GeekBrains

Архитектуры и шаблоны проектирования на Swift

Базовые паттерны. Часть 3

Паттерны flyweight, adapter, factory, template method.

Оглавление

Проект

Разбирать паттерны на этом уроке будем на примере погодного приложения, которое ученики писали на ранних курсах. Шаблон этого приложения находится по ссылке: https://drive.google.com/open?id=1UdNrgMp11VS8ab8IEn3MtASpA9mTg1VN .

Приложение использует **Alamofire** для запросов в сеть. Эти запросы реализованы в классе **WeatherService**. Приложение использует **Realm** для сохранения полученных из сети данных в БД. Из этой базы данные берутся для отображения во вью-контроллере **WeatherViewController**. Также в приложении реализованы кастомные UI-элементы, кастомные переходы и т. д. В большинстве случаев на нашем уроке они будут не важны.

Паттерн Flyweight

Разберем паттерн **flyweight** — «легковесный». В переводах классических книг он обычно называется «приспособленец».

По классификации это структурный паттерн. Его цель — в минимизации используемой памяти там, где необязательно создавать несколько объектов, а вместо этого можно переиспользовать один. Сразу перейдем в **playground**, чтобы понять, о чем будет идти речь.

Пример в Playground

Допустим, мы создаем в коде UI-компонент. Нужно установить цвет для его текста и для обводки элемента. Тогда мы напишем что-то вроде:

```
label.textColor = UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0)
layer.borderColor = UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0).cgColor
```

Для двух элементов мы применили один и тот же цвет, инициализировав его два раза с одними и теми же параметрами. Посмотрим, правильно ли мы сделали. Проведем такой эксперимент в **playground**:

```
let color1 = UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0)
let color2 = UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0)
print(color1 === color2)
```

В консоли выведется **false**. Мы создали два разных объекта, и они лежат в разных участках памяти — даже несмотря на то, что они абсолютно идентичны.

*Примечание: оператор **===** в Swift возвращает **true**, если переменные ссылаются на одно и то же место в памяти.*

Если мы напишем так:

```
let red1 = UIColor.red
let red2 = UIColor.red
print(red1 === red2)
```

… то в консоли выведется **true**. **UIColor.red** не создает каждый раз новый объект, а всегда возвращает один и тот же. Это и есть паттерн **flyweight**: вместо создания и хранения в памяти нескольких объектов, нужных для одной цели, мы создаем и храним один объект и используем его для всех случаев.

Все стандартные цвета, такие как **UIColor.red**, **UIColor.green**, **UIColor.white**, **UIColor.black** и т. д. реализованы по паттерну **flyweight**.

Пример в проекте

В любом приложении под iOS используются цвета. Как правило, дизайнер придумывает палитру из оттенков, наиболее часто используемых в дизайне приложения, а в коде хранятся переменные, которые эту палитру описывают. Самым логичным решением будет хранить эти переменные, используя паттерн **flyweight**, который в Swift реализуется очень просто.

Перейдем в проект. Собственные цвета здесь используются в кастомном ui-элементе с выбором дня недели и в навбаре. Вынесем их в расширение для **UIColor**. Это будут статические константы:

```
import UIKit

// MARK: - Brand Colors

extension UIColor {

    static let brandOrange = UIColor(red: 255.0 / 255.0, green: 100.0 / 255.0,
blue: 20.0 / 255.0, alpha: 1.0)

    static let brandWhite = UIColor(red: 250.0 / 255.0, green: 250.0 / 255.0,
blue: 255.0 / 255.0, alpha: 1.0)

    static let brandGreyRed = UIColor(red: 220.0 / 255.0, green: 110.0 / 255.0,
blue: 110.0 / 255.0, alpha: 1.0)

    static let brandGrey = UIColor(red: 150.0 / 255.0, green: 150.0 / 255.0,
blue: 150.0 / 255.0, alpha: 1.0)
}
```

Затем заменим все места в проекте, где инициализировались цвета, на эти константы. Задание цветов в **WeekDayPicker**:

```
button.setTitleColor(day.isWeekend ? .brandGreyRed : .brandGrey, for: .normal)
button.setTitleColor(.brandWhite, for: .selected)
button.tintColor = .brandOrange
```

Задание цветов для навбара:

```
self.navigationController?.navigationBar.tintColor = UIColor.brandOrange
self.navigationController?.navigationBar.backgroundColor = UIColor.brandWhite
```

Вместо инициализации нескольких объектов **UIColor**, содержащих одинаковые данные, инициализируется один, и к нему в проекте всегда есть доступ через расширение для **UIColor**.

Так же обычно поступают с палитрой шрифтов — создают расширение для **UIFont**, в котором статическими константами прописывают все шрифты, которые используются в приложении.

Теперь зайдем в класс **WeatherCell**, где нас ждет интересный вызов. Дело в том, что цвет для тени здесь не задается из палитры, а рассчитывается исходя из температуры, которую ячейка отображает — чем выше температура, тем «теплее» цвет. А температура может быть любым значением **Double**, то есть вынести все эти цвета в константы в принципе невозможно. Как бы тут сэкономить память? Ведь если разные ячейки будут отображать одинаковую температуру (а такое вполне возможно в прогнозе погоды), то они также будут отображать и одинаковые цвета, но объекты **UIColor** всегда инициализируются и не переиспользуются. Как быть?

Следующий шаг — сохранение объектов в кеш. Когда создается новый объект, который, возможно, будет переиспользоваться, он кладется в кеш (им может быть массив, словарь или любая другая структура данных). Если требуется использовать объект, сначала проверяется, нет ли его уже в кеше. Если есть — он берется оттуда, если нет — инициализируется новый и кладется в кеш для последующего переиспользования. Реализуем это — добавим еще одно расширение:

```
// MARK: - Color Store

extension UIColor {

    private static var colorsCache: [String: UIColor] = [:]

    public static func rgba(_ r: CGFloat, _ g: CGFloat, _ b: CGFloat, a:
CGFloat) -> UIColor {
        let key = "\(r)\(g)\(b)\(a)"
        if let cachedColor = self.colorsCache[key] {
            return cachedColor
        }
        self.clearColorsCacheIfNeeded()
        let color = UIColor(red: r/255.0, green: g/255.0, blue: b/255.0, alpha:
a)
        self.colorsCache[key] = color
        return color
    }

    private static func clearColorsCacheIfNeeded() {
        let maxObjectsCount = 100
        guard self.colorsCache.count >= maxObjectsCount else { return }
        colorsCache = [:]
    }
}
```

При использовании функции **UIColor.rgba(r, g, b, a)** описанное выше поведение с взятием из кеша и сохранением в кеш реализовано. Чтобы кеш не переполнялся, при необходимости очищаем его. Мы сделали наиболее простую реализацию — если в кеше превышено максимально допустимое количество объектов, то просто обнуляем его. Можно сделать более умную — очищать те объекты, которые были добавлены давно, и оставлять те, что были созданы недавно. Но это уже частности.

Осталось только использовать эти наработки в **WeatherCell**:

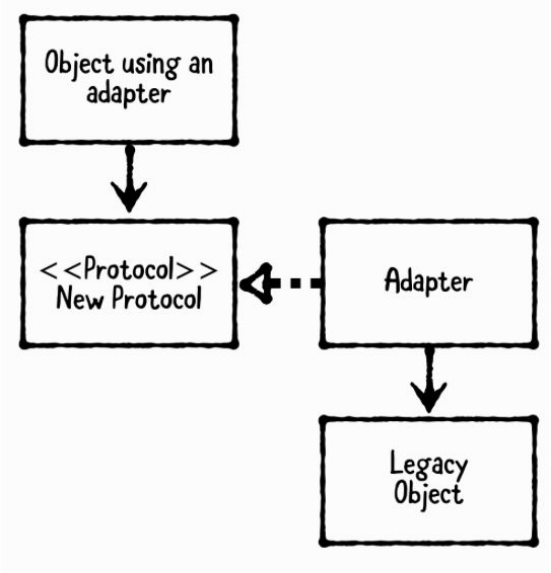
```
let color = min(abs(weather.temp) * 10, 255.0)
if weather.temp <= 0 {
    self.shadowView.layer.shadowColor =
        UIColor.rgba(0.0, 50.0, CGFloat(color), a: 1.0).cgColor
} else {
    self.shadowView.layer.shadowColor =
        UIColor.rgba(CGFloat(color), 0.0, 0.0, a: 1.0).cgColor
}
```

Теперь, если у двух разных ячеек будет одинаковая температура, то во второй раз объект **UIColor** возьмется из кеша. Это можно проверить, поставив соответствующий **breakpoint**.

Паттерн Adapter

Паттерн **Adapter** (адаптер) — поведенческий шаблон проектирования. Он позволяет несовместимым интерфейсам работать вместе. Представьте, что у вас есть код, который вы или не можете менять (например, он находится внутри закрытого фреймворка), или очень не хотите этого делать. Но использовать его трудно и хочется обращаться к нему через совсем другой интерфейс. В этом случае вы создаете такой интерфейс (в виде протокола), затем пишете объект-адаптер. Он будет поддерживать этот протокол. Таким образом, обращаясь к адаптеру, вы используете тот интерфейс, который хотели. Но сам адаптер не реализует функциональность, а лишь использует уже написанную.

Схема:



Пример в проекте

Вернемся к проекту погодного приложения и реализуем в нем паттерн «адаптер». Тут как раз для него есть подходящая задача.

Текущая реализация. Есть **WeatherService** — с помощью этого класса можно запрашивать данные о погоде в городе по заданному названию. У него один метод, торчащий наружу:

```
func loadWeatherData(city: String)
```

Он вызывает загрузку данных о погоде. Но ничего не возвращает и не вызывает асинхронно замыкание с полученными данными. Вместо этого метод после получения данных от сервера сохраняет их в **Realm**. Те, кто использует этот сервис, должны подписаться на изменения в базе данных и именно так получать данные. У такого подхода есть плюсы, но обычно он работает не очень хорошо: не очевидно, когда придут данные и откуда, сложно обрабатывать ошибки и покрывать такой код тестами. Но менять всю эту реализацию мы не будем — сохранение в базу данных может еще пригодиться. Вместо этого напишем адаптер с нужным интерфейсом — тем, что обычно используется в современном Swift.

Начнем с того, что добавим модель данных. С объектами **RLMWeather** и **RLMCity** будет работать адаптер внутри своей реализации, но объекты из базы данных мы не хотим доставать наружу. Так что адаптер будет возвращать обычные свифтовые объекты класса **Weather**, который напишем так:

```
typealias Celsius = Double
struct Weather {
    let cityName: String
    let date: Date
    let temperature: Celsius
    let pressure: Double
    let humidity: Double
    let weatherName: String
    let weatherIconName: String?
    let windSpeed: Double
    let windDegrees: Double
}
```

По сути, объект содержит то же самое, что есть в модельке **RLMWeather**. Но это более удобная структура, которая не тянет с собой ни **Realm**, ни **Objective-C**.

Теперь подумаем над адаптером. Нам бы хотелось иметь такой метод:

```
func getWeathers(inCity city: String, then completion: @escaping ([Weather]) ->
Void)
```

С помощью него мы запрашиваем данные о погоде, и он асинхронно возвращает через замыкание **completion**. Пока не будем обрабатывать ошибки. Внутри своей реализации этот метод будет вызывать **WeatherService().loadWeatherData(city: city)** и, поскольку все данные передаются в Realm, подписываться на обновление данных в нем. Затем, когда придут соответствующие данные в БД, мы переведем модельки **RLMWeather** в только что добавленную **Weather** и вызовем **completion** с массивом этих созданных моделей.

Создадим файл **WeatherAdapter.swift** и добавим в него следующий код:

```
import Foundation
import RealmSwift

final class WeatherAdapter {

    private let weatherService = WeatherService()

    private var realmNotificationTokens: [String: NotificationToken] = [:]

    func getWeathers(inCity city: String, then completion: @escaping ([Weather])
-> Void) {
        guard let realm = try? Realm()
            , let realmCity = realm.object(ofType: RLMCity.self, forPrimaryKey:
city)
            else { return }

        self.realmNotificationTokens[city]?.stop()

        let token = realmCity.weathers.addNotificationBlock { [weak self]
(changes: RealmCollectionChange) in
            guard let self = self else { return }
            switch changes {
            case .update(let realmWeathers, _, _, _):
                var weathers: [Weather] = []
                for realmWeather in realmWeathers {
                    weathers.append(self.weather(from: realmWeather))
                }
                self.realmNotificationTokens[city]?.stop()
                completion(weathers)
            case .error(let error):
                fatalError("\(error)")
            case .initial:
                break
            }
        }
        self.realmNotificationTokens[city] = token

        weatherService.loadWeatherData(city: city)
    }

    private func weather(from rlmWeather: RLMWeather) -> Weather {
        return Weather(cityName: rlmWeather.city,
                       date: Date(timeIntervalSince1970: rlmWeather.date),
                       temperature: rlmWeather.temp,
                       pressure: rlmWeather.pressure,
                       humidity: Double(rlmWeather.humidity),
                       weatherName: rlmWeather.weatherName,
                       weatherIconName: rlmWeather.weatherIcon,
                       windSpeed: rlmWeather.windSpeed,
                       windDegrees: rlmWeather.windDegrees)
    }
}
```

*Примечание: сейчас **WeatherAdapter** будет некорректно обрабатывать кейс, когда повторно вызывается метод **getWeathers**, пока предыдущий вызов этого метода с тем же названием города еще не успел отработать. Это можно исправить, но в нашем курсе необязательно.*

Теперь перейдем в **WeatherViewController** и переведем его на адаптер. Заменим старый код:

```
private let weatherService = WeatherService()
private var weathers: List<RLMWeather>!
```

на новый, использующий новые модельки и адаптер:

```
private let weatherService = WeatherAdapter()
private var weathers: [Weather] = []
```

Заменим реализацию **viewDidLoad** и удалим метод **pairTableAndRealm()**. Вместо них будет следующий код:

```
override func viewDidLoad() {
    super.viewDidLoad()
    weatherService.getWeathers(inCity: cityName) { [weak self] weathers in
        self?.weathers = weathers
        self?.collectionView.reloadData()
    }
}
```

Сейчас компилятор ругается на вызов метода конфигурации ячейки, потому что он требует передать ему модель **RLMWeather**. Заменим этот метод на новый, использующий модель **Weather**:

```
func configure(with weather: Weather) {
    let stringDate = WeatherCell.dateFormatter.string(from: weather.date)

    self.weather.text = String(Int(round(weather.temperature)))
    time.text = stringDate
    icon.image = weather.weatherIcon

    let color = min(abs(weather.temperature) * 10, 255.0)
    if weather.temperature <= 0 {
        self.shadowView.layer.shadowColor = UIColor.rgba(0.0, 50.0,
CGFloat(color), a: 1.0).cgColor
    } else {
        self.shadowView.layer.shadowColor = UIColor.rgba(CGFloat(color), 0.0,
0.0, a: 1.0).cgColor
    }
}
```

Теперь проект работает и данные успешно грузятся. Мы применили паттерн **adapter** для создания нового интерфейса, с которым работаем в клиентском коде, но не изменяли внутреннюю реализацию сервиса для получения погоды.

Паттерн Factory

Factory (фабрика) — еще один порождающий паттерн (как и **builder**, рассмотренный на прошлом уроке). Фабрика позволяет скрыть внутри себя создание сложного объекта. В отличие от **builder**, она не позволяет создавать объект пошагово. Вместо этого у фабрики есть один метод с входными параметрами, которые определяют, как и какой будет создаваться объект. И на выходе этот метод возвращает созданный объект. Никаких **setMeat**, **addSauce** и других методов, которые мы использовали в примере с **builder** и которые конфигурировали создаваемый объект пошагово.

Есть несколько вариантов использования паттерна **factory**:

- **simple factory** (простая фабрика),
- **abstract factory** (абстрактная фабрика),
- **factory method** (фабричный метод).

Мы применим **simple factory** в самой частой задаче — для преобразования **model** во **viewModel**. Абстрактная фабрика нужна, чтобы создать объект, не уточняя его конкретный класс. Пример абстрактной фабрики позже рассмотрим в плейграунде. Фабричный метод в iOS-разработке применяется редко, так как в нем используется наследование, без которого практически всегда можно обойтись.

Simple Factory. Пример в проекте

Обсудим сначала проблему, которую будем решать паттерном. Зайдем в **WeatherCell.swift**. Найдем там метод **func configure(with weather: Weather)**. Посмотрим, что он делает внутри.

Такой код создает строку из даты, используя **dateFormatter**:

```
let stringDate = WeatherCell.dateFormatter.string(from: weather.date)
```

Сам **dateFormatter** приходится держать статическим свойством у ячейки, что нелогично, ведь ячейка — всего лишь UI-элемент.

Далее внутри метода производится конвертация значения температуры в строку для отображения, затем берется картинка. По значению температуры настраивается цвет ячейки. Получается, что все преобразование сырых данных о погоде в данные для отображения на вью (тексты, цвет, иконки) происходит в самой ячейке. Это не антипаттерн, но есть у такого подхода минусы:

1. Нарушается абстракция, UI знает о моделях данных.
2. Нарушается принцип **single responsibility** (единственной ответственности) — ячейка занимается не только отображением UI, но и преобразованием данных модели в отображаемый вид.
3. Все вытекающие из предыдущих двух пунктов недостатки: затрудняется поддержка и изменение кода.

Решение этих проблем заключается в том, чтобы перенести логику преобразования сырой модели в готовый отображаемый вид в отдельный класс-фабрику. Эта фабрика будет на вход принимать модель, а возвращать специальный объект (или структуру), которая содержит только те данные,

которые используются для отображения. Причем с ними ничего больше не нужно делать — бери и используй. Этот объект — вью-модель, то есть модель для отображения.

Применим этот подход в проекте. Сначала создадим вью-модель:

```
struct WeatherViewModel {
    let weatherText: String
    let dateText: String
    let iconImage: UIImage?
    let shadowColor: UIColor
}
```

Обратите внимание, что все поля этой вью-модели будут напрямую использоваться ячейкой, без дополнительных преобразований. Теперь добавим фабрику:

```
final class WeatherViewModelFactory {

    func constructViewModels(from weathers: [Weather]) -> [WeatherViewModel] {
        return weathers.compactMap(self.viewModel)
    }

    private func viewModel(from weather: Weather) -> WeatherViewModel {
        let weatherText = String(Int(round(weather.temperature)))
        let dateText = WeatherViewModelFactory.dateFormatter.string(from:
weather.date)
        let iconImage = UIImage(named: weather.weatherIconName)
        let colorTone = min(abs(weather.temperature) * 10, 255.0)
        let shadowColor: UIColor
        if weather.temperature <= 0 {
            shadowColor = UIColor.rgba(0.0, 50.0, CGFloat(colorTone), a: 1.0)
        } else {
            shadowColor = UIColor.rgba(CGFloat(colorTone), 0.0, 0.0, a: 1.0)
        }
        return WeatherViewModel(weatherText: weatherText, dateText: dateText,
iconImage: iconImage, shadowColor: shadowColor)
    }

    private static let dateFormatter: DateFormatter = {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "dd.MM.yyyy HH.mm"
        return dateFormatter
    }()
}
```

У фабрики нужно вызвать функцию **constructViewModels(from weathers: [Weather])** и получить массив вью-моделей. Затем эти вью-модели должны передаваться ячейке. Перейдем в ячейку **WeatherCell** и добавим метод конфигурации с вью-моделью, а все остальные методы конфигурации удалим:

```
func configure(with viewModel: WeatherViewModel) {
    weather.text = viewModel.weatherText
    time.text = viewModel.dateText
    icon.image = viewModel.iconImage
    shadowView.layer.shadowColor = viewModel.shadowColor.cgColor
}
```

Теперь перейдем в **WeatherViewController**. Добавим в нем свойства:

```
private let viewModelFactory = WeatherViewModelFactory()
private var viewModels: [WeatherViewModel] = []
```

Перед обновлением **collection view** нужно сконструировать вью-модели, а в **data source** передавать ячейке соответствующую вью-модель. Для этого метод **viewDidLoad** заменим на следующую реализацию:

```
override func viewDidLoad() {
    super.viewDidLoad()
    weatherService.getWeathers(inCity: cityName) { [weak self] weathers in
        guard let self = self else { return }
        self.weathers = weathers
        self.viewModels = self.viewModelFactory.constructViewModels(from:
weathers)
        self.collectionView.reloadData()
    }
}
```

Здесь перед обновлением мы добавили создание вью-моделей и сохранили их в свойство вью-контроллера. Осталось переделать методы **data source**:

```
func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection
section: Int) -> Int {
    return viewModels.count
}

func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
IndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
"WeatherCell", for: indexPath) as! WeatherCell
    cell.configure(with: viewModels[indexPath.row])
    return cell
}
```

Все работает так же, но мы избавились от проблемы нарушения абстракции и единственной ответственности. Преобразованием модели данных в данные для отображения занимается теперь фабрика — это и есть ее единственная ответственность.

Abstract Factory. Пример в Playground

Абстрактная фабрика — усложнение простой фабрики для создания не одного и того же класса, а разных, объединенных одним протоколом. Абстрактная фабрика создает объект, не специфицируя его конкретный класс.

Рассмотрим пример. Мы хотим создать такой UI, который может имитировать внешний вид системы Windows либо macOS, причем у нас должна быть возможность менять этот интерфейс прямо во время работы программы. Для начала создадим элементы интерфейса для каждой системы — это будут конкретные классы. Создадим кнопки:

```
protocol Button {
    func setTitle(_ title: String?)
    func show()
}

class MacOSButton: Button {
    var title: String?

    func setTitle(_ title: String?) {
        self.title = title
    }

    func show() {
        print("show macos button with title \(self.title ?? "")")
    }
}

class WindowsButton: Button {
    var title: String?

    func setTitle(_ title: String?) {
        self.title = title
    }

    func show() {
        print("show windows button with title \(self.title ?? "")")
    }
}
```

Есть два класса — кнопка под Windows и кнопка под macOS. У них одинаковый интерфейс (свойства и методы), и поддерживают они один и тот же протокол **Button**. Различаются только во внутренней реализации (в примере мы это имитировали разным текстом в методе **show**).

Далее есть два варианта действий — для начала пойдем простым путем. Можно сразу создать класс абстрактной фабрики, которая будет возвращать кнопку. Она будет возвращать **Button** (это протокол), а конкретный класс будет зависеть от передаваемого параметра — для какой системы создается кнопка:

```
enum GUIType {
    case macos
    case windows
```

```
}

class GUIFactory {
    func button(with type: GUIType) -> Button {
        switch type {
        case .macos:
            return MacOSButton()
        case .windows:
            return WindowsButton()
        }
    }
}
```

Вот мы и реализовали абстрактную фабрику в наиболее простом виде. Отличие от **simple factory** хорошо видно в том, что фабрика возвращает не конкретный класс, а объект типа **Button** — а какой именно класс будет использоваться, зависит от входных параметров.

Но, как правило, объекты с помощью фабрик создаются куда более сложные, и их может быть намного больше. Поэтому поступают следующим образом: создают несколько простых фабрик с общим протоколом.

Удалим класс **GUIFactory** и вместо него создадим протокол и следующие фабрики:

```
protocol GUIFactory {

    func button() -> Button
}

class WindowsGUIFactory: GUIFactory {

    func button() -> Button {
        return WindowsButton()
    }
}

class MacOSGUIFactory: GUIFactory {

    func button() -> Button {
        return MacOSButton()
    }
}
```

В зависимости от типа, который нам нужен, будем возвращать разную фабрику:

```
func guiFactory(type: GUIType) -> GUIFactory {
    switch type {
    case .macos:
        return MacOSGUIFactory()
    case .windows:
        return WindowsGUIFactory()
    }
}
```

Посмотрим, как все это можно использовать на практике — например, во вью-контроллере:

```
class ViewController: UIViewController {
    var guiType: GUIType = .macos // можно менять в runtime

    override func viewDidLoad() {
        let factory = guiFactory(type: self.guiType)
        let button = factory.button()
        // дальше можно использовать эту кнопку
        // button здесь имеет тип Button, а не MacOSButton
    }
}
```

От подобного подхода мы получаем много плюсов: можно передать вью-контроллеру **guiType**, и, в зависимости от него, созданные элементы интерфейса (в нашем случае, кнопка **Button**) будут отличаться. Но сам вью-контроллер ничего об этих отличиях не знает, потому что работает с протоколами. Можно потом будет легко создать еще один тип интерфейса, при этом ничего не меняя во вью-контроллере, — благодаря созданной абстракции.

Паттерн Template Method

Паттерн **Template method** (шаблонный метод) — поведенческий паттерн проектирования. Он в обязательном порядке использует наследование. Базовый класс определяет набор методов, которые могут быть реализованы в наследниках, и определяет порядок их вызовов при работе этого объекта. Наследники могут переопределять методы и добавлять в них свою функциональность.

Это очень простой и интуитивный паттерн. Проиллюстрируем его на примере из жизни. Чтобы купить мороженое, вам нужно:

1. Дойти до магазина.
2. Выбрать мороженое.
3. Расплатиться.

Это общие шаги. Именно в таком порядке они вызываются в базовом классе. Базовый класс также определяет все три метода, но не будет содержать никакой реализации.

Конкретная реализация будет находиться в наследниках этого базового класса. Какой она может быть? Например, сценарий такой: вы находитесь дома. Тогда реализация первого метода (дойти до магазина) будет содержать конкретный супермаркет, который ближе к дому. Реализация второго

метода (выбрать мороженое) может состоять в том, что вы выбираете свой любимый пломбир и говорите об этом продавцу. А реализация третьего метода — отдаете рубли. А если вы находитесь в другой стране, то реализация первого метода может также содержать непростой выбор магазина по карте. Реализация второго и третьего метода тоже будет отличаться, так как сказать продавцу о своем выборе нужно на другом языке и отдать деньги в другой валюте.

Пример в Playground

Задача: нам нужны классы, которые будут печатать одну и ту же информацию, но в разном форматировании. Создадим базовый класс:

```
class FormattedTextPrinter {

    let title: String
    let text: [String]

    init(title: String, text: [String]) {
        self.title = title
        self.text = text
    }

    func printFullText() {
        self.startPrintText()
        self.startPrintHead()
        self.printHead()
        self.endPrintHead()
        self.startPrintBody()
        self.printBody()
        self.endPrintBody()
        self.endPrintText()
    }

    func startPrintText() { }

    func startPrintHead() { }

    func printHead() { }

    func endPrintHead() { }

    func startPrintBody() { }

    func printBody() { }

    func endPrintBody() { }

    func endPrintText() { }
}
```

Класс при инициализации требует указать заголовок и массив строк-абзацев, которые формируют непосредственно текст. У этого класса есть метод **printFullText()**, который должен напечатать текст по шагам. Сначала вызывается метод **startPrintText()**. В нем можно добавить функциональность, которая должна быть выполнена прямо перед началом набора текста. Далее вызывается

startPrintHead() — в этой функции должно быть описано, что нужно сделать перед тем, как начать печатать заголовок. Затем **printHead()** — напечатать заголовок. И так далее. В итоге все это разбивается на отдельные шаги, каждый из которых можно по-своему определить в наследниках, но общая структура остается одинаковой.

Сделаем две конкретные реализации: принтер простого текста и принтер текста а-ля газетная статья.

```
final class PlainTextPrinter: FormattedTextPrinter {

    override func printHead() {
        print(self.title)
    }

    override func printBody() {
        print("")
        self.text.forEach { print($0) }
    }
}
```

```
final class NewspaperTextPrinter: FormattedTextPrinter {

    override func startPrintText() {
        print("============ SENSATION ============")
        print("")
    }

    override func printHead() {
        print(self.title.uppercased())
    }

    override func startPrintBody() {
        print("")
        print("See in details:")
    }

    override func printBody() {
        self.text.forEach {
            print($0)
            print("")
        }
    }

    override func endPrintBody() {
        print("Author: Marry Poppins")
    }

    override func endPrintText() {
        print("")
        print("====================================")
    }
}
```

В **PlainTextPrinter** мы лишь определили, как будет печататься заголовок и тело, а реализацию остальных методов оставили пустой. В **NewspaperTextPrinter** мы сделали больше шагов — в начале и в конце печатаем разделитель, заголовок набираем капсом, а после того как напечатали тело, указываем автора статьи. Давайте посмотрим, как эти классы будут работать:

```
let title = "Atmospheric river"
let text = ["An atmospheric river (AR) is a narrow corridor or filament of
concentrated moisture in the atmosphere.",
            "Atmospheric rivers consist of narrow bands of enhanced water vapor
transport, typically along the boundaries between large areas of divergent
surface air flow.",
            "The term was originally coined by researchers Reginald Newell and
Yong Zhu of the Massachusetts Institute of Technology in the early 1990s, to
reflect the narrowness of the moisture plumes involved."]

let plainTextPrinter = PlainTextPrinter(title: title, text: text)
let newspaperTextPrinter = NewspaperTextPrinter(title: title, text: text)

plainTextPrinter.printFullText()
newspaperTextPrinter.printFullText()
```

От обычного принтера в консоли увидим следующий вывод:

```
Atmospheric river

An atmospheric river (AR) is a narrow corridor or filament of concentrated
moisture in the atmosphere.
Atmospheric rivers consist of narrow bands of enhanced water vapor transport,
typically along the boundaries between large areas of divergent surface air
flow.
The term was originally coined by researchers Reginald Newell and Yong Zhu of
the Massachusetts Institute of Technology in the early 1990s, to reflect the
narrowness of the moisture plumes involved.
```

Как мы и предполагали, все вывелось без форматирования. А вот от **newspaperTextPrinter** вывод будет таким:

```
============ SENSATION ============

ATMOSPHERIC RIVER

See in details:
An atmospheric river (AR) is a narrow corridor or filament of concentrated
moisture in the atmosphere.

Atmospheric rivers consist of narrow bands of enhanced water vapor transport,
typically along the boundaries between large areas of divergent surface air
flow.

The term was originally coined by researchers Reginald Newell and Yong Zhu of
the Massachusetts Institute of Technology in the early 1990s, to reflect the
narrowness of the moisture plumes involved.

Author: Marry Poppins

====================================
```

Примеры в iOS SDK

В iOS SDK шаблонный метод, наряду с паттернами делегат и синглтон, применяется достаточно часто. Главный пример — **UIViewController**. У этого класса есть методы, открытые к переопределению наследниками, — они обычно называются методами жизненного цикла вью-контроллера. Среди них:

```
func loadView()
func viewDidLoad()
func viewWillAppear(_ animated: Bool)
func viewDidAppear(_ animated: Bool)
func viewWillDisappear(_ animated: Bool)
func viewDidDisappear(_ animated: Bool)
func viewWillLayoutSubviews()
func viewDidLayoutSubviews()
func didReceiveMemoryWarning()
```

Некоторые из этих методов не содержат реализации, другие содержат и требуют в наследниках вызывать реализацию суперкласса (об этом написано в документации к методам). На протяжении своего жизненного цикла (когда вью-контроллер загружен, показан, убран и т. д.) родительский класс вызывает эти методы в определенном порядке, а в наследнике может быть своя реализация. Например, в **viewDidLoad** мы часто пишем код по настройке внешнего вида. В **viewWillAppear** можно обновлять данные (каждый раз при появлении экрана), а в **viewDidLayoutSubviews** — узнать рассчитанные размеры вью на экране и использовать их.

Другой пример из iOS SDK — **UIView**. У наследников **UIView** часто переопределяется метод **func layoutSubviews()**, реже — **func willMove(toSuperview newSuperview: UIView?)** и **func didMoveToSuperview()**. Есть и другие методы, которые можно переопределить.

У **UIResponder** есть следующие методы для обработки нажатий:

```
open func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?)
open func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?)
open func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?)
open func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?)
```

Переопределив их, мы можем делать что-то по нажатию на экран в определенной точке. Кстати, и **UIView**, и **UIViewController** являются наследниками **UIResponder**.

Но на практике при проектировании приложения и создании собственных классов мы редко прибегаем к паттерну «шаблонный метод». Основной минус — он использует наследование, а сложная иерархия классов с разными реализациями шаблонных методов и вызовами реализации суперклассов может сильно усложнить и понимание кода, и его поддержку. Кроме того, для тех же целей часто можно использовать другие паттерны, например **delegate** или **observer**. Но «шаблонный метод» широко используется в UIKit, поэтому его обязательно надо знать.

Практическое задание

1. В практическом задании вы будете работать с проектом **VK Client**, который вы делали на предыдущих курсах. Если он у вас не сохранился, можно воспользоваться готовым шаблоном с проектом по ссылке: https://drive.google.com/open?id=1kNtnuaXbbrRzPiUOlpIJbMey4PLG26m3. В этом проекте реализуйте все изученные на этом уроке шаблоны проектирования.

2. Найдите все места в проекте, где используются экземпляры **UIColor** и **UIFont**. Примените **flyweight**, чтобы избежать лишней инициализации объектов.

3. Найдите в проекте код, отвечающий за получение данных из сети: получение списка новостей, друзей, фотографий и т. д. В шаблоне по ссылке выше за это отвечает **AlamofireService**, в методах которого данные асинхронно передаются с помощью паттерна «делегат». Отрефакторите код так, чтобы данные асинхронно возвращались в замыкание, которое передается параметром (аналогично тому, как мы сделали на уроке). При этом не меняйте существующие классы — воспользуйтесь паттерном **adapter**.

4. Найдите в проекте код, который отвечает за UI — отображение пришедших с сервера данных. Это все ячейки **table view / collection view**, отображение данных во вью-контроллере, сторибордах и т. д. Найдите код, который преобразует сырые данные из моделей в информацию для отображения. Воспользуйтесь паттерном **simple factory** и создавайте вью-модели, по которым будет отрисовываться интерфейс.

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.

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».