GeekBrains

Архитектуры и шаблоны проектирования на Swift

Продвинутые паттерны. Часть 2

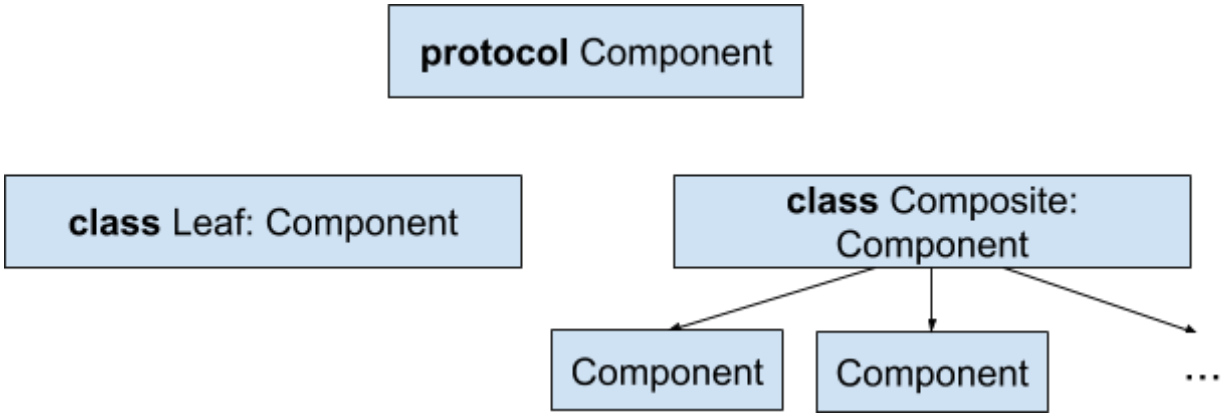
Паттерны composite, mediator, chain of responsibility.

Оглавление

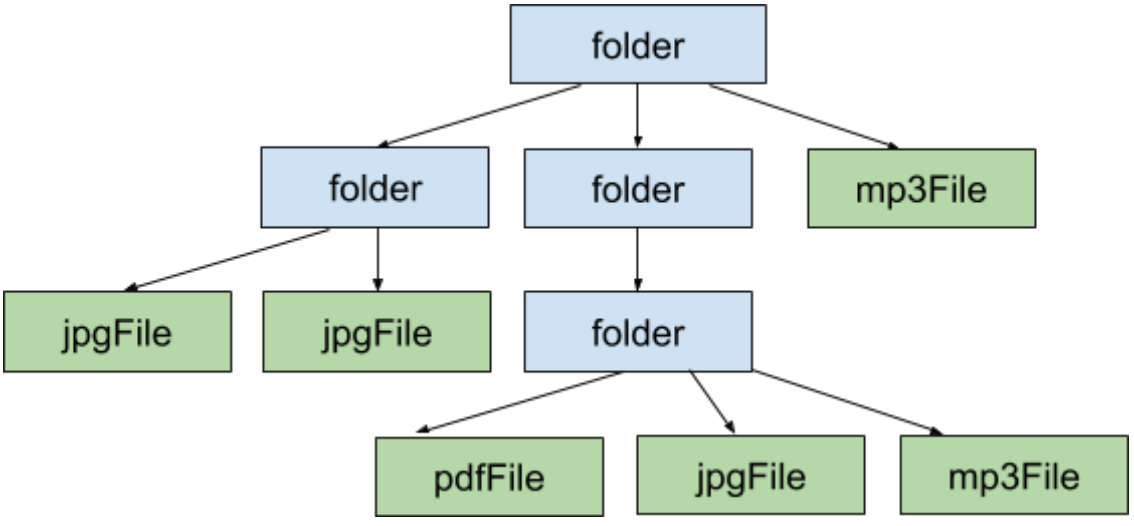
Паттерн Composite

Паттерн **Composite** (компоновщик) — структурный шаблон проектирования. Он представляет объекты в виде древовидной структуры и обеспечивает к ним доступ через единый интерфейс.

Рассмотрим схему этого паттерна. Есть протокол **Component** (название условное) — в нем описан общий интерфейс для каждого из объектов, участвующих в паттерне. Может быть сколько угодно классов, которые реализуют протокол, но их можно разделить на две части: композитные объекты и листья. Композитные объекты (**class Composite** на схеме) содержат другие объекты типа **Component**, обычно в виде массива. Листья (**class Leaf** на схеме) не содержат других компонентов — так что являются конечными объектами в структуре.

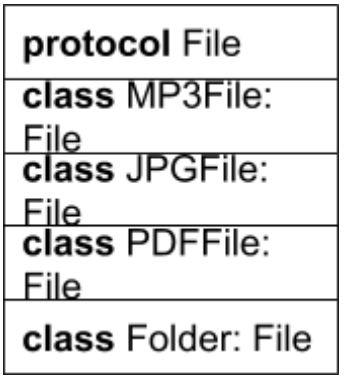


Рассмотрим пример подобной древовидной структуры. Это файловая система. Есть отдельные файлы — картинки, видео, звуковые и текстовые файлы и т. д. Все они являются листьями, так как один файл не содержит других. А есть папки — это композитные объекты, которые содержат как файлы, так и другие папки. Получается, например, вот такая структура:



Здесь листья выделены зеленым цветом — это файлы, уже не содержащие других компонентов.

Важная черта паттерна **Composite** — все объекты древовидной структуры объединены одним протоколом. То есть, например, файлами будут и **jpgFile**, и папка. Обращение к ним происходит через один протокол, назовем его **File**. Структура классов будет такой:



Пример 1

Реализуем пример с файлами и папками в плейграунде по паттерну **Composite**.

Сначала объявим протокол **File**: он требует, чтобы у файла были имя и описание, а также реализацию функции **open()**, которая должна открывать этот файл.

```
protocol File {
    var name: String { get }
    var description: String { get }
    func open()
}
```

Теперь добавим несколько конкретных файлов. Пока займемся «листьями», то есть обычными файлами, которые не содержат других файлов. Пусть, например, у нас будут **MP3File** и **PDFFile**:

```
class PDFFile: File {
    let name: String
    let author: String

    var description: String {
        return "ebook \(name) by \(author)"
    }

    init(name: String, author: String) {
        self.name = name
        self.author = author
    }

    func open() {
        print("opening \(description)")
    }
}

class MP3File: File {
    let name: String
    let artist: String

    var description: String {
        return "music \(name) by \(artist)"
    }

    init(name: String, artist: String) {
        self.name = name
        self.artist = artist
    }

    func open() {
        print("opening \(description)")
    }
}
```

Далее создадим класс папки. Папка поддерживает все тот же протокол **File**. Но у этого конкретного класса, помимо свойств из **File**, будет также свойство, хранящее массив других файлов внутри этой папки. Причем в ней могут находиться другие папки — паттерн **Composite** позволяет это сделать, так как все объекты объединены общим протоколом.

Итак, добавим класс папки:

```
class Folder: File {
    var name: String
    var files: [File] = []

    var description: String {
        let fileNames = files.compactMap { $0.description }.joined(separator: ",
")
        return "folder \(name) with files: \(fileNames)"
    }

    init(name: String) {
        self.name = name
    }

    func open() {
        print("opening \(description)")
        print("then opening files: ")
        self.files.forEach { $0.open() }
    }
}
```

В свойстве **description** мы возвращаем имя папки и имена всех файлов, содержащихся в ней. Функцию **open()** у папки мы реализовали так, что она открывает все находящиеся в ней файлы.

Создадим несколько файлов и папок, объединим их в структуру:

```
let book = PDFFile(name: "War and Peace", author: "Tolstoy")
let book2 = PDFFile(name: "Eugene Onegin", author: "Pushkin")
let music = MP3File(name: "Yesterday", artist: "The Beatles")
let music2 = MP3File(name: "Bohemian Rhapsody", artist: "Queen")
let folder1 = Folder(name: "folder1")
let folder2 = Folder(name: "folder2")
folder1.files = [book, folder2]
folder2.files = [book2, music, music2]

folder1.open()
```

Мы создали два pdf-файла, два mp3-файла и две папки. Первая папка содержит один pdf-файл и вторую папку. Вторая — другие три файла. Затем открываем папку 1. В консоли мы увидим последовательное открытие сначала pdf-файла, потом папки 2, а затем у папки 2 последовательно открываются все находящиеся в ней файлы.

Еще раз обращаем внимание, что фишка паттерна «компоновщик» — это единый интерфейс как для композитных объектов, так и для листьев.

Пример 2

На практике паттерн «компоновщик» прекрасно работает для представления иерархии view на экране. В **UIKit** он уже реализован, и мы им все это время пользовались. У объекта **UIView** есть массив **subviews: [UIView]** и метод для добавления сабвью **addSubview(_ subview: UIView)**.

Складывается иерархия построения — у одной view может быть несколько subview, у каждой из которых может быть еще subview и так далее. Вместо класса **UIView** здесь может выступать любой его сабкласс — **UIControl**, **UILabel**, **UIButton** и т. д.

Аналогичным образом можно выстроить иерархию вью-контроллеров. У класса **UIViewController** есть свойство **childViewControllers: [UIViewController]** и метод для добавления дочернего вью-контроллера: **addChild(_ childController: UIViewController)**. Точно так же вью-контроллер может содержать несколько дочерних вью-контроллеров, каждый из которых содержит еще дочерние, и так далее.

Пример 3

Паттерн **Composite** отлично подходит для асинхронных задач, которые можно сгруппировать в одну или несколько композитных. Например, чтобы получить полную информацию о товаре, необходимо сделать несколько запросов в Сеть, и еще несколько — чтобы уточнить цену (не идеальное API, но такое бывает в жизни). Тогда первая пачка запросов представляется одной группой (композитной задачей), вторая пачка запросов — второй, и вместе две эти композитные задачи образуют еще одну композитную, назовем ее задачей получения полной информации. Получается та самая древовидная структура паттерна **Composite**.

Реализуем решение такой задачи в коде. Начнем с протокола **Task** — это общий протокол как для обычной задачи, так и для композитной:

```
protocol Task {
    func run(completion: @escaping () -> Void)
}
```

Создадим конкретную задачу — реализацию протокола **Task**. Это может быть задача загрузки данных из сети. Таких задач может быть много, на каждый отдельный запрос. Мы рассмотрим их в общем. Поэтому реализация функции **run** будет состоять из фейковой задержки в 0,5 сек, после этого просто вызываем **completion**, как если бы задача была выполнена. Дадим задаче имя и обеспечим вывод в консоль для дебага.

```
class ConcreteTask: Task {
    let name: String

    init(name: String) {
        self.name = name
    }

    func run(completion: @escaping () -> Void) {
        print("start \(self.name)")
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) { [weak self] in
            print("completed \(self?.name ?? "")")
            completion()
        }
    }
}
```

Теперь добавим составную задачу. Реализация параллельного выполнения асинхронных задач обычно делается с помощью **DispatchGroup** — этим же воспользуемся и мы:

```
class CompositeTask: Task {
    private var tasks: [Task]

    init(tasks: [Task]) {
        self.tasks = tasks
    }

    func run(then completion: @escaping () -> Void) {
        let dispatchGroup = DispatchGroup()

        for task in tasks {
            dispatchGroup.enter()
            task.run { dispatchGroup.leave() }
        }

        dispatchGroup.notify(queue: .main) {
            completion()
        }
    }
}
```

Создадим несколько задач и композиций и посмотрим, как это будет работать:

```
let task1 = ConcreteTask(name: "task1")
let task2 = ConcreteTask(name: "task2")
let task3 = ConcreteTask(name: "task3")
let task4 = ConcreteTask(name: "task4")
let task5 = ConcreteTask(name: "task5")
let compositeTask1 = CompositeTask(tasks: [task1, task2, task3])
let compositeTask2 = CompositeTask(tasks: [task4, task5])
let compositeTask3 = CompositeTask(tasks: [compositeTask1, compositeTask2])

compositeTask3.run {
    print("completed")
}
```

Мы создали 5 задач — 3 в одной группе и 2 в другой. Затем обе группы объединили в третью. Таким образом, **compositeTask3** запускает параллельное выполнение двух групп, которые запускают параллельное выполнение каждой отдельной задачи. В консоли мы увидим сначала такой вывод:

```
start task1
start task2
start task3
start task4
start task5
```

А потом, через 0,5 секунд, такой:

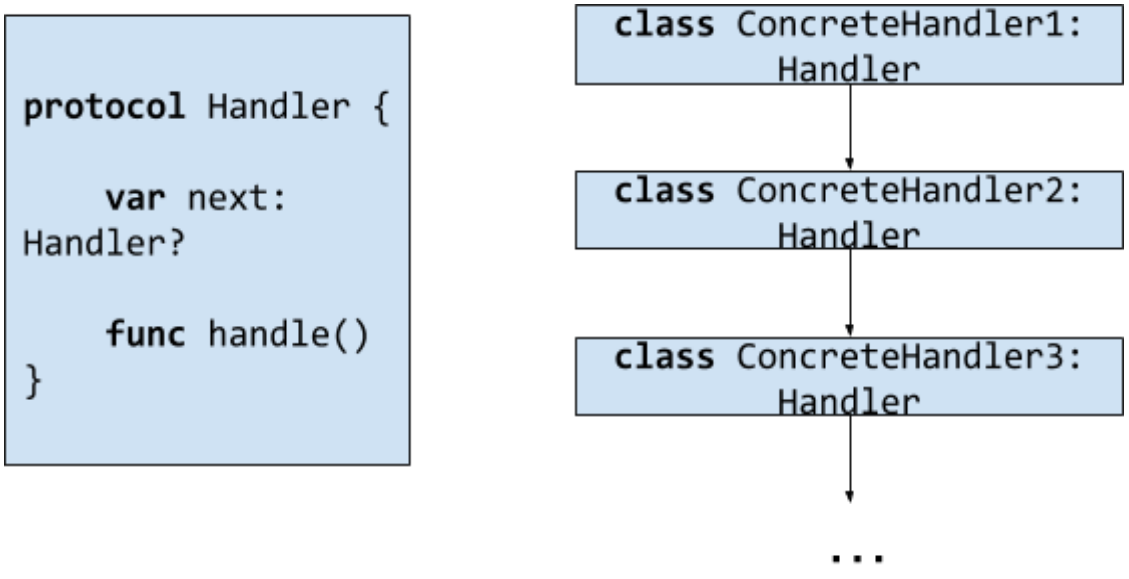
```
completed task1
completed task2
completed task3
completed task4
completed task5
completed
```

Паттерн Chain of responsibility

Паттерн **Chain of responsibility** (цепочка обязанностей) — поведенческий шаблон проектирования. Он предназначен для распределения обработки сообщения / события между разными объектами. Структура паттерна следующая:

- сначала один объект пытается обработать событие;
- если он не может этого сделать, то посылает его следующему обработчику. Все обработчики выстроены в последовательную цепочку и реализуют один протокол.

На рисунке представлена схема паттерна. Определяется протокол **Handler**, которому должны соответствовать все обработчики события. Каждый из них имеет свойство **next: Handler?**, ссылающееся на следующий обработчик по цепочке (у последнего обработчика **next** будет **nil**). Каждый обработчик должен реализовать функцию **handle()**, которая и обрабатывает событие.



В книге «банды четырех» описан классический пример применения этого паттерна: обработка нажатия пользователя на область экрана для вызова контекстной помощи. В зависимости от области нажатия нужно обработать его по-разному, и для этого создается цепочка обработчиков, каждый из которых может обработать нажатие в зависимости от входных условий.

У нас, как правило, нет необходимости в создании такой цепочки, так как уже есть механизмы **UIGestureRecognizer**, **addTarget** и т. д. Мы рассмотрим пример применения **Chain of responsibility** для обработки сетевых событий — в частности, для обработки ошибок, приходящих в ответе от сервера.

Пример

Допустим, в нашем приложении мы ожидаем, что на запрос данных в сеть можем получить одну из ошибок:

- связанную с сетью,
- связанную с авторизацией,
- общую ошибку,
- другую (их может быть намного больше).

Распишем эти варианты ошибок в перечислениях, которые поддерживают протокол **Error** (и таким образом могут быть выброшены в качестве ошибки):

```
enum LoginError: Error {
    case loginDoesNotExist
    case wrongPassword
    case smsCodeInvalid
}

enum NetworkError: Error {
    case noConnection
    case serverNotResponding
}

enum GeneralError: Error {
    case sessionInvalid
    case versionIsNotSupported
    case general
}
```

Напишем функцию, которая имитирует запрос данных из сети. В комплишне этой функции может прийти ошибка:

```
func requestData(completion: @escaping (Error?) -> Void) {
    DispatchQueue.main.asyncAfter(deadline: .now() + 0.5, execute: {
        completion(GeneralError.sessionInvalid)
    })
}
```

Теперь займемся написанием классов, которые могут обрабатывать эти ошибки. Здесь нам и пригодится паттерн «цепочка обязанностей». Каждый отдельный класс будет уметь обрабатывать только свой тип ошибки. Если обработчик не смог обработать ошибку, то он передает ее следующему обработчику по цепочке.

```swift
protocol ErrorHandler {

    var next: ErrorHandler? { get set }

    func handleError(_ error: Error)
}

class LoginErrorHandler: ErrorHandler {

    var next: ErrorHandler?

    func handleError(_ error: Error) {
        guard let loginError = error as? LoginError else {
            self.next?.handleError(error)
            return
        }
        print(loginError)
        // show tooltip
    }
}

class NetworkErrorHandler: ErrorHandler {

    var next: ErrorHandler?

    func handleError(_ error: Error) {
        guard let networkError = error as? NetworkError else {
            self.next?.handleError(error)
            return
        }
        print(networkError)
        // show alert
        // try repeat network request
    }
}

class GeneralErrorHandler: ErrorHandler {

    var next: ErrorHandler?

    func handleError(_ error: Error) {
        guard let generalError = error as? GeneralError else {
            self.next?.handleError(error)
            return
        }
        print(generalError)
        // show error view controller
        // try repeat request
        // log error
    }
}
```

Мы не стали реализовывать конкретное поведение, а написали в комментариях, как именно обработчик может обработать ошибку — записать ее в лог, показать экран ошибки, попробовать повторить запрос. Важно, все обработчики объединены общим протоколом и благодаря свойству **next** их можно расположить в цепочку.

Добавим эти обработчики:

```
let loginErrorHandler = LoginErrorHandler()
let networkErrorHandler = NetworkErrorHandler()
let generalErrorHandler = GeneralErrorHandler()
let errorHandler: ErrorHandler = loginErrorHandler

loginErrorHandler.next = networkErrorHandler
networkErrorHandler.next = generalErrorHandler
generalErrorHandler.next = nil
```

Сначала **errorHandler** (который на самом деле является **LoginErrorHandler** — первым обработчиком в цепочке) будет пытаться обработать ошибку. А если не получится, он передаст обработку другому — **NetworkErrorHandler**, а он в свою очередь передаст ошибку **GeneralErrorHandler**.

Итак, попробуем вызвать ранее написанный метод **requestData**, который в нашей реализации всегда будет возвращать ошибку, и обработать ошибку нашей цепочкой обработчиков:

```
requestData { error in
    if let error = error {
        errorHandler.handleError(error)
    }
}
```

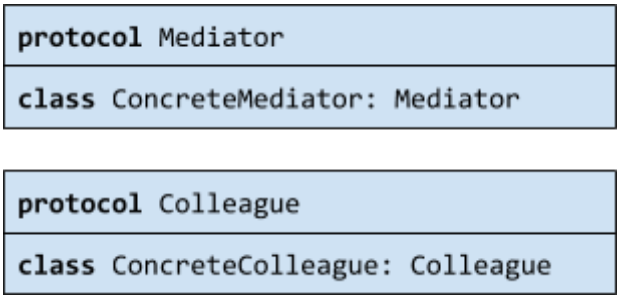
В плейграунде можно увидеть, как у обработчиков последовательно были вызваны методы, которые пробрасывали ошибку следующему обработчику, и в конце концов именно **GeneralErrorHandler** смог обработать ошибку.

Паттерн Mediator

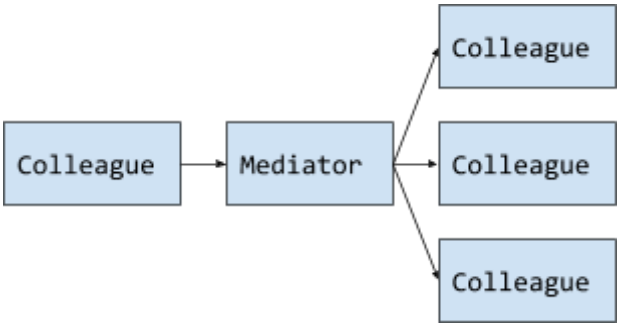
Паттерн **Mediator** (посредник) — поведенческий шаблон проектирования. Он применяется в системе, где есть много взаимодействующих друг с другом объектов, чтобы уменьшить связность между ними. Объект взаимодействует не с другими напрямую, а с посредником, который уже передает это взаимодействие всем другим объектам.

Сама эта концепция активно используется в ООП и в iOS-разработке. Вью-контроллер — посредник между вью (отображением) и моделью, в архитектуре **VIPER** это слой **presenter**, **NotificationCenter** — посредник между передатчиком данных (тем, кто постит нотификацию) и получателем (обсервером).

Мы рассмотрим классический вариант использования паттерна **Mediator**. По структуре: в паттерне два действующих лица — **mediator** и **colleague**. Оба представлены своим протоколом и соответствующими конкретными реализациями:



Colleague — объект, который хочет взаимодействовать с другими такими же объектами. Он должен делать это через посредника **mediator**. Прямые обращения одного **colleague** к другому запрещены.



Пример

Рассмотрим абстрактный пример передачи строковых сообщений между объектами одного класса. Реализуем такую передачу с помощью паттерна **Mediator**. Напишем протоколы **Mediator** и **Colleague**:

```
protocol Colleague: class {
    func colleague(_ collegue: Colleague, didSendMessage message: String)
}

protocol Mediator: class {
    func addColleague(_ colleague: Colleague)
    func sendMessage(_ message: String, by colleague: Colleague)
}
```

Colleague умеет получать сообщения от других объектов такого же типа (поддерживающих протокол **Colleague**, при этом конкретный объект может быть любым). Далее напишем реализацию медиатора:

```
class ConcreteMediator: Mediator {

    var colleagues: [Colleague] = []

    func addColleague(_ colleague: Colleague) {
        self.colleagues.append(colleague)
    }

    func sendMessage(_ message: String, by colleague: Colleague) {
        for colleagueToSend in self.colleagues where colleagueToSend !==
colleague {
            colleagueToSend.colleague(colleague, didSendMessage: message)
        }
    }
}
```

Медиатор содержит ссылки на **colleagues** в массиве. С помощью функции **sendMessage** медиатор умеет отправить сообщение от одного объекта **colleague** всем другим. Рассмотрим в качестве **colleague** объект персоны, который хочет общаться с другими такими же объектами, и сделаем так, чтобы обмен сообщениями происходил через посредника. Напишем класс **Person**, который при инициализации должен принимать объект-медиатор:

```
class Person {
    let name: String
    private weak var mediator: Mediator?

    init(name: String, mediator: Mediator) {
        self.name = name
        self.mediator = mediator
        mediator.addColleague(self)
    }

    func sendMessage(_ message: String) {
        print("\(name) send message \(message)")
        self.mediator?.sendMessage(message, by: self)
    }
}
```

При инициализации **Person** мы тут же добавляем его медиатору в массив **colleagues**. Вызов функции **sendMessage** у **Person** вызовет обращение к медиатору, который уже передаст сообщение всем остальным коллегам.

Но сначала сделаем сам класс **Person** поддерживающим протокол **Colleague**:

```
extension Person: Colleague {
    func colleague(_ collegue: Colleague, didSendMessage message: String) {
        print("\(name) did receive message \(message)")
    }
}
```

Ожидаем, что отправка сообщения одним объектом **Person** вызовет вывод в консоль того, что этот объект послал сообщения, а все остальные объекты приняли его. Проверим:

```
let mediator = ConcreteMediator()
let person1 = Person(name: "John", mediator: mediator)
let person2 = Person(name: "Steve", mediator: mediator)
let person3 = Person(name: "Joe", mediator: mediator)

person1.sendMessage("Hi!")
print("")
person2.sendMessage("Hi to you")
print("")
person3.sendMessage("Hello")
```

В консоли увидим:

```
John send message Hi!
Steve did receive message Hi!
Joe did receive message Hi!

Steve send message Hi to you
John did receive message Hi to you
Joe did receive message Hi to you

Joe send message Hellow
John did receive message Hellow
Steve did receive message Hellow
```

Практическое задание

1. Задание на паттерн **Composite**.

   Создать приложение со списком задач, где для каждой задачи можно создать подзадачи, у них — еще подзадачи, и так до бесконечности.

   UI приложения: первый экран состоит из списка корневых задач. В **navigation bar** справа должна быть кнопка + (добавить задачу). Все задачи отображаются в обычной **table view**. В ячейке укажите название задачи и количество подзадач в ней. По нажатию на ячейку переходим на новый экран, представляющий собой экран этой задачи. На нем отображается список подзадач уже этой задачи. Здесь также есть плюсик в **navigation bar** для добавления

подзадач и все они отображаются в **table view**. Таким образом, все экраны задач абсолютно одинаковы.

Реализуйте древовидную иерархию задач, применив паттерн **Composite**.

2. * Задание на паттерн **Chain of responsibility**.

   Откройте плейграунд https://drive.google.com/open?id=1TuaiBDD1oJGlAL4UxFGHy_OjT1by3zn.

   В нем находится три файла в папке **Resources**: **1.json**, **2.json** и **3.json**, а в основном файле плейграунда написана функция, которая по имени файла возвращает его данные в виде **Data**. Представьте, что ответ от сервера содержит массив персон, но формат этих данных может отличаться. В ответе **1.json** этот массив находится в корневом объекте по ключу **data**, во втором ответе по ключу **result**, а в третьем **json** сам представляет собой этот массив, без обертки в корневой объект.

   Необходимо написать цепочку парсеров, которые будут пытаться распарсить ответ от сервера в массив объектов (заранее напишите структуру **Person**). Каждый парсер умеет работать со своей структурой данных. Нужно, чтобы была написана функция, которая на вход принимает имя файла, а возвращает массив **[Person]**. И эта функция должна работать для любого из трех файлов.

Дополнительные материалы

1. Design Patterns on iOS using Swift – Part 1/2.
2. Design Patterns on iOS using Swift – Part 2/2.
3. Real World: iOS Design Patterns.
4. App Architecture and Object Composition in Swift

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Шаблон проектирования (Википедия)..
2. Паттерны ООП в метафорах.
3. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. «Приемы объектно-ориентированного проектирования. Паттерны проектирования».