Профессиональное программирование на Swift

Урок 2

Архитектура кода

Проектирование кода класса. Зависимости. Swinject.

Архитектура проекта

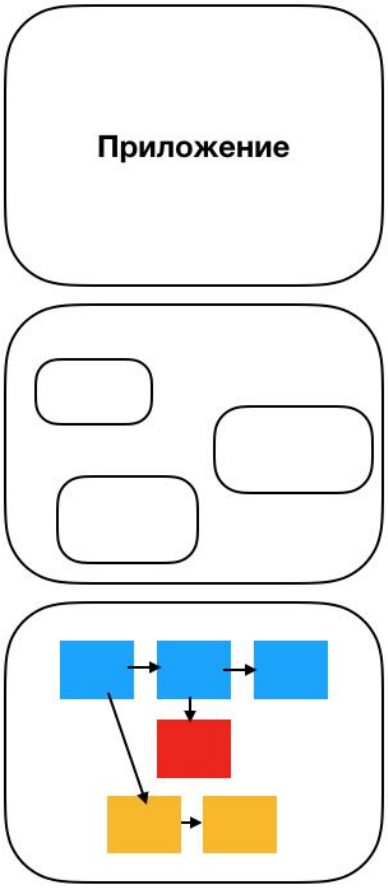
Когда ТЗ на разработку мобильного приложения уже проверено вами и все в нем понятно, спешить не стоит. Хоть заказчик и видит проект как приложение с кнопочками и экранами, нужно понимать, что крайне важна хорошо продуманная архитектура. Особенно — при работе с большими проектами. Иначе в определенный момент можно осознать, что ты уже не контролируешь проект: он превращается в тяжеловесного монстра, в него сложно внести новый функционал или доработать существующий.

Разберемся, что такое архитектура проекта, какими критериями нужно руководствоваться при ее создании?

Архитектура — это организация набора модулей, объединенных для определенной задачи, между которыми налажены связи.

Выделение подсистем/модулей проекта

Модули проекта выстраивают из компонентов логики приложения. В основном модули — это классы проекта. Но не стоит сразу «рубить» приложения на множество классов. Сначала достаточно определить подсистемы, а затем разделить их на классы.



Задача кажется простой, но, как показывает практика, это не так. Рассмотрим пример с контроллером, в котором присутствуют бизнес-логика и представление.

Бизнес-логика — это набор правил, взаимосвязи объектов в той или иной области приложения. Возвращаясь к примеру с контроллером, видим, что надо выделить 2 модуля: представление и

бизнес-логику. На таком разбиении и построен знаменитый архитектурный паттерн **MVC** (Модель-Вид-Контроллер). Он размещает реализацию бизнес-логики в модуль «Модель», а представление или взаимодействие с пользователем — в «Вид». Модуль «Контроллер» определяет, какое представление должно быть отображено в данный момент. Он может повлиять на модель и определить другое представление. Существует еще множество архитектурных паттернов: **MVP**, **MVVM**, **VIPER**... Выбор каждого из них определяется потребностями приложения.

Разбиение на модули дает много преимуществ

- **Масштабируемость** — можем расширять возможности приложения, добавляя новые модули;
- **Легкое исправление и доработка** — модуль легко тестируется/заменяется/модифицируется/отсоединяется;
- **Многократное использование**;
- **Тестируемость.**

SOLID

SOLID-принципы — пример правил «хорошей архитектуры». Их всего пять:

```
1. S - Принцип единственной ответственности (Single responsibility)
Каждый класс должен реализовать одну функциональность. Если у класса таких
функциональностей больше, его нужно разбить.

2. O - Принцип открытости/закрытости (Open-closed)
Классы должны быть расширяемыми, но закрытыми для изменения.

3. L - Принцип подстановки Барбары Лисков (Liskov substitution)
Классы могут быть заменены их наследниками без изменения логики приложения.

4. I - Принцип разделения интерфейса (Interface segregation)
Много специализированных интерфейсов лучше, чем один универсальный.

5. D - Принцип инверсии зависимостей (Dependency Inversion)
Зависимости должны строиться относительно абстракций, а не деталей.
Если классы зависят от других классов(сохраняют внутри себя объекты других
классов и т.д), заменяем эту зависимость на абстракцию.
```

Хороший класс

Но все же хорошая архитектура — это, прежде всего, выгодная архитектура. И в ее основе должно лежать понимание — подсистема, модуль, класс должны реализовать каждый только свою логику. Например, у **ViewController** простая цель — он принимает жизненный цикл и является связующим звеном для остальных компонентов. Загрузка данных от сервиса — это отдельный класс. Хранение данных — тоже.

По одной из метрик, «хороший класс» — это тот, в котором около 200 строк. Это только рекомендация, и ваш класс при необходимости может иметь в реализации и больше. Но все же, если в проекте несколько классов с большим количеством строк — это повод обратить на код пристальное внимание.

Зависимости

Каждый класс должен быть закрыт протоколом. Вспоминаем пятый принцип SOLID — принцип инверсии зависимостей (Dependency Inversion). Если один класс хранит в себе объекты других и внутри своих методов вызывает «чужие», у него присутствуют зависимости. От них нужно избавляться — заменять на протоколы. Рассмотрим простой пример:

```
class Cat {
    let name: String

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

    func info() -> String {
        return "Кот по имени - \(name)"
    }
}

class PetOwner {
    let pet = Cat(name: "Барсик")

    func infoPet() -> String {
        return "Мое животное: \(pet.info())"
    }
}
```

```
let owner = PetOwner()
print(owner.infoPet())
// Мое животное: Кот по имени - Барсик
```

Как видим, есть класс владельца, который может показать информацию о своем животном. Животное — это класс **Cat** (кошка), но не все любят кошек. Кому-то нравятся кролики (**Rabbit**). Поскольку **Cat** жестко встроен в класс **PetOwner**, последний зависит от него. Данная зависимость должна быть развязана для поддержки класса **Rabbit** или других.

Внедрение и управление зависимостями

Применим метод внедрения зависимостей. Создадим протокол **Pet**. и модифицируем класс **Cat** в соответствии с ним. Изменим класс **PetOwner**, чтобы использовать протокол **Pet**.

```
protocol Pet {
    func info() -> String
}

class Cat: Pet {
    let name: String

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

    func info() -> String {
        return "Кот по имени - \(name)"
```

```
    }
}

class PetOwner {
    let pet: Pet

    init(pet: Pet) {
        self.pet = pet
    }

    func infoPet() -> String {
        return "Мое животное: \(pet.info())"
    }
}
```

Мы применили так называемую инъекцию через инициализатор (паттерн **Initializer Injection DI**). Теперь можем создать и владельца кошки, и хозяина кролика:

```
class Rabbit: Pet {
    let name: String

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

    func info() -> String {
        return "Кролик по имени - \(name)"
    }
}
```

```
let ownerCat = PetOwner(pet: Cat(name: "Барсик"))
print(ownerCat.infoPet())

let ownerRabbit = PetOwner(pet: Rabbit(name: "Кроля"))
print(ownerRabbit.infoPet())

// Мое животное: Кот по имени - Барсик
// Мое животное: Кролик по имени - Кроля
```

Это был простой пример внедрения зависимостей (**Dependency injection**, или **DI**).

Swinject

Swinject — это фреймворк для **Dependency injection** в **Swift**.

Внедрение зависимостей (**DI**) — шаблон разработки программного обеспечения, который реализует паттерн «Инверсия управления» (**Inversion of Control**, или **IoC**) для разрешения зависимостей. **Swinject** поможет приложению разбиться на слабо связанные компоненты, которые легко разрабатывать, тестировать и поддерживать.

Рассмотрим применение **Swinject** на нашем примере **PetOwner**. Чтобы установить **Swinject** с помощью **CocoaPods**, добавьте в **Podfile**:

```
pod 'Swinject'
```

Подключаем **Swinject**:

```
import Swinject
```

Переписываем код на использование **IoC**-контейнеров от **Swinject**.

```
let container = Container()
container.register(Pet.self) { _ in Cat(name: "Барсик") }
container.register(PetOwner.self) { r in
    PetOwner(pet: r.resolve(Pet.self)!)
}

let petOwner = container.resolve(PetOwner.self)!
print(petOwner.infoPet())

// Мое животное: Кот по имени - Барсик
```

GBShop

Управление зависимостями в сетевой подсистеме

Вернемся к проекту и применим знания по управлению зависимостями в сетевой части **GBShop**.

Мы уже умеем посылать сетевые запросы и обрабатывать ответы на них с помощью фреймворка **Alamofire**. Рассмотрим пример загрузки данных о погоде для определенного города с удаленного сервиса **openweathermap.org**:

```
func loadWeatherData(city: String, completion: @escaping () -> Void ){
    // Путь для получения погоды за 5 дней
    let path = "/data/2.5/forecast"
    // Параметры, город, единицы измерения — градусы, ключ для доступа к сервису
    let parameters: Parameters = [
        "q": city,
        "units": "metric",
        "appid": apiKey
    ]
    // Составляем url из базового адреса сервиса и конкретного пути к ресурсу
    let url = baseUrl+path
    // Делаем запрос
    Alamofire.request(url, method: .get, parameters: parameters).responseData {
[weak self] response in
        if let status = repsonse.response?.statusCode {
            guard (200..<300).contains(status) else {
                print("Wrong response status: \(status)")
                return
            }
        }
        guard let data = repsonse.value else { return }
        do {
            let json = try JSON(data: data)
            let weathers = json["list"].flatMap { Weather(json: $0.1, city:
```

```
city) }
            self?.saveWeatherData(weathers, city: city)
            completion()
        } catch {
            // Если произошла ошибка, выводим ее в консоль
            print(error)
        }
    }
}
```

Уже видим, в какой клубок переплелись зависимости. И обработка ошибок, и посылка запроса, и обработка данных, и сохранение данных…

Будем работать с фреймворком **Alamofire**, поэтому именно его классы будут встречаться по выделяемым протоколам.

Выделим в группе **Core** группу **Network**:

Реализуем сетевую систему проекта через управление зависимостями. Начнем с обработки ошибок:

```
protocol AbstractErrorParser {
    func parse(_ result: Error) -> Error
    func parse(response: HTTPURLResponse?, data: Data?, error: Error?) -> Error?
}
```

Используя знания во внедрении зависимостей, выделим протокол **AbstractErrorParser** (в файле **AbstractErrorParser.swift**), который будет заниматься только обработкой ошибок. В этом протоколе

реализуем цель — обработку ошибок. Это можно сделать, получая ошибки через **Alamofire** (первый метод протокола), либо получая данные через **Alamofire** (второй метод протокола). Создадим пока простую реализацию данного протокола — **ErrorParser** (в файле **ErrorParser.swift**). Позже поймем, чего хотим от этого класса:

```
class ErrorParser: AbstractErrorParser {
    func parse(_ result: Error) -> Error {
        return result
    }

    func parse(response: HTTPURLResponse?, data: Data?, error: Error?) -> Error?
{
        return error
    }
}
```

Выделим отдельный класс обработки полученного ответа для **Alamofire** (в файле **DataRequest.swift**). По умолчанию **Alamofire** умеет разбирать только строки, данные и **json**. Расширим его возможности этим классом: это позволит разбирать класс, унаследованный от протокола **Codable**. Он будет расширением стандартного класса **DataRequest** от фреймворка **Alamofire**:

```
import Alamofire

extension DataRequest {
    @discardableResult
    func responseCodable<T: Decodable>(
        errorParser: AbstractErrorParser,
        queue: DispatchQueue? = nil,
        completionHandler: @escaping (DataResponse<T>) -> Void)
        -> Self {
            let responseSerializer = DataResponseSerializer<T> { request,
response, data, error in
                if let error = errorParser.parse(response: response, data: data,
error: error) {
                    return .failure(error)
                }
                let result = Request.serializeResponseData(response: response,
data: data, error: nil)
                switch result {
                case .success(let data):
                    do {
                        let value = try JSONDecoder().decode(T.self, from: data)
                        return .success(value)
                    } catch {
                        let customError = errorParser.parse(error)
                        return .failure(customError)
                    }
                case .failure(let error):
                    let customError = errorParser.parse(error)
                    return .failure(customError)
                }
            }
            return response(queue: queue, responseSerializer:
responseSerializer, completionHandler: completionHandler)
    }
}
```

В классе DataRequest применяем абстрактный **ErrorParser** (объект протокола **AbstractErrorParser**), что позволяет использовать любую реализацию протокола **AbstractErrorParser** по обработке ошибок. В коде класса **DataRequest** применяется стандартное **API Alamofire** по получению данных. Используя класс **JSONDecoder** в классе **DataRequest**, получаем данные нужного типа. Обратите внимание: класс **DataRequest** реализует одну цель — разбор данных.

Создадим класс фабрики запросов **AbstractRequestFactory** (в файле **DataRequest.swift**), единственной целью которой будет создание запросов:

```
import Alamofire

protocol AbstractRequestFactory {
    var errorParser: AbstractErrorParser { get }
    var sessionManager: SessionManager { get }
    var queue: DispatchQueue? { get }

    @discardableResult
    func request<T: Decodable>(
        request: URLRequestConvertible,
        completionHandler: @escaping (DataResponse<T>) -> Void)
        -> DataRequest
}

extension AbstractRequestFactory {

    @discardableResult
    public func request<T: Decodable>(
        request: URLRequestConvertible,
        completionHandler: @escaping (DataResponse<T>) -> Void)
        -> DataRequest {
            return sessionManager
                .request(request)
                .responseCodable(errorParser: errorParser, queue: queue,
completionHandler: completionHandler)
    }
}
```

Фабрика — это порождающий шаблон проектирования, предоставляет интерфейс для создания взаимосвязанных объектов, не указывая, каких конкретно классов они будут. Шаблон реализуется добавлением протокола **Factory**, который представляет методы для создания этих объектов. Далее пишутся наследуемые от **Factory** классы, реализующие эти методы.

Добавим следующий протокол **RequestRouter** (в файле **RequestRouter.swift**), который будет непосредственно реализовывать наш запрос к серверу:

```
import Alamofire

enum RequestRouterEncoding {
    case url, json
}

protocol RequestRouter: URLRequestConvertible {
    var baseUrl: URL { get }
    var method: HTTPMethod { get }
    var path: String { get }
    var parameters: Parameters? { get }
```

```
    var fullUrl: URL { get }
    var encoding: RequestRouterEncoding { get }
}

extension RequestRouter {
    var fullUrl: URL {
        return baseUrl.appendingPathComponent(path)
    }

    var encoding: RequestRouterEncoding {
        return .url
    }

    func asURLRequest() throws -> URLRequest {
        var urlRequest = URLRequest(url: fullUrl)
        urlRequest.httpMethod = method.rawValue

        switch self.encoding {
        case .url:
            return try URLEncoding.default.encode(urlRequest, with: parameters)
        case .json:
            return try JSONEncoding.default.encode(urlRequest, with: parameters)
        }

    }
}
```

Теперь попробуем реализовать логику входа в личный кабинет нашего интернет-магазина, используя API сервера. Описание находится по ссылке online-store-api.

Запрос «Войти»:

Данные запроса	{
	"username" : "Somebody",
	"password" : "mypassword",
	"cookie" : ...
	}

Данные ОК ответа JSON	{
	"result": 1,
	"user": {
	"id_user": 123,
	"user_login": "geekbrains",
	"user_name": "John",
	"user_lastname": "Doe"
	}
	}

Данные ответа JSON с ошибкой	{ result: 0, errorMessage : "Сообщение об ошибке" }

Создадим классы, описывающие ответы от нашего сервера, используя представленную выше таблицу. Для этого создадим два файла в группе **Entity**: **LoginResult.swift** и **User.swift**, соответственно со структурами **LoginResult** и **User**.

```
struct LoginResult: Codable {
```

```
    let result: Int
    let user: User
}

struct User: Codable {
    let id: Int
    let login: String
    let name: String
    let lastname: String

    enum CodingKeys: String, CodingKey {
        case id = "id_user"
        case login = "user_login"
        case name = "user_name"
        case lastname = "user_lastname"
    }
}
```

Выделим новую группу **Network** в группе **BusinessLogic**.

В ней создадим протокол **AuthRequestFactory** (в файле **AuthRequestFactory.swift**), который будет заниматься только реализацией входа в личный кабинет:

```
import Alamofire

protocol AuthRequestFactory {
    func login(userName: String, password: String, completionHandler: @escaping
(DataResponse<LoginResult>) -> Void)
}
```

Добавим класс **Auth** (в файле **Auth.swift**), который будет реализовывать протокол **AuthRequestFactory** и заниматься непосредственно обработкой запроса «Войти». Он будет знать, к какому ресурсу обращаться, по какому сетевому адресу выполнять запрос и с какими параметрами.

```
import Alamofire

class Auth: AbstractRequestFactory {
    let errorParser: AbstractErrorParser
    let sessionManager: SessionManager
    let queue: DispatchQueue?
    let baseUrl = URL(string:
"https://raw.githubusercontent.com/GeekBrainsTutorial/online-store-api/master/re
sponses/")!

    init(
        errorParser: AbstractErrorParser,
        sessionManager: SessionManager,
        queue: DispatchQueue? = DispatchQueue.global(qos: .utility)) {
        self.errorParser = errorParser
        self.sessionManager = sessionManager
        self.queue = queue
    }
}

extension Auth: AuthRequestFactory {
    func login(userName: String, password: String, completionHandler: @escaping
(DataResponse<LoginResult>) -> Void) {
```

```
        let requestModel = Login(baseUrl: baseUrl, login: userName, password:
password)
        self.request(request: requestModel, completionHandler:
completionHandler)
    }
}

extension Auth {
    struct Login: RequestRouter {
        let baseUrl: URL
        let method: HTTPMethod = .get
        let path: String = "login.json"

        let login: String
        let password: String
        var parameters: Parameters? {
            return [
                "username": login,
                "password": password
            ]
        }
    }
}
```

Обратите внимание: реализацию протоколов класса **Auth** мы вынесли в отдельные **extension**. Этот прием очень удобен для дальнейшей навигации по коду. Параметры и путь к запросу берем из описания к запросу «Войти» и по ссылке online-store-api.

Создадим класс фабрики запросов **RequestFactory** (в файле **RequestFactory.swift**). Пока фабрика будет уметь отдавать только запрос «Войти». В дальнейшем именно здесь будем создавать объекты последующих запросов.

```
import Alamofire

class RequestFactory {

    func makeErrorParser() -> AbstractErrorParser {
        return ErrorParser()
    }

    lazy var commonSessionManager: SessionManager = {
        let configuration = URLSessionConfiguration.default
        configuration.httpShouldSetCookies = false
        configuration.httpAdditionalHeaders = SessionManager.defaultHTTPHeaders
        let manager = SessionManager(configuration: configuration)
        return manager
    }()

    let sessionQueue = DispatchQueue.global(qos: .utility)

    func makeAuthRequestFatory() -> AuthRequestFactory {
        let errorParser = makeErrorParser()
        return Auth(errorParser: errorParser, sessionManager:
commonSessionManager, queue: sessionQueue)
    }
}
```

Создаем объект типа **RequestFactory** в классе **AppDelegate**:

```
let requestFactory = RequestFactory()
```

Осталось отправить запрос «Войти» и распечатать ответ. Все эти действия реализуем в методе класса:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    let auth = requestFactory.makeAuthRequestFatory()
    auth.login(userName: "Somebody", password: "mypassword") { response in
        switch response.result {
        case .success(let login):
            print(login)
        case .failure(let error):
            print(error.localizedDescription)
        }
    return true
}
```

Результат работы:

Практическое задание

1. Создать класс(ы) для регистрации, авторизации, выхода, изменения личных данных.
2. Обязательно использовать **DI**: зависимости не должны создаваться внутри класса.

Не стоит забывать о работе с гит. Каждая домашняя работа ведется в отдельной ветке. Внутри ветки — несколько коммитов. Решение о сохранении изменений в Git-репозитории — за вами. Необходимо ли кэшировать данные и с помощью чего это делать, тоже решаете сами.

На данном этапе у нас нет **UI**. Поэтому проверять работоспособность написанного кода необходимо, вызывая метод класса **AppDelegate**:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool
```

Дополнительные материалы

1. Swinject.
2. online-store-api.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Swinject.