# Design Patterns in Swift

Swift design patterns are intended to make it easier for programmers to write code. They assist developers in creating a simple and effective working environment. iOS design patterns are a set of repeatable techniques for creating apps. A design pattern is a template that tells you how to write code, but it's up to you to fit your code to this template.

Singleton is a design pattern that is very popular in development. Most of the developers are using this design pattern. This is very simple, common and **easy** to use in your project. It initializes your class instance single time only with **static** property and it will share your **class** instance **globally**.

**Here the one simple example of using class:**

```
class LocationManager{
    func requestForLocation(){
        print("Location granted")
    }

}

let location = LocationManager()
location.requestForLocation()
```

This is the class without a **singleton** pattern for access to any function we need to **initialize** class every time for avoiding these things we are using **singleton** classes with the **static** instance.

**Singleton Class:**

```
class LocationManager{
    static let shared = LocationManager()
    init(){}
    func requestForLocation(){
        print("Location granted")
    }
}
```

**Usage:**

```
LocationManager.shared.requestForLocation()
let location = LocationManager()
location.requestForLocation()
```

**Factory** is a creational design pattern which solves the problem of creating product objects without specifying their concrete classes. The Factory Method defines a method, which should be used for creating objects instead of using a direct constructor call (new operator). Subclasses can override this method to change the class of objects that will be created.

```
class PersonFactory {
    func createPerson() -> Person? {
        nil
    }
}

class DaveFactory: PersonFactory {
    override func createPerson() -> Person? {
        Person(name: "Dave")
    }
}
```

**Usage:**

```
let dave = DaveFactory().createPerson()
print (dave)
```

**Prototype** is a creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes. The prototype design pattern is

used to create clones of a base object.
This is also a creational design pattern, it is useful when you have a very basic configuration for an object and you'd like to give (clone) those predefined values to another one. Basically you're making clones from a prototype objects.

Imagine, we have a **SmartPhone** class as bellow:

```
class SmartPhone {

    var name: String
    var color: String
    var capacity: Int

    // Designed Initializer
    init(name: String, color: String, capacity: Int) {
        self.name = name
        self.color = color
        self.capacity = capacity
    }
}
```

Now we need to create 5 instances of this class look like this:

```
var iPhone7Gray16G    = SmartPhone(name: "iPhone 7", color: "Gray",
capacity: 16)
var iPhone7Gray64G    = SmartPhone(name: "iPhone 7", color: "Gray",
capacity: 64)
var iPhone7White64G   = SmartPhone(name: "iPhone 7", color: "White",
capacity: 64)
var samsungS8White64G = SmartPhone(name: "samsung S8", color:
"White", capacity: 64)
var samsungS8Blue64G  = SmartPhone(name: "samsung S8", color:
"Blue", capacity: 64)
```

We can easy to realize that many instances have the same properties so we often duplicate the code line then edit any properties differently. It’s okay but not optimized & easy to make mistakes. Let’s try to apply the **Prototype** pattern into **SmartPhone** class as bellow:

```
class SmartPhone {
    var name: String
    var manufacture: String
    var color: String
    var capacity: Int

    // Designed Initializer
    init(name: String, color: String, capacity: Int) {
        self.name = name
        self.color = color
        self.capacity = capacity

    }

    // Prototype pattern in action
    func clone(name: String? = nil, color: String? = nil, capacity: Int? = nil)
-> SmartPhone {
        let cloneName = name == nil ? self.name : name!
        let cloneColor = color == nil ? self.color : color!
        let cloneCapacity = capacity == nil ? self.capacity : capacity!

        return SmartPhone(name: cloneName, manufacture:
cloneManufacture, color: cloneColor, capacity: cloneCapacity)
    }
}
```

We declare a **clone** func with default optional parameters are the same with properties. This func returns a new instance by copying all of the properties of an existing instance then modifies some properties if needed. Now create 5 intances as bellow:

```
var iPhone7Gray16G    = SmartPhone(name: "iPhone 7", color: "Gray",
capacity: 16)
var iPhone7Gray64G    = iPhone7Gray16G.clone(capacity: 64)
var iPhone7White64G   = iPhone7Gray64G.clone(color: "White")
var samsungS8White64G = iPhone7White64G.clone(name: "Samsung
S8")
var samsungS8Blue64G  = samsungS8White64G.clone(color: "Blue")
```

## The **Adapter Pattern** is a design pattern that enables

objects with similar functionality to work together despite having incompatible interfaces.

Let’s see a common structure for an Adapter implementation:

```
protocol TargetProtocol {
    func doSomething()
}

class Adaptee {
    func doSomethingAdaptee() {
        print("doSomethingAdaptee")
    }
}

class Adapter: TargetProtocol {
    private let adaptee: Adaptee

    internal init(adaptee: Adaptee) {
        self.adaptee = adaptee
    }

    func doSomething() {
        adaptee.doSomethingAdaptee()
    }
}
```

**Facade** is a structural design pattern that provides a simplified (but limited) interface to a complex system of classes, library or framework. Facade can be recognized in a class that has a simple interface, but delegates most of the work to other classes. Usually, facades manage the full life cycle of objects they use.

```
class Engine {
    func produceEngine() {
        print("prodce engine")
    }
}

class Body {
    func produceBody() {
        print("prodce body")
    }
}

class Accessories {
    func produceAccessories() {
        print("prodce accessories")
    }
}
```

So we build a facade to provide a simple interface.

```
class FactoryFacade {
    let engine = Engine()
    let body = Body()
    let accessories = Accessories()

    func produceCar() {
        engine.produceEngine()
        body.produceBody()
        accessories.produceAccessories()
    }
}
```

**Usage:**

```
let factoryFacade = FactoryFacade()
factoryFacade.produceCar()
```

**Observer** is a behavioral design pattern that allows some objects to notify other objects about changes in their state.

```
protocol PropertyObserver : class {
    func willChange(propertyName: String, newPropertyValue: Any?)
    func didChange(propertyName: String, oldPropertyValue: Any?)
}

final class TestChambers {

    weak var observer:PropertyObserver?

    private let testChamberNumberName = "testChamberNumber"

    var testChamberNumber: Int = 0 {
        willSet(newValue) {
            observer?.willChange(propertyName:
testChamberNumberName, newPropertyValue: newValue)
        }
        didSet {
            observer?.didChange(propertyName:
testChamberNumberName, oldPropertyValue: oldValue)
        }
    }
}

final class Observer : PropertyObserver {
    func willChange(propertyName: String, newPropertyValue: Any?) {
        if newPropertyValue as? Int == 1 {
            print("Okay. Look. We both said a lot of things that you're going
to regret.")
        }
    }

    func didChange(propertyName: String, oldPropertyValue: Any?) {
        if oldPropertyValue as? Int == 0 {
            print("Sorry about the mess. I've really let the place go since you
killed me.")
        }
    }
}
```

**Usage:**

```
var observerInstance = Observer()
var testChambers = TestChambers()
testChambers.observer = observerInstance
testChambers.testChamberNumber += 1
```

**State** is a behavioral design pattern that allows an object to change the behavior when its internal state changes.The state pattern is a behavioral pattern that allows an object to change its behavior at runtime.

```
final class Context {
        private var state: State = UnauthorizedState()

    var isAuthorized: Bool {
        get { return state.isAuthorized(context: self) }
    }

    var userId: String? {
        get { return state.userId(context: self) }
    }

        func changeStateToAuthorized(userId: String) {
                state = AuthorizedState(userId: userId)
        }

        func changeStateToUnauthorized() {
                state = UnauthorizedState()
        }
}

protocol State {
        func isAuthorized(context: Context) -> Bool
        func userId(context: Context) -> String?
}

class UnauthorizedState: State {
        func isAuthorized(context: Context) -> Bool { return false }

        func userId(context: Context) -> String? { return nil }
}

class AuthorizedState: State {
        let userId: String

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

        func isAuthorized(context: Context) -> Bool { return true }

        func userId(context: Context) -> String? { return userId }
}
```

**Usage:**

```
let userContext = Context()
(userContext.isAuthorized, userContext.userId)
userContext.changeStateToAuthorized(userId: "admin")
(userContext.isAuthorized, userContext.userId) // now logged in as
"admin"
userContext.changeStateToUnauthorized()
(userContext.isAuthorized, userContext.userId)
```

**Decorator** is a structural pattern that allows adding new behaviors to objects dynamically by placing them inside special wrapper objects, called *decorators*.

```
protocol CostHaving {
    var cost: Double { get }
}

protocol IngredientsHaving {
    var ingredients: [String] { get }
}

typealias BeverageDataHaving = CostHaving & IngredientsHaving

struct SimpleCoffee: BeverageDataHaving {
    let cost: Double = 1.0
    let ingredients = ["Water", "Coffee"]
}

protocol BeverageHaving: BeverageDataHaving {
    var beverage: BeverageDataHaving { get }
}

struct Milk: BeverageHaving {

    let beverage: BeverageDataHaving

    var cost: Double {
        return beverage.cost + 0.5
    }

    var ingredients: [String] {
        return beverage.ingredients + ["Milk"]
    }
}

struct WhipCoffee: BeverageHaving {

    let beverage: BeverageDataHaving

    var cost: Double {
        return beverage.cost + 0.5
    }

    var ingredients: [String] {
        return beverage.ingredients + ["Whip"]
    }
}
```

**Usage:**

```
var someCoffee: BeverageDataHaving = SimpleCoffee()
print("Cost: \(someCoffee.cost); Ingredients: \(someCoffee.ingredients)")
someCoffee = Milk(beverage: someCoffee)
print("Cost: \(someCoffee.cost); Ingredients: \(someCoffee.ingredients)")
someCoffee = WhipCoffee(beverage: someCoffee)
```

```
print("Cost: \(someCoffee.cost); Ingredients: \(someCoffee.ingredients)")
```

**Proxy** is a structural design pattern that provides an object that acts as a substitute for a real service object used by a client. A proxy receives client requests, does some work (access control, caching, etc.) and then passes the request to a service object.While the Proxy pattern isn't a frequent guest in most Swift applications, it's still very handy in some special cases. It's irreplaceable when you want to add some additional behaviors to an object of some existing class without changing the client code.

```
protocol Car {
    func drive()
}class Sedan: Car {
    func drive() {
        print("drive a sedan")
    }
}

class AutonomousCar: Car {
    var car: Car    init(car: Car) {
        self.car = car
    }    func drive() {
        car.drive()
        print("by self-driving system")
    }
}
```

**Usage:**

```
let sedan = Sedan()
let autonomousCar = AutonomousCar(car: sedan)
autonomousCar.drive()
```

**Flyweight** is a structural design pattern that allows programs to support vast quantities of objects by keeping their memory consumption low.The pattern achieves it by sharing parts of object state between multiple objects. In other words, the Flyweight saves RAM by caching the same data used by different objects. The Flyweight pattern has a single purpose: minimizing memory intake. If your program doesn't struggle with a shortage of RAM, then you might just ignore this pattern for a while.

```
struct SpecialityCoffee {
    let origin: String
}

protocol CoffeeSearching {
    func search(origin: String) -> SpecialityCoffee?
}

// Menu acts as a factory and cache for SpecialityCoffee flyweight
objects
final class Menu: CoffeeSearching {

    private var coffeeAvailable: [String: SpecialityCoffee] = [:]

    func search(origin: String) -> SpecialityCoffee? {
        if coffeeAvailable.index(forKey: origin) == nil {
            coffeeAvailable[origin] = SpecialityCoffee(origin: origin)
        }

        return coffeeAvailable[origin]
    }
}

final class CoffeeShop {
    private var orders: [Int: SpecialityCoffee] = [:]
    private let menu: CoffeeSearching

    init(menu: CoffeeSearching) {
        self.menu = menu
    }

    func takeOrder(origin: String, table: Int) {
        orders[table] = menu.search(origin: origin)
    }

    func serve() {
        for (table, origin) in orders {
            print("Serving \(origin) to table \(table)")
        }
    }
}

let coffeeShop = CoffeeShop(menu: Menu())

coffeeShop.takeOrder(origin: "Yirgacheffe, Ethiopia", table: 1)
coffeeShop.takeOrder(origin: "Buziraguhindwa, Burundi", table: 3)

coffeeShop.serve()
```

**Iterator** is a behavioral design pattern that allows sequential traversal through a complex data structure without exposing its internal details. **As the name suggests, the pattern enables you to iterate over a collection of elements.** The iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure.

```
struct Novella {
    let name: String
}
```

```
struct Novellas {
    let novellas: [Novella]
}

struct NovellasIterator: IteratorProtocol {

    private var current = 0
    private let novellas: [Novella]

    init(novellas: [Novella]) {
        self.novellas = novellas
    }

    mutating func next() -> Novella? {
        defer { current += 1 }
        return novellas.count > current ? novellas[current] : nil
    }
}

extension Novellas: Sequence {
    func makeIterator() -> NovellasIterator {
        return NovellasIterator(novellas: novellas)
    }
}
```

**Usage:**

```
let greatNovellas = Novellas(novellas: [Novella(name: "The Mist")] )

for novella in greatNovellas {
    print("I've read: \(novella)")
}
```

**Chain of Responsibility** is behavioral design pattern that allows passing request along the chain of potential handlers until one of them handles request. The chain-of-responsibility pattern is a behavioral design pattern that allows an event to be processed by one of many handlers.

**Example:**

```
protocol Withdrawing {
    func withdraw(amount: Int) -> Bool
}

final class MoneyPile: Withdrawing {

    let value: Int
    var quantity: Int
    var next: Withdrawing?

    init(value: Int, quantity: Int, next: Withdrawing?) {
        self.value = value
        self.quantity = quantity
        self.next = next
    }

    func withdraw(amount: Int) -> Bool {

        var amount = amount

        func canTakeSomeBill(want: Int) -> Bool {
            return (want / self.value) > 0
        }

        var quantity = self.quantity

        while canTakeSomeBill(want: amount) {

            if quantity == 0 {
                break
            }

            amount -= self.value
            quantity -= 1
        }

        guard amount > 0 else {
            return true
        }

        if let next = self.next {
            return next.withdraw(amount: amount)
        }

        return false
    }
}

final class ATM: Withdrawing {

    private var hundred: Withdrawing
    private var fifty: Withdrawing
    private var twenty: Withdrawing
    private var ten: Withdrawing

    private var startPile: Withdrawing {
        return self.hundred
    }

    init(hundred: Withdrawing,
           fifty: Withdrawing,
          twenty: Withdrawing,
             ten: Withdrawing) {

        self.hundred = hundred
        self.fifty = fifty
        self.twenty = twenty
        self.ten = ten
    }

    func withdraw(amount: Int) -> Bool {
        return startPile.withdraw(amount: amount)
    }
}
```

**Usage:**

```
// Create piles of money and link them together 10 < 20 < 50 < 100.**
let ten = MoneyPile(value: 10, quantity: 6, next: nil)
let twenty = MoneyPile(value: 20, quantity: 2, next: ten)
let fifty = MoneyPile(value: 50, quantity: 2, next: twenty)
let hundred = MoneyPile(value: 100, quantity: 1, next: fifty)

// Build ATM.
var atm = ATM(hundred: hundred, fifty: fifty, twenty: twenty, ten: ten)
atm.withdraw(amount: 310) // Cannot because ATM has only 300
atm.withdraw(amount: 100) // Can withdraw - 1x100
```