VAMSHI KRISHNA

DESIGN PATTERNS IN SWIFT

A DIFFERENT APPROACH TO CODING WITH SWIFT

FV PRESS

Contents

Design Patterns in Swift

Vamshi Krishna

(Non)Fiction Vortex™

Join the Story with
our groundbreaking mobile app,
StoryShop.
We are redefining digital narrative.

Design Patterns in Swift
Vamshi Krishna

ISBN: 978-1-947655-16-4
Published by Fiction Vortex, Inc. (FV Press)
Nampa, ID 83651
http://www.fictionvortex.com

Cover by Tom Patchin

Published in the United States of America

Design Patterns in Swift

Vamshi Krishna

Preface

Design Patterns - I came across this term for the first time in my life in a job interview (mind you, I was almost two years into iOS Development then). I got back home and googled about them. It was embarrassing and interesting at the same time, as I found out that I had been using a few of those patterns without actually realising that I was doing so.

I forgot about that incident after several days and got back to my work. Fast forward two years, I was back at job searching and this time I decided to learn in depth about design patterns. As a seasoned iOS Developer, I was initially looking to learn the concepts through examples coded in Swift language.

Surprisingly, I could not find a single book or blog discussing design patterns in detail in the Swift language. There are tons of books and blogs discussing them in Java, C#, PHP, etc.

For someone who codes in Swift, it only takes a little effort to understand Java. So, I started learning the design patterns from different sources available on the Internet through Java examples. For practice, I started putting up a few examples in Swift for each of the design patterns.

Personally, cricket is something that I understand in and out. I can almost relate anything under the sun to a situation in cricket (okay, that’s a bit of an exaggeration).

So, I decided, instead of using different contexts for each of the design pattern examples, I would be using cricketing terms for all the examples I would be coding. I believe cricket is a very simple game, and even for those who do not follow the game, it should not be a big effort to relate to the cricketing terms.

That’s when I decided instead of just letting the code reside on my Mac, I would put a little more effort to take it to book form. That’s how this book was born, and I am sure your understanding on design patterns will be enhanced by the time you finish reading this book.

I would suggest you code the examples (not copy-paste, but type each and every line of the code) in your Xcode playground and see the results for yourself. Then imagine a scenario where you would apply such a design pattern, and code an example for yourself.

I believe that’s how coding is learned.

Happy learning.

Introduction

Wikipedia says, “In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design”.

In a general sense, design patterns can be stated as best practices that were implemented on a repetitive basis to solve similar problems, but that are found in different contexts.

Design patterns are not finished designs that can be directly transformed into code. But the templates can help as a bridge between the levels of a programming paradigm and concrete algorithm. These templates can also be used to solve a specific type of problem that can occur in different programming situations.

The popularity of design patterns came about after being formalised in the book ***Design Patterns: Elements of Reusable Object-Oriented Software*** by the so-called Gang of Four.

Design patterns are very popular in Java and C#, but they can be applied to all object oriented languages. They are universally relevant because we are living in a world where object oriented paradigms are used on a daily basis. Object oriented design patterns mainly shows the interactions and relationships between classes or objects.

Interestingly, most developers have been using design patterns (at least a few of them) for many years without actually realising that they are doing so.

Design patterns on a broad level can be divided into four types, namely:

1. SOLID Design Principles
2. Creational
3. Structural
4. Behavioural

SOLID design principles, introduced by Robert C. Martin, are relevant across all of the remaining three design patterns. SOLID is an acronym for the set of five design principles intended to make software designs more understandable, flexible, and maintainable. It’s better to be aware of them before moving on to the other patterns.

Prerequisites:

1. Basic understanding of Swift
2. Good understanding of object oriented programming

Contents:

Part One: SOLID

1) SOLID - Single Responsibility Principle (SRP)

Definition:

Single responsibility principle says a class should have one, and only one, reason to change. Every class should be responsible for a single part of the functionality, and that responsibility should be entirely encapsulated by the class. This makes your software easier to implement and prevents unexpected side-effects of future changes.

Usage:

Let us design an imaginary operation system for a cricket tournament. For the sake of simplicity, let’s have two major operations: 1)A TeamRegister class, which helps for checking in and checking out cricketers. 2)A TeamConveyance class, which is used to drop the players from hotel to stadium and to pick them up from the stadium after the match is over.

Every class is assigned its own responsibility and they will be responsible only for that action.

```
import UIKit
import Foundation

class TeamRegister : CustomStringConvertible{

    var teamMembers = [String]()
    var memberCount = 0

    func checkInGuest (_ name : String) -> Int{
        memberCount += 1
        teamMembers.append("\(memberCount) - \(name)")
        return memberCount - 1

    }

    func checkOutGuest (_ index : Int) {
        teamMembers.remove(at: index)
    }

    var description: String{
        return teamMembers.joined(separator: "\n")
    }
}
```

TeamRegister class conforms to CustomStringConvertible. It has two variables defined, an array named teamMembers of type String and memberCount of type Integer.

We also define two methods. checkInGuest method takes the guest name as a parameter of type String and appends the guest to teamMembers array and returns array count.

checkOutGuest takes index of type Integer as a parameter and removes the guest from register.

```
class TeamConveyance {

    func takePlayersToStadium(_ teamRegister : TeamRegister){
        print("Taking players \n \(teamRegister.description) \n to the Stadium")
    }

    func dropPlayersBackAtHotel(){
        print("Dropping all the players back at Hotel")
    }
}
```

TeamConveyance class has two major responsibilities. takePlayersToStadium takes a parameter of type TeamRegister and drops all the players at the stadium.

dropPlayersBackAtHotel gets back all the players to the hotel after the match is over. It is not concerned about anything else.

Let us now write a function called main and see the code in action.

```
func main(){
    let teamRegister = TeamRegister()
    let player1 = teamRegister.checkInGuest("PlayerOne")
    let player2 = teamRegister.checkInGuest("PlayerTwo")

    print(teamRegister)
}

main()
```

We take an instance of TeamRegister class and check in a couple of guests passing their names as parameters.

<u>Output in the Xcode console:</u>

```
1 - PlayerOne
2 - PlayerTwo
```

Let us now check out a guest and add one more guest to the team. Change the main function to:

```
func main(){
    let teamRegister = TeamRegister()
    let player1 = teamRegister.checkInGuest("PlayerOne")
    let player2 = teamRegister.checkInGuest("PlayerTwo")

    print(teamRegister)

    teamRegister.checkOutGuest(1)
    print("------------------------------------")
    print(teamRegister)

    let player3 = teamRegister.checkInGuest("PlayerThree")
    print("-----------------------------------")
    print(teamRegister)
}

main()
```

We checked out ‘PlayerTwo’ and then checked in another guest named ‘PlayerThree’.

<u>Output in the Xcode console:</u>

```
1 - PlayerOne
2 - PlayerTwo
-----------------------------------
1 - PlayerOne
-----------------------------------
1 - PlayerOne
3 - PlayerThree
```

Now change the main method to the following:

```
func main(){
    let teamRegister = TeamRegister()
    let player1 = teamRegister.checkInGuest("PlayerOne")
    let player2 = teamRegister.checkInGuest("PlayerTwo")

    print(teamRegister)

    teamRegister.checkOutGuest(1)
    print("-----------------------------------")
    print(teamRegister)

    let player3 = teamRegister.checkInGuest("PlayerThree")

    print("-----------------------------------")
    print(teamRegister)

    print("-----------------------------------")
    let teamBus = TeamConveyance()
    teamBus.takePlayersToStadium(teamRegister)

    print("-------Match Over ----------")
    teamBus.dropPlayersBackAtHotel()
}

main()
```

We are taking an instance of TeamConveyance to drop players at the stadium and get them back to the hotel after the match is over.

<u>Output in the Xcode console:</u>

```
1 - PlayerOne
2 - PlayerTwo
-----------------------------------
1 - PlayerOne
-----------------------------------
1 - PlayerOne
3 - PlayerThree
-----------------------------------
Taking players
1 - PlayerOne
3 - PlayerThree
 to the Stadium
-------Match Over ----------
Dropping all the players back at Hotel
```

2) SOLID - Open Closed Principle (OCP)

Definition:

Open closed principle says one should be able to extend a class behaviour without modifying it. This principle is the foundation for building code that is maintainable and reusable.

Any class following OCP should fulfill two criteria:

1. 1) Open for extension: This ensures that the class behaviour can be extended. In a real world scenario, requirements keep changing, and in order for us to be able to accommodate those changes, classes should be open for extension so that they can behave in a new way.

1. 2) Closed for modification: Code inside the class is written in such a way that no one is allowed to modify the existing code under any circumstances.

Usage:

Let us consider an example where we have an array of cricketers' profiles, where each entity has the name of a cricketer, his team, and his specialisation as the attributes. Now we want to build a system where the client can apply filters on the data based on different criteria like team, role of the player, etc. Let us see how we can use OCP to build this:

```
import Foundation
import UIKit
```

```
enum Team{
    case india
    case australia
    case pakistan
    case england
}

enum Role{
    case batsman
    case bowler
    case allrounder
}
```

Enumeration is a data type that allows us to define a list of possible values. We define enums for the available names of the teams and roles of the cricketers.

```
class Cricketer{

    var name:String
    var team:Team
    var role:Role

    init(_ name:String, _ team:Team, _ role:Role) {
        self.name = name
        self.team = team
        self.role = role
    }

}
```

We then define a class called Cricketer, which takes three parameters during its initialisation: name of type String, team of type Team, and role of type Role.

Now, assume one of the client requirements is to provide a filter of cricketers based on their team.

```
class CricketerFilter{
    func filterByTeam(_ cricketers:[Cricketer], _ team:Team) -> [Cricketer]{
        var filteredResults = [Cricketer]()
        for item in cricketers{
            if item.team == team{
                filteredResults.append(item)
            }
        }
        return filteredResults
    }

}
```

We write a class called CricketerFilter and define a method filterByTeam to filter the player profiles based on their team. It takes an array of type Cricketer and team of type Team as parameters and returns a filtered array of type Cricketer.

For each cricketer in the given array, we check if his team is the same as that of the given team for the filter, then add him to the filtered array. Let us see this code in action. Add the below code after CricketerFilter class.

```
func main(){
    let dhoni = Cricketer("Dhoni", .india, .batsman)
    let kohli = Cricketer("Kohli",  .india, .batsman)
    let maxi = Cricketer("Maxwell", .australia, .allrounder)
    let smith = Cricketer("Smith", .australia, .batsman)
    let symo = Cricketer("Symonds", .australia, .allrounder)
    let broad = Cricketer("Broad", .england, .bowler)
    let ali  = Cricketer("Ali", .pakistan, .batsman)
    let stokes = Cricketer("Stokes", .england, .allrounder)

    let cricketers = [dhoni, kohli, maxi, broad, ali, stokes ,smith, symo]
    print(" Indian Cricketers")
    let cricketerFilter = CricketerFilter()
    for item in cricketerFilter.filterByTeam(cricketers, .india){
        print(" \(item.name) belongs to Indian Team")
    }
}

main()
```

Output in the Xcode console:

```
 Indian Cricketers
 Dhoni belongs to Indian Team
 Kohli belongs to Indian Team
```

Assume, after a few days, we got a new requirement to be able to filter by role of the cricketer and then to filter by both team and role at once. Our CricketerFilter class would look something like this:

```
class CricketerFilter{

    func filterByRole(_ cricketers:[Cricketer], _ role:Role) -> [Cricketer]{
        var filteredResults = [Cricketer]()
        for item in cricketers{
            if item.role == role{
                filteredResults.append(item)
            }
        }
        return filteredResults
    }

    func filterByTeam(_ cricketers:[Cricketer], _ team:Team) -> [Cricketer]{
        var filteredResults = [Cricketer]()
        for item in cricketers{
            if item.team == team{
                filteredResults.append(item)
            }
        }
        return filteredResults
    }

    func filterByRoleAndTeam(_ cricketers:[Cricketer], _ role:Role, _
team:Team) -> [Cricketer]{
        var filteredResults = [Cricketer]()
        for item in cricketers{
            if item.role == role && item.team == team{
                filteredResults.append(item)
            }
        }
        return filteredResults
    }

}
```

This logic is quite similar to filterByTeam method, except with filterByRole, we check if the player’s role is the same as that of the given role. For filterByRoleAndTeam method, we use **AND** statement to check if the given condition is met.

The OCP states that classes should be closed for modification and open for extension. But, here we are clearly breaking this principle. Let us see how the same use case can be served with the help of OCP.

```
//Conditions
protocol Condition{
    associatedtype T
    func isConditionMet(_ item: T) -> Bool
}
```

We begin by defining a protocol called Condition, which basically checks if a particular item satisfies some criteria. We have a function called isConditionMet, which takes an item of generic type T and returns a boolean indicating whether the item meets the given criteria.

```
protocol Filter
{
    associatedtype T
    func filter<Cond: Condition>(_ items: [T], _ cond: Cond) -> [T]
    where Cond.T == T;
}
```

We then define a protocol named Filter that has a function called filter, which takes an array of items of generic type T and a condition of type Condition as parameters and returns the filtered array.

We now use the above generic type Filter to write conditions for role and team.

```
class RoleCondition : Condition
{
    typealias T = Cricketer
    let role: Role
    init(_ role: Role)
    {
        self.role = role
    }

    func isConditionMet(_ item: Cricketer) -> Bool {
        return item.role == role
    }
}

class TeamCondition : Condition
{
    typealias T = Cricketer
    let team: Team
    init(_ team: Team)
    {
        self.team = team
    }

    func isConditionMet(_ item: Cricketer) -> Bool {
        return item.team == team
    }
}
```

In each of the methods, we write the logic of isConditionMet protocol method to see if the item meets the criteria and returns a boolean.

```
class OCPCricketFilter : Filter
{
    typealias T = Cricketer

    func filter<Cond: Condition>(_ items: [Cricketer], _ cond: Cond)
        -> [T] where Cond.T == T
    {
        var filteredItems = [Cricketer]()
        for i in items
        {
            if cond.isConditionMet(i)
            {
                filteredItems.append(i)
            }
        }
        return filteredItems
    }
}
```

Now we define a brand new filter called OCPCricketFilter, usage of which does not violate OCP. We take items of type Cricketer, check for the condition of type Condition, and return the filtered array.

Let us now see the code in action. Change the main method to the following:

```
func main(){
    let dhoni = Cricketer("Dhoni", .india, .batsman)
    let kohli = Cricketer("Kohli",  .india, .batsman)
    let maxi = Cricketer("Maxwell", .australia, .allrounder)
    let smith = Cricketer("Smith", .australia, .batsman)
    let symo = Cricketer("Symonds", .australia, .allrounder)
    let broad = Cricketer("Broad", .england, .bowler)
    let ali  = Cricketer("Ali", .pakistan, .batsman)
    let stokes = Cricketer("Stokes", .england, .allrounder)

    let cricketers = [dhoni, kohli, maxi, broad, ali, stokes ,smith, symo]

    let ocpFilter = OCPCricketFilter()

    print(" England Cricketers")
    for item in ocpFilter.filter(cricketers, TeamCondition(.england)){
        print(" \(item.name) belongs to English Team" )
    }
}
```

We take an instance of OCPFilter and just pass the team name parameter to

TeamCondition.

Output in the Xcode console:

```
 England Cricketers
 Broad belongs to English Team
 Stokes belongs to English Team
```

In a similar way, without modifying any existing classes, we can extend the OCPCricketFilter class to as many filters as we need. Now we will see how we can write a filter for AND condition (role and team for example):

```
class AndCondition<T,
    CondA: Condition,
    CondB: Condition> : Condition
    where T == CondA.T, T == CondB.T
{

    let first: CondA
    let second: CondB
    init(_ first: CondA, _ second: CondB)
    {
        self.first = first
        self.second = second
    }

    func isConditionMet(_ item: T) -> Bool {
        return first.isConditionMet(item) && second.isConditionMet(item)
    }
}
```

This is very much similar to other filters. The only change is that it takes two conditions as arguments for its initialisation.

Change the main method to the below code:

```
func main(){
    let dhoni = Cricketer("Dhoni", .india, .batsman)
    let kohli = Cricketer("Kohli",  .india, .batsman)
    let maxi = Cricketer("Maxwell", .australia, .allrounder)
    let smith = Cricketer("Smith", .australia, .batsman)
    let symo = Cricketer("Symonds", .australia, .allrounder)
    let broad = Cricketer("Broad", .england, .bowler)
    let ali  = Cricketer("Ali", .pakistan, .batsman)
    let stokes = Cricketer("Stokes", .england, .allrounder)

    let cricketers = [dhoni, kohli, maxi, broad, ali, stokes ,smith, symo]

    let ocpFilter = OCPCricketFilter()

    print(" Australian Allrounders")

    for item in ocpFilter.filter(cricketers, 
AndCondition(TeamCondition(.australia), RoleCondition(.allrounder))){
        print(" \(item.name) belongs to Australia Team and is an Allrounder" )
    }
}
```

Output in the Xcode console:

```
 Australian Allrounders
 Maxwell belongs to Australia Team and is an Allrounder
 Symonds belongs to Australia Team and is an Allrounder
```

We can write n number of filters without modifying any existing classes. All we have to do is extend the Filter class.

3) SOLID - Liskov Substitution Principle (LSP)

Definition:

Liskov substitution principle, named after Barbara Liskov, states that one should always be able to substitute a base type for a subtype. LSP is a way of ensuring that inheritance is used correctly. If a module is using a base class, then the reference to the base class can be replaced with a derived class without affecting the functionality of the module.

Usage:

Let us understand LSP's usage with a simple example.

```
import UIKit
import Foundation

protocol Cricketer {
    func canBat()
    func canBowl()
    func canField()
}
```

We define a protocol called Cricketer, which implements three methods of canBat, canBowl, and canField.

```
class AllRounder : Cricketer{
    func canBat() {
        print("I can bat")
    }
    func canBowl() {
        print("I can bowl")
    }

    func canField() {
        print("I can field")
    }
}
```

We then define a class called AllRounder, conforming to Cricketer protocol. An all-rounder in cricket is someone who can bat, bowl, and field.

```
class Batsman : Cricketer{
    func canBat() {
        print("I can bat")
    }

    func canBowl() {
        print("I cannot bowl")
    }

    func canField() {
        print("I can field")
    }
}
```

We then define a class called Batsman, conforming to Cricketer protocol. This is a violation of LSP, as a batsman is a cricketer but cannot use Cricketer protocol because he cannot bowl. Let us now see how we can use LSP in this scenario:

```
protocol Cricketer {
    func canBat()
    func canField()
}

class Batsman : Cricketer{
    func canBat() {
        print("I can bat")
    }
    func canField() {
        print("I can field")
    }
}
```

We change the Cricketer protocol and now make the Batsman class conform to Cricketer protocol.

```
class BatsmanWhoCanBowl : Cricketer{

    func canBat() {
        print("I can bat")
    }

    func canField() {
        print("I can field")
    }

    func canBowl() {
        print("I can bowl")
    }

}

class AllRounder : BatsmanWhoCanBowl{

}
```

We then define a new class named BatsmanWhoCanBowl with super class as Cricketer and define the extra method of canBowl in this class.

4) SOLID - Interface Segregation Principle (ISP)

Definition:

The only motto of Interface segregation principle is that the clients should not be forced to implement interfaces they don't use. Clients should not have the dependency on the interfaces that they do not use.

Usage:

Let us assume we are building a screen display for mobile, tablet, and desktop interfaces of an app that is used to display live scores of a cricket match.

We will see how this can be achieved without using ISP and then using ISP.

```
import UIKit
import Foundation

// Before ISP
protocol MatchSummaryDisplay{
    func showLiveScore()
    func showCommentary()
    func showLiveTwitterFeed()
    func showSmartStats()
}
```

We define a protocol named MatchSummaryDisplay, which has four methods to show live score, commentary, twitter feed about the match, and statistics of the players.

```
enum NoScreenEstate : Error
```

```
{
    case doesNotShowLiveTwitterFeed
    case doesNotShowSmartStats
}

extension NoScreenEstate: LocalizedError {
    public var errorDescription: String? {
        switch self {
        case .doesNotShowLiveTwitterFeed:
            return NSLocalizedString("No Screen Estate to show Live Twitter Feed",
comment: "Error")
        case .doesNotShowSmartStats:
            return NSLocalizedString("No Screen Estate to show Smart Stats",
comment: "Error")
        }
    }
}
```

By default, we want to show the live score and commentary on all types of devices like mobile, tablet, and desktop. Showing twitter feed and statistics are optional, depending on the screen estate available on the device. So, we define an enum called NoScreenEstate with two possible cases. We also write an extension to it just to make the error descriptions more clear.

```
class DesktopDisplay:MatchSummaryDisplay{
    func showLiveScore() {
        print("Showing Live Score On Desktop")
    }

    func showCommentary() {
        print("Showing Commentary On Desktop")
    }

    func showLiveTwitterFeed() {
        print("Showing Live Twitter Feed On Desktop")
    }

    func showSmartStats() {
        print("Showing Smart Stats On Desktop")
    }
}
```

We start the interface design by defining a class called DesktopDisplay conforming to MatchSummaryDisplay. A desktop has enough screen space available, and we show all the available data to the user.

```
class TabletDisplay:MatchSummaryDisplay{
    func showLiveScore() {
        print("Showing Live Score On Tablet")
    }

    func showCommentary() {
        print("Showing Commentary On Tablet")
    }

    func showLiveTwitterFeed() {
        print("Showing Live Twitter Feed On Tablet")
    }

    func showSmartStats() {
        do{
            let error: Error = NoScreenEstate.doesNotShowSmartStats
            print(error.localizedDescription)
            throw error
        } catch{

        }
    }
}
```

We then define another class called TabletDisplay conforming to MatchSummaryDisplay. As the screen size of a tablet is less when compared to a desktop, we do not show smart stats on the tablet display. We throw an error in showSmartStats method.

```
class MobileDisplay:MatchSummaryDisplay{
    func showLiveScore() {
        print("Showing Live Score On Mobile")
    }

    func showCommentary() {
        print("Showing Commentary On Mobile")
    }

    func showLiveTwitterFeed() {
        do{
            let error: Error = NoScreenEstate.doesNotShowLiveTwitterFeed
            print(error.localizedDescription)
            throw error
        } catch{

        }
    }

    func showSmartStats() {
        do{
            let error: Error = NoScreenEstate.doesNotShowSmartStats
            print(error.localizedDescription)
            throw error
        } catch{

        }
    }
}
```

We then define another class called MobileDisplay conforming to MatchSummaryDisplay. As the screen size of mobile is even smaller when compared to desktop and tablet, we do not show smart stats and twitter feed on mobile display. We throw an error in showLiveTwitterFeed and showSmartStats methods.

As you can see, this approach violates ISP because TabletDisplay and MobileDisplay are forced to implement methods they are not using. Let's see how we can use ISP in this scenario.

```
//Following ISP
```

```
protocol LiveScoreDisplay{
    func showLiveScore()
    func showCommentary()
}

protocol TwitterFeedDisplay{
    func showLiveTwitterFeed()
}

protocol SmartStatsDisplay{
    func showSmartStats()
}
```

Here we define a protocol named LiveScoreDisplay, which is mandatory for all the screen sizes of the devices. Then we define different protocols called TwitterFeedDisplay and SmartStatsDisplay so that only the devices with enough screen sizes can conform to required protocols.

```
class ISPMobileDisplay:LiveScoreDisplay{
    func showLiveScore() {
        print("Showing Live Score On Mobile")
    }

    func showCommentary() {
        print("Showing Commentary On Mobile")
    }
}
```

We define a class called ISPMobileDisplay, which conforms only to LiveScoreDisplay. We don’t have to force the class to implement any unwanted methods.

```
class ISPTabletDisplay:LiveScoreDisplay, TwitterFeedDisplay{

    func showLiveScore() {
        print("Showing Live Score On Tablet")
    }

    func showCommentary() {
        print("Showing Commentary On Tablet")
    }

    func showLiveTwitterFeed() {
        print("Showing Live Twitter Feed On Tablet")
    }

}
```

We then define a class called ISPTabletDisplay, which conforms to TwitterFeedDisplay along with LiveScoreDisplay.

We can define desktop interface as follows:

```
class ISPDesktopDisplay:LiveScoreDisplay, TwitterFeedDisplay,
SmartStatsDisplay{

    func showLiveScore() {
        print("Showing Live Score On Desktop")
    }

    func showCommentary() {
        print("Showing Commentary On Desktop")
    }

    func showLiveTwitterFeed() {
        print("Showing Live Twitter Feed On Desktop")
    }

    func showSmartStats() {
        print("Showing Smart Stats On Desktop")
    }
}
```

We can observe that, in all the above three classes, we are not forcing any class to implement a method that they do not use. We achieved ISP by defining multiple protocols.

5) SOLID - Dependency Inversion Principle (DIP)

Definition:

In short, Dependency inversion principle says to depend on abstractions, not on concretions. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions. By depending on higher-level abstractions, we can easily change one instance with another instance in order to change the behaviour. DIP increases the reusability and flexibility of our code.

Usage:

Let us assume we are designing a small system where we want to list from the database all the wickets taken by a bowler in his cricketing career.

```
import Foundation
import UIKit

enum WicketsColumn{
    case wicketTakenBy
    case wicketGivenTo
}

class Cricketer{
    var name = ""

    init(_ name:String){
        self.name = name
    }

}
```

We define an enum called WicketsColumn with a list of two possible cases. We then define a class called Cricketer that takes the parameter of name of type String for its initialisation.

```
protocol WicketsTallyBrowser{
    func returnAllWicketsTakenByBowler(_ name:String) -> [Cricketer]
}
```

We define a protocol named WicketsTallyBrowser, which has a function to return all the wickets taken by a given bowler as an array of type Cricketer.

We will now define a class, which stores relationship between bowlers and batsmen.

```
class WicketsTally : WicketsTallyBrowser { //Low Level
     var wickets = [(Cricketer, WicketsColumn, Cricketer)]()

    func addToTally(_ bowler : Cricketer,_ batsman : Cricketer){
        wickets.append((bowler, .wicketTakenBy, batsman))
        wickets.append((batsman, .wicketGivenTo, bowler))
    }

    func returnAllWicketsTakenByBowler(_ name: String) -> [Cricketer] {
        return wickets.filter({$0.name == name && $1 ==
WicketsColumn.wicketTakenBy && $2 != nil})
            .map({$2})
    }

}
```

We define a class called WicketsTally conforming to WicketsTallyBrowser protocol. It has a variable called wickets, which is an array of tuples where each of the tuples has three attributes: one each of type Cricketer, WicketsColumn, and Cricketer, in that order.

Then we define a method called addToTally, which takes parameters of bowler and batsman of type Cricketer. It appends the same to the wickets array but with different relationships available from WicketsColumn enum.

In the definition of protocol method returnAllWicketsTakenByBowler, we filter the wickets array by comparing first attribute of tuple to the name of the given bowler.

```
class PlayerStats{ //High Level
    init(_ wicketsTally : WicketsTally){
        let wickets = wicketsTally.wickets
        for w in wickets where w.0.name == "BrettLee" && w.1 ==
.wicketTakenBy{
            print("Brett Lee has a wicket of \(w.2.name)")
        }
    }
}
```

We now define a class called PlayerStats, where we use the logic written in WicketsTally class to return all the wickets taken by a particular bowler.

Let us now write a main method to see this code in action.

```
func main(){
    let bowler = Cricketer("BrettLee")
    let batsman1 = Cricketer("Sachin")
    let batsman2 = Cricketer("Dhoni")
    let batsman3 = Cricketer("Dravid")

    let wicketsTally = WicketsTally()
    wicketsTally.addToTally(bowler, batsman1)
    wicketsTally.addToTally(bowler, batsman2)
    wicketsTally.addToTally(bowler, batsman3)

    let _ = PlayerStats(wicketsTally)

}
```

Output in the Xcode console:

```
Brett Lee has a wicket of Sachin
Brett Lee has a wicket of Dhoni
Brett Lee has a wicket of Dravid
```

The issue with the above approach is its violation of DIP (it states that the high-level modules should not directly depend on low-level modules), as our PlayerStats class depends upon wickets array of WicketsTally class. It should be declared as a private variable so that no other class can manipulate the data directly.

Let us now change the WicketsTally class this way:

```
class WicketsTally : WicketsTallyBrowser { //Low Level
     private var wickets = [(Cricketer, WicketsColumn, Cricketer)]()

    func addToTally(_ bowler : Cricketer,_ batsman : Cricketer){
        wickets.append((bowler, .wicketTakenBy, batsman))
        wickets.append((batsman, .wicketGivenTo, bowler))
    }

    func returnAllWicketsTakenByBowler(_ name: String) -> [Cricketer] {
        return wickets.filter({$0.name == name && $1 ==
WicketsColumn.wicketTakenBy && $2 != nil})
            .map({$2})
    }

}
```

Now change the PlayerStats class to:

```
class PlayerStats{ //High Level
    init(_ browser : WicketsTallyBrowser){
        for w in browser.returnAllWicketsTakenByBowler("BrettLee"){
            print("Brett Lee has a wicket of \(w.name)")
        }
    }
}
```

Here we can observe that, instead of directly depending on wickets array from WicketsTally, PlayerStats is dependent on abstraction from WicketsTallyBrowser. Output in the Xcode console remains the same but we are now adhering to DIP.

Output in the Xcode console:

```
Brett Lee has a wicket of Sachin
Brett Lee has a wicket of Dhoni
Brett Lee has a wicket of Dravid
```

Part Two: Creational

6) Creational - Factory Design Pattern

Definition:

Factory design pattern is also known as Virtual Constructor. It is a creational design pattern that defines an abstract class for creating objects in super class but allows the subclasses to decide which class to instantiate.

Usage:

Assume there is a BowlingMachine that delivers Red Cricket Balls (used for Test Cricket) and White Cricket Balls (used for Limited Overs Cricket) based on user input.

```
import UIKit

protocol CricketBall{
    func hitMe()
}
```

Any class conforming to CricketBall must implement hitMe method.

```
class RedBall : CricketBall{
    func hitMe() {
        print("This ball is good for Test Cricket")
    }
}

class WhiteBall : CricketBall{
    func hitMe() {
        print("This ball is good for Limited Overs Cricket")
    }
}
```

Let us start defining factories now.

```
protocol CricketBallFactory{

    init()
    func deliverTheBall (_ speed : Int) -> CricketBall
}
```

Factories conforming to CricketBallFactory must implement deliverTheBall. We should also give some input like the speed at which we want the ball to be delivered.

Now, moving out of abstract classes creating objects, we start defining subclasses for object creation.

```
class RedBallFactory{
    func deliverTheBall (_ speed : Int) -> CricketBall{
          print("Releasing Red Ball at \(speed) speed")
          return RedBall()
    }
}

class WhiteBallFactory{
    func deliverTheBall (_ speed : Int) -> CricketBall{
        print("Releasing White Ball at \(speed) speed")
        return WhiteBall()
    }
}
```

Here we are defining two factories to deliver different colours of balls. We input the speed of the ball and get a red/white ball in return.

It’s time we go to the machine and give an input to deliver the balls.

```
class BowlingMachine{
    enum AvailableBall : String{
        case redBall = "RedBall"
        case whiteBall = "WhiteBall"

        static let all = [redBall, whiteBall]
    }

    internal var factories = [AvailableBall : CricketBallFactory]()
    internal var namedFactories = [(String, CricketBallFactory)] ()

    init() {
        for ball in AvailableBall.all{
            let type = NSClassFromString("FactoryDesignPattern.\
(ball.rawValue)Factory")
            let factory = (type as! CricketBallFactory.Type).init()
            factories[ball] = factory
            namedFactories.append((ball.rawValue, factory))
        }
    }

    func setTheBall () -> CricketBall{
        for i in 0..<namedFactories.count{
            let tuple = namedFactories[i]
            print("\(i) : \(tuple.0)")
        }

        let input = Int(readLine()!)!
        return namedFactories[input].1.deliverTheBall(120)

    }
}
```

We define a class called BowlingMachine. We have an enum of available balls with redBall and whiteBall as the options. Then we have an array of all the available balls.

We have an internal variable called factories, which is a dictionary with key as the AvailableDrink and value as CricketBallFactory. Then we define a variable called namedFactories, which is a list of tuples where each entry has the name of the factory and the instance of the factory.

In the initialiser method, we initialise the factory. For each ball in available balls, we get the type from actual class. Then we construct the factory by taking the type and casting it as a CricketBallFactory and initialising it. Then we append each factory to the array of factories.

We then define a function that asks us to set the ball and returns a cricket ball. For each factory, we print out the index and the name of the factory. Then based on the input entered by the user, we return cricketBall at given speed.

Let's now define a function called main and see the code in action.

```
func main(){
    let bowlingMachine = BowlingMachine()
    print(bowlingMachine.namedFactories.count)
    let ball = bowlingMachine.setTheBall()
    ball.hitMe()
}

main()
```

Here we initialise the BowlingMachine and set the ball. Then we call the hitMe method on the instance of each ball the user inputs.

Output in the Xcode console:

```
2
AvailableBalls
0 : RedBall
1 : WhiteBall
```

If we choose 0, we print 'Releasing Red Ball at 20 speed'.
If we choose 1, we print 'Releasing White Ball at 20 speed'.

Summary:

When you are in a situation where a class does not know what subclasses will be required to create, or when a class wants its subclasses to specify the objects to be created, go for Factory design pattern.

7) Creational - Builder Design Pattern:

Definition:

Builder is a creational design pattern that helps in piecewise construction of complex objects avoiding too many initialiser arguments. It lets us produce different types and representations of an object using the same process of building.

This pattern primarily involves three types:

Product - complex object to be created
Builder - handles the creation of product
Director - accepts inputs and coordinates with the builder

Usage:

Let us assume we are creating a cricket team that consists of a captain, batsmen, and bowlers. We will see how we can use Builder pattern in this context.

We start with the Product part first.

```
import UIKit

//MARK: -Product
public struct CricketTeam{
    public let captain : Captain
    public let batsmen : Batsmen
    public let bowlers : Bowlers
}

extension CricketTeam : CustomStringConvertible{
    public var description : String{
        return "Team with captain \(captain.rawValue)"
    }
}
```

We first define CricketTeam, which has properties for captain, batsmen, and bowlers. Once a team is set, we shouldn't be able to change its composition. We also make CricketTeam conform to CustomStringConvertible.

```
public enum Captain : String{
    case Dhoni
    case Kohli
    case Rahane
}
```

We declare Captain as enum. Each team can have only one captain.

```
public struct Batsmen : OptionSet{
    public static let topOrderBatsman = Batsmen(rawValue: 1 << 0)
    public static let middleOrderBatsman = Batsmen(rawValue: 1 << 1)
    public static let lowerOrderBatsman = Batsmen(rawValue: 1 << 2)

    public let rawValue : Int
    public init(rawValue : Int){
        self.rawValue = rawValue
    }
}

public struct Bowlers : OptionSet{
    public static let fastBowler = Bowlers(rawValue: 1 << 0)
    public static let mediumPaceBowler = Bowlers(rawValue: 1 << 1)
    public static let spinBowler = Bowlers(rawValue: 1 << 2)

    public let rawValue : Int
    public init(rawValue : Int){
        self.rawValue = rawValue
    }
}
```

We define Batsmen and Bowlers as OptionSet. This allows us to try different combinations of batsmen together, like a team with two topOrderBatsman and one middleOrderBatsman. Same with Bowlers where we can choose a combination of fastBowler, mediumPaceBowler, and a spinBowler for the team.

Add the following code to make Builder:

```
//MARK: -Builder
public class CricketTeamBuilder{

    public enum Error:Swift.Error{
        case alreadyTaken
    }

    public private(set) var captain : Captain = .Dhoni
    public private(set) var batsmen : Batsmen = []
    public private(set) var bowlers : Bowlers = []
    private var soldOutCaptains : [Captain] = [.Dhoni]

    public func addBatsman(_ batsman : Batsmen){
        batsmen.insert(batsman)
    }

    public func removeBatsman(_ batsman: Batsmen) {
        batsmen.remove(batsman)
    }

    public func addBowler(_ bowler : Bowlers){
        bowlers.insert(bowler)
    }

    public func removeBowler(_ bowler: Bowlers) {
        bowlers.remove(bowler)
    }

    public func pickCaptain(_ captain: Captain) throws {
        guard isAvailable(captain) else { throw Error.alreadyTaken }
        self.captain = captain
    }

    public func isAvailable(_ captain: Captain) -> Bool {
        return !soldOutCaptains.contains(captain)
    }
    public func makeTeam() -> CricketTeam{
        return CricketTeam(captain: captain, batsmen: batsmen, bowlers: bowlers)
    }

}
```

We declare properties for captain, batsmen, and bowlers. These are declared as var so that we can change the team's composition based on the requirement. We are using private(set) for each to ensure only CricketTeamBuilder can set them directly.

Since each property is declared private, we need to provide public methods to change them. We defined methods like addBatsman, removeBatsman, addBowler, removeBowler, etc., for the purpose of building the team.

We have an interesting thing to note here. Every team by default should have a captain. Assume you are starting a team with Dhoni as captain. What if some other team tries to choose Dhoni as captain too? We should throw some error using the array of soldOutCaptains. We check the availability of the captains via isAvailable method.

We are done with the Builder. Now, let's build our Director.

```
//MARK: -Director/ Maker
public class TeamOwner {

    public func createTeam1() throws -> CricketTeam {
        let teamBuilder = CricketTeamBuilder()
        try teamBuilder.pickCaptain(.Kohli)
        teamBuilder.addBatsman(.topOrderBatsman)
        teamBuilder.addBowler([.fastBowler, .spinBowler])
        return teamBuilder.makeTeam()
    }

    public func createTeam2() throws -> CricketTeam {
        let teamBuilder = CricketTeamBuilder()
        try teamBuilder.pickCaptain(.Dhoni)
        teamBuilder.addBatsman([.topOrderBatsman, .lowerOrderBatsman])
        teamBuilder.addBowler([.mediumPaceBowler, .spinBowler])
        return teamBuilder.makeTeam()
    }

}
```

We have a class called TeamOwner, who builds their teams from the available options. Each team is built taking an instance of CricketTeamBuilder, picking up a captain and arrays of different types of batsmen and bowlers.

Now, let’s define a function called main to see the code in action.

```
func main(){
    let owner = TeamOwner()
    if let team = try? owner.createTeam1(){
        print("Hello! " + team.description)
    }

}

main()
```

We try to use method createTeam1 with captain as Kohli.

Output in the Xcode console:

```
Hello! Team with captain Kohli
```

Now, change the main() to the following:

```
func main(){
    let owner = TeamOwner()
    if let team = try? owner.createTeam1(){
        print("Hello! " + team.description)
    }

    if let team = try? owner.createTeam2(){
        print("Hello! " + team.description)
    } else{
        print("Sorry! Captain already taken")
    }
}

main()
```

After Team1, we are trying to create a Team2 with the help of createTeam2() with Dhoni as captain. But Dhoni is already taken and we throw the error.

Output in the Xcode console:

```
Hello! Team with captain Kohli
Sorry! Captain already taken
```

Summary:

If you are trying to use the same code for building different products to isolate the complex construction code from business logic, Builder design pattern fits the best.

Also, be careful when your product does not require multiple parameters for initialisation or construction. In this instance it’s advised to stay away from Builder pattern.

8) Creational - Prototype Design Pattern

Definition:

Prototype is a creational design pattern used to produce new objects that have very few differences. A prototype is basically a template of any object before the actual object is constructed. The Prototype pattern delegates a cloning process to objects themselves.

Usage:

Let us consider a simple use case where we want to create the profile of two cricketers, including their name and a custom profile that includes runs scored and wickets taken.

```
import UIKit

class Profile : CustomStringConvertible{
    var runsScored : Int
    var wicketsTaken : Int

    init(_ runsScored : Int, _ wicketsTaken : Int) {
        self.runsScored = runsScored
        self.wicketsTaken = wicketsTaken
    }

    var description: String{
        return "\(runsScored) Runs Scored & \(wicketsTaken) Wickets Taken"
    }
}
```

First, we create a Profile class that conforms to CustomStringConvertible. It has two properties, runsScored and wicketsTaken of type int. It takes the same parameters during its initialisation.

Then we define a Cricketer class that conforms to CustomStringConvertible. It has two properties, name of type String and profile of custom type Profile, which we just created.

```
class Cricketer : CustomStringConvertible {
    var name : String
    var profile : Profile

    init(_ name :String , _ profile : Profile) {
        self.name = name
        self.profile = profile
    }
    var description: String{
        return "\(name) : Profile : \(profile)"
    }

}
```

Let us now write a function called main to see the things in action.

```
func main (){
    let profile = Profile(1200, 123)
    let bhuvi = Cricketer("Bhuvi", profile)
    print(bhuvi.description)
}

main()
```

<u>In the Xcode console it prints:</u>

Bhuvi : Profile : 1200 Runs Scored & 123 Wickets Taken

Now we need to talk about copying the objects.

Just before print statement in the main function, add the following lines:

```
    var ishant = bhuvi
    ishant.name = "Ishant"
    print(ishant.description)
```

In the Xcode console it prints:

```
Ishant : Profile : 1200 Runs Scored & 123 Wickets Taken
Ishant : Profile : 1200 Runs Scored & 123 Wickets Taken
```

This is because we are only copying the references.

Now add this line just before printing ishant's description:

```
ishant.profile.runsScored = 600
```

In the Xcode console it prints:

```
Ishant : Profile : 600 Runs Scored & 123 Wickets Taken
Ishant : Profile : 600 Runs Scored & 123 Wickets Taken
```

Now we need to make sure bhuvi and ishant actually refer to different objects.

Here, we use the concept of Deep Copy. When we deep copy objects, the system will copy references, and each copied reference will be pointing to its own copied memory object. Let us now see how to implement Deep Copy interface for our use case.

```
protocol DeepCopy{
    func createDeepCopy () -> Self
}
```

First, we create a DeepCopy protocol that defines a function called createDeepCopy returning self.

Then make the classes Profile and Cricketer conform to DeepCopy protocol. Classes now look like this:

```
class Profile : CustomStringConvertible, DeepCopy{
    var runsScored : Int
    var wicketsTaken : Int

    init(_ runsScored : Int, _ wicketsTaken : Int) {
        self.runsScored = runsScored
        self.wicketsTaken = wicketsTaken
    }

    var description: String{
        return "\(runsScored) Runs Scored & \(wicketsTaken) Wickets Taken"
    }

    func createDeepCopy() -> Self {
        return deepCopyImplementation()
    }

    private func deepCopyImplementation <T> () -> T{
        return Profile(runsScored, wicketsTaken) as! T
    }
}
```

We have a private method called deepCopyImplementation, which is generic and able to figure out the type correctly. It has a type parameter ‘T’, which is actually going to be inferred (we don’t provide this type parameter anywhere) and a return type of ‘T’. We return a Profile object and force cast it to T.

Cricketer class now looks like this:

```
class Cricketer : CustomStringConvertible ,DeepCopy{
    var name : String
    var profile : Profile

    init(_ name :String , _ profile : Profile) {
        self.name = name
        self.profile = profile
    }

    var description: String{
        return "\(name) : Profile : \(profile)"
    }

    func createDeepCopy() -> Self {
        return deepCopyImplementation()
    }

    private func deepCopyImplementation <T> () -> T{
        return Cricketer(name, profile) as! T
    }

}
```

Let us define our main method as below and see the results:

```
func main(){
    let profile = Profile(1200, 123)
    let bhuvi = Cricketer("Bhuvi", profile)
    let ishant = bhuvi.createDeepCopy()
    ishant.name = "Ishant"
    ishant.profile = bhuvi.profile.createDeepCopy()
    ishant.profile.wicketsTaken = 140
    print(bhuvi.description)
    print(ishant.description)
}

main()
```

Output in the Xcode console:

```
Bhuvi : Profile : 1200 Runs Scored & 123 Wickets Taken
Ishant : Profile : 1200 Runs Scored & 140 Wickets Taken
```

We can see that bhuvi and ishant are two different objects now, and this is how Deep Copy is implemented.

Summary:

When you are in a situation to clone objects without coupling to their concrete classes, you can opt for Prototype design pattern, which also helps in reducing repetitive initialisation code.

9) Creational - Singleton Design Pattern

When discussing which patterns to drop, we found that we still love them all (Not really - I am in favour of dropping Singleton. Its usage is almost always a design smell) - Erich Gamma (one of the Gang Four)

A design pattern everyone loves to hate. Is it because it is actually bad or is it because of its abuse by the developers? Let's see.

Definition:

Singleton is a creational design pattern that provides us with one of the best ways to create an object. This pattern ensures a class has only one instance and provides a global access to it so that the object can be used by all the other classes.

Usage:

Let us take the case of an API that returns some JSON response, which when parsed looks like this:

["Sachin" : 1, "Sehwag" : 2 , "Dravid" : 3, "Kohli" : 4, "Yuvraj" : 5 ,"Dhoni" : 6 ,"Jadeja" : 7 ,"Ashwin" : 8, "Zaheer" : 9 ,"Bhuvi" : 10, "Bumrah" : 11]

This data structure is an array where each object is a key-value pair. Key represents the name of Indian Cricketer and Value represents the position at which the cricketer bats.

We would need only one instance of the SingletonDatabase class in order to save this data to our database. There is no point in initialising database class more than once, as it would just waste memory. Our code looks like this:

```
import UIKit
class SingletonDatabase{
    var dataSource = ["Sachin" : 1, "Sehwag" : 2 , "Dravid" : 3, "Kohli" : 4,
"Yuvraj" : 5   ,"Dhoni" : 6 ,"Jadeja" : 7 ,"Ashwin" : 8, "Zaheer" : 9 ,"Bhuvi" : 10,
"Bumrah" : 11]

    var cricketers = [String:Int]()

    static let instance = SingletonDatabase()
    static var instanceCount = 0

    private init(){
        print("Initialising the singleton")
        type(of: self).instanceCount += 1
        for dataElement in dataSource{
            cricketers[dataElement.key] = dataElement.value
        }
    }

}
```

We first make a private initialiser that does not take any arguments. And that's the simplest way to create on object. As it is private, no one can make another instance of the class.

But how do we let someone access the SingletonDatabase? That's where the Singleton pattern comes into play.

We initialise a static variable with the only instance of SingletonDatabase class. Making it static restricts the ability to create multiple instances of the class.

Now we add the data coming from the API call to our array of cricketers. That's it! We have our database ready.

Now, how does someone have access to this database? Assume we want to know the position at which a cricketer bats. We write a function for that just after the private init() method in SingletonDatabase class.

```
func getRunsScoredByCricketer(name:String) -> Int{
        if let position = cricketers[name]{
            print("\(name) bats at number \(position) for Indian Crikcet Team")
            return cricketers[name]!
        }

       print("Cricketer with name \(name) not found")
       return 0
}
```

This method is straightforward. It takes the name of the cricketer as an argument and returns his position in the line-up.

In order for us to access this class at some point in our code, we write it this way:

```
func main(){
    let singleton = SingletonDatabase.instance
    singleton.getRunsScoredByCricketer(name: "Sachin")
}
```

Very simple and short. We create a variable named singleton, which helps us in accessing all the functions in our SingletonDatabase class.

Now run the main() method.

```
main()
```

Output in the Xcode console:

```
Initialising the singleton
Sachin bats at number 1 for Indian Cricket Team
```

Change the name parameter to “Sach” and the output is:

```
Initialising the singleton
Cricketer with name Sach not found
```

We have not yet discussed the variable named instanceCount in our private init()

method. We can use this variable to show that there is only one instance of the SingletonDatabase class.

Change the main method this way:

```
func main(){
    let singleton1 = SingletonDatabase.instance
    print(SingletonDatabase.instanceCount)

    let singleton2 = SingletonDatabase.instance
    print(SingletonDatabase.instanceCount)

}
```

Output in the Xcode console:

```
Initialising the singleton
1
1
```

Instance count remains 1 even though we initialised the class more than once.

Adding the code snippet for another self-explanatory example here, which would enhance your understanding:

```
import UIKit

class PlayerRating : CustomStringConvertible{
    private static var _nameOfThePlayer = ""
    private static var _ratingForThePlayer = 0

    var nameOfThePlayer : String{
        get {return type(of: self)._nameOfThePlayer}
        set(value) {type(of: self)._nameOfThePlayer = value}
    }

    var ratingForThePlayer : Int{
        get {return type(of: self)._ratingForThePlayer}
        set(value) {type(of: self)._ratingForThePlayer = value}
    }

    var description: String{
        return "\(nameOfThePlayer) has got a rating of \(ratingForThePlayer)"
    }
}

func main(){
    let playerRating1 = PlayerRating()
    playerRating1.nameOfThePlayer = "Dhoni"
    playerRating1.ratingForThePlayer = 8

    let playerRating2 = PlayerRating()
    playerRating2.ratingForThePlayer = 7

    print(playerRating1)
    print(playerRating2)
}
main()
```

Output in the Xcode console:

```
Dhoni has got a rating of 7
Dhoni has got a rating of 7
```

Summary:

We should use Singleton pattern only when we have a scenario forcing us to use a single instance of an object at multiple places.

Part Three: Structural

10) Structural - Adapter Design Pattern

Definition:

Adapter is a structural design pattern that converts the interface of a class into another interface clients expect. This allows classes with incompatible interfaces to collaborate.

Usage:

Suppose you have a TestBatsman class with fieldWell() and makeRuns() methods. And also a T20Batsman class with batAggressively() method.

Let's assume that you are short on T20Batsman objects and you would like to use TestBatsman objects in their place. TestBatsmen have some similar functionality but implement a different interface (they can bat but cannot bat in the way needed for a T20 match), so we can't use them directly.

We will use the Adapter pattern. Here our client would be T20Batsman and adaptee would be TestBatsman.

Let us now write code:

```
import UIKit

protocol TestBatsman {
   func makeRuns()
   func fieldWell()
}
```

A simple protocol named TestBatsman defining two methods, makeRuns and fieldWell.

```
class Batsman1 : TestBatsman{
    func makeRuns() {
        print("I can bat well but only at StrikeRate of 80")
    }

    func fieldWell() {
        print("I can field well")
    }
}
```

We define a Batsman1 class conforming to TestBatsman protocol. This type of batsman can make runs at a strike rate of 80.

```
protocol T20Batsman{
    func batAggressively()
}
```

We have one more protocol named T20Batsman, which defines batAggressively method.

```
class Batsman2 : T20Batsman{
    func batAggressively() {
         print("I need to bat well at a StrikeRate of more than 130")
    }
}
```

We define a Batsman2 class conforming to T20Batsman protocol. This type of batsman can make runs at a strike rate of 130.

Now considering our situation, we need to make an adapter in such a way that TestBatsman can fit to be a T20Batsman.

```
class TestBatsmanAdapter : T20Batsman{
    let testBatsman : TestBatsman
    init (_ testBatsman : TestBatsman){
        self.testBatsman = testBatsman
    }
    func batAggressively() {
        testBatsman.makeRuns()
    }
}
```

We write a class named TestBatsmanAdapter, whose superclass is T20Batsman. It has a property of type TestBatsman and it takes an object of type TestBatsman for its initialisation. It is this object which we make adaptable to batAggressively method by calling makeRuns method.

Output in the Xcode console:

```
Test Batsman
I can field well
I can bat well but only at StrikeRate of 80
T20 Batsman
I need to bat well at a StrikeRate of more than 130
TestBatsmanAdapter
I can bat well but only at StrikeRate of 80
```

Summary:

When you are in a situation where you have an object that should be able to do the same task but in lots of different ways, and you do not want to expose the algorithm's implementation details to other classes, opt for Adapter design pattern.

11) Structural - Bridge Design Pattern

Definition:
Bridge is a structural design pattern that lets us connect components together through abstraction. It enables the separation of implementation hierarchy from interface hierarchy and improves the extensibility.
Usage:
Let us suppose that we have a protocol named Batsman, whose main function is to make runs for his team.

```
import Foundation
import UIKit

protocol Batsman
{
    func makeRuns(_ numberOfBalls: Int)
}
```

makeRuns takes a parameter named numberOfBalls of type Int.

Let us now define three different classes of batsmen conforming to Batsman protocol.

```
class TestBatsman : Batsman
{
    func makeRuns(_ numberOfBalls: Int) {
        print("I am a Test Batsman and I score \(0.6 * Double(numberOfBalls))
runs in \(numberOfBalls) balls")
    }
}
```

```
class ODIBatsman : Batsman
{
    func makeRuns(_ numberOfBalls: Int) {
        print("I am a ODI Batsman and I score \(1 * Double(numberOfBalls)) runs
in \(numberOfBalls) balls")
    }
}

class T20IBatsman : Batsman
{
    func makeRuns(_ numberOfBalls: Int) {
        print("I am a T20 Batsman and I score \(1.4 * Double(numberOfBalls))
runs in \(numberOfBalls) balls")
    }
}
```

We have three types of batsmen with the only difference between them being the number of runs they score in a given number of balls. Let us now define a protocol Player, whose main function is to play.

```
protocol Player
{
    func play()
}
```

We now define a Cricketer class conforming to Player protocol.

```
class Cricketer : Player
{
    var numberOfBalls: Int
    var batsman: Batsman

    init(_ batsman: Batsman, _ numberOfBalls: Int)
    {
        self.batsman = batsman
        self.numberOfBalls = numberOfBalls
    }

    func play() {
        batsman.makeRuns(numberOfBalls)
    }

}
```

Cricketer class takes two parameters during its initialisation, one of type Batsman and the other of type Int. This is where we are bridging between Batsman class and Player class by calling makeRuns method of batsman in the play method.

Let us now define our main function and see how this design pattern works.

```
func main()
{
    let testBatsman = TestBatsman()
    let odiBatsman = ODIBatsman()
    let t20Batsman = T20IBatsman()

    let cricketer1 = Cricketer(testBatsman, 20)
    let cricketer2 = Cricketer(odiBatsman, 20)
    let cricketer3 = Cricketer(t20Batsman, 20)

    cricketer1.play()
    cricketer2.play()
    cricketer3.play()

}

main()
```

Output in the Xcode console:

```
I am a Test Batsman and I score 12.0 runs in 20 balls
I am a ODI Batsman and I score 20.0 runs in 20 balls
I am a T20 Batsman and I score 28.0 runs in 20 balls
```

Summary:

When you are in a situation where you have to change the implementation object inside the abstraction, and when you need to extend a class in several independent dimensions, Bridge design pattern serves the best.

12) Structural - Composite Design Pattern

Definition:

Composite is a structural design pattern that lets us compose objects into tree structures and allows clients to work with these structures as if they were individual objects. Composition lets us make compound objects.

Usage:

Assume we are building a tree structure of a cricket team where each entity contains name, role, and grade of contract as attributes. Let’s see how we can use Composite design pattern to build such a system.

```
import UIKit
import Foundation

class CricketTeamMember : CustomStringConvertible{

    var name : String
    var role : String
    var grade : String
    var teamMembers : [CricketTeamMember]

    init(name:String, role : String, grade : String) {
        self.name = name
        self.role = role
        self.grade = grade
        self.teamMembers = [CricketTeamMember]()
    }
    func addMember(member : CricketTeamMember){
        teamMembers.append(member)
    }

    func removeMember(member : CricketTeamMember){
        teamMembers.append(member)
    }

    func getListOfTeamMembers() -> [CricketTeamMember]{
        return teamMembers
    }
    var description: String
    {
        let demo = "\(name)  \(role) \(grade)"
        return demo

    }
}
```

Let’s start with defining a class called CricketTeamMember conforming to CustomStringConvertible. It has four properties like name of type String, role of type String, grade of type String, and an array of teamMembers of type CricketTeamMember. It takes three parameters for its initialisation: name, role, and grade of type String.

We define a function called addMember, which takes a CricketTeamMember object as parameter and appends it to the teamMembers array.

We have a function named removeMember, which takes a CricketTeamMember object as parameter and removes it from teamMembers array.

We have another function called getListOfTeamMembers, which returns list of team members.

Let us now define main function and see how the Composite pattern can be used to define a tree structure.

```
func main(){
```

```
//1

    let headCoach = CricketTeamMember(name: "HeadCoach", role:
"HeadCoach", grade: "A")
    let captain = CricketTeamMember(name: "TeamCaptain", role: "Captain",
grade: "B")
    let bowlingCoach = CricketTeamMember(name: "BowlingCoach", role:
"Coach", grade: "B")
    let battingCoach = CricketTeamMember(name: "BattingCoach", role:
"Coach", grade: "B")
    let fieldingCoach = CricketTeamMember(name: "FieldingCoach", role:
"Coach", grade: "B")
    let asstBowlingCoach = CricketTeamMember(name: "ABoC1", role:
"AsstCoach", grade: "C")
    let asstBattingCoach = CricketTeamMember(name: "ABaC1", role:
"AsstCoach", grade: "C")
    let asstFieldingCoach = CricketTeamMember(name: "ABfC1", role:
"AsstCoach", grade: "C")
    let teamMember1 = CricketTeamMember(name: "TM1", role: "Player", grade:
"B")
    let teamMember2 = CricketTeamMember(name: "TM2", role: "Player", grade:
"B")

  //2

    headCoach.addMember(member: captain)
    headCoach.addMember(member: bowlingCoach)
    headCoach.addMember(member: battingCoach)
    headCoach.addMember(member: fieldingCoach)

    captain.addMember(member: teamMember1)
    captain.addMember(member: teamMember2)

    bowlingCoach.addMember(member: asstBowlingCoach)
    battingCoach.addMember(member: asstBattingCoach)
    fieldingCoach.addMember(member: asstFieldingCoach)

//3
```

```
    print(headCoach.description)
    for member in headCoach.getListOfTeamMembers(){
        print(member.description)
        for member in member.getListOfTeamMembers(){
            print(member.description)
        }
    }
}

main()
```

Let’s read this method step-by-step now.

1. 1) Here we define different team members using the instance of CricketTeamMember. We can see different roles like HeadCoach, TeamCaptain, BowlingCoach, etc.

1. 2) We then start forming trees by adding all the captains and coaches under head coach, adding team members under team captain, etc.

1. 3) Here we start printing the trees. Initially we print the description of HeadCoach and then we loop through all the team members added under him and print their descriptions too.

<u>Output in the Xcode console:</u>

```
HeadCoach  HeadCoach A
TeamCaptain  Captain B
TM1  Player B
TM2  Player B
BowlingCoach  Coach B
ABoC1  AsstCoach C
BattingCoach  Coach B
```

ABaC1 AsstCoach C
FieldingCoach Coach B
ABfC1 AsstCoach C

Summary:

When you are in a situation to simplify the code at the client's end that has to interact with a complex tree structure, then go for Composite design pattern. In other words, it should be used when clients need to ignore the difference between compositions of objects and individual objects.

13) Structural - Decorator Design Pattern

Definition:

Decorator is a structural design pattern that lets us add new behaviour to the objects without altering the class itself. It helps us in keeping the new functionalities separate without having to rewrite existing code.

Usage:

Assume we are checking if a player is fit for playing T20 game of cricket as a bowler or batsman or both or none, based on his batting and bowling statistics. Let us see how Decorator design pattern can help us here.

```
import UIKit
import Foundation

class T20Batsman{

    var strikeRate : Int = 0

    func makeRuns() -> String{
        return (strikeRate > 130) ? "Fit for T20 Team as Batsman" : "Too slow 
Batsman for T20 Team"
    }

}
```

We write a class called T20Batsman with a property called strikeRate of type Int. It has a function defined makeRuns, which tells us if the batsman is fit for T20 team based on his strikeRate. If the strike rate is more than 130, he is fit as T20 batsman, otherwise he is too slow for the game.

```
class T20Bowler{

    var economyRate : Float = 0

    func bowlEconomically () -> String{
        return (economyRate < 8.0) ? "Fit for T20 Team as Bowler" : "Too
expensive as a Bowler"
    }

}
```

We then define a class called T20Bowler with a property called economyRate of type Float. It has a function defined called bowlEconomically, which tells us if the bowler is fit for T20 team based on his economyRate. If the economy rate is less than 8.0, he is fit as T20 bowler, otherwise he is too expensive as a bowler for the game.

```
class T20AllRounder : CustomStringConvertible{
    private var _strikeRate : Int = 0
    private var _economyRate : Float = 0

    private let t20Batsman = T20Batsman()
    private let t20Bowler = T20Bowler()


    func makeRuns() -> String{
        return t20Batsman.makeRuns()
    }

    func bowlEconomically() -> String{
        return t20Bowler.bowlEconomically()
    }

    var strikeRate : Int{
        get {return _strikeRate}
        set(value){
            t20Batsman.strikeRate = value
            _strikeRate = value
        }
    }

    var economyRate : Float{
        get {return _economyRate}
        set(value){
            t20Bowler.economyRate = value
            _economyRate = value
        }
    }

    var description: String{
        if t20Batsman.strikeRate > 130 && t20Bowler.economyRate < 8 {
            return "Fit as T20 AllRounder"
        }
        else{
        var buffer = ""
        buffer += t20Batsman.makeRuns()
        buffer += " & " + t20Bowler.bowlEconomically()
        return buffer
        }
    }
}
```

We now define a class for T20AllRounder conforming to CustomStringConvertible. All rounder is someone in cricket who can bat and bowl reasonably well. It has four private variables, strikeRate and economyRate of type Int and Float, along with two more variables of type T20Batsman and T20Bowler.

This allrounder should be able to make runs and bowl well. It has two functions defined:

1. 1) makeRuns: Here we use the instance of T20Batsman variable to call the makeRuns method and see if he is fit as T20Batsman based on defined criteria for strike rate.

1. 2) bowlEconomically: Here we use the instance of T20Bowler variable to call the bowlEconomically method and see if he is fit as T20Bowler based on defined criteria for economy rate.

In the future, if we want to change the conditions for batsmen or bowler or both, we do not have to disturb the code written for allrounder class. Just changing the code in T20Batsman and T20Bowler classes will be enough.

Let us now write a main function to see the code in action.

```
func main(){

   let t20AllRounder = T20AllRounder()
   t20AllRounder.strikeRate = 120
   t20AllRounder.economyRate = 7
   print(t20AllRounder.description)

}

main()
```

We take an instance of T20AllRounder class and feed in the strikeRate and economyRate and see if a certain player is fit or not.

<u>Output in the Xcode console:</u>

Too slow Batsman for T20 Team & Fit for T20 Team as Bowler

Keep changing the inputs for strikeRate and economyRate and see if the player is fit for T20 game of cricket.

```
t20AllRounder.strikeRate = 150
t20AllRounder.economyRate = 7
```

Prints: **Fit as T20 AllRounder**

t20AllRounder.strikeRate = 150
t20AllRounder.economyRate = 9

Prints: **Fit for T20 Team as Batsman & Too expensive as a Bowler**

t20AllRounder.strikeRate = 120
t20AllRounder.economyRate = 9

Prints: **Too slow Batsman for T20 Team & Too expensive as a Bowler**

Summary:

If you are in a situation where you are looking for something more flexible than class inheritance and need to edit/update behaviours at runtime, then Decorator design pattern serves you better.

14) Structural - Facade Design Pattern

Definition:

Facade is a structural design pattern that lets us expose several patterns through a single, easy-to-use interface. Facade defines a higher-level interface that makes the subsystem easier to use by wrapping a complicated subsystem with a simpler interface.

Usage:

Assume we are building an imaginary player auction system for a private cricket league. Any team with an id and a name can buy players who have an id, role in the team, and a price. Let’s write some code for this:

```
import UIKit
import Foundation

//Team represents an object that can buy a player
public struct Team {

   public let teamId: String
   public var teamName: String
}

public struct Player {
   public let playerId: String
   public var primaryRole: String
   public var price: Double
}
```

We define Team struct that holds the properties of teamId and teamName as String. Then there is another struct for Player that holds playerId, primaryRole as String and price as Double.

```
//Any Swift type that conforms the Hashable protocol must also conform the
Equatable protocol. Because Hashable protocol is inherited from Equatable
protocol.

extension Team: Hashable {

    public var hashValue : Int{
        return teamId.hashValue
    }

    public static func == (lhs : Team, rhs : Team) -> Bool{
        return lhs.teamId == rhs.teamId
    }

}

extension Player : Hashable{

    public var hashValue : Int{
        return playerId.hashValue
    }

    public static func ==(lhs:Player, rhs:Player) ->Bool{
        return lhs.playerId == rhs.playerId
    }
}
```

We write a couple of extensions, one for Team and one for Player, each conforming to Hashable protocol. When we conform to a hashable protocol, we must have a hashValue property.

Hashable is a type that has hashValue in the form of an integer that can be compared across different types. We get the hashValue as teamId.hashValue.

Apple definesEquatable as a type that can be compared for value equality, which is part of the working definition for a hashable protocol.

We then use mandatory method related to Hashable protocol that compares the type and checks to see if they are equal.

```
public class AvailablePlayersList{
    public var availablePlayers : [Player : Int] = [:]

    public init(availablePlayers : [Player:Int]){
        self.availablePlayers = availablePlayers
    }

}

public class SoldPlayersList{
    public var soldPlayers : [Team:[Player]] = [:]
}
```

We define a class called AvailablePlayersList. It has a variable named availablePlayers of type Dictionary.

Then we have another class called SoldPlayerList, which has a variable named soldPlayers that basically maintains a list of players bought by a certain team.

Now we define our facade with the help of the classes defined above!

```
public class AuctionFacade{

    public let availablePlayersList : AvailablePlayersList
    public let soldPlayersList : SoldPlayersList

    public init(availablePlayersList:AvailablePlayersList,
soldPlayersList:SoldPlayersList){
        self.availablePlayersList = availablePlayersList
        self.soldPlayersList = soldPlayersList
    }

    public func buyAPlayer(for player: Player,
                           by team: Team) {
```

```
        print("Ready to buy \(player.primaryRole) with id '\(player.playerId)' - '\
(team.teamName)'")

        let count = availablePlayersList.availablePlayers[player, default: 0]
        guard count > 0 else {
            print("'\(player.primaryRole)' is sold out")
            return
        }

        availablePlayersList.availablePlayers[player] = count - 1

        var soldOuts =
            soldPlayersList.soldPlayers[team, default: []]
        soldOuts.append(player)
        soldPlayersList.soldPlayers[team] = soldOuts

        print("\(player.primaryRole) with \(player.playerId) " + "bought by '\
(team.teamName)'")
    }

}
```

AuctionFacade takes two parameters during its initialisation, one of type AvailablePlayersList and one of type SoldPlayerList. We then define a public method buyAplayer.

When a player is bought, the count for that type of player is reduced by one in availablePlayerList. The same player is appended to the list of soldPlayersList.

Let’s now write a main function to see our facade in action.

```
func main(){
    let bowler1 = Player(playerId: "12345", primaryRole: "Bowler", price: 123)
    let batsman1 = Player(playerId: "12365", primaryRole: "Batsman", price: 152)

    let availablePlayerList = AvailablePlayersList(availablePlayers: [bowler1 : 3,
batsman1:45])
    let auctionFacade = AuctionFacade(availablePlayersList: availablePlayerList,
soldPlayersList: SoldPlayersList())
    let team1 = Team(teamId: "XYZ-123", teamName: "Sydney")
    auctionFacade.buyAPlayer(for: bowler1, by: team1)
}

main()
```

We define bowler1 and batsman1 as Player type objects. We then initialise AvailablePlayerList with 3 bowler1 type Players and 45 batsman1 type Players.

We then take an instance of AuctionFacade and provide availablePlayerList and instance of SoldPlayerList as parameters.

Output in the Xcode console:

```
Ready to buy Bowler with id '12345' - 'Sydney'
Bowler with 12345 bought by 'Sydney'
```

Summary:

When you want to provide a simple interface to a complex subsystem and have a single interface for traversing different data structures, Facade design patterns works the best.

15) Structural - FlyWeight Design Pattern

Definition:

FlyWeight is a structural design pattern that helps in avoiding redundancy while storing data. It helps fit more objects in the available amount of RAM by reusing already existing similar kinds of objects by storing them and creating a new object when no matching object is found.

Assume you are storing first and last names in memory. When there are many people with identical first and last names, there is no point in storing them again and again as a new entity. Instead we use something like FlyWeight design pattern to save the storage space.

Usage:

Let us consider a situation where we are storing player profiles where each entity consists of player's name of type String and the teams he played for of type String array as attributes.

We write the code without using FlyWeight design pattern and check the memory occupied.

```
import UIKit
import Foundation

class PlayerProfile{
    var fullName : String
    var teamsPlayedFor : [String]

    init(_ fullName : String, _ teamsPlayedFor : [String]) {
        self.fullName = fullName
        self.teamsPlayedFor = teamsPlayedFor
    }

    var charCount: Int
    {
        var count = 0
        for team in teamsPlayedFor{
            count += team.utf8.count
        }
        count += fullName.utf8.count
        return count
    }
}
```

We define a class called PlayerProfile, which takes fullName of type String and teamsPlayedFor of type String array as parameters during its initialisation.

We then define a variable called charCount, which is an indicator of the memory occupied. Let us write our main function and check the character count.

```
func main()
{
    let dhoni = PlayerProfile("Mahendra Dhoni",["India ,Chennai"])
    let kohli = PlayerProfile("Virat Kohli",["India , Bangalore"])
    let yuvi = PlayerProfile("Yuvraj Singh",["India , Punjab"])
    print("Total number of chars used:"  ,dhoni.charCount + kohli.charCount +
yuvi.charCount)

}

main()
```

We define a few instances of PlayerProfile by passing the players' names and their teams as parameters. Then we use the charCount property on all the instances and print it to the console.

<u>Output in the Xcode console:</u>

Total number of chars used: 82

Let us now use FlyWeight design pattern for the same use case.

```
class PlayerProfileOptimised{
    static var stringsArray = [String]()
    private var genericNames = [Int]()

    init(_ fullName: String, _ teamsPlayedFor : [String])
    {
        func getOrAdd(_ s: String) -> Int
        {
            if let idx = type(of: self).stringsArray.index(of: s)
            {
                return idx
            }
            else
            {
                type(of: self).stringsArray.append(s)
                return type(of: self).stringsArray.count - 1
            }
        }
        genericNames = fullName.components(separatedBy: " ").map {
getOrAdd($0) }
        for team in teamsPlayedFor{
            genericNames = team.components(separatedBy: " ").map {getOrAdd($0)
}
        }
    }

    static var charCount: Int
    {
        return stringsArray.map{ $0.utf8.count }.reduce(0, +)
    }
}
```

We define a class called PlayerProfileOptimised. Here we define a static variable called stringsArray, which stores different strings that may or may not be repeated. We then define a non-static variable called genericNames, which is going to keep an array of indices.

In initialisation method, we have an inner function called getOrAdd, which takes a string as a parameter and returns the index of the string in stringsArray if already existing, or returns the index by adding it to the stringsArray array at the tail end.

We then initialise the genericNames array by taking the full name, splitting it into a component separated by space, and mapping it by calling the function getOrAdd with a parameter. This lets us get genericNames to be initialised to a set of indices that correspond to the strings inside the stringsArray array.

Let us now write a main function and check the character count:

```
func main()
{
   let dhoni1 = PlayerProfileOptimised("Mahendra Dhoni",["India ,Chennai"])
   let kohli1 = PlayerProfileOptimised("Virat Kohli",["India , Bangalore"])
   let yuvi1 = PlayerProfileOptimised("Yuvraj Singh",["India , Punjab"])
   print("Total number of chars used:"  ,PlayerProfileOptimised.charCount)
}

main()
```

Output in the Xcode console:

```
Total number of chars used: 63
```

For the same data, the number of characters reduced significantly. That’s how FlyWeight design pattern can be used for efficient storage of data.

Summary:

When you are in a situation to store data that might contain a significant amount of duplicate data, you can use FlyWeight design pattern. This helps in reducing the usage of available RAM.

16) Structural - Proxy Design Pattern

Definition:

Talking real world terms, your debit card is a proxy of your bank account. It's not real money, it but can be substituted for money when you want to buy something.

Proxy is a structural design pattern that uses wrapper classes to create a stand-in for a real resource. It is also called surrogate, handle, and wrapper. Proxy is used to cover the main object's complex logic from the client using it.

Usage:

Assume we are designing a small software to filter applicants for the position of head coach of a cricket team. The client only passes the number of years of the applicant's experience, and we need to write a logic to say if the applicant is fit for the role or not, without disturbing the client.

Let us see how we can use Proxy design pattern here:

```
import UIKit
import Foundation

protocol Coach
{
    func mentorTheTeam()
}

class CricketCoach : Coach
{
    func mentorTheTeam() {
        print("Mentoring the Cricket Team")
    }

}
```

We define a protocol called Coach, whose main job is to mentor the team.
Then we define a class called CricketCoach conforming to Coach protocol.

```
class CoachApplicant
{
    var numberOfYearsOfExperience: Int

    init(numberOfYearsOfExperience: Int)
    {
        self.numberOfYearsOfExperience = numberOfYearsOfExperience
    }
}
```

We write a class called CoachApplicant, which takes numberOfYearsOfExperience of type Int as parameter during its initialisation.

Now we write a proxy conforming to Coach protocol to define the logic in order to filter applicants.

```
class CricketCoachProxy : Coach
{
    private let cricketCoach = CricketCoach()
    private let coachApplicant: CoachApplicant

    init(coachApplicant: CoachApplicant)
    {
        self.coachApplicant = coachApplicant
    }

    func mentorTheTeam() {
        if coachApplicant.numberOfYearsOfExperience >= 8{
            cricketCoach.mentorTheTeam()
        } else{
            print("Not enough experience to coach the team")
        }
    }

}
```

It has two private variables, one of type CricketCoach and one of type CoachApplicant. In mentorTheTeam method, we define the logic. If the experience of the coach applicant is more than 8 years, he is through, otherwise he is rejected.

Let us now write our main method:

```
func main()
{
    let coach : Coach = CricketCoachProxy(coachApplicant:
CoachApplicant(numberOfYearsOfExperience: 8))
    coach.mentorTheTeam()
}

main()
```

Output in the Xcode console:

```
Mentoring the Cricket Team
```

Keep changing the experience parameter and check the output.

```
func main()
{
    let coach : Coach = CricketCoachProxy(coachApplicant:
CoachApplicant(numberOfYearsOfExperience: 5))
    coach.mentorTheTeam()
}

main()
```

Output in the Xcode console:

Not enough experience to coach the team

In the future, if we want to change the criteria from 8 years to 10 years or 6 years, we do not have to change any code at the client's end. We can just change the logic in the proxy and things will work just fine.

Summary:

When you want to create a wrapper around a main object to hide its complexity from the client, Proxy design pattern suits the best. It also helps in delaying the object's initialisation so that you can load the objects only when it is needed.

Part Four: Behavioural

17) Behavioural - Chain of Responsibility Design Pattern

Definition:

Chain of Responsibility is a behavioural design pattern that allows us to avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request.

Usage:

Assume we are building a small cricket video game where we choose the player characters with their default skills. But then we also give provision to add skill boosters to the player characters as gamers gain some credits. Let us see how we can use Chain of Responsibility to design such a system.

```
import Foundation

class Cricketer : CustomStringConvertible{
    var name : String
    var battingSkillRating : Int
    var bowlingSkillRating : Int
    var fieldingSkillRating : Int

    init(_ name:String, _ battingSkillRating:Int, _ bowlingSkillRating:Int, _ 
fieldingSkillRating:Int) {
        self.name = name
        self.battingSkillRating = battingSkillRating
        self.bowlingSkillRating  =  bowlingSkillRating
        self.fieldingSkillRating = fieldingSkillRating
    }
    var description: String{
        return "Cricketer : \(name) with battingRating : \(battingSkillRating),
bowlingRating : \(bowlingSkillRating), fieldingRating : \(fieldingSkillRating)"
    }
}
```

We define a class called Cricketer conforming to CustomStringConvertible. During its initialisation it takes parameters of name of type String, battingSkillRating of type Int, bowlingSkillRating of type Int, and fieldingSkillRating of type Int.

```
class SkillBooster{
    let cricketer : Cricketer
    var skillBooster : SkillBooster?

    init(_ cricketer : Cricketer) {
        self.cricketer = cricketer
    }

    func addBooster(_ booster : SkillBooster){
        if skillBooster != nil{
            skillBooster!.addBooster(booster)
        } else{
            skillBooster = booster
        }
    }

    func playTheGame(){
        skillBooster?.playTheGame()
    }
}
```

We then define a class called SkillBooster, which is meant to be a base class for different types of boosters. It takes a parameter of type Cricketer during its initialisation. We also define an optional private variable of type SkillBooster. It has two methods defined, addBooster and playTheGame. addBooster takes a parameter of type SkillBooster and adds it to existing boosters after nil check. Otherwise, skillBooster is assigned the value of incoming booster.

```
class BattingSkillBooster : SkillBooster{
    override func playTheGame() {
        print("Adding Hook Shot to \(cricketer.name) 's Batting")
        cricketer.battingSkillRating += 1
        super.playTheGame()
    }
}

class BowlingSkillBooster : SkillBooster{
    override func playTheGame() {
        print("Adding Reverse Swing to \(cricketer.name) 's Bowling")
        cricketer.bowlingSkillRating += 1
        super.playTheGame()
    }
}

class FieldingSkillBooster : SkillBooster{
    override func playTheGame() {
        print("Adding Dive Catches to \(cricketer.name) 's Fielding")
        cricketer.fieldingSkillRating += 1
        super.playTheGame()
    }
}
```

We define different types of skill boosters with SkillBooster as the base class. In all the boosters, we override the function playTheGame and improve the corresponding skill rating of the player character by 1.

```
class NoSkillBooster : SkillBooster{
    override func playTheGame() {
        print("No boosters available here")
        //don't call super
    }
}
```

We also define a dummy skill booster just to make the game more interesting.

Let us now write a main function to see the code in action.

```swift
func main(){

    let dhoni = Cricketer("Dhoni", 6, 3, 7)
    print(dhoni)
}

main()
```

<u>Output in the Xcode console:</u>

```
Cricketer : Dhoni with battingRating : 6, bowlingRating : 3, fieldingRating :
7
```

Now change the main method to the following:

```swift
func main(){

    let dhoni = Cricketer("Dhoni", 6, 3, 7)
    print(dhoni)

    let skillBooster = SkillBooster(dhoni)

    print("Adding Batting Booster to Dhoni")
    skillBooster.addBooster(BattingSkillBooster(dhoni))
    skillBooster.playTheGame()
    print(dhoni.description)

}

main()
```

<u>Output in the Xcode console:</u>

```
Cricketer : Dhoni with battingRating : 6, bowlingRating : 3, fieldingRating :
7
Adding Batting Booster to Dhoni
Adding Hook Shot to Dhoni 's Batting
Cricketer : Dhoni with battingRating : 7, bowlingRating : 3, fieldingRating :
7
```

We add BattingSkillBooster to object dhoni of type Cricketer. We can check the output in the console for an improved rating on dhoni’s batting skills.

Change the main method to the following and observe the console:

```
func main(){

    let dhoni = Cricketer("Dhoni", 6, 3, 7)

    let skillBooster = SkillBooster(dhoni)

    print("Adding Batting Booster to Dhoni")
    skillBooster.addBooster(BattingSkillBooster(dhoni))

    print("Adding Bowling Booster to Dhoni")
    skillBooster.addBooster(BowlingSkillBooster(dhoni))
    skillBooster.playTheGame()
    print(dhoni.description)
}

main()
```

Output in the Xcode console:

```
Adding Batting Booster to Dhoni
Adding Bowling Booster to Dhoni
Adding Hook Shot to Dhoni 's Batting
Adding Reverse Swing to Dhoni 's Bowling
Cricketer : Dhoni with battingRating : 7, bowlingRating : 4, fieldingRating :
7
```

Summary:

Use the Chain of Responsibility pattern when you can conceptualize your program as a chain made up of links, where each link can either handle a request or pass it up the chain. It can modify an existing behaviour by overriding an existing method using inheritance.

18) Behavioural - Strategy Design Pattern

Definition:

Strategy is a behavioural design pattern that lets you define a set of encapsulated algorithms and enables selecting one of them at runtime. An important point to observe is that these algorithm implementations are interchangeable. In other words, strategy lets the algorithm vary independently from the clients that use it.

Usage:

Consider an example of a Bowling Machine that releases balls of different colours based on the input of speed specified by the user. Assume we have three different speeds: slow, medium, and fast, which corresponds to yellow, green, and red coloured balls respectively.

```
import UIKit

enum CricketBall : String{
    case slow = "Yellow"
    case medium = "Green"
    case fast = "Red"
}
```

We now define a protocol ReleaseCricketBallStrategy, which has properties of speed and the type of cricket ball, and which also defines a method to release the ball.

```
protocol ReleaseCricketBallStrategy{
    var speed : String {get set}
    var cricketBall : CricketBall {get set}
    func releaseBall() -> String
}
```

We now define three new classes, one each for fast, medium, and slow ball strategies.

Each of the classes conform to ReleaseCricketBallStrategy protocol.
For the sake of simplicity, we define speed as a string which can be Fast, Medium, or Slow. Each of the classes has an initialiser that does not take any extra arguments.

releaseBall method returns a string implying that its implementation releases a ball with specified properties.

```
class FastBallStrategy : ReleaseCricketBallStrategy{

    var speed = "Fast"
    var cricketBall = CricketBall.fast
    init(){}

    func releaseBall() -> String {
        return "Released \(speed) ball with colour \(cricketBall.rawValue)"
    }
}

class MediumBallStrategy : ReleaseCricketBallStrategy{
    var speed = "Medium"
    var cricketBall = CricketBall.medium
    init(){}

    func releaseBall() -> String {
        return "Released \(speed) ball with colour \(cricketBall.rawValue)"
    }
}

class SlowBallStrategy : ReleaseCricketBallStrategy{
    var speed = "Slow"
    var cricketBall = CricketBall.slow
    init(){}
    func releaseBall() -> String {
        return "Released \(speed) ball with colour \(cricketBall.rawValue)"
    }
}
```

Now, we define a BowlingMachine class, which can be initialised at runtime by passing an argument of the type of strategy. We make it conform to CustomStringConvertible.

```
class BowlingMachine : CustomStringConvertible {
    private var releaseCricketBallStrategy : ReleaseCricketBallStrategy
    private var returnString = ""
    init(whatStrategy : ReleaseCricketBallStrategy){
        self.releaseCricketBallStrategy = whatStrategy
        returnString = releaseCricketBallStrategy.releaseBall()
    }
    var description: String{
        return returnString
    }
}
```

It's now time to play with our bowling machine. We define a method named main, where we initialise BowlingMachine class with different types of strategies.

```
func main(){
    var bowlingMachine = BowlingMachine(whatStrategy: FastBallStrategy())
    print(bowlingMachine.description)

    bowlingMachine = BowlingMachine(whatStrategy: SlowBallStrategy())
    print(bowlingMachine.description)

    bowlingMachine = BowlingMachine(whatStrategy: MediumBallStrategy())
    print(bowlingMachine.description)

}
```

Now, run the main() method.

```
main()
```

Output in the Xcode console:

```
Released Fast ball with colour Red
Released Slow ball with colour Yellow
Released Medium ball with colour Green
```

Summary:

Strategy pattern allows us to define a set of related algorithms and allows the client to choose any of the algorithms at runtime. It allows us to add a new algorithm without modifying existing algorithms.

19) Behavioural - Command Design Pattern:

Definition:

The definition of Command provided in the original Gang of Four book on Design Patterns states:

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Command is a behavioural design pattern that decouples the object invoking the operation from the object that knows how to perform it. It allows us to turn requests into stand-alone objects by providing request objects with all the necessary information for the action to be taken.
Usage:
Let us consider a decision review system in a cricket match where the on-field umpire is not sure if a batsman is out or not. This umpire then asks the TV umpire to check TV replays and make a decision. The TV umpire then commands the TV operator to show OUT/NOT OUT on the screen, depending upon the decision made.
Let’s see how we can design such a system with the help of Command design pattern.

```
import UIKit
import Foundation

protocol Command{
    func execute()
}
```

We define a protocol named Command with a function named execute.

```
class ScreenDisplay{
    private var showOutOnDisplay = false

    func isBatsmanOut(){
        showOutOnDisplay = true
        print("Batsman is OUT")
    }

    func isBatsmanNotOut(){
        showOutOnDisplay = false
        print("Batsman is NOTOUT")
    }

}
```

We then define a class called ScreenDisplay, which is used to display the decision made by the TV umpire. It has a private variable named showOutOnDisplay, which is initialised to false.
It has two methods defined. Based on the bool property, these methods show batsman OUT/NOT OUT on the screen.

```
class BatsmanOutCommand : Command{
    var screenDisplay : ScreenDisplay

    init(_ screenDisplay : ScreenDisplay){
        self.screenDisplay = screenDisplay
    }

    func execute() {
        screenDisplay.isBatsmanOut()
    }
}
```

```
class BatsmanNotOutCommand : Command{
    var screenDisplay : ScreenDisplay

    init(_ screenDisplay : ScreenDisplay){
        self.screenDisplay = screenDisplay
    }

    func execute() {
        screenDisplay.isBatsmanNotOut()
    }
}
```

We then write two classes, BatsmanOutCommand and BatsmanNotOutCommand, conforming to Command protocol. Both these classes take a parameter of type ScreenDisplay during their initialisation. Then we write the definition of execute method by calling isBatsmanOut/isBatsmanNotOut on ScreenDisplay object.

```
class DisplaySwitch {
    var command : Command

    init(_ command : Command) {
        self.command = command
    }

    func pressSwitch(){
        command.execute()
    }
}
```

Finally, we write a class called DisplaySwitch, which takes an object of type Command for its initialisation. We define a method called pressSwitch, which implements the execute method on Command object.
Let us write our main method and see our code in action.

```
func main(){
    let screenDisplay = ScreenDisplay()

    let outCommand = BatsmanOutCommand(screenDisplay)
    let notOutCommand = BatsmanNotOutCommand(screenDisplay)
    let displaySwitchForOut = DisplaySwitch(outCommand)
    displaySwitchForOut.pressSwitch()

    let displaySwitchForNotOut = DisplaySwitch(notOutCommand)
    displaySwitchForNotOut.pressSwitch()

}

main()
```

We take an instance of ScreenDisplay and pass it to BatsmanOutCommand and BatsmanNotOutCommand for their initialisations.

Then, based on the decision made by the TV umpire, we initialise DisplaySwitch using outCommand/notOutCommand as the parameters.

Output in the Xcode console:

```
Batsman is OUT
Batsman is NOTOUT
```

Summary:

When you want to encapsulate the logical details of an operation in a separate entity and define specific instructions for applying the command, Command design pattern serves you the best. It also helps in creating composite commands.

20) Behavioural - Iterator Design Pattern

Definition:

Iteration in coding is a core functionality of various data structures. An iterator facilitates the traversal of a data structure. Iterator is a behavioural design pattern that is used to sequentially access the elements of an aggregate object without exposing its underlying implementation.

Usage:

Assume we are making a list of top cricketers in a current lot, which includes their name and team name. We will now see how to use an iterator to traverse through the list and print the profile of each cricketer.

Let us write some code now:

```
import Foundation
struct Cricketer{
   let name : String
   let team : String
}
```

We define a struct named Cricketer, which stores name and team as String properties.

```
struct Cricketers{
   let cricketers : [Cricketer]
}
```

We define another struct named Cricketers, which stores cricketers array of custom type Cricketer.

```
struct CricketersIterator : IteratorProtocol{

    private var current = 0
    private let cricketers : [Cricketer]

    init(_ cricketers : [Cricketer]) {
        self.cricketers = cricketers
    }

    mutating func next() -> Cricketer? {
        defer {
            current += 1
        }
        if cricketers.count > current{
            return cricketers[current]
        } else{
            return nil
        }
    }

}

extension Cricketers : Sequence{
    func makeIterator() -> CricketersIterator {
        return CricketersIterator(cricketers)
    }
}
```

This is where the magic happens. We define a struct named CricketersIterator conforming to IteratorProtocol. Then we write an extension for Cricketers that conforms to Sequence protocol.

Apple says,

“The IteratorProtocol protocol is tightly linked with the Sequence protocol. Sequences provide access to their elements by creating an iterator, which keeps track of its iteration process and returns one element at a time as it advances

through the sequence.
Whenever you use a for-in loop with an array, set, or any other collection or sequence, you're using that type's iterator. Swift uses a sequence's or collection's iterator internally to enable the for-in loop language construct. Using a sequence's iterator directly gives you access to the same elements in the same order as iterating over that sequence using a for-in loop."

Back to our code, we defined two private properties, current of type Int (with default value of 0) and an array cricketers of type Cricketer.

We define a method next that returns an object of type Cricketer (notice the optional - we may not have any element left in the array after we reach the last element).

Let us now write our main method:

```
func main(){
    let cricketers = Cricketers(cricketers: [Cricketer(name: "Kohli", team: 
"India"), Cricketer(name: "Steve", team: "Australia"), Cricketer(name: "Kane", 
team: "Kiwis"), Cricketer(name: "Root", team: "England")])
    for crick in cricketers{
        print(crick)
    }
}

main()
```

<u>Output in the Xcode console:</u>

```
Cricketer(name: "Kohli", team: "India")
Cricketer(name: "Steve", team: "Australia")
Cricketer(name: "Kane", team: "Kiwis")
Cricketer(name: "Root", team: "England")
```

Adding the code snippet for another self-explanatory example here, which would enhance your understanding:

```
import Foundation
```

```swift
class Cricketer : Sequence
{
    var totalRunsScored = [Int](repeating: 0, count: 3)

    private let _testRuns = 0
    private let _ODIRuns = 1
    private let _t20Runs = 2

    var testRuns: Int
    {
        get { return totalRunsScored[_testRuns] }
        set(value) { totalRunsScored[_testRuns] = value }
    }

    var ODIRuns: Int
    {
        get { return totalRunsScored[_ODIRuns] }
        set(value) { totalRunsScored[_ODIRuns] = value }
    }

    var t20Runs: Int
    {
        get { return totalRunsScored[_t20Runs] }
        set(value) { totalRunsScored[_t20Runs] = value }
    }

    var totalRuns: Int
    {
        return totalRunsScored.reduce(0, +)
    }

    subscript(index: Int) -> Int
    {
        get { return totalRunsScored[index] }
        set(value) { totalRunsScored[index] = value }
    }

    func makeIterator() -> IndexingIterator<Array<Int>>
    {
        return IndexingIterator(_elements: totalRunsScored)
    }
}

func main()
{
    let cricketer = Cricketer()
    cricketer.testRuns = 1200
    cricketer.ODIRuns = 1800
    cricketer.t20Runs = 600

    print("Total Runs Scored = \(cricketer.totalRuns)")

    for s in cricketer
    {
        print(s)
    }
}

main()
```

Output in the Xcode console:

```
Total Runs Scored = 3600
1200
1800
600
```

Summary:

When you are in a situation where you want to hide the complexity of a data structure from clients and have a single interface for traversing the data structures, Iterator design pattern serves you the best.

21) Behavioural - Interpreter Design Pattern

Definition:

Interpreters are present everywhere around us. In design patterns, interpreter is a component that processes structured text data. It falls under behavioural design pattern.

Usage:

Let us use interpreter design pattern to get a way to print the elements of a collection object in sequential manner.

```
import UIKit
import Foundation

protocol Interpreter{
    func hasNext() -> Bool
    func next() -> String
}

protocol Container{
   func getInterpreter() -> Interpreter
}
```

We define two protocols named Interpreter and Container. Interpreter has two functions to check if the next element is present in array after every iteration and to return the element (if present). Container has a method to return the interpreter.

```
class NameRepo : Container{
    let names = ["India" ,"Australia", "England", "NewZealand"]
    func getInterpreter() -> Interpreter {
        return NameInterpreter(names)
    }
}
```

We then define a class called NameRepo conforming to Container protocol. In our main function, we use iterator to print all the values present in the names array.

```
private class NameInterpreter : Interpreter{
    var index = -1
    var names = [String]()

    init(_ names : [String]){
        self.names = names
    }

    func hasNext() -> Bool {
        if index < names.count {
            return true
        }
        return false
    }

    func next() -> String {
        if self.hasNext(){
            index = index + 1
            return names[index]
        } else{
            return ""
        }
    }
}
```

We define a class called NameIterator conforming to Iterator protocol. It takes a parameter of type String array during its initialisation.

Let us write a main function to see the code in action.

```
func main(){
    let nr = NameRepo()
    let interpreter = NameInterpreter(nr.names)

    for _ in nr.names{
        interpreter.hasNext()
        print(interpreter.next())
    }
}


main()
```

Output in the Xcode console:

```
India
Australia
England
NewZealand
```

Summary:

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees.

22) Behavioural - Mediator Design Pattern

Definition:

Mediator is a behavioural design pattern that lets us define a component that encapsulates relationships between a set of components (that absolutely makes no sense to have direct references to one another) to make them independent of each other. Mediator pattern prevents direct communication between individual components by sending requests to a central component that knows where to redirect those requests.

Usage:

Let us assume we are designing a TV umpire decision review system for a cricket match. When an on-field umpire does not have enough evidence to rule a batsman out, he sends the request to the TV umpire who takes a look at the replays and sends a command to the monitor operator who displays the final decision on the big screen.

Let's see how we can use Mediator design pattern to design such a decision review system.

```
import UIKit
import Foundation

protocol Command{
    func displayStatus()
}
```

We write a protocol Command that defines a function called displayStatus.

```
protocol RemoteUmpire{
    func registerTVDisplay(tvDisplay :TVDisplay)
    func registerTVOperator(tvOperator : TVOperator)
    func isDecisionMade() -> Bool
    func setDecisionStatus(status : Bool)
}
```

We write another protocol called RemoteUmpire that defines a handful of functionalities.

1. 1) The remote umpire (also called TV umpire) has to register for TV display by passing a parameter of type TVDisplay (to be defined).
2. 2) The remote umpire (also called TV umpire) has to register for TV operator by passing a parameter of type TVOperator (to be defined).
3. 3) A function named isDecisionMade, which returns a boolean.
4. 4) Another function to set the status of decision by passing a boolean argument.

```
class TVOperator : Command{
    var tvUmpire:TVUmpire

    init(_ tvUmpire : TVUmpire){
        self.tvUmpire = tvUmpire
    }

    func displayStatus() {
        if tvUmpire.isDecisionMade(){
            print("Decision Made and Batsman in OUT")
            tvUmpire.setDecisionStatus(status: true)
        } else{
             print("Decision Pending")
        }
    }

    func getReady(){
        print("Ready to Display Decision")
    }
}
```

We define a class called TVOperator conforming to Command protocol. It takes an object of type TVUmpire (to be defined) during its initialisation.

It has a method named displayStatus, which based on the TV umpire's decision, displays a batsman out or not on the big screen in the stadium.

```
class TVDisplay : Command{
    var tvUmpire:TVUmpire

    init(_ tvUmpire : TVUmpire) {
        self.tvUmpire = tvUmpire
        tvUmpire.setDecisionStatus(status: true)
    }

    func displayStatus() {
        print("Decision made and permission granted to display the decision on TV
Display")
        tvUmpire.setDecisionStatus(status: true)
    }
}
```

We define a class called TVDisplay conforming to Command protocol. This class also takes an object of type TVUmpire (to be defined) for its initialisation. Its main functionality is to display the status of the decision based on the input given by the TV umpire.

```
class TVUmpire : RemoteUmpire{
    private var tvOperator : TVOperator?
    private var tvDisplay : TVDisplay?
    private var decisionMade : Bool?

    func registerTVDisplay(tvDisplay: TVDisplay) {
        self.tvDisplay = tvDisplay
    }

    func registerTVOperator(tvOperator: TVOperator) {
        self.tvOperator = tvOperator
    }

    func isDecisionMade() -> Bool {
        return decisionMade!
    }
    func setDecisionStatus(status: Bool) {
        self.decisionMade = status
    }
}
```

We then define our most important class named TVUmpire conforming to RemoteUmpire protocol. It has three private optional variables defined: tvOperator of type TVOperator, tvDisplay of type TVDisplay, and a boolean named decisonMade.

It registers for TV display by assigning its property of tvDisplay to parameter of type TVDisplay from the method registerTVDisplay.

It registers for TV operator by assigning its property of tvOperator to parameter of type TVOperator from the method registerTVOperator.

Let’s now write our main function and see how the above code comes into action.

```
func main(){
    let tvUmpire = TVUmpire()
    let tvDisplayAtGround = TVDisplay(tvUmpire)
    let tvOperatorAtGround = TVOperator(tvUmpire)
    tvUmpire.registerTVDisplay(tvDisplay: tvDisplayAtGround)
    tvUmpire.registerTVOperator(tvOperator: tvOperatorAtGround)
    tvOperatorAtGround.getReady()
    tvDisplayAtGround.displayStatus()
    tvOperatorAtGround.displayStatus()
}

main()
```

We take an instance of TVUmpire and pass the same instance to initialise TVDisplay and TVOperator.

TVUmpire then registers for TVDisplay and TVUmpire by passing instances of TVDisplay and TVUmpire respectively.

Once the TV umpire has made his decision, TV operator on the ground gets ready to display the status of the decision accordingly on the display at the ground.

Output in the Xcode console:

```
Ready to Display Decision
Decision made and permission granted to display the decision on TV Display
Decision Made and Batsman in OUT
```

Summary:

Use a Mediator design pattern when the complexity of the object communication begins to hinder object reusability. Mediator engages in two-way communication with its connected components.

23) Behavioural - Memento Design Pattern

Definition:

Memento is a behavioural design pattern that lets us save the snapshots of the object's internal state at every point of time without exposing its internal structure. This helps us in rolling back to the state when the snapshot was taken.

Usage:

Assume we are adding the stats of a cricketer (number of runs scored) year by year in our program, and at some point we want to trace back to a year in the past and check his stats up until that point in time.

Let's define a Memento class, which takes an argument of number of runs scored in its initialisation.

```
import UIKit
class Memento {
    let numberOfRunsScored : Int

    init(_ numberOfRunsScored : Int){
        self.numberOfRunsScored = numberOfRunsScored
    }
}
```

Let's assume an imaginary hardware named StatsHolder, which displays the stats. It conforms to CustomStringConvertible protocol. It takes an argument of number of runs scored in its initialisation.

```
class StatsHolder : CustomStringConvertible{
    private var numberOfRunsScored : Int
    private var snapshots : [Memento] = []
    private var currentIndex = 0

    init(_ numberOfRunsScored : Int) {
        self.numberOfRunsScored = numberOfRunsScored
        snapshots.append(Memento(numberOfRunsScored))
    }

    var description: String{
        return "Total Runs scored = \(numberOfRunsScored)"
    }
}
```

All the properties are declared private, as we do not want to expose the internal structure of our hardware to the client.

We maintain an array of snapshots of type Memento so that we can restore past stats just by passing a memento. When the class is initialised, currentIndex starts at zero.

We now add a function named ‘addStatsToHolder’, which takes number of runs as parameter and returns us a snapshot of type Memento.

```
 func addStatsToHolder (_ runsToBeAdded : Int) -> Memento{
        numberOfRunsScored += runsToBeAdded
        let snapshot = Memento(runsToBeAdded)
        snapshots.append(snapshot)
        currentIndex += 1
        return snapshot
    }
```

We keep appending the snapshot of Memento initialised to the array and increment the currentIndex by 1.

We need a function that lets us restore a past stat by passing a parameter of type Memento.

```
 func restoreToPastStat(_ memento : Memento?){
        if let snap = memento{
           numberOfRunsScored = snap.numberOfRunsScored
           snapshots.append(snap)
           currentIndex = snapshots.count - 1
        }
    }
```

Note that memento parameter is optional because for the currentIndex value of 0, we do not have anything to restore to. We change the numberOfRunsScored to the value of snapshot. We append the snapshot of parameter to our array and decrement the currentIndex by 1.

Now, we need methods to undo and redo stats.

```
func undoAStat() -> Memento?{
        if currentIndex > 0{
           currentIndex -= 1
           let snap = snapshots[currentIndex]
           numberOfRunsScored = snap.numberOfRunsScored
           return snap
        }
        return nil
    }

    func redoAStat() -> Memento?{
        if currentIndex+1 < snapshots.count{
           currentIndex += 1
           let snap = snapshots[currentIndex]
           numberOfRunsScored = snap.numberOfRunsScored
           return snap
        }
        return nil
    }
```

Note that the returned Memento is optional, as we may not have anything to undo for the first addition of the stat and nothing to redo after the final addition of the stat. In these cases, we return a nil.

In undoAStat method, we check if the currentIndex > 0. If no, we return nil. If yes, we decrement the currentIndex by 1 and get the snapshot at currentIndex. We then change the numberOfRunsScored to the value present in the snapshot.

In redoAStat method, we check if the currentIndex is less than the count of snapshots decremented by 1. If no, we return nil. If yes, we increment the currentIndex by 1 and get the snapshot at the currentIndex. We then change the numberOfRunsScored to the value present in the snapshot.

We are done with our set up of Memento design pattern. Let's see how we implement this pattern.

```
func main(){
    let statsHolder = StatsHolder(1200) //1200 is the first stat (number of runs) we
add to stats holder
    let stat1 = statsHolder.addStatsToHolder(1400)
    let stat2 = statsHolder.addStatsToHolder(700)

    print("a - ", statsHolder)

    //undo Top most operation
    statsHolder.undoAStat()
    print("b - ", statsHolder)

    //undo Top most operation
    statsHolder.undoAStat()
    print("c - ", statsHolder)

    //restoreToStat2
    statsHolder.redoAStat()
    print("d - ", statsHolder)
    //There is no memento/snapshot when the StatsHolder is initialised
}
```

In the main method, we initialise the StatsHolder by passing 1200 runs as parameter. We then add a couple of stats by using addStatsToHolder method by passing 1400 and 700 runs as parameters.

We then undo the last two stat additions by using undoAStat method on statsHolder. We then redo the last operation by using redoAStat method on statsHolder.

Now run the main() method.

```
main()
```

Output in the Xcode console:

```
a -  Total Runs scored = 3300
b -  Total Runs scored = 1400
c -  Total Runs scored = 1200
d -  Total Runs scored = 1400
```

Initially, we added three stats, which takes the total to 3300. Then we undo one operation, which takes the total to 1400 (subtracting 700). One more undo takes us to the initial state of 1200. We redo the last undo operation, which again takes us back to 1400.

Summary:

If your application demands to save checkpoints as the user progresses through the app, go for Memento design pattern. This helps in restoring the checkpoints at a later point of time.

24) Behavioural - Null Object Design Pattern

Definition:

In Object Oriented Programming, null is an object that has no referenced value. When an object A tries to use an object B, object A assumes that object B is not nil. When there is no option of telling A not to use instance of B when it has no value, Null Object design pattern comes into play. Null Object is a behavioural design pattern that simplifies the use of undefined dependencies.

Usage:

Let's see how this design pattern can be used in code.

Assume we have a cricket match happening at a stadium and users receive live updates of the score on their devices (iPad or iPhone). By default, we show the score on the interface. But on iPad, along with the live score, we also show bowlers and batsman stats, as there is available screen estate. But the same interface looks congested on an iPhone display. So, we refrain ourselves from showing batsman and bowler stats for iPhone display.

```
import Foundation

protocol Log
{
   func bowlerStatsFromCurrentMatch(_ stats: String)
   func batsmenStatsFromCurrentMatch(_ stats: String)
}
```

We have a protocol named Log that defines two methods to show bowlers and batsmen stats from the current match, which takes stats as input in String format.

```
class StatsDisplayLog : Log
{
    func bowlerStatsFromCurrentMatch(_ stats: String) {
        print(stats)
    }

    func batsmenStatsFromCurrentMatch(_ stats: String) {
         print(stats)
    }
}

class NoDisplayStatsLog : Log
{
    func bowlerStatsFromCurrentMatch(_ stats: String) {}
    func batsmenStatsFromCurrentMatch(_ stats: String) {}
}
```

We now define two classes, StatsDisplayLog and NoDisplayStatsLog, both conforming to Log protocol. The only difference is the implementation of these methods in the classes, which is very straight forward. We show the stats in StatsDisplayLog and do not show any stats in NoDisplayStatsLog.

```
class UserInterface
{
    var log: Log
    var runsScored = 0
    var wicketsTaken = 0

    init(_ log: Log)
    {
        self.log = log
    }

    func wicketTaken (){
        wicketsTaken += 1
        log.bowlerStatsFromCurrentMatch("Total Wickets : \(wicketsTaken)")
    }
    func runsScored(numberOFRunsScored : Int){
        runsScored += numberOFRunsScored
        log.batsmenStatsFromCurrentMatch("Total Runs : \(runsScored)")
    }

}
```

We define a class called UserInterface, which takes care of the logic behind what to display to the users on their devices. This class takes a parameter of type Log during its initialisation. There are two methods to update the number of wickets taken and number of runs scored, and when an event happens in the match. With the help of instance of Log class, these stats are shown on the interface.
Let's now write a function called main.

```
func main()
{
    let ipadLog = StatsDisplayLog()
    let iPAdUserInterface = UserInterface(ipadLog)
    iPAdUserInterface.runsScored(numberOFRunsScored: 4)
    iPAdUserInterface.runsScored(numberOFRunsScored: 3)
    iPAdUserInterface.wicketTaken()
}

main()
```

Output in the Xcode console:

```
Total Runs : 4
Total Runs : 7
Total Wickets : 1
```

This is what a user sees on his iPad, as we are taking an instance of StatsDisplayLog for iPad interface. Now, add the below code to the main function:

```
let iPhoneLog = NoDisplayStatsLog()
let iPhoneUserInterface = UserInterface(iPhoneLog)
iPhoneUserInterface.runsScored(numberOFRunsScored: 6)
iPhoneUserInterface.runsScored(numberOFRunsScored: 2)
```

We can observe that there is no change in the output in the Xcode console. This is because we are using an instance of NoDisplayStatsLog, which is used for an iPhone interface.

Summary:

Null Object design pattern can be used in situations where real objects are replaced by null objects when the object is expected to do nothing.

25) Behavioural - Observer Design Pattern

Definition:

Observer design pattern is used when we want an object (called observable), which maintains a list of objects (called observers), to notify them when observable does something or its properties change or some external change occurs. The process of notifying is done through events generated by observable.

Usage:

```
import Foundation

protocol Invocable : class
{
    func invoke(_ data: Any)
}

public protocol Disposable
{
    func dispose()
}

public class Event<T>
{
    public typealias EventHandler = (T) -> ()

    var eventHandlers = [Invocable]()

    public func raise(_ data: T)
    {
        for handler in eventHandlers
        {
            handler.invoke(data)
        }
    }

    public func addHandler<U: AnyObject>
      (target: U, handler: @escaping (U) -> EventHandler) -> Disposable
    {
        let subscription = Subscription(
            target: target, handler: handler, event: self)
        eventHandlers.append(subscription)
        return subscription
    }
}

class Subscription<T: AnyObject, U> : Invocable, Disposable
{
    weak var target: T?
    let handler: (T) -> (U) -> ()
    let event: Event<U>

    init(target: T?,
         handler: @escaping (T) -> (U) -> (),
         event: Event<U>)
    {
        self.target = target
        self.handler = handler
        self.event = event
    }

    func invoke(_ data: Any) {
        if let t = target {
            handler(t)(data as! U)
        }
    }

    func dispose()
    {
        event.eventHandlers = event.eventHandlers.filter { $0 as AnyObject? !==
self }
    }
}
```

This is how we define observers and observables. You can feel free to copy and paste this without any modifications to use Observer design patterns.
I will not be discussing it in detail, as the main intention is to learn about design patterns.

Now, assume a cricket match is happening. There is a scoreboard on the ground, which is updated whenever any event happens (run hit/batsman out/fielder taking catch, etc). For the sake of simplicity, let us only talk about the batting team making runs event.

Let us define scoreboard class.

```
class ScoreBoardInTheGround{
    let batsmenHitRun = Event<Int>()
    init(){}
    func updateScore(){

    }
}
```

ScoreBoardInTheGround can be initialised without any arguments. It has an event batsmenHitRun and a method to update the score. This is our observable, which broadcasts events whenever batsman hits runs.

When the scoreboard on the ground is updated, the same update has to reach the servers of a mobile app, where millions of people follow the match updates. There are many such servers, which are called observers, and they need to know whenever the state of ScoreBoardInTheGround changes. Let us define our observable class.

```
class ScoreUpdateInServers{
    init(){
        let scoreBoard = ScoreBoardInTheGround()
        let subscriber = scoreBoard.batsmenHitRun.addHandler(target: self,
handler: ScoreUpdateInServers.showScoreInApp)

        //simualte batsman hitting runs in the ground
        scoreBoard.batsmenHitRun.raise(6)
        //get rid of the description
        subscriber.dispose()
    }
    func showScoreInApp(score: Int){
        print("Score Now is : \(score) runs")
    }
}
```

When ScoreUpdateInServers is initialised, we take an instance of ScoreBoardInTheGround and add a subscriber to batsmanHitRun event with the help of a handler.

Then we are simulating an event of a batsman hitting six runs in the match and the scoreboard on the ground needs to broadcast this event to the servers. Then we also get rid of the subscription with the help of a dispose function.

Servers of the mobile app have a method to show the score in the display, which takes the score as the input parameter. For the sake of simplicity, we just print the score in this method.

Now, add the below code snippet to see the code in action.

```
func main(){
    let dummy = ScoreUpdateInServers()
}

main()
```

Output in the Xcode console:

```
Score Now is : 6 runs
```

Summary:

If you want to subscribe/unsubscribe to objects dynamically, Observer design pattern is the best possible way. Note the fact that subscribers are notified in random order.

26) Behavioural - State Design Pattern

Definition:

State is a behavioural design pattern that is used to alter the behaviour of an object as its internal state changes. The object will appear to change its class.

Usage:

Let’s assume a player auction is going on for some private cricket league. A player is either in unsold state or sold state with the name of the team attached to him. We now see how State design pattern can be used in this context.

```
import UIKit

protocol State {
   func isSold(playerAuction: PlayerAuction) -> Bool
   func whichTeam(playerAuction: PlayerAuction) -> String?
}
```

We have a protocol named State, which defines two methods:

1. 1) isSold takes an object of type PlayerAuction (to be defined) and returns true/false.
2. 2) whichTeam takes an object of type PlayerAuction (to be defined) and returns an optional (as an unsold player does not have any team associated with him).

We then define states in which a player object can be in.

```
class IsUnsoldState: State {
    func isSold(playerAuction: PlayerAuction) -> Bool { return false }

    func whichTeam(playerAuction: PlayerAuction) -> String? { return nil }
}

class IsSoldState: State {
    let teamName: String

    init(teamName: String) {
        self.teamName = teamName
    }

    func isSold(playerAuction: PlayerAuction) -> Bool {
        return true
    }

    func whichTeam(playerAuction: PlayerAuction) -> String? {
        return teamName
    }
}
```

1. 1) First one is IsUnsoldState and it conforms to State protocol. It returns false when called isSold method and a nil when called whichTeam.
2. 2) Second one is IsSoldState and it conforms to State protocol. It takes the argument of teamName of type String for its initialisation. It returns true when called isSold method and a nil when called whichTeam.

We now define the most important class, PlayerAuction:

```
class PlayerAuction {
    private var state: State = IsUnsoldState()

    var isSold: Bool {
        get {
            return state.isSold(playerAuction: self)
        }
    }

    var teamName: String? {
        get {
            return state.whichTeam(playerAuction: self)
        }
    }

    func changeStateToSold(teamName: String) {
        state = IsSoldState(teamName: teamName)

    }

    func changeStateToUnSold() {
        state = IsUnsoldState()
    }

}
```

It has a private variable of type State, which has a default value of IsUnsoldState. We then get and set two variables, isSold and teamName, with the help of state property and passing an argument of type self.

We then define two methods:

1. 1) changeStateToSold, which takes an argument of type String, and the state of player object is changed to IsSoldState by passing the argument.
2. 2) changeStateToUnsold, where player's state is changed to instance of IsUnsoldState.

Let us now write a main method to see this design pattern in action:

```
func main(){
    let playerAuction = PlayerAuction()
    print(playerAuction.isSold, playerAuction.teamName)
    playerAuction.changeStateToSold(teamName: "Chennai Super Kings")
    print(playerAuction.isSold, playerAuction.teamName!)
}

main()
```

We start with taking an instance of class PlayerAuction. Then change the default status of unsold to sold state by passing a team name Chennai Super Kings.

<u>Output in the Xcode console:</u>

```
false nil
true Chennai Super Kings
```

For the unsold state, isSold returns a false and whichTeam returns nil. As the state is changed, isSold returns true and whichTeam returns Chennai Super Kings.

<u>Summary:</u>

When you are in a situation where the behaviour of an object should be influenced by its state, the number of states is big, and the state related code changes frequently, State design pattern serves your purpose.

27) Behavioural - Template Design Pattern

<u>Definition:</u>

Template is a behavioural design pattern that helps to divide algorithms into common parts and specifics through inheritance. In simple words, base class declares algorithm 'placeholders' and derived classes write the concrete implementation of placeholders (or algorithms).

<u>Usage:</u>

Let us consider a situation where we are building a template for a cricket team. The main motto is to decide the team's composition based on the pitch and weather conditions (how many batsmen to play, how many bowlers to play, how many fast bowlers, how many spinners, etc). A single template for all the pitch conditions will not help.

Let us see how we can use Template design pattern for the same use case.

```
import UIKit
import Foundation

class TeamTemplate{

    func buildTeam(){
        pickBatsmen()
        pickBowlers()
        pickAllRounders()
        pickWicketKeeper()
        print("\nTeam Set For the match")
    }
```

```swift
    func pickBatsmen(){

    }

    func pickBowlers(){

    }

    func pickAllRounders(){

    }

    private func pickWicketKeeper(){
        print("Only one WK available and he is picked by default")
    }
}
```

We define a base class called TeamTemplate, which will later be inherited by other classes.

We define a function called pickWicketKeeper and declare it private because there is only one WicketKeeper available in the squad, and he is in the playing team by default. No one has the authority to change it.

We also define empty functions called pickBatsmen, pickBowlers, and pickAllRounders, whose implementations are defined in the subclasses through inheritance.

We now define three different classes with TeamTemplate as BaseClass where we write the concrete implementation of teamBuilding method.

```swift
class SeamingPitchTeamTemplate : TeamTemplate{
    override func pickBatsmen() {
        print("Picking 6 batsmen")
    }

    override func pickBowlers() {
        print("Picking 3 Fast Bowlers")
    }

    override func pickAllRounders() {
        print("Picking 1 Pace AllRounder")
    }
}

class SpinPitchTeamTemplate : TeamTemplate{
    override func pickBatsmen() {
        print("Picking 5 Batsmen")
    }

    override func pickBowlers() {
        print("Picking 2 Fast Bowlers and 2 Spinners")
    }

    override func pickAllRounders() {
        print("Picking 2 Spin AllRounder")
    }
}

class BattingPitchTeamTemplate : TeamTemplate{
    override func pickBatsmen() {
        print("Picking 7 Batsmen")
    }

    override func pickBowlers() {
        print("Picking 2 Fast Bowlers and 1 Spinners")
    }

    override func pickAllRounders() {
        print("Picking 1 Batting AllRounder")
    }
}
```

We define three templates named SeamingPitchTeamTemplate, SpinPitchTeamTemplate, and BattingPitchTeamTemplate, with different method definitions for pickBatsmen, pickBowlers, and pickAllRounders.

Let us now write our main method and see the code in action.

```
func main(){
    var finalTeam : TeamTemplate = SeamingPitchTeamTemplate()
    finalTeam.buildTeam()


}

main()
```

Assume we are picking a team for seam friendly pitch and we take an instance of SeamingPitchTeamTemplate and call the buildTeam method.

Output in the Xcode console:

```
Picking 6 batsmen
Picking 3 Fast Bowlers
Picking 1 Pace AllRounder
Only one WK available and he is picked by default

Team Set For the match
```

Now, we change the team compositions by taking instances of other templates and observe the output.

```
func main(){
    var finalTeam : TeamTemplate = SeamingPitchTeamTemplate()
    finalTeam.buildTeam()

    print("\n***Pitch Changed***\n")
    finalTeam = SpinPitchTeamTemplate()
    finalTeam.buildTeam()

}

main()
```

Output in the Xcode console:

```
Picking 6 batsmen
Picking 3 Fast Bowlers
Picking 1 Pace AllRounder
Only one WK available and he is picked by default

Team Set For the match

***Pitch Changed***

Picking 5 Batsmen
Picking 2 Fast Bowlers and 2 Spinners
Picking 2 Spin AllRounder
Only one WK available and he is picked by default

Team Set For the match
```

Now change the main method to:

```
func main(){
    var finalTeam : TeamTemplate = SeamingPitchTeamTemplate()
    finalTeam.buildTeam()

    print("\n***Pitch Changed***\n")
    finalTeam = SpinPitchTeamTemplate()
    finalTeam.buildTeam()

    print("\n***Pitch Changed***\n")
    finalTeam = BattingPitchTeamTemplate()
    finalTeam.buildTeam()
}

main()
```

Output in the Xcode console:

```
Picking 6 batsmen
Picking 3 Fast Bowlers
Picking 1 Pace AllRounder
```

```
Only one WK available and he is picked by default

Team Set For the match

***Pitch Changed***

Picking 5 Batsmen
Picking 2 Fast Bowlers and 2 Spinners
Picking 2 Spin AllRounder
Only one WK available and he is picked by default

Team Set For the match

***Pitch Changed***

Picking 7 Batsmen
Picking 2 Fast Bowlers and 1 Spinners
Picking 1 Batting AllRounder
Only one WK available and he is picked by default

Team Set For the match
```

<u>Summary:</u>

When you are in a situation to build a template/base class, which is open for extension but closed for modification, or in simple words, subclasses should be able to extend the base algorithm without altering its structure, Template design pattern suits the best.

28) Behavioural - Visitor Design Pattern

Definition:

Visitor is a behavioural design pattern that lets us define a new operation without changing the classes of the objects on which it operates. We use it when we do not want to keep modifying every class in the hierarchy.

Usage:

Consider a situation where we are designing a check-out counter in a shop that sells cricket accessories. It offers discounts on selective brands and selective items. Let us see how we can use Visitor pattern to design such a system.

```
import Foundation
import UIKit

protocol CricketAccessory{
    func accept(counter : CheckoutCounter) -> Int
}
```

We define a protocol called CricketAccessory with a function called accept, which takes a parameter of type CheckoutCounter (to be defined) and returns an integer.

```
class CricketBat : CricketAccessory{
    private var price : Double
    private var brand : String

    init(_ price : Double, _ brand:String) {
        self.price = price
        self.brand = brand
    }

    public func getPrice() -> Double{
        return price
    }

    public func getBrand() -> String{
        return brand
    }
    func accept(counter : CheckoutCounter) -> Int {
        return counter.moveToCounter(bat: self)
    }
}
```

We then define a class called CricketBat conforming to CricketAccessory protocol. It has two private variables defined: price of type Double and brand of type String. We also define two public methods called getPrice and getBrand to return price and brand of the bat respectively.

```
class CricketBall : CricketAccessory{

    private var type : String
    private var price : Double

    init(_ type : String, _ price : Double){
        self.type = type
        self.price = price

    }

    public func getType() -> String{
        return type
    }

    public func getPrice() -> Double{
        return price
    }
    func accept(counter : CheckoutCounter) -> Int {
         return counter.moveToCounter(ball: self)
    }

}
```

We define another class called CricketBall conforming to CricketAccessory protocol. This is very much similar to CricketBat class.

```
protocol CheckoutCounter {
    func moveToCounter(bat : CricketBat) -> Int
    func moveToCounter(ball : CricketBall) -> Int
}
```

We define a protocol called CheckoutCounter, which has two methods with the same name but differs when it comes to parameter types.

```
class CashCounter :CheckoutCounter{
    func moveToCounter(bat: CricketBat) -> Int {
        var cost : Int = 0
        if bat.getBrand() == "MRF"{
            cost = Int(0.9 * bat.getPrice())
        } else{
            cost = Int(bat.getPrice())
        }
        print("Bat brand : \(bat.getBrand()) and price is : \(cost) ")
        return cost
    }

    func moveToCounter(ball: CricketBall) -> Int {

        print("Ball Type : \(ball.getType()) and price is : \(ball.getPrice()) ")
        return Int(ball.getPrice())
    }
}
```

We now define a class called CashCounter conforming to CheckoutCounter. We can see that, for a cricket bat of brand MRF, we give a discount of 10%. In the future, if we want to add any new brands or remove discounts on existing

brands, we can make all the changes here with no changes required at the client end.
Let us now write a main function to see the code in action.

```
func main(){
    print("Main")
    func finalPriceCalculation(accessories : [CricketAccessory]) -> Int{
        var checkout = CashCounter()
        var cost = 0
        for item in accessories{
            cost += item.accept(counter: checkout)
        }
        print("Total cart value : \(cost)")
        return cost
    }

    var cartItems = [CricketAccessory]()
    let mrfBat = CricketBat(2000, "MRF")
    let brittaniaBat = CricketBat(1500, "Brittania")
    let tennisBall = CricketBall("Tennis", 120)
    let leatherBall = CricketBall("Leather", 200)
    cartItems.append(mrfBat)
    cartItems.append(brittaniaBat)
    cartItems.append(tennisBall)
    cartItems.append(leatherBall)

    var cost = finalPriceCalculation(accessories: cartItems)
    print("Checked Out with Bill Amount : \(cost)")
}

main()
```

We define a function called finalPriceCalculation, which takes an array of type CricketAccessory and returns the final cart value.

<u>Output in the Xcode console:</u>

```
Main
Bat brand : MRF and price is : 1800
Bat brand : Brittania and price is : 1500
```

```
Ball Type : Tennis and price is : 120.0
Ball Type : Leather and price is : 200.0
Total cart value : 3620
Checked Out with Bill Amount : 3620
```

You can see the MRF bat is checked out at discounted price.

Let us now change the CashCounter class to the following:

```
class CashCounter :CheckoutCounter{
    func moveToCounter(bat: CricketBat) -> Int {
        var cost : Int = 0
        if bat.getBrand() == "Brittania"{
            cost = Int(0.8 * bat.getPrice())
        } else{
            cost = Int(bat.getPrice())
        }
        print("Bat brand : \(bat.getBrand()) and price is : \(cost) ")
        return cost
    }

    func moveToCounter(ball: CricketBall) -> Int {

        print("Ball Type : \(ball.getType()) and price is : \(ball.getPrice()) ")
        return Int(ball.getPrice())
    }
}
```

Now, we are removing the discount on MRF and giving a 20% discount on Brittania bats.

The same main method gives a different output in the Xcode console:

```
Main
Bat brand : MRF and price is : 2000
Bat brand : Brittania and price is : 1200
Ball Type : Tennis and price is : 120.0
Ball Type : Leather and price is : 200.0
Total cart value : 3520
```

Checked Out with Bill Amount : 3520

<u>Summary:</u>

When you are in a situation where you might want to add a new action and have that new action entirely defined within one of the visitor classes rather than spread out across multiple classes, Visitor design pattern serves you the best.

Final note:

<u>Final note:</u>

It is not necessary to learn all the patterns and their applications by heart. The main intention is to identify the use cases and problems, which the design patterns are meant to address. Then applying a specific design pattern is just a matter of using the right tool at the right time for the right job. It's the job that must be identified and understood before the tool can be chosen.

Happy Coding!!!