iOS Architecture Patterns

MVC, MVP, MVVM, VIPER, and VIP in Swift

—

Raúl Ferrer García

Apress®

iOS Architecture Patterns

MVC, MVP, MVVM, VIPER, and VIP in Swift

Raúl Ferrer García

Apress®

iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift

Raúl Ferrer García
Barcelona, Spain

ISBN-13 (pbk): 978-1-4842-9068-2 ISBN-13 (electronic): 978-1-4842-9069-9
https://doi.org/10.1007/978-1-4842-9069-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on the Github repository: https://github.com/Apress/iOS-Architecture-Patterns. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

We're here to put a dent in the universe. Otherwise, why else even be here?

—Steve Jobs

They didn't know it was impossible, so they did it.

—Mark Twain

Table of Contents

About the Author

Raúl Ferrer García holds a doctorate in chemistry, but he has always had a great interest in the world of computer science and software development, where he began his foray programming with a ZX Spectrum at the age of 14. For just over ten years, and in a self-taught way, he has entered the world of mobile development, first as an iOS Developer and then as Mobile Tech Lead at Editorial Vicens Vives, and he has dedicated himself completely to the development and management of mobile applications. He also maintains a blog in which he tries to explain everything he's learned and studied about the world of mobile development.

About the Technical Reviewer

Massimo Nardone has more than 25 years of experience in security, web and mobile development, cloud, and IT architecture. His true IT passions are security and Android. He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a CISO, CSO, security executive, IoT executive, project manager, software engineer, research engineer, chief security architect, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years. His technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, and more.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas). He is currently working for Cognizant as head of cybersecurity and CISO to help both internally and externally with clients in areas of information and cybersecurity, like strategy, planning, processes, policies, procedures, governance, awareness, and so forth. In June 2017, he became a permanent member of the ISACA Finland Board.

Massimo has reviewed more than 45 IT books for different publishing companies and is the co-author of *Pro Spring Security: Securing Spring Framework 5 and Boot 2-based Java Applications* (Apress, 2019), *Beginning EJB in Java EE 8* (Apress, 2018), *Pro JPA 2 in Java EE 8* (Apress, 2018), and *Pro Android Games* (Apress, 2015).

Acknowledgments

First of all, I would like to thank my family for their support, their words of encouragement, their inspiration… during the preparation and writing of this book and, in general, ever since. This book is dedicated to them.

Second, I would like to thank the entire Apress team for the opportunity to write this book, starting with Aaron Black who contacted me and raised the possibility of writing a book on iOS development, and not forgetting Jessica Vakili and Nirmal Selvaraj for their great work in managing and editing this book.

Learning is a journey that never ends, so I would also like to thank the work of all those who in one way or another teach us something new every day.

Finally, thanks to you, reader. I hope that once you have read this book you consider it a wise decision and that, to a greater or lesser extent, it has helped you in your evolution as a developer.

Introduction

As we will see throughout the book, various architectural patterns have been developed to apply in the development of our applications – some well known (and older), such as MVC or MVVM, and others more innovative, such as VIPER or VIP.

If you are just starting to develop applications or have been at it for some time, you have surely searched for information on how an application is built and what architecture pattern is the best to apply. But possibly you have also reached the same conclusion as me: from a global point of view, there is no perfect architecture pattern, they all have advantages and disadvantages, and it almost always depends on how we apply said pattern that our code is readable, testable, and scalable.

In addition, you will also have noticed that an architecture pattern comes to mark some kind of application rules, but that later many developers adapt or modify it looking to improve its features or solve some of its possible drawbacks.

Who Is This Book For?

This book is aimed both at those developers who are starting now and who want to know what architecture patterns they can apply to their applications, as well as those developers who have been developing applications for some time but who want to know other possible architectures to apply.

Therefore, this book is for you if what you want is

- Learn to develop applications following some of the architecture patterns explained
- Understand the advantages and disadvantages of each of the architecture patterns explained and choose the one that suits you best
- Understand the advantages of developing a readable, testable, and scalable code

In this book I have not sought to delve into the use of each of the architectures explained, but rather to serve as a point of introduction to their use, to understand why they are important, and from here on you will be able to choose one or those that suit you best, know how to delve into them, apply them, and evolve in your career as a developer.

How to Use This Book?

Apart from a theoretical introduction to each of the architectural patterns presented (there are numerous articles for each of the architectures that we will cover that talk about their features, advantages, and disadvantages), this book is eminently practical. In Chapters 2–6 (MVC, MVP, MVVM, VIPER, and VIP architecture patterns), the development of an application (MyToDos) following each of these patterns is presented.

For the sake of simplicity, although the main parts of the code are presented (depending on the concept explained), you will be able to observe omitted parts of the code (marked with "..."). However, you can find the full code for each of the projects in this book's repository.

Therefore, I am going to assume that you have some knowledge of both Swift and Xcode that should allow you to follow the course of the book without problems.

CHAPTER 1

Introduction

Assume the following situation: you and your team have received a new project to develop a mobile application. A project, whether it originates from our idea or is commissioned by a client, will present a series of specifications, functionalities, behaviors, etc.

Continuing with our assumption, we are going to consider that all these specifications and functionalities have already been studied and transformed into user stories (i.e., how a functionality would be described from the point of view of a user: for example, "As a user, I want to login in the application") and that we could already start developing the application by writing the first lines of code.

It has happened to all of us that when we have a new project, we want to start writing code. However, if we work in this way, without proposing a project structure or taking into account the type of application, we can end up developing an application that works, but whose code is later difficult to maintain.

To avoid this situation, before starting to write code we have to determine what structure we are going to give it, what the Software Architecture is going to be, and what architecture pattern is the most suitable for our project.

Before starting to see the most used architecture patterns in the development of iOS applications, we are going to make an introduction to what Software Architecture is, what architecture patterns are, why their use is necessary, and how to choose the most suitable one for our projects.

R. Ferrer García, *iOS Architecture Patterns*, https://doi.org/10.1007/978-1-4842-9069-9_1

What Is Software Architecture?

The Software Architecture defines how the software structure is, what are the components that form it, how they are joined, and how they communicate with each other.

All these points that intervene in the definition of the Software Architecture can be represented according to different models or views, the following three being the main ones (and an example of their application is one of the architectures that we will study later, the MVVM or Model–View–ViewModel):

- **Static view**: It indicates which components make up the structure. In MVVM, these components would be View, Model, and ViewModel.
- **Dynamic view**: It establishes the behavior of the different components and the communication between them over time.
- **Functional view**: It shows us what each component does. For example, in the example we are seeing, each of the components would have the following functionalities:
  - **Model**: It contains the classes and structures responsible for storing and transferring an application's data. It also includes business logic.
  - **View**: It represents the interface with the elements that form it, the interaction with the user, and how it is updated to show the user the information received.

- **ViewModel**: It acts as an intermediary between the View and the Model; it usually includes presentation logic, that is, those methods that allow the data received from the Model to be transformed to be presented in the View.

Architecture Patterns

We have just seen that Software Architecture helps us to give structure to our project. However, not all projects are the same or have the same purpose, so, logically, the architectures used are different and appropriate to each project.

Developers have been finding different solutions when facing their projects. The fact that the requirements and functionalities of a project, its components, or the way of communicating between them vary from one project to another has given rise to different architectural solutions and architecture patterns.

Keep in mind that an architecture pattern is like a template that offers us some rules or guidelines on how to develop and structure the different components of the application. Each architectural pattern has its advantages and disadvantages, as we will see later.

Why Do We Need an Architecture Pattern for Our Applications?

The selection and use of an architecture pattern when developing our applications will allow us to avoid a series of problems that applications that have been developed without using these patterns can present.

Some of these problems are as follows:

- These are applications that are difficult to maintain. This problem increases if new developers are involved, as it is more costly for them to understand the software they are working on.
- It usually increases the development time and, therefore, increases its cost.
- As it is not a structured code, it is more complicated to add new features or scale it.
- They are very likely to have duplicate, unused, and messy code. All this makes it more prone to errors.

Therefore, the use of a good architecture pattern will allow us to reduce these problems:

- We will work with a code that is simpler, more organized, and easier to understand (and following good development practices, such as the SOLID principles) for all developers since they will act according to the same rules.
- The written code will be easily testable and less prone to errors.
- A correct distribution of the components and their responsibilities will lead to a structure that is easy to maintain, modify, and extend.
- All of this will result in reduced development time and, by extension, reduced costs.

Design from High Level to Low Level

In this book, we are going to study and apply different architectural patterns in iOS applications, so we will also see the code with which we implement them and the best way to implement it.

As we have seen, an architecture pattern gives us guidelines to build an application that is efficient, scalable, and easy to maintain, among other advantages.

But we can think of an architecture pattern as the high-level design of the application, and then we have to go down to the code (Figure 1-1).

This is where the design patterns and their implementation through the programming language come in, trying to follow some principles (SOLID), which will make the code flexible, stable, maintainable, and reusable.

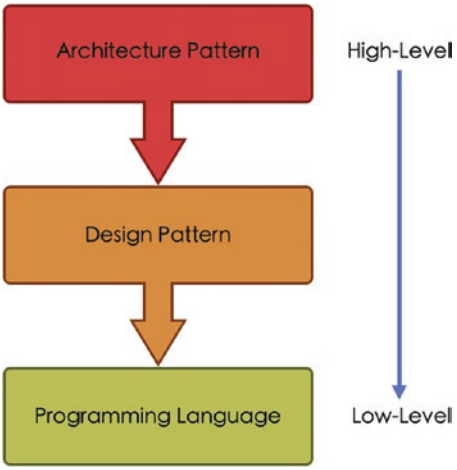

Figure 1-1. *Levels of design and implementation in an application*

Let's see briefly what design patterns are (some of which we will apply) and how to work following the SOLID principles.

Design Patterns

In the same way that an architecture pattern is a generic solution to a certain problem when it comes to the structure of our software (i.e., it will affect the entire project), design patterns give us solutions to recurring problems that affect a project component.

There are 23 design patterns, which are described in the book *Design Patterns: Elements of Reusable Object-Oriented Software*.[1] They described the 23 design patterns and classified them into three groups: structural patterns, creational patterns, and behavioral patterns.

Creational Patterns

Creational patterns are those that allow us to create objects. These patterns encapsulate the procedure for creating an object and generally work through interfaces.

Factory Method

It provides an interface that allows you to create objects in a superclass, but delegates the implementation and alteration of objects to subclasses.

Abstract Factory

It provides an interface that allows groups of related objects to be generated, but without specifying their specific classes or implementation.

[1] *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, E. Johnson, and J. Vlissides. Addison Wesley, 1st Ed, 1994.

Builder

It allows building complex objects step by step, separating the creation of the object from its structure. In this way, we can use the same construction process to obtain different types and representations of an object.

Singleton

This pattern ensures that a class has only one possible instance, which is globally accessible.

Prototype

It allows you to copy or clone an object without requiring our code to depend on its classes.

Structural Patterns

Structural patterns specify how objects and classes relate to each other to form more complex structures so that they are flexible and efficient. They rely on inheritance to define interfaces and obtain new functionality.

Adapter

It is a structural pattern that allows two objects with incompatible interfaces to collaborate, through an intermediary with which they communicate and interact.

Bridge

In this pattern, an abstraction is decoupled from its implementation, so it can evolve independently.

Composite

It allows you to create objects with a tree-like structure and then work with these structures as if they were individual objects. In this case, all the elements of the structure use the same interface.

Decorator

This pattern allows you to add new features to an object (including this object in a container that contains the new features) without changing the behavior of objects of the same type.

Façade

It provides a simplified interface to a complex structure (such as a library or set of classes).

Flyweight

It is a pattern that allows you to save RAM by having many objects share common properties on the same object, instead of maintaining these properties on every object.

Proxy

It is an object that acts as a simplified version of the original. A proxy controls access to the original object, allowing it to perform some tasks before or after accessing that object. This pattern is often used for Internet connections, device file access, etc.

Behavioral Patterns

They are the most numerous, and they focus on communication between objects and are responsible for managing algorithms, relationships, and responsibilities between these objects.

Chain of Responsibility

It allows requests to be passed through a chain of handlers. Each of these handlers can either process the request or pass it on to the next. In this way, the transmitter and the final receiver are decoupled.

Command

It transforms a request into an object that encapsulates the action and information you need to execute it.

Interpreter

It is a pattern that, given a language, defines a representation for its grammar and the mechanism for evaluating it.

Iterator

It allows to cycle through the elements of a collection without exposing its representation (list, stack, tree…).

Mediator

It restricts direct communications between objects and forces communication through a single object, which acts as a mediator.

Memento

It allows you to save and restore an object to a previous state without revealing the details of its implementation.

Observe

It allows establishing a subscription mechanism to notify different objects of the events that occur in the object that they observe.

State

It allows an object to change its behavior when its internal state changes.

Strategy

It allows defining that, from a family of algorithms, we can select one of them at execution time to perform a certain action.

Template Method

This pattern defines the skeleton of an algorithm in a superclass but allows subclasses to override some methods without changing their structure.

Visitor

It allows algorithms to be separated from the objects with which they operate.

SOLID Principles

These are five principles that will allow us to create reusable components, easy to maintain, and with higher quality code.

SOLID is an acronym that comes from the first letter of the five principles.

Single-Responsibility Principle (SRP)

A class should have one reason, and one reason only, to change. That is, a class should only have one responsibility.

Open–Closed Principle (OCP)

We must be able to extend a class without changing its behavior. This is achieved by abstraction.

Liskov Substitution Principle (LSP)

In a program, any class should be able to be replaced by one of its subclasses without affecting its operation.

Interface Segregation Principle (ISP)

It is better to have different interfaces (protocols) that are specific to each client than to have a general interface. Also, a client would not have to implement methods that it does not use.

Dependency Inversion Principle (DIP)

High-level classes should not depend on low-level classes. Both should depend on abstractions. Abstractions should not depend on details. The details should depend on the abstractions. What is sought is to reduce dependencies between modules and thus achieve less coupling between classes.

How to Choose the Right Architectural Pattern

We have just seen the advantages that our applications have a good architecture offers us. But how do we choose the right architecture pattern for our project?

In the first place, we have to know some information about our project and the technology that we are going to use since we have seen that some architecture patterns are better adapted to some projects and other patterns to others.

Therefore, we must take into account, for example:

- The type of project
- The technologies used to develop it
- Support infrastructure (servers, clouds, databases…)
- User interface (usability, content, navigation…)
- Budget and development time
- Future scalability and the addition of new functionalities

If we take into account everything seen so far, the choice of a good architecture pattern (along with the use of design patterns and SOLID principles) will allow us to have the following:

- **A scalable application**: A good architecture pattern should allow us to add new features and even change some of the technologies used, without having to modify the entire application.
- **Separation of interests**: Each component should be independent of the code point of view. That is, to function correctly, a component should only be aware of those around it and nothing else. This will allow us, for example, to reuse these components or simply change them for others.
- **A code easy to maintain**: Well-written, structured code without repetition makes it easier to understand, review, or modify. Also, any new developer joining the project will require less time to get hold of.

- **A testable code**: The previous points result in the fact that it is easier to test a code if the functionalities are correctly separated than if they are not.
- **A solid, stable, reliable, and durable code over time**.

Most Used Architecture Patterns

From a generic point of view of software development, there are numerous architecture patterns, but we will focus on the most used for the development of iOS applications:

- Model–View–Controller (MVC)
- Model–View–Presenter (MVP)
- Model–View–ViewModel (MVVM)
- View–Interactor–Presenter–Entity–Router (VIPER)
- View–Interactor–Presenter (VIP)

We will start with the best-known model and the one that every developer usually starts working with, the MVC. From here we will work on models that derive from it, such as the MVP and the MVVM, to end up with much more elaborate models of higher complexity, such as the VIPER and the VIP.

After these architectural patterns, we will see, in a more summarized way, some more patterns, perhaps not so used or known, but that can give us a better perspective of how to structure an application. Examples of these types of patterns are RIBs (developed by Uber) and Redux (based on an initial idea of Facebook for a one-way architecture).

In Search of a "Clean Architecture"

The "Clean Architecture" concept was introduced by Robert C. Martin in 2012,[2] and it is not an architecture, but a series of rules that, together with the SOLID principles, will allow us to develop software with responsibilities separate, robust, easy to understand, and testable.

Clean Architecture Layers

According to this philosophy, for an architecture to be considered "clean" it must have at least the following three layers: Domain Layer, Presentation Layer, and Data Layer (Figure 1-2).

[2] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

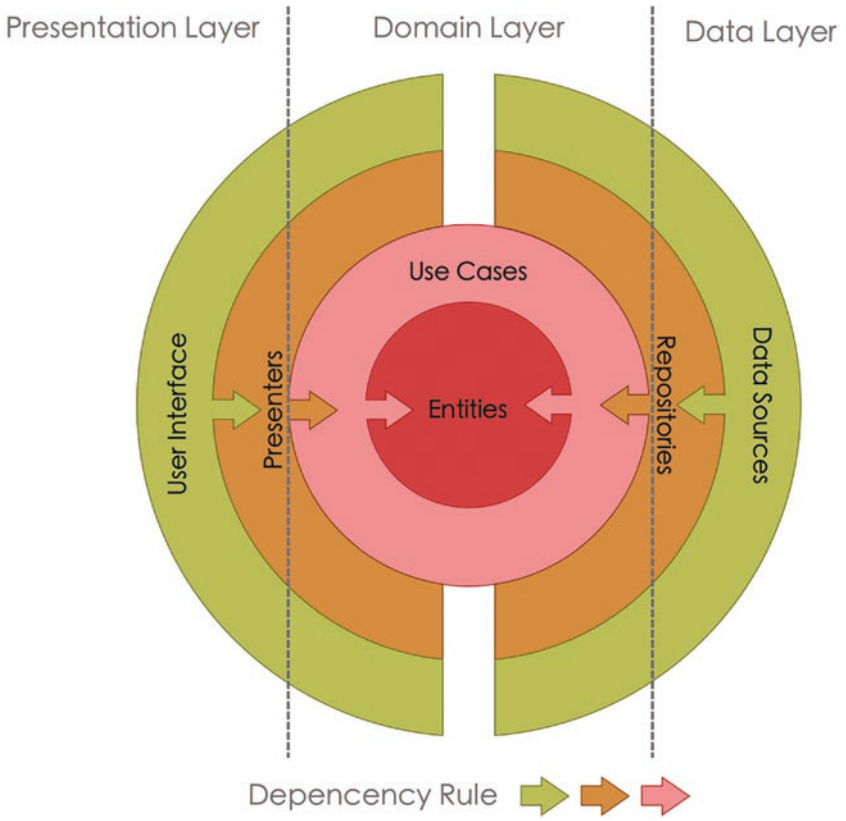

Figure 1-2. *Scheme of the layer structure in Clean Architecture. Dependency rule arrows show how the outermost layers depend on the innermost ones and not the other way around*

Domain Layer

It is the core of this architecture and contains the application logic and business logic. In this layer we find the Use Cases or Interactors, the Entities, and the Interfaces of the Repositories: