Хрестоматия Swift паттернов. На всякий...

v.2.0

Дима Малеев и Сережа Суханов

Оглавление

Привет

Привет, Друг! Очень мило, что ты решил прочитать эту книгу. Ну, или просто скачал, чтобы посмотреть, что тут находится. А находятся здесь просто примеры реализации паттернов GoF для iOS. Так как примеры писаны на Swift, вероятнее всего их можно использовать и для Mac, но так как я круче программы, чем “Hello, World!” под Mac не писал – то утверждать не могу.

Что тебя ждет дальше? Невероятно, но паттерны! И вероятнее всего – грамматические и орфографические ошибки. Хоть это и вторая версия книги (исправленная и доработанная), в которой были исправлены старые ошибки, в ней наверняка появились и новые ☺. Поэтому я торжественно клянусь, что исправлю каждую ошибку, которую ты мне пришлешь на ssuhanov@gmail.com. Я очень надеюсь, что ты не граммар-наци и поможешь мне исправить все их. И, конечно же, оценишь те знания, которыми я пытался поделиться с тобой.

Для кого эта книга?

Книга будет полезна всем Swift разработчикам, потому как книги просто полезно читать. Говорят, это улучшает память. Для начинающих разработчиков, можно будет прочитать про паттерны и примеры их реализации, в будущем про это вас спросят на собеседовании. Для более продвинутых ребят, книга может послужить небольшой напоминалкой про забытое описание паттернов. Книга однозначно не являет собой учебник по Swift, и прочитав ее вы вряд ли сможете написать второй Instagram, однако некоторые проблемы она вам все же решить поможет. Паттерны вообще хорошо знать, чтобы структурировать свои знания в голове. ☺

Почему эта книга?

По моему глубокому личному убеждению – знания должны быть бесплатны. Просто представьте, как далеко было бы человечество, если бы у нас культура была направлена не на бесконечное зарабатывание денег, а на продвижение человечества к звездам. К сожалению, я не фармацевт, который придумал лекарство от рака, и не ученый, который придумал телепортацию. ☺

Поэтому, я попытаюсь поделиться знаниями, которые есть у меня. Что меня радует – я далеко не первый и не второй, и даже решение о написании этой книги пришло мне в голову из-за успеха Андрея Будая с его книжкой (http://andriybuday.com/book), которую я вам очень рекомендую почитать. Внимательный читатель увидит, что книги сами по себе очень похожи, разве что язык примеров другой (ну и сами примеры, кроме одного).

Распространение книги

Книга бесплатна. ☺
Глупо писать другое, особенно если вы прочитали предыдущий пункт. Распространяться книга будет путем скачивания откуда угодно. Потому, просто пару правил:

1. Книга не должна продаваться. Книга может только бесплатно распространяться в любом виде, но за просто так. Совсем.

2. Книгу можно перепечатывать, копировать себе в блог, отсылать голубиной почтой и даже менять имя автора. Если вспомните вашего покорного слугу – респект вам и уважуха, если нет – это тоже отлично, наверное, у вас была причина.

3. Естественно, автор не несет ответственности за те знания, которые вы тут получили. ☺ Ну серьезно – делайте добро, вы ж не политики.

4. Вы можете дописывать книгу, изменять в любом месте, но тогда будьте добры, меняйте автора.

5. Если вы поменяли контент книги, даже одну буковку – это уже ваша книга. ☺ Наслаждайтесь! Вы стали автором! Во второй версии книги именно так и произошло. ☺

6. Кстати да, менять можно все, кроме этих простых правил.

7. После прочтения книги задумайтесь, какими знаниями можете поделиться вы. Вероятнее всего, вы можете рассказать уникальные и интересные вещи, которые поменяют что-то в этом мире! Пользуйтесь! Это ваша суперсила!

Напутственное слово

Читайте.

История изменений

Книга же не бумажная, поэтому имеет возможность эволюционировать и меняться!

v.1.0 – базовый контент книги
v.1.1 – «подкрашен» код, немного пофиксаны ошибки, книга выложена в опен сорс
v.2.0 – описания паттернов дополнены и в них исправлены ошибки, примеры кода написаны на Swift.

Prototype

Прототип – один из самых простых паттернов, который позволяет нам получить точную копию необходимого объекта. То есть – использовать как прототип для нового объекта.

Когда использовать:

1. У нас есть семейство схожих объектов, разница между которыми только в состоянии их полей.
2. Чтобы создать объект вам надо пройти через огонь, воду и медные трубы. Особенно если этот объект состоит из еще одной кучи объектов, многие из которых для заполнения требуют подгрузки данных из базы, веб-сервисов и подобных источников. Часто легче скопировать объект и поменять в нем несколько полей.
3. Да и в принципе, нам особо и не важно, как создается объект. Ну есть и есть.
4. Нам страшно лень писать иерархию фабрик (читай дальше), которые будут инкапсулировать всю противную работу создания объекта.

Да, и есть еще частое заблуждение (вероятнее всего из названия), что прототип – это архетип, который никогда не должен использоваться, и служит только для создания себе подобных объектов. Хотя, прототип, как и архетип – тоже достаточно популярный кейс. Собственно, нам ничего не мешает делать прототипом любой объект, который у нас в подчинении.

Поверхностное и глубокое копирование
Тут нет особой разницы с другими языками программирования. Есть указатель, есть значение в куче.

Поверхностное копирование – это просто создание нового на те же самые байты в куче. То есть – в результате мы можем получить два объекта, которые указывают на одно и то же значение.

К примеру создадим объект:

```
public class Person {
    public var name: String
    public var surname: String
    public var age: Int

    public init(name: String,
                surname: String,
                age: Int) {

        self.name = name
        self.surname = surname
        self.age = age
    }
}
```

А теперь давайте просто создадим два объекта и посмотрим что же получится:

```
let firstPerson = Person(name: "Adam",
                         surname: "Adams",
                         age: 32)
print("First Person:\(firstPerson.name) \(firstPerson.surname)")

let secondPerson = firstPerson
secondPerson.name = "Bobby"

print("Second Person: \(secondPerson.name) \(secondPerson.surname)")
print("First Person: \(firstPerson.name) \(firstPerson.surname)")
```

Как видим лог достаточно ожидаемый:

```
First Person: Adam Adams
Second Person: Bobby Adams
First Person: Bobby Adams
```

Заметьте, что хоть и меняли мы имя для secondPerson, но и у firstPerson имя поменялось. Просто потому что мы создали два указателя на один и тот же объект.

Для таких задач, стоит использовать глубокое копирование, которое в Swift сделано в принципе очень похоже как и в .NET:

Для этого надо реализовать протокол NSCopying, и перегрузить:

```
public func copy(with: NSZone? = nil) -> Any
```

Расширим наш класс:

```
extension Person: NSCopying {
    public func copy(with: NSZone? = nil) -> Any {
        let copy = Person(name: self.name,
                          surname: self.name,
                          age: self.age)
        return copy
    }
}
```

И немного изменим код нашего тестового приложения:

```
let firstPerson = Person(name: "Adam",
                         surname: "Adams",
                         age: 32)
print("First Person:\(firstPerson.name) \(firstPerson.surname)")

let secondPerson = firstPerson.copy() as! Person
secondPerson.name = "Bobby"

print("Second Person: \(secondPerson.name) \(secondPerson.surname)")
print("First Person: \(firstPerson.name) \(firstPerson.surname)")
```

Ну и, естественно, лог:

```
First Person: Adam Adams
Second Person: Bobby Adam
First Person: Adam Adams
```

Как видим, мы в результате получили два независимых объекта, один из которых сделан по подобию первого.

Код примера.

Factory Method

Еще один порождающий паттерн, довольно прост и популярен. Паттерн позволяет переложить создание специфических объектов, на наследников родительского класса, потому можно манипулировать объектами на более высоком уровне, не заморачиваясь объект какого класса будет создан. Частенько этот паттерн называют виртуальный конструктор, что по моему мнению лучше выражает его предназначение.

Когда использовать:

1. Мы не до конца уверены объект какого типа нам необходим.
2. Мы хотим чтобы не родительский объект решал какой тип создавать, а его наследники.

Почему хорошо использовать:

Объекты, созданные фабричным методом – схожи, потому как у них один и тот же родительский объект. Потому, если локализировать создание таких объектов, то можно добавлять новые типы, не меняя при этом код, который использует фабричный метод.

Пример:

Давайте представим, что мы такой неправильный магазин, в котором тип товара определяется по его цене. ☺ На данный момент товары есть двух типов: Игрушки и Одежда.

В чеке мы получаем только цены, и нам надо сохранить объекты, которые куплены.

Для начала, создадим протокол Product. В нем будут два свойства: имя и цена, и метод сохранения (для примера):

```
public protocol Product {
    var name: String { get }
    var price: Int { get }

    func saveObject()
}

extension Product {
    func saveObject() {
        print("I am saving the object into product database")
    }
}
```

Теперь создадим две реализации этого протокола.

Игрушка:

```
public class Toy: Product {
    public var name: String
    public var price: Int

    public init(name: String, price: Int) {
        self.name = name
        self.price = price
    }

    public func saveObject() {
        print("Saving object into Toys database")
    }
}
```

И одежда:

```
public class Dress: Product {
    public var name: String
    public var price: Int

    public init(name: String, price: Int) {
        self.name = name
        self.price = price
    }

    public func saveObject() {
        print("Saving object into Dress database")
    }
}
```

Теперь мы, практически, подошли вплотную к нашему паттерну. Собственно, теперь нужно создать метод, который будет по цене определять, что же за продукт у нас в чеке и создавать объект необходимого типа.

```
public class ProductGenerator {
    public init() { }

    public func getProduct(price: Int) -> Product? {
        if (0..<100).contains(price) {
            return Toy(name: "Teddy Bear", price: price)
        } else if price >= 100 {
            return Dress(name: "Little Black", price: price)
        } else {
            return nil
        }
    }
}
```

Вот собственно, все. Теперь добавим метод, который будет считать и записывать расходы:

```
func saveExpenses(price: Int) {
    let productGenerator = ProductGenerator()
    let expense = productGenerator.getProduct(price: price)
    expense?.saveObject()
}
```

Пробуем:

```
saveExpenses(price: 50)
saveExpenses(price: 56)
saveExpenses(price: 79)
saveExpenses(price: 100)
saveExpenses(price: 123)
saveExpenses(price: 51)
```

Лог:

```
Saving object into Toys database
Saving object into Toys database
Saving object into Toys database
Saving object into Dress database
Saving object into Dress database
Saving object into Toys database
```

Код примера.

Abstract Factory

Абстрактная фабрика – еще один очень популярный паттерн, который как в названии, так и в реализации слегка похож на фабричный метод.

Итак, что же делает абстрактная фабрика:
Абстрактная фабрика дает простой интерфейс для создания объектов, которые принадлежат к тому или иному семейству объектов.

Отличия от фабричного метода:

1. Фабричный метод порождает объекты одного и того же типа, фабрика же может создавать независимые объекты.
2. Чтобы добавить новый тип объекта нужно поменять интерфейс фабрики, а в абстрактной фабрике легче менять внутренности метода, который отвечает за порождение объектов.

Давайте представим ситуацию: у нас есть две фабрики по производству IPhone и IPad. Одна – оригинальная, компании Apple, а другая – хижина дядюшки Хуа. И вот, мы хотим производить эти товары: если в страны третьего мира – то товар от дядюшки, в остальные страны – товар любезно предоставлен компанией Apple.

Пусть у нас есть фабрика, которая умеет производить и айпэды и айфоны:

```
public protocol DeviceFactory {
    func getIPhone() -> GenericIPhone
    func getIPad() -> GenericIPad
}
```

Естественно, нам необходимо реализовать продукты, которые фабрика будет производить:

```
public protocol GenericIPad {
    var osName: String { get }
    var productName: String { get }
    var screenSize: Float { get }
}

public protocol GenericIPhone {
    var osName: String { get }
    var productName: String { get }
}
```

Но продукты немного отличаются. Пусть у нас есть оригинальные продукты Apple:

```
public class AppleIPhone: GenericIPhone {
    public let osName: String
    public let productName: String

    public init() {
        self.osName = "iOS"
        self.productName = "iPhone"
    }
}

public class AppleIPad: GenericIPad {
    public let osName: String
    public let productName: String
    public let screenSize: Float

    public init() {
        self.osName = "iOS"
        self.productName = "iPad"
        self.screenSize = 7.7
    }
}
```

И продукты, которые произведены трудолюбивым дядушкой Хуа:

```
public class ChinaPhone: GenericIPhone {
    public let osName: String
    public let productName: String

    public init() {
        self.osName = "Android"
        self.productName = "Chi Huan Hua Phone"
    }
}

public class ChinaPad: GenericIPad {
    public let osName: String
    public let productName: String
    public let screenSize: Float

    public init() {
        self.osName = "Android 8.0"
        self.productName = "Buan Que iPado Killa"
        self.screenSize = 12.5
    }
}
```

Разные телефоны, конечно же, производят на различных фабриках, добавим фабрику Apple:

```
public class AppleFactory: DeviceFactory {
    public func getIPhone() -> GenericIPhone {
        let iphone = AppleIPhone()
        return iphone
    }

    public func getIPad() -> GenericIPad {
        let ipad = AppleIPad()
        return ipad
    }
}
```

И фабрика нашего китайского дядюшки:

```
public class ChinaFactory: DeviceFactory {
    public func getIPhone() -> GenericIPhone {
        let phone = ChinaPhone()
        return phone
    }

    public func getIPad() -> GenericIPad {
        let pad = ChinaPad()
        return pad
    }
}
```

Интерфейсы у фабрик одинаковые, а девайсы у них получаются разные ☺.

Вот, собственно, и все. Мы готовы к демонстрации. Теперь давайте напишем небольшой метод, который будет возвращать нам фабрику, которую мы хотим (и тут у нас все таки будет фабричный метод):

```
func getFactory(isThirdWorld: Bool) -> DeviceFactory {
    if isThirdWorld {
        return ChinaFactory()
    } else {
        return AppleFactory()
    }
}
```

Теперь создадим девайсы:

```
let factory = getFactory(isThirdWorld: false)
let iphone = factory.getIPhone()
let ipad = factory.getIPad()

print("iPhone named: \(iphone.productName) with OS: \(iphone.osName)")
print("iPad named: \(ipad.productName) with OS: \(ipad.osName) and
screensize: \(ipad.screenSize)")
```

Посмотрим в лог:

```
iPhone named: iPhone with OS: iOS
iPad named: iPad with OS: iOS and screensize: 7.7
```

А теперь передадим в метод `getFactory` – `true`:

```
let factory = getFactory(isThirdWorld: true)
```

Лог будет совсем другим:

```
iPhone named: Chi Huan Hua Phone with OS: Android
iPad named: Buan Que iPado Killa with OS: Android 8.0 and screensize: 12.5
```

Код примера.

Builder

Вот представьте, что у нас есть фабрика. Но в отличие от фабрики из предыдущего поста, она умеет создавать только телефоны на базе андроида, и еще при этом различной конфигурации. То есть – есть один объект, но при этом его состояние может быть совершенно разным, а еще представьте, если его очень трудно создавать, и во время создания этого объекта еще и создается миллион дочерних объектов. Именно в такие моменты нам очень помогает такой паттерн как строитель.

Когда использовать:

1. Создание сложного объекта.
2. Процесс создания объекта тоже очень нетривиальный – к примеру: получение данных из базы и манипуляция ими.

Сам паттерн состоит из двух компонент: Builder и Director. Builder занимается именно построением объекта, а Director – знает какой Builder использовать, чтобы выдать необходимый продукт. Приступим!

Пусть у нас есть телефон, который обладает следующими свойствами:

```
public class AndroidPhone {
    public init() { }

    public var osVersion: String?
    public var name: String?
    public var cpuCodeName: String?
    public var RAMsize: Int?
    public var osVersionCode: Double?
    public var launcher: String?
}
```

Создадим абстрактного строителя:

```
public protocol AndroidPhoneBuilder {
     var phone: AndroidPhone { get }

    func setOSVersion() -> Self
    func setName() -> Self
    func setCPUCodeName() -> Self
    func setRAMSize() -> Self
    func setOSVersionCode() -> Self
    func setLauncher() -> Self

    func getPhone() -> AndroidPhone
}
```

Добавим реализацию метода getPhone по-умолчанию:

```
extension AndroidPhoneBuilder {
    public func getPhone() -> AndroidPhone {
        return self.phone
    }
}
```

А теперь добавим несколько реализаций строителей. Так будет выглядеть строитель для дешевого телефона:

```
public class LowPricePhoneBuilder: AndroidPhoneBuilder {
    public let phone: AndroidPhone

    public init() {
        self.phone = AndroidPhone()
    }

    public func setOSVersion() -> Self {
        self.phone.osVersion = "Android 4.1"
        return self
    }

    public func setName() -> Self {
        self.phone.name = "Low price phone"
        return self
    }

    public func setCPUCodeName() -> Self {
        self.phone.cpuCodeName = "Some shitty CPU"
        return self
    }

    public func setRAMSize() -> Self {
        self.phone.RAMsize = 256
        return self
    }

    public func setOSVersionCode() -> Self {
        self.phone.osVersionCode = 3.0
        return self
    }

    public func setLauncher() -> Self {
        self.phone.launcher = "Hia Tsung"
        return self
    }
}
```

А вот так строительство дорогого телефона:

```
public class HighPricePhoneBuilder: AndroidPhoneBuilder {
    public let phone: AndroidPhone

    public init() {
        self.phone = AndroidPhone()
    }

    public func setOSVersion() -> Self {
        self.phone.osVersion = "Android 9.0"
        return self
    }

    public func setName() -> Self {
        self.phone.name = "High price phone"
        return self
    }

    public func setCPUCodeName() -> Self {
        self.phone.cpuCodeName = "Some shitty but expensive CPU"
        return self
    }

    public func setRAMSize() -> Self {
        self.phone.RAMsize = 8192
        return self
    }

    public func setOSVersionCode() -> Self {
        self.phone.osVersionCode = 9.0
        return self
    }

    public func setLauncher() -> Self {
        self.phone.launcher = "Samsung Launcher"
        return self
    }
}
```

Кто-то же должен использовать строителей, поэтому давайте создадим объект, который будет с помощью строителей создавать разные телефоны:

```
public class FactorySalesMan {
    var builder: AndroidPhoneBuilder

    public init(builder: AndroidPhoneBuilder) {
        self.builder = builder
    }

    public func update(builder: AndroidPhoneBuilder) {
        self.builder = builder
    }

    public func getPhone() -> AndroidPhone {
        return self.builder.getPhone()
    }

    public func constructPhone() {
        self.builder
            .setOSVersion()
            .setName()
            .setCPUCodeName()
            .setRAMSize()
            .setOSVersionCode()
            .setLauncher()
    }
}
```

Ну и, конечно, куда же мы без теста и кода:

```
let cheapPhoneBuilder = LowPricePhoneBuilder()
let expensivePhoneBuilder = HighPricePhoneBuilder()

let salesMan = FactorySalesMan(builder: cheapPhoneBuilder)
salesMan.constructPhone()
var phone = salesMan.getPhone()
print("\n")
print("Phone name: \(phone.name!)")
print("OS version: \(phone.osVersion!)")
print("CPU code name: \(phone.cpuCodeName!)")
print("RAM size: \(phone.RAMsize!)")
print("OS version code: \(phone.osVersionCode!)")
print("Launcher: \(phone.launcher!)")

salesMan.update(builder: expensivePhoneBuilder)
salesMan.constructPhone()
phone = salesMan.getPhone()
print("\n")
print("Phone name: \(phone.name!)")
print("OS version: \(phone.osVersion!)")
print("CPU code name: \(phone.cpuCodeName!)")
print("RAM size: \(phone.RAMsize!)")
print("OS version code: \(phone.osVersionCode!)")
print("Launcher: \(phone.launcher!)")
```

Мы создали различных строителей и, сказав директору (FactorySalesMan) какого из них хотим использовать, получили нужный нам девайс.

Лог выглядит так:

```
Phone name: Low price phone
OS version: Android 4.1
CPU code name: Some shitty CPU
RAM size: 256
OS version code: 3.0
Launcher: Hia Tsung


Phone name: High price phone
OS version: Android 9.0
CPU code name: Some shitty but expensive CPU
RAM size: 8192
OS version code: 9.0
Launcher: Samsung Launcher
```

Код примера.

Singleton

Кто вообще бы мог подумать, что Singleton такой не самый простой паттерн в iOS? На самом деле, непростым он был раньше в Objective-C, но с появлением Swift, реализация этого паттерна стала самой простой из всех языков программирования. На собеседованиях на Swift-разработчика иногда задают вопрос: сколько строк кода нужно, чтоб написать Singleton на Swift. И большинство программистов с радостью неправильно отвечают, что одна. Давайте разберемся детально. ☺

Начнем с описания: Singleton – это такой объект, который существует в единственном экземпляре на весь жизненный цикл приложения. Часто используется для хранения глобальных значений, например настроек приложения.

Итак, напишем первую «однострочную» версию синглтона в Swift:

```
public class SingletonObject {
    public static let shared = SingletonObject()
}
```

Такая реализация решает почти все проблемы, которые возникают в других языках. Статическая константа гарантирует нам, что значение в нее будет записано только один раз. Нам не нужно проверять был ли объект создан ранее, также не нужно контролировать работу нескольких потоков по доступу к этому значению. Прям идиллия (на самом деле нет). Почему? Потому что никто не мешает вам в любом месте своего приложения сделать вот так:

```
let myObject1 = SingletonObject()
let myObject2 = SingletonObject()
```

Мы только что, легким движением руки, создали еще два объекта нашего «синглтон»-класса. А значит, что реализация «в одну строку» совсем не гарантирует нам один-единственный объект. Решение, как всегда на поверхности, и это – приватный инициализатор:

```
public class SingletonObject {
    private init() { }
    public static let shared = SingletonObject()
}
```

Вот теперь – это настоящий Singleton. Давайте добавим в него еще одну переменную для теста:

```
public class SingletonObject {
    private init() { }
    public static let shared = SingletonObject()

    public var tempProperty: String?
}
```

И напишем немного кода, который покажет как этим всем пользоваться:

```
print(SingletonObject.shared.tempProperty)
SingletonObject.shared.tempProperty = "Hello. How are you doing?"
print(SingletonObject.shared.tempProperty)
```

Получаем вполне ожидаемый лог:

```
nil
Optional("Hello. How are you doing?")
```

Код примера.

Adapter

Тяжело найти более красочное описание паттерна Адаптер, чем пример из жизни каждого, кто покупал технику из США. Розетка! Вот почему не сделать одинаковую розетку всюду? Но нет, в США розетка с квадратными дырками, в Европе - с круглыми, а в некоторых странах - вообще с треугольными. Следовательно, потому вилки на зарядных устройствах, и других устройствах питания тоже различные.

Представьте, что вы едете в командировку в США. У вас есть, допустим, ноутбук купленный в Европе – следовательно вилка на проводе от блока питания имеет круглые окончания. Что делать? Покупать зарядку для американского типа розетки? А когда вы вернетесь домой – она будет лежать у вас мертвым грузом?

Потому, вероятнее всего, вы приобретете один из адаптеров, которые надеваются на вилку, и которая позволяет Вам использовать старую зарядку и заряжаться от совершенно другой розетки.

Так и с Адаптером – он конвертирует интерфейс класса на такой, который ожидается.

Сам паттерн состоит из трех частей: Цели (target), Адаптера (adapter), и Адаптируемого (adaptee).

В описанном выше примере:

1. Target – ноутбук со старой зарядкой.
2. Adapter – переходник.
3. Adaptee – розетка с квадратными дырками.

Реализаций паттерна Adapter может быть две (вероятно даже и больше, но я вижу две).

Итак, первая – более простая реализация. Пусть у нас будет объект Bird, который реализует протокол BirdProtocol:

```
public protocol BirdProtocol {
    func sing()
    func fly()
}
```

```
public class Bird: BirdProtocol {
    public init() { }

    public func sing() {
        print("Tew-tew-tew")
    }

    public func fly() {
        print("OMG! I am flying!")
    }
}
```

И пусть у нас будет объект Raven, который выглядит так:

```
public class Raven {
    public init() { }

    public func flySearchAndDestroy() {
        print("I am flying and seek for killing!")
    }

    public func voice() {
        print("Kaaaar-kaaaaar-kaaaaar!")
    }
}
```

Чтоб использовать ворона в методах, которые ждут птицу ☺, стоит этого ворона адаптировать:

```
public class RavenAdapter: BirdProtocol {
    private var raven: Raven

    public init(raven: Raven) {
        self.raven = raven
    }

    public func sing() {
        self.raven.voice()
    }

    public func fly() {
        self.raven.flySearchAndDestroy()
    }
}
```

Ну и, конечно же, тест:

```
func makeTheBirdTest(bird: BirdProtocol) {
    print("\n")
    bird.fly()
    bird.sing()
}

let simpleBird = Bird()
let simpleRaven = Raven()
let ravenAdapter = RavenAdapter(raven: simpleRaven)

makeTheBirdTest(bird: simpleBird)
makeTheBirdTest(bird: ravenAdapter)
```

Результат можно легко увидеть в логе:

```
OMG! I am flying!
Tew-tew-tew


I am flying and seek for killing!
Kaaaar-kaaaaar-kaaaaar!
```

А теперь – уже более сложная реализация, которая все еще зависит от протоколов, но при этом использует и делегаты. Вернемся к нашему ноутбуку с зарядкой. Допустим у нас есть базовый класс Charger:

```
public class Charger {
    public func charge() {
        print("C'mon, I am charging!")
    }
}
```

И есть протокол для европейской зарядки:

```
public protocol EuropeanNotebookChargerDelegate: AnyObject {
    func chargeNotebookRoundHoles(charger: Charger)
}
```

Если сделать просто реализацию, то получится то же самое, что и в предыдущем примере ☺. Поэтому, давайте добавим делегат:

```
public class EuropeanNotebookCharger: Charger,
EuropeanNotebookChargerDelegate {
    weak var delegate: EuropeanNotebookChargerDelegate?

    public override init() {
        super.init()
        self.delegate = self
    }

    public override func charge() {
        super.charge()
        self.delegate?.chargeNotebookRoundHoles(charger: self)
    }

    public func chargeNotebookRoundHoles(charger: Charger) {
        print("Charging with 220 and round holes.")
    }
}
```

Как видим, у нашего класса есть свойство, которое реализует тип EuropeanNotebookChargerDelegate. Так как наш класс этот протокол реализует, он может этому свойству присвоить самого себя, поэтому когда происходит вызов

```
self.delegate?.chargeNotebookRoundHoles(charger: self)
```

просто вызывается свой же метод. Вы увидите дальше для чего это сделано. Теперь давайте глянем что ж за зверь такой – американская зарядка:

```
public class USANotebookCharger {
    public init() { }

    public func chargeNotebookRectHoles(charger: Charger) {
        print("Charge notebook with rect holes.")
    }
}
```

Как видим, в американской зарядке совсем другой метод и мировоззрение. Давайте создадим адаптер для нее:

```
public class USANotebookEuropeanAdapter: Charger,
EuropeanNotebookChargerDelegate {
    var usaCharger: USANotebookCharger

    public init(charger: USANotebookCharger) {
        self.usaCharger = charger
    }

    public override func charge() {
        let euroCharge = EuropeanNotebookCharger()
        euroCharge.delegate = self
        euroCharge.charge()
    }

    public func chargeNotebookRoundHoles(charger: Charger) {
        self.usaCharger.chargeNotebookRectHoles(charger: charger)
    }
}
```

Наш адаптер реализует протокол EuropeanNotebookChargerDelegate и его метод chargeNotebookRoundHoles. Поэтому, когда вызывается метод charge – на самом деле, создается тип европейской зарядки, ей присваивается наш адаптер как делегат, и вызывается ее метод charge. Так как делегатом присвоен наш адаптер, при вызове метода chargeNotebookRoundHoles, будет вызван этот метод нашего адаптера, который в свою очередь, вызовет метод зарядки США ☺.

Давайте посмотрим тест-код и вывод лога:

```
func makeTheNotebookCharge(charger: Charger) {
    print("\n")
    charger.charge()
}

let euroCharger = EuropeanNotebookCharger()
makeTheNotebookCharge(charger: euroCharger)

let charger = USANotebookCharger()
let adapter = USANotebookEuropeanAdapter(charger: charger)
makeTheNotebookCharge(charger: adapter)
```

В логе будет следующее:

```
C'mon, I am charging!
Charging with 220 and round holes.


C'mon, I am charging!
Charge notebook with rect holes.
```

Код примера.

Bridge

Представьте себе, что у нас есть что-то однотипное, к примеру, у нас есть телефон и куча наушников. Если бы у каждого телефона был свой разъем, то мы могли бы пользоваться только одним типом наушников. Но Бог миловал! Собственно та же штука и с наушниками. Они могут выдавать различный звук, иметь различные дополнительные функции, но основная их цель – просто звучание ☺. И хорошо, что во многих случаях штекер у них одинаковый (я не говорю про различные студийные наушники ☺).

Собственно, Мост (Bridge) позволяет разделить абстракцию от реализации, так чтобы реализация в любой момент могла быть изменена, не меняя при этом абстракции.

Когда использовать:

1. Вам совершенно не нужна связь между абстракцией и реализацией.
2. Собственно, как абстракцию так и имплементацию могут наследовать независимо.
3. Вы не хотите, чтобы изменения в реализации имели влияние на клиентский код.

Давайте создадим теперь базовую абстракцию наушников:

```
public protocol BaseHeadphones {
    func playSimpleSound()
    func playBassSound()
}
```

И теперь два элемента – дорогие наушники и дешевые ☺

```
// Наушники обычные – китайские
public class CheapHeadphones: BaseHeadphones {
    public init() { }

    public func playSimpleSound() {
        print("beep-beep-bhhhrhrhrep")
    }

    public func playBassSound() {
        print("puf-puf-pufhrrr")
    }
}
```

```
// Наушники дорогие, тоже китайские
public class ExpensiveHeadphones: BaseHeadphones {
    public init() { }

    public func playSimpleSound() {
        print("Beep-Beep-Beep Taram-Rararam")
    }

    public func playBassSound() {
        print("Bam-Bam-Bam")
    }
}
```

И, собственно, плеер через который мы будем слушать музыку:

```
public class MusicPlayer {
    public var headPhones: BaseHeadphones?

    public init() { }

    public func playMusic() {
        self.headPhones?.playBassSound()
        self.headPhones?.playBassSound()
        self.headPhones?.playSimpleSound()
        self.headPhones?.playSimpleSound()
    }
}
```

Как видите, одно из наших свойств нашего плеера – наушники. Их можно подменять в любой момент, так как свойство того же типа, от которого наши дешевые и дорогие наушники наследуются.

Итак, тест:

```
let player = MusicPlayer()
let cheapHeadphones = CheapHeadphones()
let expensiveHeadphones = ExpensiveHeadphones()

player.headPhones = cheapHeadphones
player.playMusic()
print("==========")
player.headPhones = expensiveHeadphones
player.playMusic()
```

И, конечно же, log:

```
puf-puf-pufhrrr
puf-puf-pufhrrr
beep-beep-bhhhrhrhrep
beep-beep-bhhhrhrhrep
==========
Bam-Bam-Bam
Bam-Bam-Bam
Beep-Beep-Beep Taram-Rararam
Beep-Beep-Beep Taram-Rararam
```

Код примера.

Facade

Многие сложные системы состоят из огромной кучи компонент. Так же и в жизни, очень часто для совершения одного основного действия, мы должны выполнить много маленьких.

К примеру, чтобы пойти в кино нам нужно:

1. Посмотреть расписание фильмов, выбрать фильм, посмотреть когда есть сеансы, посмотреть когда у нас есть время.
2. Необходимо купить билет, для этого ввести номер карточки, секредный код, дождаться снятия денег, распечатать билет.
3. Приехать в кинотеатр, запарковать машину, купить попкорн, найти места, смотреть.

И все это для того, чтобы просто посмотреть фильм, который нам, очень вероятно, не понравится.

Или же возьмем пример Amazon – покупка с одного клика – как много систем задействовано в операции покупки? И проверка вашей карточки, и проверка вашего адреса, проверка товара на складе, проверка или возможна доставка данного товара в данную точку мира. В результате, очень много действий которые происходят всего по одному клику.

Для таких вот процессов был изобретен паттерн Фасад (Facade), который предоставляет унифицированный интерфейс к большому количеству интерфейсов системы, вследствие чего система становится гораздо проще в использовании.

Давайте, попробуем создать систему, которая нас переносит в другую точку мира с одного нажатия кнопки! Сначала нам нужна система, которая проложит путь от нашего места пребывания в место назначения:

```
public class PathFinder {
    public init() { }

    public func findCurrentLocation() {
        print("Finding your location. Hmmm, here you are!")
    }

    public func findLocationToTravel(location: String) {
        print("So you wanna travel to \(location)")
    }

    public func makeARoute() {
        print("Using Google maps...")
        // looking for path in Google maps
    }
}
```

Естественно, нам необходима сама система заказа транспорта и, собственно, путешествия:

```
public class TravelEngine {
    public init() { }

    public func findTransport() {
        print("Okay, to travel there you will need a dragon!")
    }

    public func orderTransport() {
        print("I need to order a dragon.")
        print("Yes, green one.")
        print("With fire, of course.")
    }

    public func travel() {
        print("Finally, I'm flying on a dragon. Woo-hoo!")
    }
}
```

Ну и какие же путешествия без билетика:

```
public class TicketPrintingSystem {
    public init() { }

    public func createTicket() {
        print("Connecting to our ticket system...")
    }

    public func printTicket() {
        print("Hmmm, ticket for travelling on the green dragon.")
        print("Interesting...")
    }
}
```

И создадим единый доступ ко всем этим системам:

```
public class TravelSystemFacade {
    public init() { }

    public func travelTo(location: String) {
        let pathFinder = PathFinder()
        let travelEngine = TravelEngine()
        let ticketPrintingSystem = TicketPrintingSystem()

        pathFinder.findCurrentLocation()
        pathFinder.findLocationToTravel(location: location)
        pathFinder.makeARoute()

        travelEngine.findTransport()
        travelEngine.orderTransport()

        ticketPrintingSystem.createTicket()
        ticketPrintingSystem.printTicket()

        travelEngine.travel()
    }
}
```

Как видим, наш фасад знает все про все системы, потому в одном методе он берет и транспортирует нас куда следует. Код теста элементарен:

```
let facade = TravelSystemFacade()
facade.travelTo(location: "Lviv")
```

Давайте посмотрим лог:

```
Finding your location. Hmmm, here you are!
So you wanna travel to Lviv
Using Google maps...
Okay, to travel there you will need a dragon!
I need to order a dragon.
Yes, green one.
With fire, of course.
Connecting to our ticket system...
Hmmm, ticket for travelling on the green dragon.
Interesting...
Finally, I'm flying on a dragon. Woo-hoo!
```

Код примера.

Mediator

Медиатор – паттерн, который определяет внутри себя объект, в котором реализуется взаимодействие между некоторым количеством объектов. При этом, эти объекты могут даже не знать про существование друг друга, потому взаимодействий, реализованных в медиаторе, может быть огромное количество.

Когда стоит использовать:

1. Когда у вас есть некоторое количество объектов, и очень тяжело реализовать взаимодействие между ними. Яркий пример – умный дом. Однозначно есть несколько датчиков и несколько устройств. К примеру, датчик температуры следит за тем, какая на данный момент температура, а кондиционер умеет охлаждать воздух. Причем кондиционер не обязательно знает о существовании датчика температуры. Есть центральный компьютер, который получает сигналы от каждого из устройств и понимает, что делать в том или ином случае.
2. Тяжело повторно использовать объект, так как он взаимодействует и общается с огромным количеством других объектов.
3. Логика взаимодействия должна легко настраиваться и расширяться.

Собственно, пример медиатора даже писать бессмысленно, потому как это любой контроллер который мы используем во время нашей разработки. Посудите сами – на View есть очень много элементов управления, и все правила взаимодействия мы прописываем в контроллере. Элементарно.

И все же пример не будет лишним. Давайте все же создадим пример, который показывает создание а-ля умного дома.

Пусть у нас есть оборудование, которое может взаимодействовать с нашим умным домом:

```
public class SmartHousePart {
    private let processor: CentralProcessor

    public init(processor: CentralProcessor) {
        self.processor = processor
    }

    public func numbersChanged() {
        self.processor.valueChanged(part: self)
    }
}
```

Теперь, создадим сердце нашего умного дома:

```
public class CentralProcessor {
    public weak var thermometer: Thermometer?
    public weak var condSystem: ConditioningSystem?

    public init() { }

    public func valueChanged(part: SmartHousePart) {
        print("Value changed! We need to do smth!")

        // detecting that changes are done by thermometer
        if part is Thermometer {
            print("Oh, the change is temperature")
            self.condSystem?.startCondition()
        }
    }
}
```

Тут очень интересный момент: класс `CentralProcessor` должен знать про существование `SmartHousePart`, собственно и `SmartHousePart` должен знать про существование `CentralProcessor`. Если все свойства сделать по умолчанию, то будет «любимый» многими разработчиками retain-cycle, поэтому в `CentralProcessor` мы определяем свойства с ключевым словом `weak`.

Дальше в классе `CentralProcessor` в методе `valueChanged` мы определяем, с какой деталью и что произошло, чтобы адекватно среагировать. В нашем примере – изменение температуры приводит к тому, что мы включаем кондиционер.

А вот и код термометра и кондиционера:

```
public class Thermometer: SmartHousePart {
    private var temperature = 0

    public func temperatureChanged(temperature: Int) {
        self.temperature = temperature
        self.numbersChanged()
    }
}

public class ConditioningSystem: SmartHousePart {
    public func startCondition() {
        print("Conditioning...")
    }
}
```

Как видим, в результате у нас есть два объекта, которые друг про друга не в курсе. И, все-таки, они взаимодействуют друг с другом посредством нашего медиатора `CentralProcessor`.

Код для теста:

```
let processor = CentralProcessor()
let thermometer = Thermometer(processor: processor)
let condSystem = ConditioningSystem(processor: processor)

processor.thermometer = thermometer
processor.condSystem = condSystem

thermometer.temperatureChanged(temperature: 45)
```

И, конечно же, лог:

```
Value changed! We need to do smth!
Oh, the change is temperature
Conditioning...
```

Код примера.

Observer

Что такое паттерн Observer? Вот вы когда-нибудь подписывались на газету? Вы подписываетесь, и каждый раз, когда выходит новый номер газеты, вы получаете ее к своему дому. Вы никуда не ходите, просто даете информацию про себя, и организация, которая выпускает газету, сама знает, куда и какую газету отнести. Второе название этого паттерна: **Publish – Subscriber**.

Как описывает этот паттерн наша любимая GoF-книга, Observer определяет одно-ко-многим отношение между объектами, и если изменения происходят в объекте – все подписанные на него объекты тут же узнают про это изменение.

Идея проста: объект, который мы называем Subject, дает возможность другим объектам, которые реализуют интерфейс (протокол) Observer, подписываться и отписываться от изменений, происходящих в Subject. Когда изменение происходит – всем заинтересованным объектам высылается сообщение, что изменение произошло. В нашем случае – Subject – это издатель газеты, Observer – это мы с вами, те кто подписывается на газету, ну и собственно изменение – это выход новой газеты, а оповещение – отправка газеты всем подписчикам.

Когда используется паттерн:

1. Когда вам необходимо сообщить всем объектам, подписанным на изменения, что изменение произошло. При этом, вы не знаете типы этих объектов.
2. Изменения в одном объекте требуют, чтобы состояние изменилось в других объектах. Причем, количество объектов может меняться.

Реализация этого паттерна может быть в двух вариантах:

1. Notificaton.

Notification – механизм использования возможностей NotificationCenter, который уже встроен в iOS SDK. Использование NotificatonCenter позволяет объектам общаться, даже не знаю друг о друге. Это очень удобно использовать, когда у вас в параллельном потоке пришел push-notification, или же обновилась база, и вы хотите дать об этом знать активному на данный момент View.

Чтобы послать такое сообщение стоит использовать конструкцию типа:

```
let broadcastMessage = Notification(name: "broadcastMessage",
                                    object: self)
let notificationCetner = NotificationCenter.default
```

Как видим, мы создали объект типа `Notification`, в котором мы указали имя нашего оповещения: `"broadcastMessage"` и, собственно сообщили о нем через `NotificationCenter`.

Чтобы подписаться на событие в объекте, который заинтересован в изменении стоит использовать следующую конструкцию:

```
let notificationCetner = NotificationCenter.default
notificationCenter..addObserver(self,
                                selector: #selector(update:),
                                name: "broadcastMessage",
                                object: nil)
```

Собственно, из кода все более-менее понятно: мы подписываемся на событие, и вызывается метод, который задан в свойстве selector.

2. Стандартный метод.

Стандартный метод – это реализация этого паттерна тогда, когда Subject знает про всех подписчиков, но при этом не знает ничего об их типах. Давайте начнем с того, что создадим протоколы для Subject и Observer:

```
public protocol StandardObserver: AnyObject {
    func valueChanged(name: String, newValue: String)
}

public protocol StandardSubject: AnyObject {
    func addObserver(_ observer: StandardObserver)
    func removeObserver(_ observer: StandardObserver)
    func notifyObjects()
}
```

Теперь, давайте создадим реализацию Subject:

```
public class StandardSubjectImplementation: StandardSubject {
    private var valueName: String = ""
    private var newValue: String = ""
    private var observerCollection: [StandardObserver] = []

    public init() { }

    public func changeValue(name: String, newValue: String) {
        self.valueName = name
        self.newValue = newValue
        self.notifyObjects()
    }

    public func addObserver(_ observer: StandardObserver) {
        self.observerCollection.append(observer)
    }

    public func removeObserver(_ observer: StandardObserver) {
        self.observerCollection = self.observerCollection.filter {
            $0 !== observer
        }
    }

    public func notifyObjects() {
        self.observerCollection.forEach { observer in
            observer.valueChanged(name: self.valueName,
                                  newValue: self.newValue)
        }
    }
}
```

Ну и куда же без обсерверов:

```
public class SomeObserver: StandardObserver {
    public init() { }

    public func valueChanged(name: String, newValue: String) {
        print("Some observer:")
        print("value \(name) changed to \(newValue)")
    }
}

// и второй наблюдатель

public class OtherObserver: StandardObserver {
    public init() { }

    public func valueChanged(name: String, newValue: String) {
        print("Other observer:")
        print("value \(name) changed to \(newValue)")
    }
}
```

Собственно – все ☺, теперь демо-код:

```
let subj = StandardSubjectImplementation()
let someObserver = SomeObserver()
let otherObserver = OtherObserver()

subj.addObserver(someObserver)
subj.addObserver(otherObserver)

subj.changeValue(name: "strange value", newValue: "newValue")
```

И, естественно, лог:

```
Some observer:
value strange value changed to newValue
Other observer:
value strange value changed to newValue
```

Ну и, конечно же, без KVO описание паттерна выглядело бы неполным. Одна из моих любимых особенностей Obj-C (да, это имеет отношение и к Swift) – это key-value coding. Про него очень клево написано в официальной документации, но если объяснять на валенках, то это – возможность изменять значения свойств объекта с помощью строчек – которые указывают именно на само название свойства. Как пример, такие две конструкции могут быть идентичны:

```
kvoSubj.changeableProperty = "new value"

kvoSubj.setValue("new value", forKey: "changeableProperty")
```

Такая гибкость дает нам доступ к еще одной очень замечательной возможности, которая называется key-value observing. Опять же: все круто описано в документации, но если объяснять на валенках ☺, то это – возможность подписаться на изменение любого свойства у любого объекта, который KV compliant любым объектом. На самом деле, легче объяснить на примере.

Давайте создадим класс с одним свойством, которое мы будем менять:

```
public class KVOSubject: NSObject {
    @objc dynamic public var changeableProperty: String = ""
}
```

Обратите внимание, что для того, чтоб все работало – мы должны унаследоваться от `NSObject`, а наше свойство должно быть с аннотацией `@objc` и ключевым словом `dynamic`.

Теперь создадим объект, который будет слушать изменение этого свойства:

```
public class KVOObserver: NSObject {
    public override func observeValue(forKeyPath keyPath: String?,
                                      of object: Any?,
                                      change: [NSKeyValueChangeKey : Any]?,
                                      context: UnsafeMutableRawPointer?) {

        print("KVO: Value changed;")
    }
}
```

Как видим, этот класс реализует только один метод: `observeValue`. Этот метод будет вызван когда поменяется свойство объекта за которым мы наблюдаем.

Теперь тест:

```
let kvoSubj = KVOSubject()
let kvoObserver = KVOObserver()

kvoSubj.addObserver(kvoObserver,
                    forKeyPath: "changeableProperty",
                    options: .new,
                    context: nil)

kvoSubj.changeableProperty = "new value"
kvoSubj.removeObserver(kvoObserver, forKeyPath: "changeableProperty")
```

Как видно из примера: мы для объекта, за которым наблюдаем, выполняем функцию `addObserver` – где устанавливаем, кто будет наблюдать за изменениями, за изменениями какого свойства мы будем наблюдать и остальные опции. Дальше – меняем значение свойства, и в конце – удаляем наблюдателя с нашего объекта.

Лог говорит сам за себя:

```
KVO: Value changed;
```

Код примера.

Composite

Вы задумывались как много в нашей жизни древовидных структур? Начиная, собственно, от самих деревьев, и заканчивая структурами компаний. Да даже, ладно компаний – целые страны используют древовидные структуры, чтобы построить власть.

Во главе компании или страны частенько стоит один человек, у него есть с десяток помощников. У них тоже есть с десяток помощников, и так далее… Если нарисовать их отношения на листе бумаги – увидим дерево!

Очень часто, и мы используем такие типы данных, которые лучше всего хранятся в древовидной структуре. Возьмите к примеру стандартный UI: в начале у нас есть View, в нем находятся Subview, в которых могут быть или другие View, или все такие компоненты. Та же самая структура. ☺

Именно для хранения таких типов данных, а вернее их организации, используется паттерн – Композит.

Когда использовать такой паттерн?

Собственно, когда вы работаете с древовидными типами данных, или хотите отобразить иерархию данных таким образом.

Давайте разберем более детально структуру:

Вначале всегда есть контейнер, в котором находятся все остальные объекты. Контейнер может хранить как другие контейнеры – ветки нашего дерева, так и объекты, которые контейнерами не являются – листья нашего дерева. Несложно представить, что контейнеры второго уровня могут хранить как другие контейнеры, так и листья.

Давайте пример!

Начнем с создания протокола для наших объектов:

```
public protocol CompositeObjectProtocol {
    func getData() -> String
    func add(component: CompositeObjectProtocol)
}
```

Создадим объект листа:

```
public class LeafObject: CompositeObjectProtocol {
    public var leafValue: String

    public init(leafValue: String) {
        self.leafValue = leafValue
    }

    public func getData() -> String {
        return "<\(self.leafValue)>"
    }

    public func add(component: CompositeObjectProtocol) {
        print("Can't add component. Sorry, man.")
    }
}
```

Как видим, наш объект не может добавлять себе детей (ну он же не контейнер ☺) , и может возвращать свое значение с помощью метода `getData`.

Теперь нам очень необходим контейнер:

```
public class Container: CompositeObjectProtocol {
    private var components: [CompositeObjectProtocol] = []

    public init() { }

    public func getData() -> String {
        var valueToReturn = "<ContainerValues>"

        self.components.forEach { component in
            valueToReturn += "\n"
            valueToReturn += component.getData()
        }

        valueToReturn += "\n"
        valueToReturn += "</ContainerValues>"

        return valueToReturn
    }

    public func add(component: CompositeObjectProtocol) {
        self.components.append(component)
    }
}
```

Как видим, наш контейнер может добавлять в себя детей, которые могут быть как типа `Container`, так и типа `LeafObject`. Метод `getData` же, бегает по всем объектам в массиве `components`, и вызывает тот же самый метод в детях. Вот, собственно, и все.

Теперь, конечно же пример:

```
let rootContainer = Container()
let object = LeafObject(leafValue: "level1 value")
rootContainer.add(component: object)

let firstLevelContainer1 = Container()
let object2 = LeafObject(leafValue: "level2 value")
firstLevelContainer1.add(component: object2)
rootContainer.add(component: firstLevelContainer1)

let firstLevelContainer2 = Container()
let object3 = LeafObject(leafValue: "level2 value 2")
firstLevelContainer2.add(component: object3)
rootContainer.add(component: firstLevelContainer2)

print(rootContainer.getData())
```

И, конечно же, лог:

```
<ContainerValues>
<level1 value>
<ContainerValues>
<level2 value>
</ContainerValues>
<ContainerValues>
<level2 value 2>
</ContainerValues>
</ContainerValues>
```

Если добавить немного форматирования, то будет ясно видна структура:

```
<ContainerValues>
    <level1 value>
    <ContainerValues>
        <level2 value>
    </ContainerValues>
    <ContainerValues>
        <level2 value 2>
    </ContainerValues>
</ContainerValues>
```

Код примера.

Iterator

Я задумался о том, какой бы пример из жизни привести, чтобы показать пример как работает паттерн итератор, и оказалось что это не такое простое задание. И, как показывает практика, самый простой пример – это обычная вендинг машина. (сам пример взят из книги Pro Objective-C Design Patterns for iOS). У нас есть контейнер, который разделен на секции, каждая из которых содержит определенный вид товара, к примеру набор бутылок Coca-Cola. Когда мы заказываем товар, то нам выпадет следующий из коллекции (образно говоря, команда cocaColaCollection.next). Две независимые части – контейнер и итератор.

Паттерн итератор позволяет последовательно обращаться к коллекции объектов, не особо вникая, что же это за коллекция.

Разделяют два вида итераторов – внутренний и внешний. Как видно из названия, внешний итератор – итератор, про который знает клиент, и собственно он сам (клиент) скармливает коллекцию по которой надо бегать итератору. Внутренний итератор – это внутренняя кухня самой коллекции, которая предоставляет интерфейс клиенту для итерирования.

При внешнем итераторе клиенту нужно:

1. Вообще знать о существовании итератора, хоть это и дает больше контроля.
2. Создавать и управлять итератором.
3. Можно использовать различные итераторы, для различных алгоритмов итерации.

При внутреннем итераторе:

1. Клиенту совершенно неизвестно о существовании итератора. Он просто дергает интерфейс коллекции.
2. Коллекция сама создает и управляет итератором.
3. Коллекция может менять различные итераторы, не трогая при этом код клиента.

Когда использовать итератор:

1. Вам необходимо добраться к объектам коллекции, но при этом не важны детали реализации внутренностей этой коллекции.
2. Вам нужно обходить объекты коллекции различными способами (вспомните Композит – коллекция, может быть древовидной).
3. Вам необходимо дать унифицированный интерфейс для различных подходов итерации.

Самый просто пример внешнего итератора – использование класса `NSEnumerator`:

```
let enumerator: NSEnumerator = (internalArrayCollection as
NSArray).objectEnumerator()

while let element = enumerator.nextObject() as? String {
    print(element)
}
```

Как видим, мы просто вызываем у коллекции `internalArrayCollection` метод `objectEnumerator` и получаем необходимый нам итератор. Вообще можно не заморачиваться, и использовать цикл `for-in`:

```
for element in internalArrayCollection {
    print(element)
}
```

Я не уверен, что смогу правильно объяснить разницу между созданием итератора и циклом `for-in`, потому этот момент будет опущен.

Одним из примеров реализации внешнего итератора – может быть итерация с помощью блоков:

```
internalArrayCollection.forEach { obj in
    if obj.localizedCaseInsensitiveCompare("Dima") == .orderedSame {
        print("Dima has been found!")
    }
}
```

Радость этого метода в том, что сам алгоритм итерации может написать другой программист, вам же необходимо будет только использовать блок, написанный этим программистом. Все выглядит приблизительно так:

```
// создание блока поиска Dima в массиве строк
let simpleDimaSearchBlock: (String) -> Void = { obj in
    if obj.localizedCaseInsensitiveCompare("Dima") == .orderedSame {
        print("Dima has been found!")
    }
}

// использование этого блока в вызове forEach
internalArrayCollection.forEach(simpleDimaSearchBlock)
```

Приятно же ☺

Теперь давайте создадим свой итератор, а то и два. ☺
Пусть у нас будет коллекция товаров, одни из которых будут сломаны, а другие – целые. Создадим два итератора, которые будут бегать по разным типам товаров. Итак, для начала – сам класс товаров:

```
public class ItemInShop {
    public var name: String
    var isBroken: Bool

    public init(name: String, isBroken: Bool) {
        self.name = name
        self.isBroken = isBroken
    }
}
```

Как видим, не густо – два свойства и инициализатор. Теперь давайте создадим склад, в котором эти товары и будут находиться:

```
public class ShopWarehouse {
    private var goods: [ItemInShop] = []

    public init() { }

    public var goodItemsEnumerator: GoodItemsEnumerator {
        return GoodItemsEnumerator(items: self.goods)
    }

    public var badItemsEnumerator: BadItemsEnumerator {
        return BadItemsEnumerator(items: self.goods)
    }

    public func add(item: ItemInShop) {
        self.goods.append(item)
    }
}
```

Как видим, наш склад умеет добавлять товары, а также возвращать два таинственных объекта с названиями `goodItemsEnumerator` и `badItemsEnumerator`.

Собственно, их назначение – очевидно. Давайте взглянем на реализацию. Сначала добавим протокол, который они будут конформить и typealias для удобства:

```
public typealias BasicEnumerator = BasicEnumeratorProtocol &
NSEnumerator

public protocol BasicEnumeratorProtocol {
    init(items: [ItemInShop])
    func allObjects() -> [ItemInShop]
    func nexObject() -> Any?
}
```

Как видим `BasicEnumerator` требует реализацию инициализатора и двух методов (получить все объекты и получить следующий объект), а так же предполагает наследование от `NSEnumerator`.

Давайте добавим два итератора, как и планировали:

```
public class BadItemsEnumerator: BasicEnumerator {
    private var items: [ItemInShop]
    private var internalEnumerator: NSEnumerator

    public required init(items: [ItemInShop]) {
        self.items = items.filter { item in item.isBroken }
        self.internalEnumerator = (self.items as
NSArray).objectEnumerator()
    }

    public func allObjects() -> [ItemInShop] {
        return self.items
    }

    public func nexObject() -> Any? {
        return self.internalEnumerator.nextObject()
    }
}
```

Я не привожу код для `GoodItemsEnumerator`, потому как отличие будет лишь в одной строчке:

```
self.items = items.filter { item in !item.isBroken }
```

Как видите, во время инициализации мы фильтруем товары, оставляя только плохие. Также, создаем свой внутренний итератор.

Ну что, тестируем:

```
// создание тестовых данных
let shopWareHouse = ShopWarehouse()
shopWareHouse.add(item: ItemInShop(name: "Item1", isBroken: false))
shopWareHouse.add(item: ItemInShop(name: "Item2", isBroken: false))
shopWareHouse.add(item: ItemInShop(name: "Item3", isBroken: true))
shopWareHouse.add(item: ItemInShop(name: "Item4", isBroken: true))
shopWareHouse.add(item: ItemInShop(name: "Item5", isBroken: false))
```

Сам тест:

```
let goodIterator = shopWareHouse.goodItemsEnumerator
let badIterator = shopWareHouse.badItemsEnumerator

while let element = goodIterator.nexObject() as? ItemInShop {
    print("Good item = \(element.name)")
}

while let element = badIterator.nexObject() as? ItemInShop {
    print("Bad Item = \(element.name)")
}
```

В логе увидим совпадение ожиданий с реальностью:

```
Good item = Item1
Good item = Item2
Good item = Item5
Bad Item = Item3
Bad Item = Item4
```

Код примера.

Visitor

Вот у каждого дома, вероятнее всего, есть холодильник. В ВАШЕМ доме – ВАШ холодильник. Что будет, если холодильник сломается? Некоторые пойдут почитают в интернете как чинить холодильник: узнают модель, попробуют поколдовать над ним и, разочаровавшись, вызовут мастера по ремонту холодильников. Заметьте: холодильник ваш, но функцию «чинить холодильник» выполняет совершенно другой человек, про которого вы ничего не знаете, а попросту – обычный визитер.

Паттерн визитер – позволяет вынести из наших объектов логику, которая относится к этим объектам, в отдельный класс, что позволяет нам легко изменять/добавлять алгоритмы, при этом не меняя логику самого класса.

Когда мы захотим использовать этот паттерн:

1. Когда у нас есть сложный объект в котором содержится большое количество различных элементов, и вы хотите выполнять различные операции в зависимости от типа этих элементов.
2. Вам необходимо выполнять различные операции над классами, и при этом вы не хотите писать вагон кода внутри реализации этих классов.
3. В конце концов, вам нужно добавлять различные операции над элементами, и вы не хотите постоянно обновлять классы этих элементов.

Что ж, давайте вернемся к примеру из прошлой главы, только теперь будет сложнее – у нас есть несколько складов, в каждом из них может храниться товар. Один визитер будет смотреть склады, а другой визитер будет называть цену товара в складе.

Итак, для начала сам товар:

```
public class WarehouseItem {
    public var name: String
    public var isBroken: Bool
    public var price: Int

    public init(name: String,
                isBroken: Bool,
                price: Int) {

        self.name = name
        self.isBroken = isBroken
        self.price = price
    }
}
```

И сам склад:

```
public class Warehouse {
    private var items: [WarehouseItem] = []

    public init() { }

    public func add(item: WarehouseItem) {
        self.items.append(item)
    }

    public func accept(visitor: BasicVisitor) {
        visitor.visit(self)
        self.items.forEach { item in visitor.visit(item) }
    }
}
```

Как видим, наш склад умеет хранить и добавлять товар, но также обладает таинственным методом `accept`, который принимает в себя визитера и вызывает его метод `visit`. Чтобы картинка сложилась, давайте создадим протокол `BasicVisitor` и различных визитеров:

```
public protocol BasicVisitor {
    func visit(_ object: AnyObject)
}
```

Как видим, протокол требует реализацию только одного метода. Теперь давайте перейдем к самим визитерам:

```
public class QualityCheckerVisitor: BasicVisitor {
    public init() { }

    public func visit(_ object: AnyObject) {
        switch object {
        case let item as WarehouseItem:
            if item.isBroken {
                print("Item \(item.name) is broken")
            } else {
                print("Item \(item.name) is pretty cool!")
            }
        case is Warehouse:
            print("Hmmm, nice warehouse!")
        default:
            break
        }
    }
}
```

Как видим, при вызове своего метода `visit`, визитер определяет тип объекта (в операторе `switch`), который ему передали и выполняет определенные функции в зависимости от этого типа. Данный объект просто говорит цела или сломана вещь на складе, а также, что ему нравится сам склад. ☺

```
public class PriceCheckerVisitor: BasicVisitor {
    public init() { }

    public func visit(_ object: AnyObject) {
        switch object {
        case let item as WarehouseItem:
            print("Item \(item.name) has price \(item.price)")
        case is Warehouse:
            print("Hmmm, I don't know how much Warehouse costs!")
        default:
            break
        }
    }
}
```

Этот визитер, в принципе, делает то же самое. Только в случае склада, он признается, что растерян, а в случае товара – называет цену товара.

Теперь давайте запустим то, что у нас получилось. Сначала сгенерируем данные:

```
let localWarehouse = Warehouse()
localWarehouse.add(item: WarehouseItem(name: "Item1",
                                       isBroken: false,
                                       price: 25))
localWarehouse.add(item: WarehouseItem(name: "Item2",
                                       isBroken: false,
                                       price: 32))
localWarehouse.add(item: WarehouseItem(name: "Item3",
                                       isBroken: true,
                                       price: 45))
localWarehouse.add(item: WarehouseItem(name: "Item4",
                                       isBroken: false,
                                       price: 33))
localWarehouse.add(item: WarehouseItem(name: "Item5",
                                       isBroken: false,
                                       price: 12))
localWarehouse.add(item: WarehouseItem(name: "Item6",
                                       isBroken: true,
                                       price: 78))
localWarehouse.add(item: WarehouseItem(name: "Item7",
                                       isBroken: true,
                                       price: 34))
localWarehouse.add(item: WarehouseItem(name: "Item8",
                                       isBroken: false,
                                       price: 51))
localWarehouse.add(item: WarehouseItem(name: "Item9",
                                       isBroken: false,
                                       price: 25))
```

А теперь – сам тестовый код:

```
let visitor = PriceCheckerVisitor()
let qualityVisitor = QualityCheckerVisitor()

print("==========")

localWarehouse.accept(visitor: visitor)
localWarehouse.accept(visitor: qualityVisitor)
```

Итак, при вызове метода `accept` нашего склада, визитер сначала проведывает наш склад, а потом проведывает каждый товар на этом складе. При этом мы можем менять как визитера, так и алгоритм. И это не повлечет изменения в коде клиента. ☺

Традиционный лог:

```
Hmmm, I don't know how much Warehouse costs!
Item Item1 has price 25
Item Item2 has price 32
Item Item3 has price 45
Item Item4 has price 33
Item Item5 has price 12
Item Item6 has price 78
Item Item7 has price 34
Item Item8 has price 51
Item Item9 has price 25
==========
Hmmm, nice warehouse!
Item Item1 is pretty cool!
Item Item2 is pretty cool!
Item Item3 is broken
Item Item4 is pretty cool!
Item Item5 is pretty cool!
Item Item6 is broken
Item Item7 is broken
Item Item8 is pretty cool!
Item Item9 is pretty cool!
```

Код примера.

Decorator

Классный пример декоратора – различные корпусы для новых телефонов. Как-то я сразу с конца начал. ☺

Для начала: у нас есть телефон, но так как он дорогой – мы будем счастливы, если он не разобьется при любом падении. Поэтому мы покупаем для него чехол. То есть – к уже существующему предмету мы добавляем функционал защиты от падения. Ну а еще, мы взяли стильный и красивый чехол, поэтому теперь наш телефон еще и отлично выглядит. А потом мы докупили съемный объектив, с помощью которого можно делать фотографии с эффектом «рыбьего глаза». По сути – декорировали наш телефон дополнительным функционалом. ☺

Вот, приблизительно так выглядит реальное описание паттерна «Декоратор». Теперь описание из GoF: декоратор добавляет некий функционал уже существующему объекту.

Когда использовать этот паттерн:

1. Вы хотите добавить определенному объекту дополнительные возможности, при этом не задевая и не меняя других объектов.
2. Дополнительные возможности – опциональны.

Радость Swift в данном случае – это использование расширений (extensions). Я не буду детально описывать расширения в этой книге, но в двух словах все же скажу: расширения – это возможность расширить любой класс дополнительными методами без использования наследования.

Давайте посмотрим на примере. Возьмем системную структуру `Date` и добавим к ней новый метод.

Допустим, нам нужно иметь возможность любую дату как-то определенно отформатировать и получить в виде строки. Для начала напишем расширение:

```
extension Date {
    func convertToString() -> String {
        let formatter = DateFormatter()
        formatter.dateFormat = "yyyy/MM/dd"

        return formatter.string(from: self)
    }
}
```

Как видим, наше расширение определяет только один метод `convertToString`, который форматирует дату в какой-то свой формат. Вы будете смеяться, но в принципе, вот и все. ☺

Код тестирования выглядит следующим образом:

```
let dateNow = Date()
print(dateNow)
print("Date is \(dateNow.convertToString())")
```

Посмотрим что у нас в логе. Все ожидаемо:

```
2019-02-11 17:08:59 +0000
Date is 2019/02/11
```

Кстати, Здесь можно увидеть точную дату и время написания этой главы. ☺

Код примера.

П.С. Приведенный в этой главе пример, не является классическим паттерном «Декоратор». Если по классике, то декоратор – это отдельная сущность, которая выступает оберткой над классом, который декорирует. Но возможности языка Swift позволяют пользоваться расширениями в большинстве случаев и не плодить дополнительные классы в приложении. Я считаю, что это очень удобно.

Chain of responsibility

Паттерн с моим любимым названием ☺

Представьте себе очередь людей, которые пришли за посылками. Выдающий посылки человек дает первую посылку первому в очереди человеку, он смотрит на имя-фамилию на коробке, видит, что посылка не для него, и передает посылку дальше. Второй человек делает собственно тоже самое, и так пока не найдется получатель.

Цепочка ответственности (chain of responsibility) – позволяет вам передавать объект по цепочке объектов-обработчиков, пока не будет найден необходимый обработчик.

Когда использовать этот паттерн:

1. У вас более чем один обработчик.
2. У вас есть несколько обработчиков и вы не хотите явно указывать какой из них должен обрабатывать объект в данный момент времени.

Как всегда, пример:

Представим, что у нас есть конвейер, который обрабатывает различные предметы, находящиеся на нем: игрушки, электронику и т.п.

Для начала создадим классы объектов, которые могут быть обработаны нашими обработчиками:

```
public class BasicItem {
    public init() { }
}

public class Toy: BasicItem { }

public class Electronics: BasicItem { }

public class Trash: BasicItem { }
```

Теперь добавим обработчики:

```
public typealias BasicHandler = BasicHandlerClass &
                                BasicHandlerProtocol

public protocol BasicHandlerProtocol {
    func handle(item: BasicItem)
}

public class BasicHandlerClass {
    var nextHandler: BasicHandler?

    public init() { }

    public init(nextHandler: BasicHandler) {
        self.nextHandler = nextHandler
    }
}
```

Как видим, наш базовый обработчик состоит из протокола с методом `handle(item:)`, который обрабатывает объекты типа `BasicItem`. И, самое важное – он имеет ссылку на следующий обработчик (как в нашей очереди, про людей передающих посылку).

Теперь давайте напишем код обработчика игрушки:

```
public class ToysHandler: BasicHandler {
    public func handle(item: BasicItem) {
        if item is Toy {
            print("Toy found")
            print("Handling\n")
        } else {
            print("Toy not found")
            print("Handling using next handler")
            self.nextHandler?.handle(item: item)
        }
    }
}
```

Из кода видно, что если обработчик получил объект класса `Toy`, то он его обрабатывает, а если нет – передает этот объект следующему обработчику.

Аналогично создадим еще два обработчика для электроники и мусора:

```
// хэндлер электроники
public class ElectronicsHandler: BasicHandler {
    public func handle(item: BasicItem) {
        if item is Electronics {
            print("Electronics found")
            print("Handling\n")
        } else {
            print("Electronics not found")
            print("Handling using next handler")
            self.nextHandler?.handle(item: item)
        }
    }
}

// хэндлер мусора
public class OtherItemsHandler: BasicHandler {
    public func handle(item: BasicItem) {
        print("Found undefined item")
        print("Destroying")
    }
}
```

Как видим, `OtherItemsHandler` в случае, когда до него дошло дело – просто «уничтожает» объект, и не обращается к следующему обработчику.

Давайте тестировать:

```
let otherItemsHandler = OtherItemsHandler()
let electronicsHandler = ElectronicsHandler(nextHandler:
otherItemsHandler)
let toysHandler = ToysHandler(nextHandler: electronicsHandler)

let toy = Toy()
let electronics = Electronics()
let trash = Trash()

toysHandler.handle(item: toy)
toysHandler.handle(item: electronics)
toysHandler.handle(item: trash)
```

Сначала мы создаем наши обработчики, причем в обратном порядке, чтоб при инициализации обработчика мы могли указать ему сразу следующий за ним обработчик. Затем создаем различные элементы и «кормим» их всех самому первому обработчику.

Посмотрим в лог:

```
Toy found
Handling

Toy not found
Handling using next handler
Electronics found
Handling

Toy not found
Handling using next handler
Electronics not found
Handling using next handler
Found undefined item
Destroying
```

Код примера.

Template Method

Вы заметили как много в нашей жизни шаблонов? Ну к примеру: наше поведение, когда мы приходим в незнакомый дом:

1. Зайти.
2. Поздороваться с хозяевами.
3. Раздеться и разуться, молясь о том, чтоб наши носки не были дырявыми.
4. Пройти и охать, удивляясь какая большая/уютная/клевая квартира.

Или же когда мы приходим в квартиру, в которой происходит ремонт:

1. Зайти.
2. Поздороваться с хозяевами.
3. Не разуваться, поскольку в квартире пол засыпан штукатуркой.
4. Поохать когда хозяин квартиры поведает нам смелость его архитектурной мысли. ☺

В целом, все происходит практически одинаково, но со своей изюминкой в каждом случае. ☺ Наверное, потому это и называется шаблоном поведения. Шаблонный метод задает алгоритму пошаговую инструкцию. Элементы алгоритма же, определяются в наследующих классах.

Сам паттерн – очень интуитивный, и я уверен, что многие давно уже пользуются им. Потому давайте попробуем сделать пример и разобраться с его деталями.

Вернемся к старой практике и будем писать примеры по созданию телефонов.

Для начала напишем шаблонный класс, с помощью которого будем создавать телефон:

```
public class AnyPhoneTemplate {
    public init() { }

    public func makePhone() {
        self.takeBox()
        self.takeCamera()
        self.takeMicrophone()
        self.assemble()
    }

    public func takeBox() {
        print("Taking a box")
    }

    public func takeCamera() {
        print("Taking a camera")
    }

    public func takeMicrophone() {
        print("Taking a microphone")
    }

    public func assemble() {
        print("Assembling everything")
    }
}
```

Как вы уже, наверное, догадались, сам шаблонный метод – это метод `makePhone`, который задает последовательность вызовов методов необходимых для производства телефона. Давайте теперь научимся создавать айфоны:

```
public class IPhoneMaker: AnyPhoneTemplate {
    public override func takeBox() {
        self.design()
        super.takeBox()
    }

    public func design() {
        print("Putting label 'Designed by Apple in California'")
    }
}
```

Как видим, у сборщика яблочных телефонов есть один дополнительный метод `design`, а также перегруженный метод `takeBox`, в котором дополнительно вызывается метод `design`, и после этого вызывается родительский метод `takeBox`.

На очереди – сборка Android:

```
public class AndroidMaker: AnyPhoneTemplate {
    public override func assemble() {
        self.addCPU()
        self.addRAM()
        super.assemble()
    }

    public func addRAM() {
        print("Installing 4 more CPUs")
    }

    public func addCPU() {
        print("Installing 8GB of RAM")
    }
}
```

Итак, у сборщика Андроида – два дополнительных метода и перегруженный метод `assemble`.

Тест здесь, конечно же – элементарный:

```
let iPhoneMaker = IPhoneMaker()
let androidMaker = AndroidMaker()

iPhoneMaker.makePhone()
print("==========")
androidMaker.makePhone()
```

В логе все ожидаемо:

```
Putting label 'Designed by Apple in California'
Taking a box
Taking a camera
Taking a microphone
Assembling everything
==========
Taking a box
Taking a camera
Taking a microphone
Installing 8GB of RAM
Installing 4 more CPUs
Assembling everything
```

Код примера.

Strategy

Если ваша девушка злая, то, скорее всего, вы будете общаться с ней осторожно. Если на вашем проекте завал, то, вероятнее всего, вы не будете предлагать команде дернуть пива вечерком или поиграть в компьютерные игры. В различных ситуациях, у нас могут быть очень разные стратегии поведения. К примеру, в приложении вы можете использовать различные алгоритмы сжатия, в зависимости от того с каким форматом картинки вы работаете, или же как вы хотите после этого картинку использовать. Вот мы и добрались до паттерна «Стратегия».

Также отличным примером может быть MVP-паттерн. В нем презентер говорит view что нужно сделать с UI, и при этом мы можем для одного и того же презентера использовать разные view.

Паттерн «Стратегия» определяет семейство алгоритмов, которые могут быть взаимозаменяемыми:

Когда использовать паттерн:

1. Вам необходимы различные алгоритмы.
2. Вы очень не хотите использовать кучу вложенных if-ов.
3. В различных случаях ваш класс работает по-разному.

Давайте напишем пример: RPG-игра, в которой у вас есть различные стратегии нападения вашими персонажами. ☺ Каждый раз, когда вы делаете ход, ваши персонажи делают определенное действие. Итак, для начала управление персонажами.

Создадим базовую стратегию:

```
public protocol BasicStrategy {
    func actionCharacter1()
    func actionCharacter2()
    func actionCharacter3()
}
```

Как видно из кода стратегии – у нас есть три персонажа, каждый из которых умеет совершать одно действие.

Давайте научим персонажей нападать.

```
public class AttackStrategy: BasicStrategy {
    public init() { }

    public func actionCharacter1() {
        print("Character 1: Attack all enemies!")
    }

    public func actionCharacter2() {
        print("Character 2: Attack all enemies!")
    }

    public func actionCharacter3() {
        print("Character 3: Attack all enemies!")
    }
}
```

Как видим, при использовании такой стратегии наши персонажи нападают на все, что движется. Теперь настало время научить их защищаться:

```
public class DefenceStrategy: BasicStrategy {
    public init() { }

    public func actionCharacter1() {
        print("Character 1: Attack all enemies!")
    }

    public func actionCharacter2() {
        print("Character 2: Healing Character 1!")
    }

    public func actionCharacter3() {
        print("Character 3: Protecting Character 2!")
    }
}
```

Как видим, во время защитной стратегии наши персонажи действуют иначе: первый атакует, второй его лечит, а третий защищает второго. ☺

И теперь как-то нужно это все использовать. Давайте создадим нашего игрока:

```
public class Player {
    private var strategy: BasicStrategy

    public init(strategy: BasicStrategy) {
        self.strategy = strategy
    }

    public func makeAction() {
        self.strategy.actionCharacter1()
        self.strategy.actionCharacter2()
        self.strategy.actionCharacter3()
    }

    public func change(strategy: BasicStrategy) {
        self.strategy = strategy
    }
}
```

Как видим, наш игрок может быть создан только со стратегией (если нужно иметь возможность создавать игрока с пустой стратегией – тогда свойство `strategy` нужно сделать опциональным и убрать его из инициализатора). Также игрок умеет менять стратегию, и действовать по текущей стратегии.

Код для тестирования:

```
let attackStrategy = AttackStrategy()
let player = Player(strategy: attackStrategy)
player.makeAction()
print("==========")

let defenceStrategy = DefenceStrategy()
player.change(strategy: defenceStrategy)
player.makeAction()
```

Собственно, все предельно ясно. ☺ В первом случае наши персонажи будут активно атаковать и не думать о защите. А после смены стратегии – будут действовать более командно и рассудительно.

Посмотрим лог:

```
Character 1: Attack all enemies!
Character 2: Attack all enemies!
Character 3: Attack all enemies!
==========
Character 1: Attack all enemies!
Character 2: Healing Character 1!
Character 3: Protecting Character 2!
```

Код примера.

Command

Стоять, лежать, сидеть – все это команды, которые нам очень часто давали на уроках физкультуры. Так как это очень часто происходит в нашей жизни, глупо было бы предполагать, что кто-нибудь не придумает шаблон с одноименным названием.

Итак, паттерн «Команда»: позволяет инкапсулировать всю информацию, необходимую для выполнения определенных операций, которые могут быть выполнены потом, используя объект команды.

Образно говоря, если взять наш с вами пример физрука, родители давным-давно инкапсулировали в нас команду «сидеть», потому физрук использует ее чтобы мы сели, не объясняя при этом как это сделать.

Когда использовать паттерн:

Ну, собственно, ответ один, и следует он из описания – когда вы хотите инкапсулировать определенную логику в отдельный класс-команду. Отличный пример – операции do/undo. У вас, вероятнее всего, будет, так называемый, CommandManager, который будет запоминать, что делает команда. И, при желании, отменять предыдущее действие, если выполнить команд undo (кстати, это может быть и просто метод).

Собственно, есть два пути реализации этого паттерна:

Для начала создадим базовую команду:

```
public protocol BaseCommand {
    func execute()
    func undo()
}
```

Как видим у нашей команды аж два метода – сделать и вернуть обратно.

Теперь реализация наших команд:

```
public class FirstCommand: BaseCommand {
    private var originalString: String
    private var currentString: String

    public init(argument: String) {
        self.originalString = argument
        self.currentString = argument
    }

    public func execute() {
        self.currentString = "This is a new string"
        self.printString()
        print("Execute command called")
    }

    public func undo() {
        self.currentString = self.originalString
        self.printString()
        print("Undo of execute command called")
    }

    private func printString() {
        print("Current string is equal to \(self.currentString)")
    }
}
```

Как видим, наша первая команда просто умеет менять одну строчку. Причем всегда хранит оригинал, чтобы можно было отменить изменение.

Вторая наша команда:

```
public class SecondCommand: BaseCommand {
    private var originalNumber: Int
    private var currentNumber: Int

    public init(argument: Int) {
        self.originalNumber = argument
        self.currentNumber = argument
    }

    public func execute() {
        self.currentNumber += 1
        self.printNumber()
    }

    public func undo() {
        if self.currentNumber > self.originalNumber {
            currentNumber -= 1
        }

        self.printNumber()
    }

    private func printNumber() {
        print("Current number is \(self.currentNumber)")
    }
}
```

Вторая команда делает все то же самое, только с числом.

Давайте теперь создадим объект, который будет получать команду и выполнять ее:

```
public class CommandExecutor {
    private var commands: [BaseCommand] = []

    public func add(command: BaseCommand) {
        self.commands.append(command)
    }

    public func executeCommands() {
        self.commands.forEach { $0.execute() }
    }

    public func undoAll() {
        self.commands.forEach { $0.undo() }
    }
}
```

Как видим, наш менеджер может получать очередь команд, и выполнять их все, или даже отменять все действия.

Итак, наш тестовый код:

```
let commandExecutor = CommandExecutor()
let firstCommand = FirstCommand(argument: "A test string")
let secondCommand = SecondCommand(argument: 3)

commandExecutor.add(command: firstCommand)
commandExecutor.add(command: secondCommand)
commandExecutor.executeCommands()
print("==========")
commandExecutor.undoAll()
```

И, конечно же, лог:

```
Current string is equal to A new string
Execute command called
Current number is 4
==========
Current string is equal to A test string
Undo of execute command called
Current number is 3
```

Код примера.

Flyweight

Я задумался как объяснить да и перевести этот паттерн на примеры из реальной жизни, и потерпел полнейшее фиаско. ☺ Потому – сразу к описанию и примерам.

Flyweight – паттерн, который помогает нам отделять определенную информацию для того, чтобы в будущем делиться этой информацией со многими объектами.

Как пример, возьмем любую стратегическую игру: представьте, что у вас тысяча солдат одного типа. Если каждый будет лезть на диск и пробовать подгрузить картинку с диска – вероятнее всего у вас или память закончится или производительность просядет. Очень классно этот пример рассмотрен в книге Андрея Будая «Дизайн патерни – просто, як двері».

Потому, не найдя ничего лучше, я решил просто портировать пример.

Когда использовать этот паттерн:

1. У вас ооочень много однотипных объектов в приложении.
2. Много объектов сохранены в памяти, от чего производительность вашего приложения страдает.
3. Вы видите, что несколько объектов, которые могут быть «расшарены» - спасут вас от создания тонны других объектов.

Итак, пример:
Пусть, мы пишем игру, где есть два типа персонажей: гоблины и драконы. Для начала, создадим протокол для всех юнитов:

```
public protocol BasicUnit {
    var name: String { get set }
    var health: Int { get set }
    var image: UIImage? { get }
}
```

Как видим, у каждого юнита есть свойство image, которое является типом UIImage, и может потребовать подгрузки картинки для каждого юнита. Как же сделать загрузку только единожды? Ну, собственно, с этим то и справится наш паттерн.

```
public class FlyweightImageFactory {
    private static var imageDictionary: [String : UIImage] = [:]

    public static func get(imageName: String) -> UIImage? {
        var image = imageDictionary[imageName]
        if image == nil {
            image = UIImage(named: "\(imageName).png")
            imageDictionary[imageName] = image
            print("Loading image of the \(imageName)")
        }

        return image
    }
}
```

Итак, наш Flyweight имеет только один статический метод, который и возвращает картинку по имени. Если картинки под таким именем нет в его словаре – то она подгрузится из бандла, если же есть – мы получим ссылку на нее. Каждый раз, когда картинка грузится из бандла – мы пишем в лог сообщение. Это сделано для того, чтобы увидеть сколько раз происходит подгрузка изображения из бандла.

Теперь нам нужно в конструкторе наших юнитов загружать картинку не напрямую, а через наш паттерн:

```
public class Dragon: BasicUnit {
    public var name: String
    public var health: Int
    public let image: UIImage?

    public init() {
        self.name = "Dragon"
        self.health = 150
        self.image = FlyweightImageFactory.get(imageName: "dragon")
    }
}

public class Goblin: BasicUnit {
    public var name: String
    public var health: Int
    public let image: UIImage?

    public init() {
        self.name = "Goblin"
        self.health = 20
        self.image = FlyweightImageFactory.get(imageName: "goblin")
    }
}
```

Ну и, конечно же, тест:

```
var units: [BasicUnit] = []

for _ in 0..<500 {
    units.append(Dragon())
}

for _ in 0..<500 {
    units.append(Goblin())
}
```

И, как ожидается, хоть мы и создаем тысячу юнитов, лог срабатывает только два раза:

```
Loading image of the dragon
Loading image of the goblin
```

Код примера.

Proxy

Ох, все, кто работает в большой компании – ненавидит доступ к интернету через прокси. ☺ Что делает прокся? – ну многие из нас уверены, что в основном она режет скорость интернета. Хотя, вероятнее всего, она делает еще очень много положительных вещей:

1. Логирует кто куда ходит.
2. Смотрит, чтобы не ходили куда не следует.
3. Смотрит, чтобы по нашему коннекшну к нам не ходили.

… и так далее. Все эти активности взяты из головы, но они показывают использование прокси в реальной жизни – давать стандартный доступ к чему-либо, при этом, обворачивая стандартные вызовы в проксю и добавляя свою логику.

Паттерн «Прокси» – подменяет реальный объект, и шлет ему запросы через свои интерфейсы. При этом, может добавлять дополнительную логику, или создавать реальный объект, если тот еще не создан.

Как пример: вы можете иметь обычных и премиум пользователей приложения. К примеру – премиум пользователи могут скачивать файлы на большей скорости, чем обычные пользователи. Потому, как объекту, который отвечает за скачивание файлов в вашем приложении, не обязательно знать про существование разных типов пользователей, вы оборачиваете этот объект в прокси. Которая в свою очередь знает про таких пользователей, и говорит объекту скачивания на какой скорости пользователь должен получить файл.

Когда использовать паттерн:

1. Возможно, у вас есть два сервера: тестовый и продуктовый. Когда вы дебажите – скорее всего, вы будете использовать тестовый сервер. А когда компилируете приложение для продакшена – скорее всего, реальный. Эту логику можно реализовать в проксе.
2. Добавление различных валидаций и проверок безопасности.
3. Миллион других возможных ситуаций.

Давайте создадим пример. Пусть у нас будет объект, который отвечает за скачку файлов:

```
public class FileDownloader {
    public init() { }

    public func slowDownload() {
        print("Slooooooowly downloading...")
    }

    public func fastDownload() {
        print("Shuuuh! Already downloaded!")
    }
}
```

Как видим, наш объект умеет скачивать быстро и медленно. При этом, ему все равно какой пользователь и есть ли коннект к интернету.

Давайте создадим нашу прокси. У прокси обязательно должна быть ссылка на реальный объект. В добавок к этому – прокси, обычно, наследуется от того же класса:

```
public class FileDownloaderProxy: FileDownloader {
    private var downloader: FileDownloader?
    public var isPremiumUser: Bool = false

    public override func slowDownload() {
        self.checkNetworkConnectivity()
        self.checkDownloader()
        self.downloader?.slowDownload()
    }

    public override func fastDownload() {
        if !self.isPremiumUser {
            self.slowDownload()
            return
        }

        self.checkNetworkConnectivity()
        self.checkDownloader()
        self.downloader?.fastDownload()
    }

    private func checkNetworkConnectivity() {
        print("Checking network connectivity...")
    }

    private func checkDownloader() {
        if self.downloader == nil {
            self.downloader = FileDownloader()
            print("Dowonloader initialized")
        }
    }
}
```

Как видим, проксятник не намного умнее:

1. Он знает про тип пользователя, и даже если вызвали метод `fastDownload`, но пользователь – не премиум, у `downloader`-а будет вызван метод `slowDownload`.
2. Он умеет проверять доступ к интернету (пусть это и просто выписка лога).
3. Он проверяет создан ли сам `downloader`, и если нет – создает его.

Ну что, протестируем:

```
let proxy = FileDownloaderProxy()
proxy.isPremiumUser = false
proxy.fastDownload()

print("==========")

proxy.isPremiumUser = true
proxy.fastDownload()
```

Традиционный лог:

```
Checking network connectivity...
Dowonloader initialized
Slooooooowly downloading...
==========
Checking network connectivity...
Shuuuh! Already downloaded!
```

Код примера.

Memento

Ах, как же не хватает в жизни таких штук как Quick Save и Quick Load. На худой конец Ctrl+Z. Это я вам как геймер давнейший говорю. Частенько, такой функционал очень полезен для реализации в приложении. Очень правильно также защитить наше записанное состояние от других классов, чтобы в них не смогли внести изменения.

Итак, что же это за паттерн такой? Memento – паттерн, который позволяет, не нарушая инкапсуляции, зафиксировать и сохранить внутреннее состояние объекта, чтобы позже восстановить его состояние.

Состояние, как таковое, может сохраняться как в файловую систему, так и в базу данных. Яркий пример использования – может быть сворачивание и выключение вашего приложения – во время выключения приложения вы можете сохранить все данные с формы, или настройки, или еще что вам угодно – в базу данных (через CoreData, например), чтоб при последующем включении, восстановить все это.

Когда использовать паттерн:

1. Вам необходимо сохранять состояние объекта как слепок (snapshot) за определенный период.
2. Вы хотите скрыть интерфейс получения состояния объекта.

В данном паттерне используются три ключевых объекта: Caretaker (объект, который скрывает реализацию сохранения объекта), Originator (собственно, сам объект) и, конечно же, сам Mement (объект, который сохраняет состояние Originator-а).

Давайте небольшой пример:

```
public struct OriginatorState {
    public var intValue: Int
    public var stringValue: String
}
```

Допустим, у нас есть состояние, в котором всего лишь два значения: целое число и строка.

```
public class Originator {
    private var localState: OriginatorState

    public init() {
        self.localState = OriginatorState(intValue: 100,
                                          stringValue: "Hello, World!")
    }

    public func changeValues() {
        self.localState.intValue += 1
        self.localState.stringValue += "!"
        self.showCurrentState()
    }

    public func getState() -> OriginatorState {
        return self.localState
    }

    public func update(state oldState: OriginatorState) {
        self.localState = oldState
        print("\nLoad completed.")
        self.showCurrentState()
    }

    private func showCurrentState() {
        print("\nCurrent state:")
        print("intValue: \(self.localState.intValue)")
        print("stringValue: \(self.localState.stringValue)")
    }
}
```

Как видим, мы можем изменять состояние объекта состояния, а так же получить состояние и загрузить состояние.

Пусть у нас есть Memento – объект, который будет заведовать состоянием нашего объекта:

```
public class Memento {
    private var localState: OriginatorState

    public init(state: OriginatorState) {
        self.localState = state
    }

    public func getState() -> OriginatorState {
        return self.localState
    }
}
```

То есть – наш объект Memento умеет хранить состояние, и, конечно же, отдавать его. ☺

Ну и теперь, соединим все это в единый паззл, создавая Caretaker:

```
public class Caretaker {
    private var originator: Originator!
    private var memento: Memento!

    public init() { }

    public func changeValue() {
        if self.originator == nil {
            self.originator = Originator()
        }

        self.originator.changeValues()
    }

    public func saveState() {
        self.memento = Memento(state: self.originator.getState())
        print("\nSaved state.")
        let mementoState = self.memento.getState()
        print("\nState is:")
        print("intValue: \(mementoState.intValue)")
        print("stringValue: \(mementoState.stringValue)")
    }

    public func loadState() {
        self.originator.update(state: self.memento.getState())
    }
}
```

Как видим, Caretaker умеет держать в себе сохраненное состояние (для примера оно все очень просто, но здесь может быть и стек состояний, и так далее), а также загружать его.

Давайте протестируем:

```
let caretaker = Caretaker()
caretaker.changeValue()
caretaker.saveState()
caretaker.changeValue()
caretaker.changeValue()
caretaker.changeValue()
caretaker.loadState()
```

Лог, как пример работы паттерна:

```
Current state:
intValue: 101
stringValue: Hello, World!!

Saved state.

State is:
intValue: 101
stringValue: Hello, World!!

Current state:
intValue: 102
stringValue: Hello, World!!!

Current state:
intValue: 103
stringValue: Hello, World!!!!

Current state:
intValue: 104
stringValue: Hello, World!!!!!

Load completed.

Current state:
intValue: 101
stringValue: Hello, World!!
```

Код примера.

Послесловие

Пару лет назад я начал заниматься iOS разработкой. Однажды на своей первой работе, я услышал беседу двух других разработчиков. Ухо зацепилось за слова «синглтон» и «фабрика». Тогда я уже догадывался о существовании шаблонов/паттернов, но еще весьма поверхностно.

Я начал искать материалы по этой теме и наткнулся на предыдущую версию этой книги. Она мне помогла «влиться в тему», как говорится, но была сложность с пониманием примеров кода. Дело в том, что для написания новых iOS приложений уже активно использовали Swift, особенно в стартапах (а где же еще искать первую работу ☺). И в принципе, по динамике развития языка было понятно, что он очень скоро станет основным языком, который и будет продвигать Apple.

Тогда я решил, что когда стану более опытным разработчиком – обязательно выпущу новую версию этой книги с примерами на языке Swift. Именно поэтому вы сейчас читаете эти строки.

Кроме примеров кода изменилось и название книги. Раньше там было слово «iOS», а теперь вместо него – слово «Swift». Это потому, что хотя этот язык все еще является довольно узкоспециализированным (в основном на нем пишут приложения под iOS и MacOS), но он развивается и может быть использован в других сферах.

Например, я знаю людей и небольшие компании, которые уже используют его для написания server-side-решений. Также, компания IBM, продвигая свои продукты IBM Watson и IBM Cloud Runtime for Swift, неоднозначно намекает на свою веру в развитие этого языка. Я тоже верю, что у него есть будущее и вне системы Apple.

Спасибо за прочтение. Надеюсь, то что вы почерпнули здесь, будет для вас полезным.