Swift Design Patterns

Reusable solutions for Swift development with practical examples

Mihir Das

www.bpbonline.com

First Edition 2024

ISBN: 978-93-55516-800

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Dedicated to

My beloved wife:

Priyanka

and

My parents

About the Author

With over 13 years of experience in iOS development, **Mihir Das** excels in both native and cross-platform mobile app development using Xamarin and Xamarin.Forms. He has successfully delivered numerous B2B and B2C applications for iOS and Android, demonstrating proficiency with Xcode, Android Studio, and Visual Studio. Mihir is skilled in Object-Oriented Techniques, MVC Architecture, and mobile automation using Appium and TestNG. He possesses advanced knowledge of RxSwift, MVVM, and SwiftUI, advocating strongly for SOLID principles and software design patterns.

Outside of work, Mihir is deeply passionate about continuous learning, often found exploring new skills in front of his computer. When he is not coding, he enjoys spending quality time with his family.

About the Reviewer

Balraj Verma is a passionate and experienced iOS developer. Ever since the first Apple SDK was launched, he has been working in the mobile development field. He is currently employed by a reputable consulting firm as a lead consultant. He has contributed to over 60 apps, some of which have millions of users, across a variety of industries, including Augmented Reality, telematics, banking, health, and Telecom. Driven by his enthusiasm for imparting knowledge and contributing to the community, he frequently contributes to Medium and Stack Overflow, where he writes about mobile development and shares his insights and experiences.

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing, especially my wife Priyanka and my parents.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book with the valuable participation and collaboration of reviewers, technical experts, and editors.

Finally, I would like to acknowledge the valuable contributions of my colleagues and co-workers during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Preface

Building modern applications is a complex task that requires a deep understanding of both the latest technologies and effective design principles. Swift and its robust ecosystem have become essential tools in the development of high-quality applications for Apple's platforms.

This book is designed to provide a comprehensive guide to mastering design patterns in Swift. It covers a broad spectrum of topics, starting with the fundamentals of Swift programming, moving through advanced concepts such as reactive programming with RxSwift, and exploring the use of design patterns to build robust, scalable, and maintainable applications. It also explores how to use Figma and Zeplin effectively.

Throughout the book, you will explore the key features of Swift and how to leverage them effectively to implement classic design patterns. You will gain insights into best practices and will be equipped with practical examples to solidify your understanding of each pattern.

This book is intended for developers who are new to Swift and want to learn how to apply design patterns in their projects. It is equally valuable for experienced developers seeking to deepen their knowledge of Swift and enhance their application design skills.

This book will help you acquire the knowledge and skills needed to become a proficient developer in crafting well-designed applications using Swift. I hope you find it informative and useful.

Chapter 1: Introduction to Swift Programming – This chapter explains the fundamentals of Swift, Apple's powerful and intuitive programming language for iOS, macOS, watchOS, and tvOS development. It explores Swift's modern syntax and features, which make it both beginner-friendly and highly efficient for experienced developers.

Chapter 2: Fundamentals of SwiftUI – In this chapter, we will explore the fundamentals of SwiftUI, Apple's innovative framework for building user interfaces across all its platforms. Delves how SwiftUI simplifies UI development with its declarative syntax, allowing for the creation of dynamic, responsive, and visually appealing interfaces. We will cover essential concepts, including views, state management, and data binding, providing you with the foundational knowledge needed to start building modern and efficient user interfaces in Swift applications.

Chapter 3: Why Design Patterns – Here, we will examine the importance of design patterns in software development, particularly within the Swift programming language. It explores how design patterns provide reusable solutions to common problems, promoting best practices and improving code maintainability.

Chapter 4: Creational Design Patterns – This chapter explores Creational Design Patterns, which focuses on the efficient and flexible creation of objects in Swift applications. It covers patterns such as Singleton, Factory, and Builder, demonstrating how they provide solutions to control the instantiation process, enhance scalability, and promote code reuse. Through detailed explanations and practical examples, this chapter equips readers with the skills to implement Creational Design Patterns effectively in their Swift projects.

Chapter 5: The Structural Patterns – In this chapter, we will explore Structural Design Patterns to understand how objects and classes are composed to form larger structures while ensuring flexibility and efficiency. It covers patterns such as Adapter, Composite, and Decorator, demonstrating how they facilitate the creation of complex and scalable systems. Through comprehensive explanations and practical examples, this chapter provides readers with the knowledge to implement Structural Design Patterns in their projects effectively.

Chapter 6: The Behavioral Patterns – This chapter covers Behavioral Design Patterns, which emphasizes the interactions and responsibilities among objects to ensure effective communication and responsibility distribution. It covers patterns such as Observer, Strategy, and Command, illustrating how they can optimize the flow of control and data within Swift applications.

Chapter 7: SOLID Principles – In this chapter, we will examine the SOLID principles, a set of five fundamental design principles aimed at creating more understandable, flexible, and maintainable software. It covers the Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle.

Chapter 8: Architecture Patterns – This chapter explores Architecture Patterns, which provide high-level structures for organizing and designing software systems. It covers patterns such as **Model-View-Controller** (**MVC**), **Model-View-ViewModel** (**MVVM**), and VIPER, demonstrating how they help manage complexity, enhance scalability, and improve code maintainability in Swift applications.

Chapter 9: Design System with Effective Use of Zeplin and Figma – This chapter focuses on creating a cohesive Design System using Zeplin and Figma, two powerful tools that streamline collaboration between designers and developers. It explores how

Design Systems unify visual style, components, and guidelines across projects, ensuring consistency and efficiency in Swift application development. Readers will learn how to leverage Zeplin for translating designs into developer-friendly specs and Figma for collaborative design creation and iteration.

Chapter 10: Reactive Programming with RxSwift – In this chapter, we will explore the fundamentals of Reactive Programming using RxSwift, a powerful framework for Swift and iOS development. It covers key concepts such as observables, observers (subscribers), operators, and schedulers, demonstrating how they enable declarative and responsive programming paradigms.

Chapter 11: Testing Code with Unit and UI Tests – This chapter explores the fundamentals of testing Swift code using Unit Tests for isolated component validation and UI Tests for automated interaction with user interfaces. Readers will learn essential XCTest practices, including writing assertions and managing test environments effectively.

Chapter 12: Anti-Patterns and Common Mistakes – This chapter highlights detrimental practices in Swift development, such as tight coupling and spaghetti code, that can impair scalability and maintainability. By recognizing and addressing these pitfalls with practical examples and alternative strategies, developers can improve code quality and foster more efficient Swift applications.

Chapter 13: Conclusion and Looking Ahead – This chapter offers a summary of essential insights into Swift development, emphasizing best practices and common pitfalls. It also explores upcoming trends and future directions in Swift, providing developers with a forward-looking perspective on evolving technologies and methodologies.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/yd14r9k

The code bundle for the book is also hosted on GitHub at
https://github.com/bpbpublications/Swift-Design-Patterns.
In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Introduction to Swift Programming

Introduction

Swift was created by *Apple* to build software for multiple platforms. Thereafter, it rapidly gained popularity to become the primary coding language for *Apple* ecosystem. With its expressive and elegant syntax, combined with safety and performance, Swift is perfect for mastering design patterns successfully.

Throughout this book, we will explore the intersection of two powerful concepts: timeless best practices blend with an advanced coding language to create an efficient app development environment. This book is a thorough handbook for mastering Swift and related design concepts.

Structure

In this chapter, we will discuss the following topics:

- Introducing Swift
- Control flow and functions
- Optionals and error handling
- Protocol and extension
- Concurrency
- Automatic Reference Count

Objectives

After studying this chapter, you will understand all the key concepts of Swift programming. We will be using these concepts throughout the book to understand various design patterns.

Introducing Swift

In 2014, Swift succeeded Objective-C, as announced by *Apple* during its launch in June of that year. Modernization was required as the current language and Objective-C no longer kept pace with the increasingly complex demands of software creation. Developers faced numerous difficulties, and therefore, Swift was created to provide a natural and effortless user experience.

Variables and constants

In Swift, data is stored and managed using variables and constants. Here is an overview.

Variables

Variables store data that can change over time. To declare a variable, use the **`var`** keyword followed by the variable name and optional type annotations.

In the following line of code, we are declaring age as var of type **`Int`**. We are also initializing it with a value of **`25`**. Here **`age`** cannot be optional, which means it will always hold some value. We cannot assign nil value to **`age`** here:

```
var age: Int = 25
```

In the following case, **`age`** is optional, which means it may or may not hold a value. **`?`** is used to denote optional type. We can assign **`nil`** value to **`age`**:

```
var age: Int? = 25
```

Constant

Once set, constants are used for storing data that does not change. By employing **`let`**, we declare constants. Following is an example of declaring constant:

```
let sex: String = "Male"
```

Data types

Swift has several basic data types, including:

- **`Int`**: Represents whole numbers (for example, **`42`**).
- **`Double`**: Represents floating-point numbers with decimal places (for example, 3.14).

- **`Bool`**: Represents Boolean values, either true or false.
- **`String`**: Represents text and character data (for example, **`Hello, World!`**).

Operators

Swift has a selection of operators that can be applied to values, including arithmetic operators (**`+`**, **`-`**, **`*`**, **`/`**), comparison operators (**`==`**, **`!=`**, **`<`**, **`>`**), and logical operators (**`&&`**, **`||`**, **`!`**). This enables developers to perform a vast array of operations. The following example shows **`+`** and **`>`** operator in action:

```
let x = 10
let y = 5
let sum = x + y  // sum is now 15
let isGreater = x > y  // isGreater is true
```

String and string interpolation

Swift provides powerful tools for working with strings, including string interpolation, which allows you to embed variables and expressions within string literals. In this example, we declare **`name`** as a **`String`** and **`age`** as an **`Int`**. Then, we interpolate **`name`** and **`age`** to declare **`greeting`**:

```
let name = "Rahul"
let age = 30
let greeting = "Hello, my name is \(name) and I am \(age) years old."
```

Collections

Collections are fundamental data structures of Swift that allow us to store, organize, and manipulate groups of values. Arrays, Dictionaries, and Sets are the three primary collection types available in Swift. Each has its own distinct qualities and applications.

Let us explore each of them in detail.

Arrays

An array is an ordered collection of values of the same type, indexed by integers. Since elements are stored in a specific order, an element can be accessed by its index. Duplicate values can be present inside an array.

Declaration and initialization

In the following code, we are declaring an array of Strings. Since we are declaring it as **`var`**, we will be able to modify it later:

```
var players = ["Sachin", "Rahul", "Ganguly"]
```

Accessing elements

In the following code, we are accessing 1st element of the players array, which is **`Sachin`**:

```
let firstPlayer = players[0]
```

Modifying arrays

In the following code, we will be modifying the players array by doing add, insert, and remove operation:

```
players.append("Robin")     // Adding an element to the end
players.insert("Yuvraj", at: 2) // Inserting an element at a specific index
players.remove(at: 1)         // Removing an element by index
players[0] = "Ramesh"           // Modifying an element
```

Iterating through arrays

We use for loop to iterate over array, as shown in the following example:

```
for player in players {
    print(player)
}
```

Dictionaries

A dictionary is an unsorted collection of key-value pairs. Each key must be distinct in a dictionary. To access value from the dictionary, we use the unique key. It can hold duplicate values with distinct keys.

Declaration and initialization

We are declaring person as a dictionary with **`String`** key type holding **`name`** and **`age`** as keys:

```
var person = ["name": "Nikhil", "age": 25]
```

Accessing elements

In the following example, we are accessing the **`name`** key from the dictionary using subscript:

```
let name = person["name"]
```

Modifying dictionaries

In the following code example, we are modifying the dictionary:

```
person["city"] = "New Delhi" // Adding a new key-value pair
person["age"] = 26          // Modifying a value by key
person.removeValue(forKey: "city") // Removing a key-value pair
```

Iterating through dictionaries

The following code example shows how we can iterate the dictionary using the **`for`** loop:

```
for (key, value) in person {
    print("\(key): \(value)")
}
```

Sets

A set is an unsorted collection of distinct values. When the order of the elements is irrelevant and you need to check for the existence of a value, sets are frequently used.

Declaration and initialization

The following code declares and initializes a **`Set`** of type **`String`**:

```
var colors: Set<String> = ["red", "green", "blue"]
```

Adding and removing elements

Following code shows adding and removing elements from **`Set`**:

```
colors.insert("yellow")       // Adding an element
colors.remove("green")        // Removing an element
```

Performing Set operations

We can do mathematical operation such as intersection, union, and so on, as shown in the following code:

```
let otherColors: Set<String> = ["blue", "orange"]
let commonColors = colors.intersection(otherColors)  // Intersection
let allColors = colors.union(otherColors)            // Union
let uniqueColors = colors.symmetricDifference(otherColors)  // Symmetric
difference
```

Checking for membership

To check if an element exists inside the **`Set`**, we can use the following code:

```
if colors.contains("red") {
    print("Red is in the set")
}
```

Type safety

The fundamental idea behind Swift's type safety is that each variable and constant must have a unique data type. The use of different types of data is strictly regulated by the Swift compiler. This safeguards against data mixing or misuse by accident, and shields programs from a variety of common programming mistakes.

Type annotations

In Swift, we have the choice to explicitly specify the data type of a variable or constant by utilizing a type annotation. Although not always required due to Swift's exceptional type inference, incorporating Type Annotations can enhance code clarity and self-explanatory nature.

For example, we explicitly declare **age** as **Int** in the following code:

```
var age: Int
age = 30
```

Type annotations serve a useful purpose by clarifying the expected data type for variables or constants. However, in Swift, the language itself often deduces the appropriate type through inference, eliminating the need for explicit annotations. This happens when the compiler can determine the data type based on its initial value.

Type inference

Swift's type inference system effortlessly determines the data type of a variable or constant based on its context and initial value. This remarkable feature minimizes the need for explicit type annotations, leading to cleaner and more concise code. Moreover, it plays a crucial role in identifying type related errors during compilation, therefore enhancing the reliability of the code.

In the following code, Swift infers the **name** is of type **String** and the **score** is of type **Int**:

```
let name = "Rohit" //Swift infers that name is of type String
let score = 95   // Swift infers that score is of type Int
```

Type safety with functions

Swift ensures type safety not only for variables and constants but also for function parameters and return types. When defining functions, users are required to specify the data types of their parameters and return values. This promotes consistency and predictability when invoking functions. Following example defines a function, when invoking the function swift compiler checks function parameters and return type as **Int**:

```
func add(x: Int, y: Int) -> Int {
    return x + y
}

let result = add(x: 5, y: 3)  // The compiler checks that both x and y are Int
```

Optionals and type safety

Swift's Optionals play a crucial role in type safety by allowing you to handle the absence of values explicitly. Optionals indicate that a variable might have a value or might be nil