Chapter 52: Design Patterns - Creational

Design patterns are general solutions to problems that frequently occur in software development. The following are templates of standardized best practices in structuring and designing code, as well as examples of common contexts in which these design patterns would be appropriate.

Creational design patterns abstract the instantiation of objects to make a system more independent of the process of creation, composition, and representation.

Section 52.1: Singleton

Singletons are a frequently used design pattern which consists of a single instance of a class that is shared throughout a program.

In the following example, we create a `static` property that holds an instance of the `Foo` class. Remember that a `static` property is shared between all objects of a class and can't be overwritten by subclassing.

```
public class Foo
{
    static let shared = Foo()

    // Used for preventing the class from being instantiated directly
    private init() {}

    func doSomething()
    {
        print("Do something")
    }
}
```

Usage:

```
Foo.shared.doSomething()
```

Be sure to remember the `private` initializer:

> This makes sure your singletons are truly unique and prevents outside objects from creating their own instances of your class through virtue of access control. Since all objects come with a default public initializer in Swift, you need to override your init and make it private. KrakenDev

Section 52.2: Builder Pattern

> The builder pattern is an **object creation software design pattern**. Unlike the abstract factory pattern and the factory method pattern whose intention is to enable polymorphism, the intention of the builder pattern is to find a solution to the telescoping constructor anti-pattern. The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors. Instead of using numerous constructors, the builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once.
>
> *-Wikipedia*

The main goal of the builder pattern is to setup a default configuration for an object from its creation. It is an intermediary between the object will be built and all other objects related to building it.

Example:

To make it more clear, let's take a look at a *Car Builder* example.

Consider that we have a *Car* class contains many options to create an object, such as:

- Color.
- Number of seats.
- Number of wheels.
- Type.
- Gear type.
- Motor.
- Airbag availability.

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

class Car: CustomStringConvertible {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability: \(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
```

```
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}
```

Creating a car object:

```
let aCar = Car(color: UIColor.black,
               numberOfSeats: 4,
               numberOfWheels: 4,
               type: .saloon,
               gearType: .automatic,
               motor: Motor(id: "101", name: "Super Motor",
                            model: "c4", numberOfCylinders: 6),
               shouldHasAirbags: true)

print(aCar)

/* Printing
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 4
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "101", name: "Super Motor", model: "c4", numberOfCylinders: 6)
 Airbag Availability: true
*/
```

The **problem** arises when creating a car object is that the car requires many configuration data to be created.

For applying the Builder Pattern, the initializer parameters should have default values *which are changeable if needed*.

CarBuilder class:

```
class CarBuilder {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

The `CarBuilder` class defines properties that could be changed to to edit the values of the created car object.

Let's build new cars by using the `CarBuilder`:

```
var builder = CarBuilder()
// currently, the builder creates cars with default configuration.
```

```
let defaultCar = builder.buildCar()
//print(defaultCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
*/

builder.shouldHasAirbags = true
// now, the builder creates cars with default configuration,
// but with a small edit on making the airbags available

let safeCar = builder.buildCar()
print(safeCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
 */

builder.color = UIColor.purple
// now, the builder creates cars with default configuration
// with some extra features: the airbags are available and the color is purple

let femaleCar = builder.buildCar()
print(femaleCar)
/* prints
 color: UIExtendedSRGBColorSpace 0.5 0 0.5 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
*/
```

The **benefit** of applying the Builder Pattern is the ease of creating objects that should contain much of configurations by setting default values, also, the ease of changing these default values.

Take it Further:

As a good practice, all properties that need default values should be in a *separated protocol*, which should be implemented by the class itself and its builder.

Backing to our example, let's create a new protocol called `CarBluePrint`:

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
```

```
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }
}

class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability: \(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
```

```
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

The benefit of declaring the properties that need default value into a protocol is the forcing to implement any new added property; When a class conforms to a protocol, it has to declare all its properties/methods.

Consider that there is a required new feature that should be added to the blueprint of creating a car called "battery name":

```
protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }

    // adding the new property
    var batteryName: String { get set }
}
```

After adding the new property, note that two compile-time errors will arise, notifying that conforming to `CarBluePrint` protocol requires to declare 'batteryName' property. That guarantees that `CarBuilder` will declare and set a default value for `batteryName` property.

After adding `batteryName` new property to `CarBluePrint` protocol, the implementation of both `Car` and `CarBuilder` classes should be:

```
class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool
    var batteryName: String

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\nType: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)\nBattery Name: \(batteryName)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool, batteryName: String) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags
        self.batteryName = batteryName
    }
```

```
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.red
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false
    var batteryName: String = "Default Battery Name"

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags, batteryName:
batteryName)
    }
}
```

Again, let's build new cars by using the `CarBuilder`:

```
var builder = CarBuilder()

let defaultCar = builder.buildCar()
print(defaultCar)
/* prints
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: Default Battery Name
*/

builder.batteryName = "New Battery Name"

let editedBatteryCar = builder.buildCar()
print(editedBatteryCar)
/*
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: New Battery Name
 */
```

Section 52.3: Factory Method

> In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. Wikipedia reference

```
protocol SenderProtocol
{
```

```
    func send(package: AnyObject)
}

class Fedex: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Fedex deliver")
    }
}

class RegularPriorityMail: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Regular Priority Mail deliver")
    }
}

// This is our Factory
class DeliverFactory
{
    // It will be responsable for returning the proper instance that will handle the task
    static func makeSender(isLate isLate: Bool) -> SenderProtocol
    {
        return isLate ? Fedex() : RegularPriorityMail()
    }
}

// Usage:
let package = ["Item 1", "Item 2"]

// Fedex class will handle the delivery
DeliverFactory.makeSender(isLate:true).send(package)

// Regular Priority Mail class will handle the delivery
DeliverFactory.makeSender(isLate:false).send(package)
```

By doing that we don't depend on the real implementation of the class, making the `sender()` completely transparent to who is consuming it.

In this case all we need to know is that a sender will handle the deliver and exposes a method called `send()`. There are several other advantages: reduce classes coupling, easier to test, easier to add new behaviours without having to change who is consuming it.

> Within object-oriented design, interfaces provide layers of abstraction that facilitate conceptual explanation of the code and create a barrier preventing dependencies.Wikipedia reference

Section 52.4: Observer

> The observer pattern is where an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern.Wikipedia reference

Basically the observer pattern is used when you have an object which can notify observers of certain behaviors or state changes.

First lets create a global reference (outside of a class) for the Notification Centre :

```
let notifCentre = NotificationCenter.default
```

Great now we can call this from anywhere. We would then want to register a class as an observer...

```
notifCentre.addObserver(self, selector: #selector(self.myFunc), name: "myNotification", object:
nil)
```

This adds the class as an observer for "readForMyFunc". It also indicates that the function myFunc should be called when that notification is received. This function should be written in the same class:

```
func myFunc(){
    print("The notification has been received")
}
```

> One of the advantages to this pattern is that you can add many classes as observers and thus perform many actions after one notification.

The notification can now simply be sent(or posted if you prefer) from almost anywhere in the code with the line:

```
notifCentre.post(name: "myNotification", object: nil)
```

> You can also pass information with the notification as a Dictionary

```
let myInfo = "pass this on"
notifCentre.post(name: "myNotification", object: ["moreInfo":myInfo])
```

But then you need to add a notification to your function:

```
func myFunc(_ notification: Notification){
    let userInfo = (notification as NSNotification).userInfo as! [String: AnyObject]
    let passedInfo = userInfo["moreInfo"]
    print("The notification \(moreInfo) has been received")
    //prints - The notification pass this on has been received
}
```

Section 52.5: Chain of responsibility

> In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of `command` objects and a series of `processing` objects. Each `processing` object contains logic that defines the types of command objects that it can handle; the rest are passed to the next `processing` object in the chain. A mechanism also exists for adding new `processing` objects to the end of this chain. Wikipedia

Setting up the classes that made up the chain of responsibility.

First we create an interface for all the `processing` objects.

```
protocol PurchasePower {
var allowable : Float { get }
  var role : String { get }
  var successor : PurchasePower? { get set }
}

extension PurchasePower {
  func process(request : PurchaseRequest){
    if request.amount < self.allowable {
      print(self.role + " will approve $ \(request.amount) for \(request.purpose)")
    } else if successor != nil {
      successor?.process(request: request)
    }
  }
}
```

Then we create the `command` object.

```
struct PurchaseRequest {
  var amount : Float
  var purpose : String
}
```

Finally, creating objects that made up the chain of responsibility.

```
class ManagerPower : PurchasePower {
  var allowable: Float = 20
  var role : String = "Manager"
  var successor: PurchasePower?
}

class DirectorPower : PurchasePower {
  var allowable: Float = 100
  var role = "Director"
  var successor: PurchasePower?
}

class PresidentPower : PurchasePower {
  var allowable: Float = 5000
  var role = "President"
  var successor: PurchasePower?
}
```

Initiate and chaining it together :

```
let manager = ManagerPower()
let director = DirectorPower()
let president = PresidentPower()

manager.successor = director
director.successor = president
```

> The mechanism for chaining up objects here is property access

Creating request to run it :

```
manager.process(request: PurchaseRequest(amount: 2, purpose: "buying a pen"))  // Manager will
```

```
approve $ 2.0 for buying a pen
manager.process(request: PurchaseRequest(amount: 90, purpose: "buying a printer")) // Director will
approve $ 90.0 for buying a printer

manager.process(request: PurchaseRequest(amount: 2000, purpose: "invest in stock")) // President
will approve $ 2000.0 for invest in stock
```

Section 52.6: Iterator

> In computer programming an iterator is an object that enables a programmer to traverse a container, particularly lists. Wikipedia

```
struct Turtle {
  let name: String
}

struct Turtles {
  let turtles: [Turtle]
}

struct TurtlesIterator: IteratorProtocol {
  private var current = 0
  private let turtles: [Turtle]

  init(turtles: [Turtle]) {
    self.turtles = turtles
  }

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

extension Turtles: Sequence {
  func makeIterator() -> TurtlesIterator {
    return TurtlesIterator(turtles: turtles)
  }
}
```

And usage example would be

```
let ninjaTurtles = Turtles(turtles: [Turtle(name: "Leo"),
                                     Turtle(name: "Mickey"),
                                     Turtle(name: "Raph"),
                                     Turtle(name: "Doney")])
print("Splinter and")
for turtle in ninjaTurtles {
  print("The great: \(turtle)")
}
```