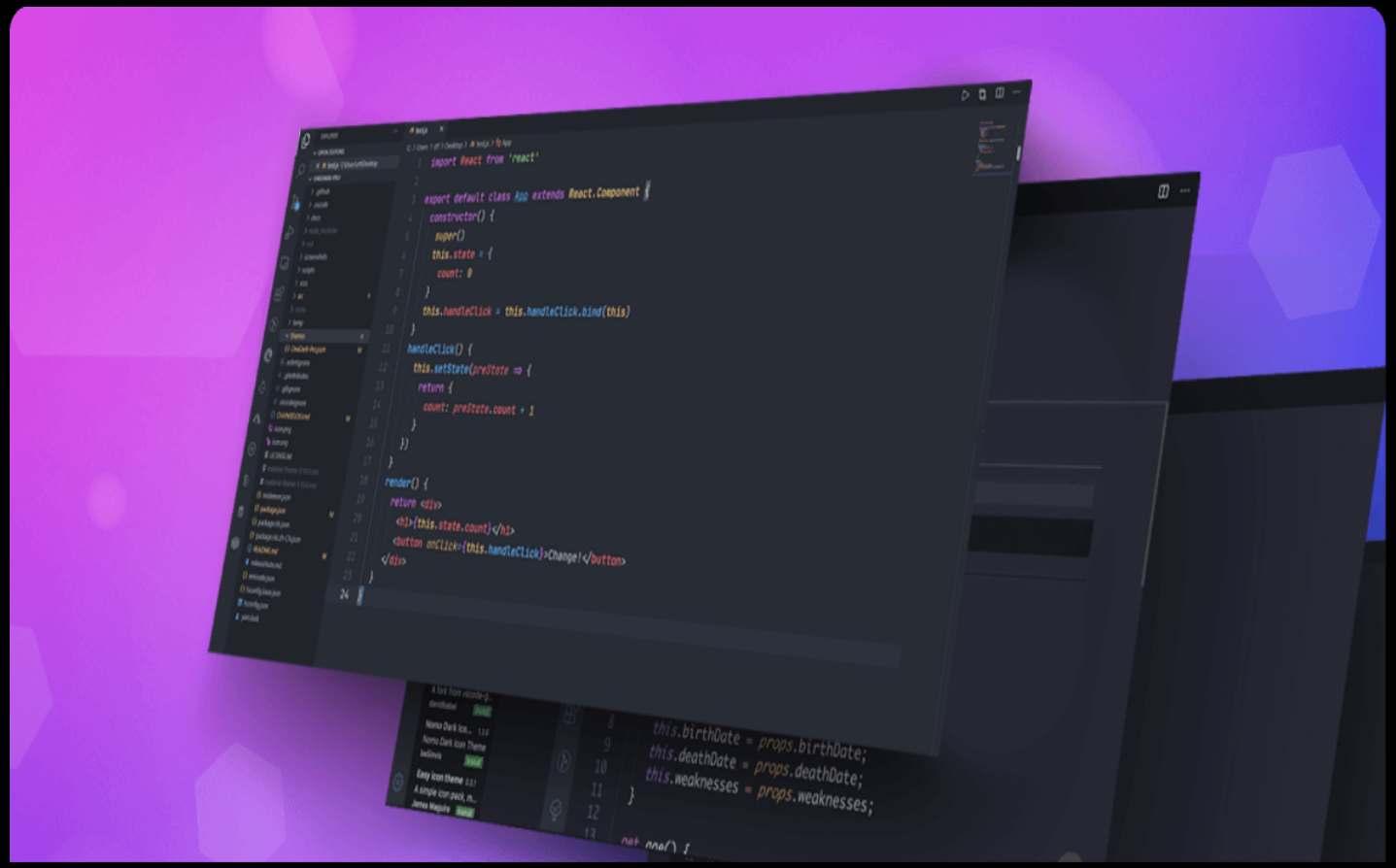
VIPER Architecture Pattern Example



- In this example, the ProductCatalogView class represents the view layer. It is responsible for displaying the product catalog to the user and allowing them to add products to their cart.
- The ProductCatalogInteractor class represents the interactor layer. It is responsible for fetching the product catalog from the repository and adding products to the cart.
- The ProductCatalogPresenter class represents the presenter layer. It is responsible for mediating between the view and the interactor. It receives user interactions from the view and passes them to the interactor. It also receives responses from the interactor and updates the view accordingly.
- The ProductRepository class represents the entity layer. It contains the data model for the product catalog.
- The ProductCatalogRouter class represents the router layer. It is responsible for navigating to the cart screen when the user clicks on the floating action button.

E-commerce app using the VIPER Pattern

Factors to Consider When Choosing an Pattern



01 Project Size

02 Team Expertise

03 Maintainability Goals

04 Specific Application Requirements