GeekBrains

Архитектуры и шаблоны проектирования на Swift

Продвинутые паттерны. Часть 3. Антипаттерны

Паттерны decorator, proxy. Антипаттерны в разработке. Как не стоит писать код.

Оглавление

Введение

Сегодня мы завершаем разбор паттернов проектирования. На следующих уроках нас ждут архитектурные паттерны, а пока рассмотрим еще два важных классических паттерна: **decorator** (декоратор) и **proxy** (прокси). Они описаны в книге «банды четырех» (см. раздел «Используемая литература»). Также на этом уроке мы поговорим об антипаттернах: о том, как не стоит писать код, к чему может привести пренебрежение паттернами или, наоборот, их чрезмерное применение, какие паттерны не очень хорошо работают и почему.

Паттерн Decorator

Decorator (декоратор) — структурный шаблон проектирования, расширяющий функциональность объекта без использования наследования.

Рассмотрим классический пример с композицией объектов. Есть класс **Orc**, который представляет противника в игре. У него может быть много разных свойств, но в нашем пример у этого класса будет одно свойство — **health** (здоровье). Мы хотим, чтобы у нас были и базовые орки (непосредственно класс **Orc**) со слабыми характеристиками, и более сильные — орки-ветераны, орки-боссы и другие. Возможна и композиция вроде орк-ветеран-босс, и любые другие. Очевидно, что используя наследование, мы придем к гигантскому количеству классов — **Orc**, **OrcVeteran**, **OrcBoss**, **OrcVeteranBoss** и т. д. Посмотрим, как такая задача решается с помощью паттерна «декоратор».

Создадим протокол **Character** и класс **Orc**, который реализует протокол **Character**. Зачем протокол? Потому что у нас могут быть и другие юниты, например класс **Elf**, который тоже реализует **Character**.

```
protocol Character {
    var health: Double { get }
}

class Elf: Character {
    var health: Double {
        return 10.0
    }
}

class Orc: Character {
    var health: Double {
        return 20.0
    }
}
```

Далее создадим сам декоратор. Характеристики «босс», «ветеран» и тому подобные — это декораторы для объекта **Character**. Поэтому все декораторы также нужно объединить общим протоколом — **CharacterDecorator**:

```
protocol CharacterDecorator: Character {
    var base: Character { get }
    init(base: Character)
}
```

В протоколе мы потребовали, чтобы декоратор инициализировался объектом **base** типа **Character** — это и будет объект, к которому декоратор будет добавлять новую функциональность. Далее создадим конкретные декораторы:

```
class Veteran: CharacterDecorator {
    let base: Character

    var health: Double {
        return base.health + 50.0
    }

    required init(base: Character) {
        self.base = base
    }
}

class Boss: CharacterDecorator {
    let base: Character

    var health: Double {
        return base.health + 500.0
    }

    required init(base: Character) {
        self.base = base
    }
}
```

Теперь применим это все. Сначала нам понадобится базовый объект — пусть это будет **Elf**. Затем можно сделать ветерана-эльфа, проинициализировав его с базовым. Можно делать сколько угодно таких комбинаций, последовательно инициализируя новый декоратор с предыдущим объектом в качестве базового. Можно, например, создать двойного ветерана, ветерана-босса и т. д. В коде это выглядит так:

```
let elf = Elf()
let veteranElf = Veteran(base: elf)
let bossElf = Boss(base: elf)
let bossVeteranElf = Boss(base: veteranElf)
let doubleVeteran = Veteran(base: veteranElf)
```

Вывод параметра **health** даст ожидаемые результаты:

```
print(elf.health) // 10.0
print(veteranElf.health) // 60.0
print(bossElf.health) // 510.0
print(bossVeteranElf.health) // 560.0
print(doubleVeteran.health) // 110.0
```

Паттерн Proxy

Паттерн Proxy (прокси, «заместитель») — структурный шаблон проектирования. Представляет собой объект, который контролирует доступ к другому объекту, перехватывая все вызовы. При этом объект-прокси имеет тот же интерфейс, что и настоящий объект.

Рассмотрим простейший пример реализации **Proxy**. Есть объект **Calculator**, который умеет выполнять элементарные математические операции:

```
protocol CalculatorInterface {
    func add(x: Double, y: Double) -> Double
    func subtract(x: Double, y: Double) -> Double
    func multiply(x: Double, y: Double) -> Double
    func divide(x: Double, y: Double) -> Double
}

class Calculator: CalculatorInterface {
    func add(x: Double, y: Double) -> Double {
        return x + y
    }
    func subtract(x: Double, y: Double) -> Double {
        return x - y
    }
    func multiply(x: Double, y: Double) -> Double {
        return x * y
    }
    func divide(x: Double, y: Double) -> Double {
        return x / y
    }
}
```

Теперь создадим объект, проксирующий (замещающий) калькулятор. Заместителю обязательно нужно иметь ссылку на настоящий объект, ведь именно последний будет выполнять всю реальную работу.

```
class CalculatorProxy: CalculatorInterface {
    let calculator: CalculatorInterface
    init(calculator: CalculatorInterface) {
        self.calculator = calculator
    }

    func add(x: Double, y: Double) -> Double {
        return calculator.add(x: x, y: y)
    }
    func subtract(x: Double, y: Double) -> Double {
        return calculator.subtract(x: x, y: y)
    }
    func multiply(x: Double, y: Double) -> Double {
        return calculator.multiply(x: x, y: y)
    }
    func divide(x: Double, y: Double) -> Double {
        return calculator.divide(x: x, y: y)
    }
}
```

Теперь можно создать настоящий объект и объект-прокси и обращаться к прокси, чтобы выполнять действия:

```
let calculator = Calculator()
let proxy = CalculatorProxy(calculator: calculator)
proxy.add(x: 2, y: 4)
```

В плейграунде можно проследить, что сначала идет вызов метода **add** у класса **CalculatorProxy**, а затем у класса **Calculator**.

В этом суть паттерна **Proxy**. В принципе, на этом можно было бы и закончить, но у вас наверняка есть вопрос: зачем вообще нужен прокси? Это что, шутка?

На самом деле, мы описали лишь основу паттерна **Proxy** — его структуру. А вот зачем его применять — уже другой вопрос. Вариантов много (например, по ссылке на статью в Википедии в разделе «Используемая литература»), но мы остановимся на реально применимых в iOS-разработке.

Но сначала вместо примера с калькулятором воспользуемся другим. На третьем уроке мы писали функциональность в погодном приложении, и там был класс **WeatherService**, который выполнял запросы в сеть для получения погоды в определенном городе. Мы не будем копировать оттуда код, просто примерно повторим его интерфейс:

```
struct Weather {
    let city: String
    let temperature: Double
    let date: Date
}

protocol WeatherServiceInterface {

    func getWeathers(for city: String, completion: @escaping ([Weather]) ->
Void)
}

class WeatherService: WeatherServiceInterface {

    func getWeathers(for city: String, completion: @escaping ([Weather]) ->
Void) {
        /// имитируем запрос в сеть задержкой в 0.5 сек
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
            /// создадим несколько объектов погоды для имитации ответа сервера
            let weather1 = Weather(city: city, temperature: 0.0, date: Date())
            let weather2 = Weather(city: city, temperature: -1.0, date:
Date(timeIntervalSinceNow: 3600))
            let weather3 = Weather(city: city, temperature: +2.0, date:
Date(timeIntervalSinceNow: 2 * 3600))
            completion([weather1, weather2, weather3])
        }
    }
}

let weatherService = WeatherService()
weatherService.getWeathers(for: "Moscow") { weathers in
    print(weathers)
}
```

Теперь создадим прокси для сервиса получения погоды:

```
class WeatherServiceProxy: WeatherServiceInterface {
    let weatherService: WeatherService
    init(weatherService: WeatherService) {
        self.weatherService = weatherService
    }

    func getWeathers(for city: String, completion: @escaping ([Weather]) ->
Void) {
        self.weatherService.getWeathers(for: city, completion: completion)
    }
}
```

Пока мы просто повторили структуру паттерна, но прокси ничего нового не делает. Перейдем к вариантам, как воспользоваться этим паттерном в данном случае:

1. **Кешировать ответ реального объекта (кеширующий прокси).**

   В **WeatherService** может быть не реализовано кеширование — то есть при повторном вызове метода **getWeathers** с тем же городом этот сервис снова будет запрашивать погоду из сети. Разумно сделать так, чтобы запросы кешировались, например, на 5 минут (потому что потом лучше снова запросить погоду, вдруг прогноз изменился). В этом случае прокси пригодится для реализации этого кеширования.

   В этом случае прокси **WeatherServiceProxy** может представлять собой подобный код:

```
class WeatherServiceProxy: WeatherServiceInterface {
    let weatherService: WeatherService

    var cache: [String: [Weather]] = [:]

    init(weatherService: WeatherService) {
        self.weatherService = weatherService
    }

    func getWeathers(for city: String, completion: @escaping ([Weather]) ->
Void) {
        if let weathersFromCache = self.cache[city] {
            return weathersFromCache
        }
        self.weatherService.getWeathers(for: city) { [weak self] weathers in
            self?.cache[city] = weathers
            completion(weathers)
        }
    }
}
```

2. **Логировать вызовы методов (протоколирующий прокси).**

По условию мы не хотим или не можем менять реализацию реального объекта, но хотим сохранять в лог все обращения к методам. Тогда прокси может выглядеть так:

```
class WeatherServiceProxy: WeatherServiceInterface {
    let weatherService: WeatherService
    init(weatherService: WeatherService) {
        self.weatherService = weatherService
    }

    func getWeathers(for city: String, completion: @escaping ([Weather]) ->
Void) {
        self.weatherService.getWeathers(for: city, completion: completion)
        print("called func getWeathers with city=\(city)")
    }
}
```

Мы дописали в прокси **print**, но в реальной ситуации это может быть сохранение в файл, на диск или отправка нового запроса на логирующий сервер.

3. **Обеспечивать доступ к реальному объекту, только если на это есть права (защищающий прокси).**

   Это еще один пример применения прокси для дополнительного контроля доступа. Прокси может решать, давать доступ к объекту или нет, учитывая при этом заданные условия. Пример:

```
class User {
    static let shared = User()
    private init() { }
    var accessGranted = false
}

class WeatherServiceProxy: WeatherServiceInterface {
    let weatherService: WeatherService
    init(weatherService: WeatherService) {
        self.weatherService = weatherService
    }

    func getWeathers(for city: String, completion: @escaping ([Weather]) ->
Void) {
        guard User.shared.accessGranted else { return }
        self.weatherService.getWeathers(for: city, completion: completion)
    }
}

let weatherService = WeatherService()
let proxy = WeatherServiceProxy(weatherService: weatherService)

User.shared.accessGranted = false
// вызов не произойдет
proxy.getWeathers(for: "Moscow") { weathers in
    print(weathers)
```

```
}

User.shared.accessGranted = true
// теперь вызов произойдет
proxy.getWeathers(for: "Moscow") { weathers in
    print(weathers)
}
```

4. **Синхронизирующий прокси.**

   Бывает, что работа с реальным объектом происходит в асинхронной многопоточной среде, а вызовы функций и обращение к свойствам объекта не потокобезопасно. В таком случае может понадобиться создать прокси, который бы реализовал потокобезопасное обращение к объекту. Пожалуй, в iOS-разработке это наиболее важное применение паттерна **Proxy**, но такой пример заведет нас в глубины многопоточности, а это уже совсем другая история (выходит за рамки нашего курса).

   *Важное замечание! Мы используем прокси тогда, когда не хотим или не можем изменять реализацию того объекта, вызов к которому мы проксируем. В этом смысле **Proxy** похож на **Adapter** — наиболее родственный ему паттерн. Отличие в том, что адаптер пишется с новым интерфейсом, а прокси с тем же, что и у базового объекта.*

Антипаттерны

Обсудим интересную и важную с точки зрения качества кода тему — антипаттерны. Очевидно, что это противоположность паттернам, то есть то, как не надо писать код. И не просто рекомендации типа «создавайте классы, называйте переменные понятно», а ситуации, в которые попадают почти все программисты. Знание антипаттернов не защищает полностью от таких ситуаций, еще требуется много опыта, но знать антипаттерны уж точно лучше, чем не знать их.

Антипаттерны — это шаблоны ошибок, которые возникают при решении задач. Из-за них код хуже поддерживается, и впоследствии его будет сложно или невозможно менять без глубокого рефакторинга. Часть практики хорошего программирования — как раз избегать антипаттернов, а не просто применять паттерны. Антипаттернов описано множество, мы рассмотрим наиболее важные из них.

God Object (божественный объект)

Божественный объект — тот, который делает «слишком много». Если у класса уйма обязанностей, то его экземпляры становятся **god object**'ами.

В iOS-разработке класс делегата приложения **AppDelegate** легко вырастает в божественный объект, если обязанности недостаточно разнесены между классами. В **AppDelegate** происходит первичная настройка приложения после запуска, он обрабатывает deep link'и, ловит переходы между состояниями приложения, регистрирует токен для пушей. А еще **AppDelegate** — это синглтон, к которому можно обратиться из любого места через **UIApplication.shared.delegate**. В **AppDelegate** можно хранить свойством объекты, а затем через синглтон из любой точки приложения иметь к ним доступ. Есть все предпосылки для превращения **AppDelegate** в божественный объект, дальше дело за разработчиком.

Другая проблема god object'ов в iOS-разработке — это массивные вью-контроллеры (**massive view controller** — **MVC**). В архитектуре **MVC** (**model — view — controller**) слои view и слой модели, как

правило, тонкие, а вся основная логика помещается во вью-контроллеры, из-за чего они могут разрастаться на тысячи строк кода. И здесь опять присутствуют черты божественного объекта. Перечислим, что делает вью-контроллер: занимается настройкой view, знает о жизненном цикле view, выполняет переходы между экранами (навигацию), хранит бизнес-логику (как должно работать приложение при действиях юзера), запрашивает данные из сети и из БД и т. д. Слишком много для одного объекта. Такая проблема возникает, конечно, не всегда, но она есть. Как ее решить с помощью других архитектурных паттернов, обсудим на следующих уроках.

Чем плох божественный объект? Из-за слишком большого количества обязанностей разобраться в том, что он делает, бывает крайне тяжело. К тому же, такой объект сопровождается многострочным кодом, поэтому поддерживать и изменять этот класс сложно. И не только его, ведь божественный объект, беря на себя множество функций, способствует тому, что эти функции используются из других классов. Получается большая связность, и распутать этот клубок бывает очень тяжело. Это и есть главная проблема божественных объектов.

Какие есть решения? Нужно изначально проектировать систему так, чтобы в ней соблюдался принцип единой ответственности (одна ответственность у одного класса). В архитектуре MVC можно частично избавиться от проблемы **massive view controller**, разделяя сложные экраны на сабмодули, каждый из которых представляет собой **child view controller** со своей view.

Hard code и волшебные числа

Волшебные числа и строки появляются, когда в код вносят числовые и строковые константы, не объясняя их смысл. Рассмотрим пример, где в части кода нужно пройтись в цикле по всем объектам — их 42:

```
let max = 42
for i in 0...max {
    ...
}
```

Такой код выглядит нормально только на первый взгляд, ведь через пару недель никто и не вспомнит, откуда взялось число 42. Придется потратить время на выяснение этого, и если ничего не поменять в коде, то каждому разработчику придется разбираться заново.

Поговорим о решении проблемы. Если 42 — это действительно константа, которая не будет меняться в ближайшем будущем, значит, она просто нуждается в документации. Необязательно в код вставлять комментарии, лучше давать такие названия переменным, которые будут документироваться сами, например:

```
struct Constants {
    static let cellsCount = 42
}

for i in 0...Constants.cellsCount {
    ...
}
```

Другой вариант — константу 42 внесли для проверки какого-то поведения или просто по незнанию, но на самом деле это число должно варьироваться в зависимости от условий. Например, вы отображаете блоки с ячейками в приложении, и в большинстве случаев в них действительно 42 ячейки. Но есть несколько блоков с другим количеством ячеек, да и вообще это число в любой

момент может измениться. Здесь мы сталкиваемся с антипаттерном **hard code** (жесткое кодирование) — внесение жестко заданных данных об окружении системы в ее реализацию, в то время как эти данные могут меняться. С этим антипаттерном надо бороться сразу при первичной реализации, хорошо продумывая архитектуру решения. В приведенном примере количество ячеек лучше рассчитывать, ориентируясь на данные, пришедшие с сервера.

Soft code

Soft code (мягкое кодирование) — противоположность **hard code**. Это стремление вынести все данные об окружении в файлы конфигурации или на сервер, чтобы система была максимально гибкой. Например, не хардкодить **url** до документа, а получать его с сервера в отдельном запросе. Излишнее следование принципу **soft code** чрезвычайно усложняет программу. Многие вещи можно и захардкодить — главное понимать, где и когда это может выстрелить.

Теперь рассмотрим методологические антипаттерны.

Золотой молоток

Так называют уверенность разработчика в том, что однажды успешно примененное решение, технология или паттерн будут работать всегда и везде. Например, вы изучили паттерн **State** на четвертом уроке, и он отлично подошел в приложении, где в каждый момент игра имеет определенное состояние. Но бездумное применение этого паттерна в других проектах может больше навредить, чем помочь.

Важно всегда до конца исследовать проблему, не поддаваясь соблазну применить ранее сработавшее решение. Не исключено, что оно действительно будет лучшим в данной ситуации, но, возможно, и нет. Для каждой задачи есть несколько решений, и важно найти наиболее подходящее.

Практическое задание

1. Задание на паттерн **Decorator**.

   Еще один классический пример: создание различных вариаций кофе (простой, с молоком, с сахаром, с молоком и сахаром, со сливками, с молоком, сахаром и сливками и т. д.) с помощью декорирования — вместо создания большого количества сабклассов кофе.

   Создайте протокол **Coffee** и класс **SimpleCoffee**, реализующий протокол **Coffee**. Создайте протокол **CoffeeDecorator**, который «наследуется» от протокола **Coffee**. Далее создайте реализации декоратора — **Milk**, **Whip**, **Sugar** и другие, которые будут ингредиентами, которые добавляют в кофе. В протоколе **Coffee** объявите переменную **cost: Int** — это цена кофе. Каждый ингредиент должен увеличивать цену кофе на свою стоимость. Реализуйте описанную систему объектов с помощью паттерна **Decorator** аналогично тому, как это было сделано на уроке.

2. Задание на паттерн **Proxy**.

   Примените «протоколирующий прокси» для логирования сетевых запросов в приложении **VK Client** (которое использовалось в практическом задании урока 3).

3. Прочитайте статьи об антипаттернах из раздела «Дополнительные материалы»:

   - Что такое анти-паттерны?

- 9 анти-паттернов, о которых должен знать каждый программист.
- Singleton is not an anti-pattern.

Дополнительные материалы

1. Design Patterns on iOS using Swift – Part 1/2.
2. Design Patterns on iOS using Swift – Part 2/2.
1. Real World: iOS Design Patterns.
3. App Architecture and Object Composition in Swift
4. Что такое анти-паттерны?
5. 9 анти-паттернов, о которых должен знать каждый программист.
6. Singleton is not an anti-pattern

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Шаблон проектирования (Википедия).
2. Антипаттерн (Википедия).
3. Паттерны ООП в метафорах.
4. Что такое анти-паттерны?
5. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования».