Хрестоматия iOS паттернов. На всякий…

V1.1

Дима Малеев

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 Objective-C, вероятнее всего их можно использовать и для Mac, но так как я круче програмы чем “Hello, World!” под Mac не писал – то утверждать не могу.
Что тебя ждет дальше? Невероятно, но паттерны! И вероятнее всего грамматические ошибки, потому как я просто физически не умею писать без
ошибок. Но, я торжественно клянусь, что исправлю каждую ошибку которую ты мне пришлешь на diwingless@gmail.com. Я очень надеюсь, что ты не граммар наци, и поможешь мне исправить ошибки, и конечно же оценишь те знания которыми я попытался поделиться с тобой.

Для кого эта книга?

Книга будет полезна всем iOS 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 Obj-C, и прочитав ее вы вырядил сможете написать вторую Angry Birds, однако некоторые проблемы она вам все же решить поможет. Паттерны вообще хорошо знать чтобы структурировать свои знания в голове☺️
☺️

Почему эта книга?

По моему глубоко личному убеждению – знания должны быть бесплатны. Просто, представьте как далеко было бы человечество, если бы у нас культура была бы 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☺️☺️ Потому, я попытаюсь поделиться знаниями которые есть у меня. Что меня радует – я далеко не первый, и даже решение о написании этой книги пришло мне в голову из-за успеха Андрея Будая с его книжкой, 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☺️☺️ Глубоко писать другое, особенно если вы прочитали предыдущий пункт. Распространятся 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!
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, потому имеет возможность меняться и эволюционировать!

V1.0 – базовый контент книги
V1.1 – «подкрашен» код, немного пофиксаны ошибки, книга выложена в опен сорс

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 подгрузку даных из базы, веб сервисов и тому подобных источников. Часто, легче скопировать объект и поменять 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) – прототип – это архетип, которые никогда не должен использоваться, и служит только для создания себе подобных объектов. Хотя, прототип, как архетип – тоже достаточно популярный кейс. Собственно, нам ничего не мешает делать прототипом любой объект который у нас в подчинении.

Поверхностное и глубокое копирование
Тут особенно разницы с .NET нету. Есть указатель, есть значение в куче.

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

К примеру создадим ~~объект~~ класс:

```
class Person {
    var name: String!
    var surname: String!
    var age: Int!
}
```

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

```
let firstPerson = Person()
firstPerson.name = "Dima"
firstPerson.surname = "Surname"

let secondPerson = firstPerson
print("First Person name = \(firstPerson.name!) and surname = \
(firstPerson.surname!)")

secondPerson.name = "Roma"
print("Second Person name = \(secondPerson.name!) and surname = \
(firstPerson.surname!)")
print("First Person name = \(firstPerson.name!) and surname = \
(firstPerson.surname!)")
```

Как видим лог достаточно ожидаемый:

2013-01-21 01:31:20.986 PrototypePattern[1961:11303] First Person name = Dima
2013-01-21 01:31:20.987 PrototypePattern[1961:11303] Second Person name = Roma
2013-01-21 01:31:20.987 PrototypePattern[1961:11303] First Person name = Roma

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~~ класс.

Для таких задач, стоит использовать глубокое копирование, которое в ~~Objective-C~~ сделано в принципе очень похоже как и в .NET:

Для этого надо реализовать протокол ~~NSCopying, и перегрузить~~ Copying:

```
protocol Copying {
    init(instance: Self)
}

extension Copying {
    func copy() -> Self {
        return Self.init(instance: self)
    }
}
```

~~И да, не стоит переживать что мы не реализуем метод “copy”. Он уже есть у класса NSObject. Если вызвать этот метод, и не реализовать copyWithZone – то мы получем ошибку типа NSInvalidArgumentException.~~

Потому ~~интерфейс нашего объекта~~ теперь наш класс будет выглядеть следующим образом:

```
class Person: Copying {
    var name: String!
    var surname: String!

    init() {}

    required init(instance: Person) {
        self.name = instance.name
        self.surname = instance.surname
    }
}
```

Теперь немного изменим код нашего тестового приложения:

```
let firstPerson = Person()
firstPerson.name = "Dima"
firstPerson.surname = "Surname"

let secondPerson = firstPerson.copy()
print("First Person name = \(firstPerson.name!) and surname = \
(firstPerson.surname!)")

secondPerson.name = "Roma"
print("Second Person name = \(secondPerson.name!) and surname = \
(firstPerson.surname!)")
print("First Person name = \(firstPerson.name!) and surname = \
(firstPerson.surname!)")
```

Ну и естественно лог:

2013-01-21 01:48:36.538 PrototypePattern[2090:11303] First Person name = Dima and surname = Surname
2013-01-21 01:48:36.539 PrototypePattern[2090:11303] Second Person name = Roma and surname = Surname
2013-01-21 01:48:36.540 PrototypePattern[2090:11303] First Person name = Dima and surname = Surname

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

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 более 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 это код который использует фабричный метод.

Пример:

Давайте представим, что мы такой неправильный магазин в котором тип товара оценивается по цене:) На данный момент товар есть 2х типов – Игрушки и Одежда.

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

Для начала создадим ~~класс~~ протокол Product. Его реализация нас особо не интересует, хотя он может содержать в себе общие для разных типов товаров методы (сделано
для примера, мы их особо не используем):

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

    func getTotalPrice(sum: Int) -> Int
    func saveObject()
}

extension Product {
    func getTotalPrice(sum: Int) -> Int {
        return price + sum
    }
}
```

Теперь создадим две реализации этого ~~интерфейса~~ протокола.

Игрушка:

```
class Toy: Product {
    var price = 50
    var name = "Toy"

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

И одежда:

```
class Dress: Product {
    var price = 150
    var name = "Dress"

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

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

```
class ProductGenerator {
    func getProduct(price: Int) -> Product? {
        switch price {
        case 0..<100:
            return Toy()
        case 100..<Int.max:
            return Dress()
        default:
            return nil
        }
    }
}
```

Ну вот собственно и все. Теперь просто создадим метод, который будет считать и записывать расходы:

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

Попробуем!

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

Лог:

2013-01-23 23:27:54.223 FactoryMethodPattern[8833:11303] Saving object into Toys database
2013-01-23 23:27:54.226 FactoryMethodPattern[8833:11303] Saving object into Toys database
2013-01-23 23:27:54.226 FactoryMethodPattern[8833:11303] Saving object into Toys database
2013-01-23 23:27:54.227 FactoryMethodPattern[8833:11303] Saving object into Dress database
2013-01-23 23:27:54.227 FactoryMethodPattern[8833:11303] Saving object into Dress database
2013-01-23 23:27:54.228 FactoryMethodPattern[8833:11303] Saving object into Toys database

Abstract Factory

Абстрактная фабрика – еще один очень популярный паттерн, который как и в названии так и в реализации слегка похож на фабричный метод.
Итак, что же делает абстрактная фабрика:

Абстрактная фабрик 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, фабрике же может создавать независимые объекты
2. Чтобы добавить новый тип объекта – надо поменять интерфейс фабрики, в фабричном методе же легко просто поменять внутренности метода, который ответственный за порождение объектов.

Давайте представим ситуацию: у нас есть две фабрики по производству iPhone и iPad. Одна оригинальная, компании Apple, другая – хижина дядюшки Хуа. И вот, мы хотим производить эти товары: если в страны 3-го мира – то товар от дядюшки, в другие страны – товар любезно предоставлен компанией Apple.

Итак, пускай у нас есть фабрика, которая умеет производить и айпады и айфоны:

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

Естественно, нам необходимо реализовать продукты которые фабрика будет производить:

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

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

Но, продукты немного отличаются.

Пускай у нас есть два типа продуктов, оригинальные Apple и продукты которые произведены трудолюбивым дядюшкой Хуа:

```
class AppleIPhone: GenericIPhone {
    var osName = "iOS"
    var productName = "iPhone"
}

class AppleIPad: GenericIPad {
    var osName = "iOS"
    var productName = "iPad"
    var screenSize = 7.7
}
```

Дядюшкофоны:

```
class ChinaPhone: GenericIPhone {
    var osName = "Android"
    var productName = "Chi Huan Hua Phone"
}

class ChinaPad: GenericIPad {
    var osName = "Windows CE"
    var productName = "Buan Que Ipado Killa"
    var screenSize = 12.5
}
```

Разные телефоны, конечно же, производятся на различных фабриках, потому мы просто обязаны их создать! Приблизительно так должны выглядеть фабрика Apple:

```
class AppleFactory: IPhoneFactory {
    func getIPhone() -> GenericIPhone {
        return AppleIPhone()
    }
    func getIPad() -> GenericIPad {
        return AppleIPad()
    }
}
```

Конечно же у нашего китайского дядюшки тоже есть своя фабрика:

```
class ChinaFactory: IPhoneFactory {
    func getIPhone() -> GenericIPhone {
        return ChinaPhone()
    }
    func getIPad() -> GenericIPad {
        return ChinaPad()
    }
}
```

Как видим, фабрики одинаковые, а вот девайсы у них получатся разные☺️☺️

Ну вот собственно и все, мы приготовили все что надо для демонстрации! Теперь, давайте напишем небольшой метод который будет возвращать нам фабрику которую мы хотим (кстати, тут фабричный метод таки будет):

```
var isThirdWorld = true

func getFactory() -> IPhoneFactory {
    return isThirdWorld ? AppleFactory() : ChinaFactory()
}
```

Теперь, давайте как создадим несколько телефонов:

```
isThirdWorld = false

let factory = getFactory()
let ipad = factory.getIPad()
let iphone = factory.getIPhone()

print("IPad named = \(ipad.productName), osname = \(ipad.osName),
screensize = \(ipad.screenSize)")
print("IPhone named = \(iphone.productName), osname = \
(iphone.osName)")
```

Лог будет выглядеть следующим образом:

2013-01-26 20:00:56.663 AbstractFactory[13093:11303] IPad named = Buan Que Ipado Killa, osname = Windows CE, screensize = 12.5
2013-01-26 20:00:56.665 AbstractFactory [13093:11303] IPhone named = Chi Huan Hua Phone, osname = Android

Теперь, просто поменяв значение переменной isThirdWorld на false, и лог будет совсем другой:

2013-01-26 20:02:21.745 AbstractFactory [13115:11303] IPad named = IPad, osname = iOS, screensize = 7.7
2013-01-26 20:02:21.747 AbstractFactory [13115:11303] IPhone named = IPhone, osname = iOS

Builder

Вот представьте что у нас есть фабрика. Но в отличии 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 не тривиальный – к примеру получение данных из базы и манипуляция ими.

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

Пускай у нас есть телефон, который обладает следующими свойствами:

```
class AndroidPhone {
    var osVersion: String!
    var name: String!
    var cpuCodeName: String!
    var RAMsize: Int!
    var osVersionCode: Double!
    var launcher: String!
}
```

Давайте создадим дженерик строителя от которого будут наследоваться конкретные строители:

```
protocol BPAndroidPhoneBuilder {
    var phone: AndroidPhone { get }

    func setOSVersion()
    func setName()
    func setCPUCodeName()
    func setRAMSize()
    func setOSVersionCode()
    func setLauncher()
}
```

Ну а теперь напишем код для конкретных строителей. К примеру, так бы выглядел строитель для дешевого телефона:

```
class LowPricePhoneBuilder: BPAndroidPhoneBuilder {
    var phone = AndroidPhone()

    func setOSVersion() {
        phone.osVersion = "Android 2.3"
    }
    func setName() {
        phone.name = "Low price phone!"
    }
    func setCPUCodeName() {
        phone.cpuCodeName = "Some shitty CPU"
    }
    func setRAMSize() {
        phone.RAMsize = 256
    }
    func setOSVersionCode() {
        phone.osVersionCode = 3.0
    }
    func setLauncher() {
        phone.launcher = "Hia Tsung!"
    }
}
```

И конечно же строительство дорогого телефона:

```
 class HighPricePhoneBuilder: BPAndroidPhoneBuilder {
     var phone = AndroidPhone()

     func setOSVersion() {
         phone.osVersion = "Android 4.1"
     }
     func setName() {
         phone.name = "High price phone!"
     }
     func setCPUCodeName() {
         phone.cpuCodeName = "Some shitty but expensive CPU"
     }
     func setRAMSize() {
         phone.RAMsize = 1024
     }
     func setOSVersionCode() {
         phone.osVersionCode = 4.1
     }
     func setLauncher() {
         phone.launcher = "Samsung Launcher"
     }
 }
```

Кто-то же должен использовать строителей, потому давайте создадим объект который будет с помощью строителей создавать дешевые или дорогие телефоны:

```
class FactorySalesMan {

    private var builder: BPAndroidPhoneBuilder!

    var phone: AndroidPhone {
        return builder.phone
    }

    func setBuilder(builder: BPAndroidPhoneBuilder) {
        self.builder = builder
    }

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

Ну и конечно же куда мы без теста и кода:

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

let salesMan = FactorySalesMan()
salesMan.setBuilder(builder: cheapPhoneBuilder)
salesMan.constructPhone()

var phone = salesMan.phone
print("Phone Name = \(phone.name!), osVersion = \(phone.osVersion!),
cpu code name = \(phone.cpuCodeName!), ram size = \(phone.RAMsize!),
osversion code = \(phone.osVersionCode), launcher = \
(phone.launcher!)")

salesMan.setBuilder(builder: expensivePhoneBuilder)
salesMan.constructPhone()

phone = salesMan.phone
print("Phone Name = \(phone.name!), osVersion = \(phone.osVersion!),
cpu code name = \(phone.cpuCodeName!), ram size = \(phone.RAMsize!),
osversion code = \(phone.osVersionCode), launcher = \
(phone.launcher!)")
```

Как видим, мы создали различных строителей, и сказав директору (FactorySalesMan) какой строитель мы хотим использовать, мы получаем тот девайс который нам необходим:

Традиционный лог:

2013-01-28 00:38:51.863 BuilderPattern[708:11303] Phone Name = Low price phone!, osVersion = Android 2.3, cpu code name = Some shitty CPU, ram size = 256, os version code = 3, launcher = Hia Tsung!
2013-01-28 00:38:51.867 BuilderPattern[708:11303] Phone Name = High price phone!, osVersion = Android 4.1, cpu code name = Some shitty but expensive CPU, ram size = 1024, os version code = 4.1, launcher = Samsung Launcher

Singleton

//реализация паттерна, которая будет приведена тут, подразумевает использование ~~GDC и ARC~~.

Кто вообще бы мог подумать, что Singleton такой не самый просто паттерн в iOS? Вернее, что есть столько версий. Собственно, в .NET, помнится, наблюдалась точно такая же штука, но там в основном были просто апдейты к самой простой версии паттерна. Я вообще считаю, что сколько людей - столько и версий синглтона.

Итак, давайте начнем с простого – с описания.

Singleton - это такой объект, который существует в системе только в единственном экземпляре. Очень часто используется для хранения каких-то глобальных переменных, например настроек приложения.

~~Итак, как и все в Obj-C начнем мы естественно с создания интерфейса:~~

Как видим, обычный объект с одним свойством и класс методом. ~~Естественно, просто от интерфейса мы не получим всего чего ожидаем~~:

```
class SingletonObject {

    private static let obj: SingletonObject = {
        return SingletonObject()
    }()

    class func singleton() -> SingletonObject {
        return obj
    }

    var tempProperty: String!
}
```

Собственно вот и все:) iOS сам за нас позаботится о том, чтобы создан был только один экземпляр нашего объекта.

~~Тут стоит сделать шаг назад и описать проблему, которая является головной болью любого кто более-менее близко работал с потоками:~~

1. ~~Представьте что есть 2 потока.~~
2. ~~И тут каждый, одновременно создает singleton.Вроде бы и должен создаться только один объект, но потому что все происходит в один момент – бывают случаи когда создается два объекта.~~

~~“Но ведь можно сделать проверку на nil!” – скажете Вы.~~

~~А теперь, представьте более сложную ситуацию: объекта singleton не существует. Два потока хотят его создать одновременно:~~

1. ~~Поток 1 делает проверку или объект существует. Видит что его нет, и проходит этап проверки.~~
2. ~~Поток 2 делает проверку на существование объекта, и хоть и поток 1 проверку УЖЕ прошел, но объект ЕЩЕ не существует.~~

~~Для решения таких проблем, в .NET использовали locks – блокирование кода, для других потоков, пока он исполняется в каком – либо потоке. Собственно dispatch_once делает тоже самое – он просто синхронный:) Потому, ни один поток не может зайти в этот код, пока он занят.~~

~~Собственно, без GCD такое создать тоже можно, тогда наш код бы выглядел следующим образом:~~

```
+(SingletonObject *) singleton
{
    static SingletonObject *singletonObject = nil;
    @synchronized(self)
    {
            if (singletonObject == nil )
            {
                singletonObject = [[self alloc] init];
            }
    }

    return singletonObject;
}
```

~~Получится тоже самое. Единственное, что dispatch_once() по документации быстрее:) Ну и семантически более правильнее.~~

~~Можно вообще бахнуть по хардкору, и создать макрос:~~

```
#define
DEFINE_SHARED_INSTANCE_USING_BLOCK(block) \
static dispatch_once_t pred = 0; \
__strong static id _sharedObject = nil; \
dispatch_once(&pred, ^{ \
_sharedObject = block(); \ }); \
return _sharedObject; \
```

~~Тогда сама реализация создания объекта будет выглядеть следующим образом:~~

```
+(SingletonObject *) singleton
{
    DEFINE_SHARED_INSTANCE_USING_B
        LOCK(^{ return [[self alloc] init];
    });
}
```

Ну, а использование простое:

```
SingletonObject.singleton().tmpProperty = "Hello 2 You!"
print(SingletonObject.singleton().tmpProperty)
```

Adapter

Тяжело найти более красочно 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 треугольные. Следовательно – потому вилки на зарядный 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 конвертит интерфейс класса – на такой, который ожидается.

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

В нашей с вами проблеме:

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

Имплементация паттерна Адаптер в ~~Objective-C~~ может быть 2 (вероятно даже больше, но я вижу две):

Итак, первая – это простенькая имплементация. Пускай у нас есть объект Bird, который реализует протокол BirdProtocol:

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

//реализация

```
class Bird: BirdProtocol {
    func sing() {
        print("Tew-tew-tew")
    }
    func fly() {
        print("OMG! I am flying!")
    }
}
```

И пускай у нас есть объект Raven, у которого есть свой интерфейс:

```
class Raven {
    func flySearchAndDestroy() {
        print("I am flying and seak for killing!")
    }
    func voice() {
        print("Kaaaar-kaaaaar-kaaaaaaar!")
    }
}
```

Чтобы использовать ворону в методах которые ждут птицу:) стоит создать так называемый адаптер:

```
class RavenAdapter: BirdProtocol {

    private var raven: Raven

    init(adaptee: Raven) {
        raven = adaptee
    }

    func sing() {
        raven.voice()
    }
    func fly() {
        raven.flySearchAndDestroy()
    }
}
```

Ну и конечно же тест:

```
func makeTheBirdTest(bird: BirdProtocol) {
    bird.fly()
    bird.sing()
}

let simpleBird = Bird()
let simpleRaven = Raven()

let ravenAdapter = RavenAdapter(adaptee: simpleRaven)

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

Результат можно легко увидеть в логе:

2013-02-03 15:43:14.447 AdapterPattern[5985:11303] OMG! I am flying!
2013-02-03 15:43:14.449 AdapterPattern[5985:11303] Tew-tew-tew
2013-02-03 15:43:14.449 AdapterPattern[5985:11303] I am flying and seak for killing!
2013-02-03 15:43:14.450 AdapterPattern[5985:11303] Kaaaar-kaaaaar-kaaaaaaar!

Теперь более сложная реализация, которая все еще зависит от протоколов, но уже использует делегирование. Вернемся к нашему несчастному ноутбуку и зарядке: Допустим у нас есть базовый ~~класс~~ протокол Charger:

```
protocol Charger {
    func charge()
}
```

И есть протокол для европейской зарядки:

```
protocol EuropeanNotebookChargerDelegate {
    func chargetNotebookRoundHoles(charger: Charger)
}

extension EuropeanNotebookChargerDelegate {
    func chargetNotebookRoundHoles(charger: Charger) {
        print("Charging with 220 and round holes!")
    }
}
```

Если сделать просто реализацию – то получится тоже самое, что и в прошлом примере:) Потому, давайте добавим делегат:

```
class EuropeanNotebookCharger: Charger, EuropeanNotebookChargerDelegate {

    var delegate: EuropeanNotebookChargerDelegate!

    init() {
        delegate = self
    }

    func charge() {
        delegate.chargetNotebookRoundHoles(charger: self)
    }
}
```

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

```
delegate.chargetNotebookRoundHoles(charger: self)
```

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

```
class USANotebookCharger {
    func chargeNotebookRectHoles() {
        print("Charge Notebook Rect Holes")
    }
}
```

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

```
class USANotebookEuropeanAdapter: Charger, EuropeanNotebookChargerDelegate {

    var usaCharger: USANotebookCharger!

    init(charger: USANotebookCharger) {
        usaCharger = charger
    }

    func chargetNotebookRoundHoles(charger: Charger) {
        usaCharger.chargeNotebookRectHoles()
    }

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

}
```

Как видим, наш адаптер реализует интерфейс EuropeanNotebookChargerDelegate и его метод chargetNotebookRoundHoles. Потому, 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 chargetNotebookRoundHoles, будет вызван этот метод нашего адаптера, который в свою очередь вызывает метод зарядки США:)

Давайте посмотрим тест код и вывод лога:

```
func makeTheNotebookCharge(charger: Charger) {
    charger.charge()
}

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

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

Лог нам выведет:

2013-02-03 15:57:42.624 AdapterPattern[6179:11303] Charging with 220 and round holes!
~~2013-02-03 15:57:42.626 AdapterPattern[6179:11303] C'mon I am charging~~
2013-02-03 15:57:42.626 AdapterPattern[6179:11303] Charge Notebook Rect Holes
~~2013-02-03 15:57:42.627 AdapterPattern[6179:11303] C'mon I am charging~~

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 наушникам. 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 поменяна, 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 имело влияния на клиентский код.

Давайте создадим теперь базовую абстракцию наушников:

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

И теперь два элемента – дорогие наушники и дешевые:)

```
    //Наушники обычные - китайские

class CheapHeadphones: BaseHeadphones {
    func playSimpleSound() {
        print("beep - beep - bhhhrhrhrep")
    }
    func playBassSound() {
        print("puf - puf - pufhrrr")
    }
}
```

```
    //наушники дорогие, тоже китайские


class ExpensiveHeadphones: BaseHeadphones {
    func playSimpleSound() {
        print("Beep-Beep-Beep Taram - Rararam")
    }
    func playBassSound() {
        print("Bam-Bam-Bam")
    }
}
```

И собственно плеер, через который мы будем слушать музыку:

```
class MusicPlayer {
    var headPhones: BaseHeadphones!

    func playMusic() {
        headPhones.playBassSound()
        headPhones.playBassSound()
        headPhones.playSimpleSound()
        headPhones.playSimpleSound()
    }
}
```

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

```
let p = MusicPlayer()

let ch = CheapHeadphones()
let ep = ExpensiveHeadphones()

p.headPhones = ch
p.playMusic()

p.headPhones = ep
p.playMusic()
```

И конечно же log:

2013-02-06 23:03:52.378 BridgePattern[3397:c07] puf – puf – pufhrrr
2013-02-06 23:03:52.379 BridgePattern[3397:c07] puf – puf – pufhrrr
2013-02-06 23:03:52.380 BridgePattern[3397:c07] beep – beep – bhhhrhrhrep 2013-02-06 23:03:52.380 BridgePattern[3397:c07] beep – beep – bhhhrhrhrep
2013-02-06 23:03:52.380 BridgePattern[3397:c07] Bam-Bam-Bam
2013-02-06 23:03:52.381 BridgePattern[3397:c07] Bam-Bam-Bam
2013-02-06 23:03:52.381 BridgePattern[3397:c07] Beep-Beep-Beep Taram – Rararam
2013-02-06 23:03:52.381 BridgePattern[3397:c07] Beep-Beep-Beep Taram – Rararam

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 надо:

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, секретный код, дождаться снятия денег, распечатать билет.
3. Приехать в кинотеатр, припарковать 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 даную 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 интерфейса к большому количеству интерфейсов системы, в следствии чего систему стает гораздо проще в использовании.

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

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

    func findLocationToTravel(location: String) {
        print("So you wanna travell to " + location)
    }
    func makeARoute() {
        print("Okay, to travell to this location we are using google
maps....")
    }
}
```

Естественно нам необходима сама система заказа транспорта и собственно путешествия:

```
class TravellEngine {
    func findTransport() {
        print("Okay, to travell there you will probabply need dragon!
Arghhhhh")
    }
    func orderTransport() {
        print("Maaaam, can I order a dragon?... Yes... Yes, green one... Yes,
with fire!... No, not a dragon of death... Thank you!")
    }
    func travel() {
        print("Maaan, you are flying on dragon!")
    }
}
```

Ну и какие же путешествия без билетика:

```
class TicketPrinitingSystem {
    func createTicket() {
        print("Connecting to our ticketing system...")
    }
    func printingTicket() {
        print("Hmmm, ticket for travelling on the green
dragon.Interesting...")
    }
}
```

А теперь, давайте создадим единый доступ ко всем этим системам:

```
 class TravellSystemFacade {
     func travellTo(location: String) {
         let pf = PathFinder()
         let te = TravellEngine()
         let tp = TicketPrinitingSystem()

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

         te.findTransport()
         te.orderTransport()

         tp.createTicket()
         tp.printingTicket()

         te.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 = TravellSystemFacade()

facade.travellTo(location: "Lviv")
```

Давайте посмотрим лог:

2013-02-09 17:46:28.442 FacadePattern[2410:c07] Finding your location. Hmmm, here you are!
2013-02-09 17:46:28.444 FacadePattern[2410:c07] So you wanna travell to Lviv
2013-02-09 17:46:28.445 FacadePattern[2410:c07] Okay, to travell to this location we are using google maps….
2013-02-09 17:46:28.446 FacadePattern[2410:c07] Okay, to travell there you will probabply need dragon!Arghhhhh
2013-02-09 17:46:28.446 FacadePattern[2410:c07] Maaaam, can I order a dragon?… Yes… Yes, green one… Yes, with fire!… No, not a dragon of death… Thank you!
2013-02-09 17:46:28.447 FacadePattern[2410:c07] Connecting to our ticketing system…
2013-02-09 17:46:28.447 FacadePattern[2410:c07] Hmmm, ticket for travelling on the green dragon.Interesting…
2013-02-09 17:46:28.448 FacadePattern[2410:c07] Maaan, you are flying on dragon!

Mediator

Медиатор – паттерн которые определяет внутри себя объект, в котором реализуется взаимодействие между некоторым количеством объектов. При этом эти объекты, могут даже не знать про существования 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. При чем кондиционер, не обязательно что знает про существования датчика температуры. Есть центральный компьютер, который получает сигналы от каждого из устройств и понимает, что делать в том или ином случает.
2. Тяжело переиспользовать объект, так как он взаимодействует и коммуницирует 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 контролов, и все правила взаимодействия мы прописываем в контроллере. Элементарно.

И все же пример не будет лишним Давайте все же создадим пример который показывает создание аля умного дома.

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

```
class SmartHousePart {
    private var processor: CentrallProcessor

    init(processor: CentrallProcessor) {
        self.processor = processor
    }

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

Теперь, создадим сердце нашего умного дома:

```
class CentrallProcessor {

    var thermometer: Thermometer!
    var condSystem: ConditioningSystem!

    func valueChanged(part: SmartHousePart) {
        print("Value changed! We need to do smth!")
        if part is Thermometer {
            condSystem.startCondition()
        }
    }
}
```

~~Тут очень интересный момент – класс CentrallProcessor должен знать про существование SmartHousePart, собственно и SmartHousePart должен знать про существование CentrallProcessor. Если все сделать простым import – то проект не скомпилируется.Потому, в SmartHousePart.h мы добавили @class CentrallProcessor; для того чтобы XCode знал что за объект будет использован, и при этом не импортил файл заголовков CentrallProcessor.~~

~~Заголовки же мы импортим в SmartHousePart.m.~~

Дальше в классе CentrallPart 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:

```
class Thermometer: SmartHousePart {
    private var temperature: Int!
    func temperatureChanged(temperature: Int) {
        self.temperature = temperature
        numbersChanged()
    }
}

class ConditioningSystem: SmartHousePart {
    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 CentrallProcessor.

Код для тестинга:

```
let proccessor = CentrallProcessor()
let therm = Thermometer(processor: proccessor)
let condSystem = ConditioningSystem(processor: proccessor)

proccessor.condSystem = condSystem
proccessor.thermometer = therm

therm.temperatureChanged(temperature: 45)
```

И конечно же лог:

2013-02-12 18:45:06.790 MediatorPattern[8809:c07] Value changed! We need to do smth!
2013-02-12 18:45:06.793 MediatorPattern[8809:c07] Oh, the change is temperature
2013-02-12 18:45:06.793 MediatorPattern[8809:c07] Conditioning…

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 отнесут. 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 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 кто подписался.

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 чтоб состояние изменилось в других объектах, при чем количество объектов может быть разное.

Реализация этого паттерна возможно двумя способами:

1. Notification

Notificaiton – механизм использования возможностей NotificationCenter самой операционной системы. Использование NSNotificationCenter позволяет объектам комуницировать, даже не зная друг про друга. Это очень удобно использовать когда у вас в паралельном 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:
Notification.Name("broadcastMessage"), object: self)
NotificationCenter.default().post(broadCastMessage)
```

Как видим мы создали объект типа NSNotification 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:

```
NotificationCenter.default().addObserver(self, selector: #selector(update),
name: Notification.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 их типа. Давайте начнем с того, что создадим протокол Subject и класс Observer:

```
func ==(lhs: StandardObserver, rhs: StandardObserver) -> Bool {
    return lhs.hashValue == rhs.hashValue
}

class StandardObserver: Hashable {
    var hashValue: Int { return "\(Mirror(reflecting:
self).subjectType)".hashValue }
    func valueChanged(name: String, value: String) {}
}

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

Теперь, давайте создадим реализацию Subject:

```
class StandardSubjectImplementation: StandardSubject {

    private var name: String!
    private var value: String!

    var observerCollection = Set<StandardObserver>()

    func addObserver(observer: StandardObserver) {
        observerCollection.insert(observer)
    }
    func removeObserver(observer: StandardObserver) {
        observerCollection.remove(observer)
    }
    func notifyObjects() {
        for observer in observerCollection {
            observer.valueChanged(name: name, value: value)
        }
    }
    func changeValue(name: String, value: String) {
        self.name = name
        self.value = value
        notifyObjects()
    }
}
```

Ну и куда же без обсерверов:

```
class SomeSubscriber: StandardObserver {
    override func valueChanged(name: String, value: String) {
        print("And some subscriber tells: Hmm, value \(value) changed to \
(name)")
    }
}
class OtherSubscriber: StandardObserver {
    override func valueChanged(name: String, value: String) {
        print("And some another subscriber tells: Hmm, value \(value) changed
to \(name)")
    }
}
```

Собственно – все:) теперь демо-код:

```
let subj = StandardSubjectImplementation()
let someSubscriber = SomeSubscriber()
let otherSubscriber = OtherSubscriber()

subj.addObserver(observer: someSubscriber)
subj.addObserver(observer: otherSubscriber)

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

И естественно log:

2013-02-16 17:31:43.176 ObserverPattern[24332:c07] And some subscriber tells: Hmm, value strange value changed to newValue
2013-02-16 17:31:43.177 ObserverPattern[24332:c07] And some another subscriber tells: Hmm, value strange value changed to newValue

Ну и конечно же без использования KVO описание паттерна выглядело бы неполным.

Одна из моих самых любимых особенностей ~~Obj-C~~ – это key-value coding. Про него очень клево описанно в официальной документации, но если объяснять на валенках – то это возможность изменять значения свойств объекта с помощью строчек – которые указывают именно само название свойства. Как пример такие две конструкции идентичны:

```
changeableProperty = "new value"
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 compilant, любым

объектом. На самом деле легче объяснить на примере.

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

```
class KVOSubject: NSObject {
    var changeableProperty: String!
}
```

И создадим объект который будет слушать изменение свойства changeableProperty:

```
class KVOObserver: NSObject {
    override func observeValue(forKeyPath keyPath: String?, of object:
AnyObject?, change: [NSKeyValueChangeKey : AnyObject]?, context:
UnsafeMutablePointer<Void>?) {
        print("KVO: Value changed;")
    }
}
```

Как видим, этот класс реализует только один метод: observeValueForKeyPath. 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.setValue("new value", forKey: "changeableProperty")
kvoSubj.removeObserver(kvoObserver, forKeyPath: "changeableProperty")
```

Как видно из примера, мы для объекта за которым мы 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 так как мы все это проделываем на нажатие кнопки – в конце мы удаляем наблюдателя с нашего объекта, что бы память не текла. Лог говорит сам за себя:

2013-02-17 11:41:58.051 ObserverPattern[26689:c07] KVO: Value changed;

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 10 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 даных, 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 даных таким образом.

Давайте разберем более детально структуру:

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

Давайте пример!

Начнем с создания протокола для наших объектов:

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

Создадим ~~объект~~ класс листа:

```
class LeafObject: CompositeObjectProtocol {

    var leafValue: String!

    func getData() -> String {
        return "\n" + "<\(leafValue!)/>"
    }
    func addComponent(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:

```
class Container: CompositeObjectProtocol {

    private var components = [CompositeObjectProtocol]()

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

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

        valueToReturn += "</ContainerValues>"

        return valueToReturn
    }
    func addComponent(component: CompositeObjectProtocol) {
        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()
object.leafValue = "level1 value"
rootContainer.addComponent(component: object)

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

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

print(rootContainer.getData())
```

И конечно же лог:

```
2013-02-17 13:04:09.470
CompositePattern[27392:c07] <ContainerValues>
      <level1 value/>
      <ContainerValues
      >
            <level2 value/>
      </ContainerValues>
      <ContainerValues>
            <level2 value 2/
      > </ContainerValues>
</ContainerValues>
```

Iterator

Я задумался о том, какой бы пример из жизни привести, чтобы показать пример как работает паттерн итератор, и оказалось что это не такой 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 надо:

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

При внутренем:

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

Когда использовать итератор:

1. Вам необходимо доступиться к объектам коллекции, без того чтобу щупать внутренности коллекции.

2. Вам надо обходить объекты коллекции различными способами (вспомните Композит – у вас коллекция может быть древовидной)

3. Вам необходимо дать унифицированный интерфейс для различных подходов итерации.

Самый просто пример внешнего итератора, это использование класса ~~NSEnumerator~~:

```
let internallArrayCollection = ["A", "B", "C"]
var iterator = internallArrayCollection.makeIterator()

while let str = iterator.next() {
    print(str)
}
```

Как видим, мы просто вызываем у объекта internallArrayCollection метод ~~objectEnumerator~~ – и получаем необходимый нам итератор. Вообще можно не заморачиваться, и использовать обычный цикл for:

```
for str in internallArrayCollection {
    print(str)
}
```

Я не уверен что смогу правильно объяснить разницу между созданием итератора и цикла for (если она есть), потому этот момент будет упущен.

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

```
var block: (Int, String, inout Bool) -> () = { index, str, stop in
    print("Index: \(index) Value: \(str)")
    if index == 1 {
        stop = true
    }
}

for (index, str) in internallArrayCollection.enumerated() {
    var stop = false
    block(index, str, &stop)
    if stop { break }
}
```

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

Приятно же:)

Теперь давайте создадим свой итератор, а то и два:) Пускай у нас будет коллекция товаров, одни их них будут сломаны, другие же – целыми. Создадим два итератора, которые будут бегать по разным типам товаров. Итак, для начала сам клас товаров:

```
class ItemInShop {
    let name: String
    let isBroken: Bool

    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 собственно товары то и будут:

```
class ShopWarehouse {
    private var allItems = [ItemInShop]()

    var goodItemsIterator: GoodItemsIterator {
        return GoodItemsIterator(items: allItems)
    }

    var badItemsIterator: BadItemsIterator {
        return BadItemsIterator(items: allItems)
    }

    func addItem(item: ItemInShop) {
        allItems.append(item)
    }
}
```

Как видим, наш склад умеет добавлять товары, а также возвращать два таинственных объекта под названием GoodItemsIterator и BadItemsIterator.

Собственно их назначение очевидно, давайте посмотрим на реализацию. Для начала создадим базовый ~~класс~~ протокол для обоих:

```
protocol BasicIterator: IteratorProtocol {
    init(items: [ItemInShop])
    func allObjects() -> [ItemInShop]
}
```

Как видим, это просто интерфейс, который предполагает реализацию ~~3х методов~~: инициализация, вернуть все объекты, и вернуть следующий объект. Давайте создадим два итератора как и задумывалось:

```
class BadItemsIterator: BasicIterator {

    typealias Element = ItemInShop

    private var items: [ItemInShop]
    private var internalIterator: IndexingIterator<[ItemInShop]>

    required init(items: [ItemInShop]) {
        self.items = items.filter { $0.isBroken }
        internalIterator = self.items.makeIterator()
    }

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

    func next() -> Element? {
        return internalIterator.next()
    }
}
```

~~Я не привожу код GoodItemsIterator, потому как разнится они будут только в одной строчке:~~

```
class GoodItemsIterator: BasicIterator {

    typealias Element = ItemInShop

    private var items: [ItemInShop]
    private var internalIterator: IndexingIterator<[ItemInShop]>

    required init(items: [ItemInShop]) {
        self.items = items.filter { !$0.isBroken }
        internalIterator = self.items.makeIterator()
    }

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

    func next() -> Element? {
        return internalIterator.next()
    }
}
```

Как видим во время инициализации, мы создаем свою копию данных, в которых только плохие товары. Так же создаем свой внутренний итератор, из стандартных ~~Cocoa~~.

Ну что, тестим:

```
let shopWarehouse = ShopWarehouse()
shopWarehouse.addItem(item: ItemInShop(name: "Item1", isBroken: false))
shopWarehouse.addItem(item: ItemInShop(name: "Item2", isBroken: false))
shopWarehouse.addItem(item: ItemInShop(name: "Item3", isBroken: true))
shopWarehouse.addItem(item: ItemInShop(name: "Item4", isBroken: true))
shopWarehouse.addItem(item: ItemInShop(name: "Item5", isBroken: false))
```

Сам тест:

```
let goodIterator = shopWarehouse.goodItemsIterator
let badIterator = shopWarehouse.badItemsIterator

while let element = goodIterator.next() {
    print("Good Item = \(element.name)")
}

while let element = badIterator.next() {
    print("Bad Item = \(element.name)")
}
```

И конечно же лог:

2013-02-25 01:18:10.401 IteratorPattern[5000:c07] Good Item = Item1
2013-02-25 01:18:10.403 IteratorPattern[5000:c07] Good Item = Item2
2013-02-25 01:18:10.403 IteratorPattern[5000:c07] Good Item = Item5
2013-02-25 01:18:10.404 IteratorPattern[5000:c07] Bad Item = Item3
2013-02-25 01:18:10.405 IteratorPattern[5000:c07] Bad Item = Item4

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, какая 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.

Чтож, давайте вернемся к примеру из прошлой статьи, только теперь сложнее – у нас есть несколько складов, в каждом складе может хранится товар. Один визитер будет смотреть склады, другой визитер будет называть цену товара в складе.

Итак, для начала сам товар:

```
class WarehouseItem {
    let name: String
    let isBroken: Bool
    let price: Int

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

И естественно сам склад:

```
class Warehouse {
    private var itemsArray = [WarehouseItem]()

    func addItem(item: WarehouseItem) {
        itemsArray.append(item)
    }

    func accept(visitor: BasicVisitor) {
        for item in itemsArray {
            visitor.visit(object: item)
        }
    }
}
```

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

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

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

```
class QualityCheckerVisitor: BasicVisitor {
    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
        }
    }
}
```

Если почитать код, то сразу видно что визитер при вызове своего метода visit определяет тип объекта который ему передался, и выполняет определенные функции в зависимости от этого типа. Данный объект просто говорит или вещи на
складе поломаны, а так же что ему нравится склад:)

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

В принципе этот визитер делает тоже самое, только в случае склада он признается что растерян, а в случае товара говорит цену товара!

Теперь давайте запустим то что у нас получилось! Код генерации тестовых даных:

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

И собственно сам тестовый код:

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

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

Итак, при вызове метода accept нашего склада, визитер в начале 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:

2013-02-26 00:19:47.756 VisitorPattern[8748:c07] Hmmm, I don't know how much Warehouse costs!
2013-02-26 00:19:47.759 VisitorPattern[8748:c07] Item: Item1 have price = 25
2013-02-26 00:19:47.759 VisitorPattern[8748:c07] Item: Item2 have price = 32
2013-02-26 00:19:47.760 VisitorPattern[8748:c07] Item: Item3 have price = 45
2013-02-26 00:19:47.761 VisitorPattern[8748:c07] Item: Item4 have price = 33
2013-02-26 00:19:47.762 VisitorPattern[8748:c07] Item: Item5 have price = 12
2013-02-26 00:19:47.763 VisitorPattern[8748:c07] Item: Item6 have price = 78
2013-02-26 00:19:47.763 VisitorPattern[8748:c07] Item: Item7 have price = 34
2013-02-26 00:19:47.764 VisitorPattern[8748:c07] Item: Item8 have price = 51
2013-02-26 00:19:47.765 VisitorPattern[8748:c07] Item: Item9 have price = 25
2013-02-26 00:19:47.765 VisitorPattern[8748:c07] Hmmm, nice warehouse!
2013-02-26 00:19:47.766 VisitorPattern[8748:c07] Item: Item1 is pretty cool!
2013-02-26 00:19:47.767 VisitorPattern[8748:c07] Item: Item2 is pretty cool!
2013-02-26 00:19:47.767 VisitorPattern[8748:c07] Item: Item3 is broken

2013-02-26 00:19:47.768 VisitorPattern[8748:c07] Item: Item4 is pretty cool!
2013-02-26 00:19:47.769 VisitorPattern[8748:c07] Item: Item5 is pretty cool!
2013-02-26 00:19:47.837 VisitorPattern[8748:c07] Item: Item6 is broken
2013-02-26 00:19:47.837 VisitorPattern[8748:c07] Item: Item7 is broken
2013-02-26 00:19:47.837 VisitorPattern[8748:c07] Item: Item8 is pretty cool!
2013-02-26 00:19:47.838 VisitorPattern[8748:c07] Item: Item9 is pretty cool!

Decorator

//уже после того как я решил оформить все в книгу, в блог посте в котором изначально я описал этот паттерн, некто по имени Саша сказал что я привожу не классический пример этого шаблона. И он действительно прав, пример тут использует категории, хотя в классическом исполнении сам декоратор должен быть отдельным объектом. Классическая реализация будет добавлена позже

Классный пример декоратора – различные корпуса для новых телефонов. Как-то я сразу с конца начал:) Для начала у нас есть телефон. Но так как телефон дорогой, мы будем очень счастливы если он не разобьется при любом падении – потому мы покупаем чехол для него. То есть, к уже существующему предмету мы добавили функционал защиты от падения. Ну еще мы взяли стильный чехол – теперь наш телефон еще и выглядит отлично. А потом мы докупили съемный объектив, с помощью которого можно делать фотографии с эффектом “рыбьего глаза”. Декорировали наш телефон дополнительным функционалом:)

Вот, приблизительно так выглядит реально описание паттерна декоратор. Теперь описание 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 класс – опциональный

Радость ~~Objective-C~~ в данном случае – это использование категорий. Я не буду детально описывать категории в этой книге, но в двух словах все же расскажу: Категории – это возможность расширить любой объект дополнительными методами ~~(ТОЛЬКО МЕТОДАМИ)~~ без унаследования от него. Давайте возьмем супер простой пример – декорирование Cocoa классов. К примеру добавим новый метод для объекта NSDate:

К примеру, нам нужно иметь возможность любую дату в нашем приложении как то определенно отформатировать и получить в виде строчки. Для начала создадим категорию:

```
extension Date {
    var convertDateToString: String {
        let formatter = DateFormatter()
        formatter.dateFormat = "yyyy/dd/MM"
        return formatter.string(from: self)
    }
}
```

Как видим наша категория определяет только один ~~метод~~ “convertDateToString”, который дату форматирует в какой-то совсем странный формат, но у нас такая задача:)
~~Теперь план-капкан, сделать эту категорию пользуемой для всех объектов NSDate в нашем приложении. Для этого в файле appName-Prefix, добавляем строчку:~~

```
#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
    #import "NSDate+StringDate.h"
#endif
```

Вы будете смеяться, но в принципе вот и все:) Примерный код тестирования выглядит следующим образом:

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

Традиционный log:

2013-03-01 00:30:18.328 DecoratorPattern[11731:c07] Date is 2013/01/03

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 объекте по цепочке объектов-обработчиков, пока не будет найден необходимый объект обработчик.

Когда использовать этот паттерн:

1. У вас более чем один объект-обработчик.
2. У вас есть несколько объектов обработчика, при этом вы не хотите специфицировать который объект должен обрабатывать в данный момент времени.

Как всегда – пример:

Представим что у нас есть конвейер, который обрабатывает различные предметы которые на нем: игрушки, электронику и другие.
Для начала создадим классы объектов которые могут быть обработаны нашими обработчиками:

```
    //базовый объект

protocol BasicItem {}


    //игрушка

class Toy: BasicItem {}

    //электроника

class Electronics: BasicItem {}


    //и мусор

class Trash: BasicItem {}
```

Теперь создадим обработчики:

```
protocol BasicHandler {
    var nextHandler: BasicHandler? { get set }
    func handleItem(item: BasicItem)
}
```

Как видим, наш базовый обработчик, умеет обрабатывать объекты типа BasicItem.
И самое важное – он имеет ссылку на следующий обработчик (как в нашей очереди, про людей передающие посылку).
Давайте создадим код обработчика игрушки:

```
class ToysHandler: BasicHandler {

    var nextHandler: BasicHandler?

    func handleItem(item: BasicItem) {
        switch item {
        case is Toy:
            print("Toy found. Handling")
        default:
            print("Toy not found. Handling using next handler")
            nextHandler?.handleItem(item: item)
        }
    }
}
```

Как видим, если обработчик получает объект класса Toy – то он его обрабатывает, если нет – то обработчик передает объект следующему обработчику.
По аналогии создадим два следующих обработчика: для электроники, и мусора:

```
    //хэндлер электроники

class ElectronicsHandler: BasicHandler {
    var nextHandler: BasicHandler?

    func handleItem(item: BasicItem) {
        switch item {
        case is Electronics:
            print("Electronics found. Handling")
        default:
            print("Electronics not found. Handling using next handler")
            nextHandler?.handleItem(item: item)
        }
    }
}


    //хэндлер мусора

class OtherItemsHandler: BasicHandler {
    var nextHandler: BasicHandler?

    func handleItem(item: BasicItem) {
        print("Found undefined item. Destroying")
    }
}
```

Как видим OtherItemsHandler в случае, когда до него дошло дело – объект уничтожает, и не пробует дергать следующий обработчик.

Давайте тестировать:

```
let toysHandler = ToysHandler()
let electronicsHandler = ElectronicsHandler()
let otherItemHandler = OtherItemsHandler()

electronicsHandler.nextHandler = otherItemHandler
toysHandler.nextHandler = electronicsHandler

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

toysHandler.handleItem(item: toy)
toysHandler.handleItem(item: electronic)
toysHandler.handleItem(item: trash)
```

Как видим мы в начале создаем обработчики, потом скрепляем их в цепь, и пытаемся обработать различные элементы.Традиционно лог:

2013-03-02 15:35:35.668 ChainOfResponsibility[16777:c07] Toy found. Handling
2013-03- 02 15:35:35.671 ChainOfResponsibility[16777:c07] Toy not found. Handling using next handler
2013-03- 02 15:35:35.672 ChainOfResponsibility[16777:c07] Electronics found. Handling
2013-03- 02 15:35:35.673 ChainOfResponsibility[16777:c07] Toy not found. Handling using next handler
2013-03-02 15:35:35.673 ChainOfResponsibility[16777:c07] Electronics not found. Handling using next handler
2013 -03-02 15:35:35.674 ChainOfResponsibility[16777:c07] Found undefined item. Destroying

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, молясь о том что у нас носки не дырявые
4. Пройти, и охать удивляясь какая большая/уютная/забавная квартира.

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

1. Зайти
2. Поздороваться с хозяевами
3. Не разуваться, потому как грязно!
4. Поохать когда хозяин квартиры поведает нам смелость его архитектурной мысли!

В целом, все происходит практически одинаково, но с изюминкой в каждом различно случае:) Наверное потому то, это и называется шаблоном поведения. Шаблонный метод задает алгоритму пошаговоую инструкцию. Элементы алгоритма же, определяются в наследующих класах.

Сам паттерн ну очень интуитивный, и я уверен что многие давно уже использовали его. Потому давайте попробуем сделать пример.
Вернемся к старой практике, писать примеры по созданию телефонов!

Итак, напишем наш шаблонный класс, с помощью которого будем создавать телефон:

```
class AnyPhoneTemplate {
    func makePhone() {
        takeBox()
        takeCamera()
        takeMicrophone()
        assemble()
    }
    func takeBox() {
        print("Taking a box")
    }
    func takeCamera() {
        print("Taking a camera")
    }
    func takeMicrophone() {
        print("Taking a microphone")
    }
    func assemble() {
        print("Assembling everythig")
    }
}
```

Как вы уже наверное догадались – сам шаблонный метод, это метод makePhone – который задает последовательность вызовов методов необходимых для складывания телефонов. Давайте теперь научим нашу программу создавать айфоны:

```
class iPhoneMaker: AnyPhoneTemplate {

    func design() {
        print("Putting label 'Designed in California'")
    }

    override func takeBox() {
        design()
        super.takeBox()
    }
}
```

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:

```
class AndroidMaker: AnyPhoneTemplate {

    func addRam() {
        print("Installing 2Gigs of RAM")
    }
    func addCPU() {
        print("Installing 4 more CPUs")
    }

    override func assemble() {
        addRam()
        addCPU()
        super.assemble()
    }
}
```

Как видим у сборщика андроида аж целых два дополнительных метода, и перегружен метод assemble.

Тест здесь конечно же – элементарный:

```
let android = AndroidMaker()
let iPhine = iPhoneMaker()

android.makePhone()
iPhine.makePhone()
```

Традиционный log:

2013-03-03 22:56:28.996 TemplateMethod[21040:c07] Taking a box
2013-03-03 22:56:28.998 TemplateMethod[21040:c07] Taking a camera
2013-03-03 22:56:28.999 TemplateMethod[21040:c07] Taking a microphone
2013-03-03 22:56:29.000 TemplateMethod[21040:c07] Installing 4 more CPUs
2013-03-03 22:56:29.000 TemplateMethod[21040:c07] Installing 2Gigs of RAM

2013-03-03 22:56:29.001 TemplateMethod[21040:c07] Assembling everythig
2013-03-03 22:56:29.001 TemplateMethod[21040:c07] Putting label ‘Designed in California’
2013-03-03 22:56:29.002 TemplateMethod[21040:c07] Taking a box
2013-03-03 22:56:29.003 TemplateMethod[21040:c07] Taking a camera
2013-03-03 22:56:29.003 TemplateMethod[21040:c07] Taking a microphone
2013-03-03 22:56:29.003 TemplateMethod[21040:c07] Assembling everything

Strategy

Если Ваша девушка злая, вы скорее всего будете общаться с ней осторожно. Если на вашем проекте завал, то вероятнее всего вы не будете предлагать в команде вечерком дернуть пива 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 куда вы хотите после этого картинку деть. Вот мы и добрались до паттерна Стратегия.

Также отличным примером может быть MVC паттерн – в разных случаях мы можем использовать разные контроллеры для одного и того же View (к примеру авторизованный и не авторизованный пользователь).

Паттерн Стратегия определяет семейство алгоритмов, которые могут взаимозаменяться.

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:

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

Как видно из кода стратегии – у нас есть 3 персонажа, каждый из которых может совершать одно действие! Давайте научим персонажей нападать!

```
class AttackStrategy: BasicStrategy {
    func actionCharacter1() {
        print("Character 1: Attack all enemies!")
    }
    func actionCharacter2() {
        print("Character 2: Attack all enemies!")
    }
    func actionCharacter3() {
        print("Character 3: Attack all enemies!")
    }
}
```

Как видим, при использовании такой стратегии наши персонажи нападают на все что движется! Давайте научим их защищаться:

```
class DefenceStrategy: BasicStrategy {
    func actionCharacter1() {
        print("Character 1: Attack all enemies!")
    }
    func actionCharacter2() {
        print("Character 2: Healing Character 1!")
    }
    func actionCharacter3() {
        print("Character 3: Protecting Character 2!")
    }
}
```

Как видим во время защитной стратегии, наши персонажи действуют по-другому – кто атакует, кто лечит, а некоторый даже защищают:) Ну, теперь как-то надо это все использовать. Давайте создадим нашего игрока:

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

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

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

Как видим наш игрок может только менять стратегию и действовать в зависимости от этой стратегии. Код
для тестирования:

```
let p = Player()
let a = AttackStrategy()
let d = DefenceStrategy()

p.changeStrategy(strategy: a)
p.makeAction()
p.changeStrategy(strategy: d)
p.makeAction()
```

Собственно все предельно ясно:) В первом случае наши персонажи будут активно атаковать, а после смены стратегии уйдут в глухую оборону.
Традиционный лог:

2013-03-04 23:57:44.797 StrategyPatterns[22420:c07] Character 1: Attack all enemies!
2013-03-04 23:57:44.799 StrategyPatterns[22420:c07] Character 2: Attack all enemies!
2013-03-04 23:57:44.800 StrategyPatterns[22420:c07] Character 3: Attack all enemies!
2013-03-04 23:57:44.800 StrategyPatterns[22420:c07] Character 1: Attack all enemies!
2013-03-04 23:57:44.801 StrategyPatterns[22420:c07] Character 2: Healing Character 1!
2013-03-04 23:57:44.801 StrategyPatterns[22420:c07] Character 3: Protecting Character 2!

Command

Стоять, лежать, сидеть – все это команды которые нам очень часто давали учителя 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 выходит он из описания, когда вы хотите инкапсулировать определенную логику в отдельный класс команду. Отличный пример – do/undo операции. У вас, вероятнее всего, будет так называемый CommandManager, которые будет запоминать что делает команда, и при желании отменять предыдущее действие если выполнить команду undo (кстати, это может быть и просто метод).

Собственно, есть два пути реализации этого паттерна:

Для начала создадим базовую команду:

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

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

Теперь реализации наших команд:

```
class FirstCommand: BaseCommand {

    private var originalString: String
    private var currentString: String

    init(argument: String) {
        originalString = argument
        currentString = argument
    }

    func printString() {
        print("Current string is equal to " + currentString)
    }

    func execute() {
        currentString = "This is a new string"
        printString()
        print("Execute command called")
    }
    func undo() {
        currentString = originalString
        printString()
        print("Undo of execute command called")
    }
}
```

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

Вторая наша команда:

```
class SecondCommand: BaseCommand {

    private var originalNumber: Int
    private var currentNumber: Int

    init(number: Int) {
        originalNumber = number
        currentNumber = number
    }

    func execute() {
        currentNumber += 1
        printNumber()
    }
    func undo() {
        if currentNumber > originalNumber {
            currentNumber -= 1
        }
        printNumber()
    }

    func printNumber() {
        print("current number is \(currentNumber)")
    }
}
```

Вторая команда делает тоже самое, но с числом.

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

```
class CommandExecutor {
    private var arrayOfCommands = [BaseCommand]()

    func addCommand(command: BaseCommand) {
        arrayOfCommands.append(command)
    }

    func executeCommands() {
        for command in arrayOfCommands {
            command.execute()
        }
    }

    func undoAll() {
        for command in arrayOfCommands {
            command.undo()
        }
    }
}
```

Как видим, наш менеджер может получать очередь команд, и выполнять их все, или даже отменять все действия (пример простой и с багами:)). Итак, наш тестовый код:

```
let commandE = CommandExecutor()
let cmdF = FirstCommand(argument: "This is a test string")
let cmdS = SecondCommand(number: 3)

commandE.addCommand(command: cmdF)
commandE.addCommand(command: cmdS)

commandE.executeCommands()
commandE.undoAll()
```

И конечно же лог:

2013-03-06 22:40:47.392 CommandPattern[9871:c07] Current string is equal to This is a new string
2013-03-06 22:40:47.393 CommandPattern[9871:c07] Execute command called
2013-03-06 22:40:47.393 CommandPattern[9871:c07] current number is 4 2013-03-06 22:40:47.394 CommandPattern[9871:c07] Current string is equal to This is a test string
2013-03-06 22:40:47.394 CommandPattern[9871:c07] Undo of execute command called
2013-03-06 22:40:47.395 CommandPattern[9871:c07] current number is 3

~~2. Второй метод реализации паттерна – это уже использование внутренностей самой Cocoa – NSInvocation:~~

~~NSInvocation – это объект, который можно использовать чтобы передать возможно вызова метода одного класса – другому, и при это передать в него несколько аргументов, а так же объект который вызывал этот метод.~~

Давайте напишем в наш CommandExecutor добавим два метода и одно приватное поле:

Как видим, теперь наш объект сохраняет объект типа NSInvocation и может его запустить когда требуется. Давайте теперь в нашем основном контроллере напишем функцию которую мы будем вызывать:

А теперь создадим объект типа NSInvocation который и будет в результате нашей командой:

Как видим, в самом начале мы создаем объект который хранит сигнатуры нашего метода, и создаем объект NSInvocation. После этого мы передаем в него
аргументы – не стоит удивляться что индекс начинается с цифры 2 – index 0 и index 1 зарезервированы под target и selector:)

Ну и конечно же лог:

2013-03-06 23:18:26.624 CommandPattern[10479:c07] Method called with first argument = 3 and second argument = This is a string argument

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 с многими объектами.

Как пример возьмем любую стратегическую игру – представьте что у вас 1 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 будет желать лучшего. 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 разшарены – спасут вас от создания тонны других объектов

Итак пример:
Пускай мы пишем игру, где есть два типа персонажей – гоблины и драконы. Для начала создадим базовый класс для всех юнитов:

```
typealias PseudoImage = String

protocol BasicUnit {
    var name: String { get }
    var health: Int { get }
    var image: PseudoImage { 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!

```
class FlyweightImageFactory {

    private static var imageDictionary = [String: PseudoImage]()

    class func getImage(name: String) -> PseudoImage {
        if imageDictionary[name] == nil {
            imageDictionary[name] = "Image" + name
            print("Loading image of the " + name)
        }
        return imageDictionary[name]!
    }
}
```

Как видим, наш flyweight иммет только один класс метод, который картинку по имени то и возвращает. Если картинки под таким именем нету в его словаре – то она грузится из бандла, если же есть – то просто передается ссылка на нее. Каждый раз когда картинка грузится из бандла мы логируем сообщение, это сделано для того чтобы увидеть сколько раз происходит подгрузка изображения из бандла.

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

```
class Dragon: BasicUnit {
    var name = "Dragon"
    var health = 150
    var image = FlyweightImageFactory.getImage(name: "dragon")
}

class Goblin: BasicUnit {
    var name = "Goblin"
    var health = 20
    var image = FlyweightImageFactory.getImage(name: "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 1 тысячу юнитов, лог срабатывает только два раза:

2013-03-09 11:08:45.002 FlyweightPattern[5595:c07] Loading image of the dragon
2013-03-09 11:08:45.006 FlyweightPattern[5595:c07] Loading image of the goblin

Proxy

Ох, все кто работает в большой компании – ненавидит доступ к интернету через прокси:) Что делает прокся? Ну многие из нас уверенны, 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 коннекшену 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 отворачивая стандартные вызовы в проксю и добавляют 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 пользовать тестовый сервер, ну 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. Пускай у нас есть объект который отвечает за скачку файлов:

```
class FileDownloader {
    init() {
        print("Downloader created")
    }
    func slowDownload() {
        print("Slooooooowly downloading...")
    }
    func fastDownload() {
        print("Shuuuuuh, already downloaded...")
    }
}
```

Как видим, наш объект умеет скачивать быстро и медленно. При том, ему все равно какой пользователь и есть ли коннект к интернету.
Давайте создадим нашу прокси:

```
class FileDownloaderProxy {

    private let downloader = FileDownloader()

    var isPremiumUser = false

    func fastDownload() {
        checkNetworkConnectivity()

        if !isPremiumUser {
            slowDownload()
        } else {
            downloader.fastDownload()
        }
    }

    func slowDownload() {
        downloader.slowDownload()
    }

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

Как видим проксятник незначительно умнее:

1. Он знает про тип пользователя, и даже если дернули метод fastDownload но пользователь не премиум – будет вызван метод slowDownload.
2. Он умеет проверять доступ к интернету (пусть это и просто выписка лога).
3. ~~Он проверяет, или тип объекта FileDownloader создан, и если нет – создает его.~~

Ну что, протестируем:

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

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

Традиционный лог:

2013-03-10 13:27:50.312 ProxyPattern[10775:c07] Downloader created 2013-03-10 13:27:50.313 ProxyPattern[10775:c07] Checking network connectivity…
2013-03-10 13:27:50.313 ProxyPattern[10775:c07] Slooooooowly downloading…
2013-03-10 13:27:50.314 ProxyPattern[10775:c07] Checking network connectivity…
2013-03-10 13:27:50.314 ProxyPattern[10775:c07] Shuuuuuh, already downloaded…

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 за паттерн такой? Memento – паттерн который позволяет, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта, чтобы позже восстановить его состояние.

Состояние, как таковое, может сохраняться как в файловую систему, так и в базу даных. Яркий пример использование – 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 даных через CoreData, чтобы восстановить при включении.

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 используется три ключевые объекта: Caretaker (объект который скрывает реализацию сохранения состояния объекта), originator (собственно сам объект) и конечно же сам Memento (объект который сохраняет состояние originator).

Давайте небольшой пример:

```
class OriginatorState {
    var intValue: Int = 0
    var stringValue: String = ""
}
```

Допустим, у нас есть состояние, в котором всего лишь два значение – целочисленное и строчка.

```
class Originator {

    private var localState = OriginatorState()

    init() {
        localState.intValue = 100
        localState.stringValue = "Hello World!"
    }

    func changeValues() {
        localState.intValue += 1
        localState.stringValue += "!"

        print("Current state int = \(localState.intValue), string = \
(localState.stringValue)")
    }

    func getState() -> OriginatorState {
        return localState
    }

    func setState(oldState: OriginatorState) {
        localState = oldState
        print("Load completed. Current state: int = \(localState.intValue),
string = \(localState.stringValue)")
    }
}
```

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

Пускай у нас есть Memento – объект который будет заведовать состояние нашего объекта:

```
class Memento {
    private let localState = OriginatorState()

    init(state: OriginatorState) {
        localState.intValue = state.intValue
        localState.stringValue = state.stringValue
    }

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

То есть наш объект Memento – умеет хранить состояние, и конечно же отдавать состояние:)
Ну и теперь, соединим все это в единый пазл создавая Caretaker:

```
class Caretaker {
    private let originator = Originator()
    private var memento: Memento!

    func changeValue() {
        originator.changeValues()
    }
    func saveState() {
        let state = originator.getState()
        memento = Memento(state: state)
        print("Saved state. State int = \(state.intValue) and string = \
(state.stringValue)")
    }
    func loadState() {
        originator.setState(oldState: 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 так же загружать его:)

Давайте протестим:

```
let crtaker = Caretaker()

crtaker.changeValue()
crtaker.saveState()
crtaker.changeValue()
crtaker.changeValue()
crtaker.changeValue()
crtaker.loadState()
```

Лог как пример работы паттерна:

2013-03-11 23:23:30.711 MementoPattern[14985:c07] Current state int = 101, string = Hello World! !
2013-03-11 23:23:30.712 MementoPattern[14985:c07] Saved state. State int = 101 and string = Hello World! !

2013-03-11 23:23:30.712 MementoPattern[14985:c07] Current state int = 102, string = Hello World! ! !
2013-03-11 23:23:30.713 MementoPattern[14985:c07] Current state int = 103, string = Hello World! ! ! !
2013-03-11 23:23:30.713 MementoPattern[14985:c07] Current state int = 104, string = Hello World! ! ! ! !
2013-03-11 23:23:30.713 MementoPattern[14985:c07] Load completed. Current state: int = 101, string = Hello World! !