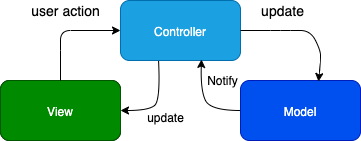
**Common Architectural Patterns**

Choosing the right architectural pattern is crucial for ensuring your app is maintainable and scalable. Here are the most common patterns used in iOS development:

**1. MVC (Model-View-Controller)**

MVC is the most traditional and widely adopted pattern in iOS development, especially in applications using UIKit.

* Model: Contains the data and business logic. It's responsible for the application's core information and rules.
* View: Represents the UI components that display the data from the model and send user actions to the controller.
* Controller: Acts as an intermediary between the view and the model, handling user input and updating the view accordingly.



Pros:

* Simple and easy to understand.
* Directly supported by UIKit, making it easy to implement with standard iOS components.

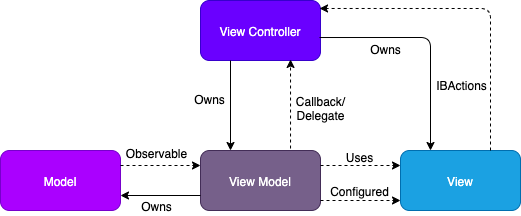
Cons:

* Tends to result in "massive view controllers" where the controller takes on too many responsibilities.
* Can be harder to maintain and test due to tight coupling between components.

**2. MVVM (Model-View-ViewModel)**

MVVM aims to solve the problems of MVC by introducing a ViewModel that handles the presentation logic.

* Model: Contains the data and business logic.
* View: Displays the data and binds to the ViewModel.
* ViewModel: Acts as a mediator between the view and the model, converting the model's data into a format that can be displayed by the view.



Pros:

* Clear separation of concerns, making the code more modular and testable.
* Reduces the responsibilities of view controllers, simplifying the UI layer.

Cons:

* Requires more boilerplate code and can introduce complexity in data binding.

**3. VIPER (View-Interactor-Presenter-Entity-Router)**

VIPER is a robust architecture that separates an application into five distinct layers, promoting a very clear separation of concerns.

* View: Displays the data and sends user actions to the presenter.
* Interactor: Contains the business logic and interacts with the data layer.
* Presenter: Prepares the data for the view and handles user actions.
* Entity: Represents the data model used by the interactor.
* Router: Manages navigation and routing between views.

Pros:

* Highly modular and testable, with each component having a single responsibility.
* Facilitates better code organization and separation of concerns.

Cons:

* Can be overkill for small projects and introduces complexity with multiple layers.

**4. Clean Architecture**

Clean Architecture focuses on creating a boundary between the application’s core business logic and its UI or external dependencies.

* Entities: Business objects that are central to the application.
* Use Cases (Interactors): Application-specific business rules, handling the core logic.
* Interface Adapters (Presenters/Controllers): Adapt data from the use cases for the UI and from the UI to the use cases.
* Frameworks & Drivers: External interfaces such as UI frameworks, databases, and network services.

Pros:

* High-level design principles ensure that business logic is decoupled from external frameworks.
* Greatly improves testability and maintainability by keeping core logic independent of the UI.

Cons:

* Complex and can require significant effort to implement and maintain, especially in smaller projects.