Хрестоматия Swift паттернов. На всякий…

v.2.0

Дима Малеев и Сережа Суханов

**Оглавление**

Привет 3

История изменений 5

Prototype 6

Factory Method 9

Abstract Factory 12

Builder 16

Singleton 21

Adapter 23

# ****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 писали – то утверждать не могу.

Что тебя ждет дальше? Невероятно, но паттерны! И вероятнее всего – грамматические ошибки. Хоть это и вторая версия книги, исправленная и доработанная – старые ошибки хоть и были исправлены (вряд ли все), но к ним добавились еще и новые. Поэтому я торжественно клянусь, что исправлю каждую ошибку, которую ты мне пришлешь на [ssuhanov@gmail.com](mailto:ssuhanov@gmail.com). Я очень надеюсь, что ты не граммар-наци и поможешь мне исправить ошибки. 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. Я не очень знаю, какие лицензии в ходу, да и в данном случае мне откровенно все равно. 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! В этой 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 – исправлены ошибки в описаниях, примеры кода переписаны с Objective-C на Swift. Книга переведена на английский и украинский языки.

# ****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](https://github.com/ssuhanov/ios_patterns_2/tree/master/PrototypePattern.playground).

# 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](https://github.com/ssuhanov/ios_patterns_2/tree/master/FactoryMethodPattern.playground).

# 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](https://github.com/ssuhanov/ios_patterns_2/tree/master/AbstractFactoryPattern.playground).

# 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](https://github.com/ssuhanov/ios_patterns_2/tree/master/BuilderPattern.playground).

# 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](https://github.com/ssuhanov/ios_patterns_2/tree/master/SingletonPattern.playground).

# 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](https://github.com/ssuhanov/ios_patterns_2/tree/master/AdapterPattern.playground).

авы