GeekBrains

Архитектуры и шаблоны проектирования на Swift

Базовые паттерны. Часть 2

Паттерны strategy, facade, observer, builder.

Оглавление

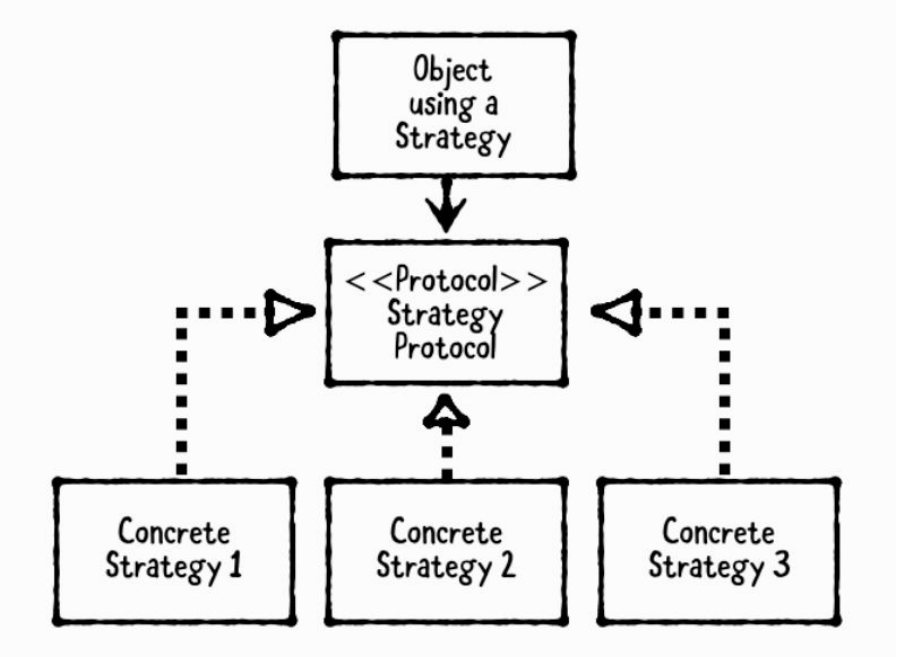
Паттерн Strategy

Паттерн **Strategy** (стратегия) — поведенческий шаблон проектирования. Он определяет семейство объектов, каждый из которых реализует один и тот же интерфейс разными алгоритмами. В рантайме объекты могут заменяться друг на друга, что позволяет при одних и тех же действиях юзера, но разных условиях использовать разную реализацию.

Другими словами, стратегия позволяет выбрать путь, которым мы получим результат. Паттерн очень прост в использовании и практически не имеет минусов. Единственный недостаток — нужно создавать дополнительные классы и сущности, но это свойственно большинству паттернов проектирования.

Пример в Playground

Посмотрим на схему паттерна Strategy. Сама стратегия описывается протоколом **StrategyProtocol**, он обычно определяет как минимум один метод. Есть несколько реализаций этого протокола — **ConcreteStrategy1**, **ConcreteStrategy2** и т. д. В общем случае их может быть сколько угодно, но как минимум две. И есть объект, который использует стратегию. В iOS-разработке это чаще всего вью-контроллер. Объект обращается к стратегии через протокол, а конкретный объект, реализующий стратегию, может быть любым из перечисленных.



В качестве примера реализуем стратегию сжатия изображения. Пусть на входе у стратегии будет картинка **UIImage**, а на выходе мы должны получить сырые данные **Data**, сжатые по одному из алгоритмов: **png** или **jpeg**. Сначала создадим протокол стратегии:

```
protocol ImageToDataConvertingStrategy {
    func data(from image: UIImage) -> Data?
}
```

Сделаем три конкретные стратегии: сжатие без потерь в **png**, сжатие в **jpeg** с минимальной компрессией (высоким качеством), сжатие в **jpeg** с максимальной компрессией (низким качеством):

```
class ImageToPNGDataConverter: ImageToDataConvertingStrategy {
    func data(from image: UIImage) -> Data? {
        return image.pngData()
    }
}

class ImageToJPEGDataConverter: ImageToDataConvertingStrategy {
    func data(from image: UIImage) -> Data? {
        return image.jpegData(compressionQuality: 0.95)
    }
}

class ImageToJPEGLowQualityDataConverter: ImageToDataConvertingStrategy {
    func data(from image: UIImage) -> Data? {
        return image.jpegData(compressionQuality: 0)
    }
}
```

Теперь можно воспользоваться этими стратегиями. Добавим в **Resources** плейграунда картинку **bird** и посмотрим, как она будет выглядеть после сжатия.

```
let image = UIImage(named: "bird")!

var convertingStrategy: ImageToDataConvertingStrategy =
ImageToPNGDataConverter()
if let data = convertingStrategy.data(from: image) {
    UIImage(data: data) // кликнуть в плейграунде на значок просмотра
}
convertingStrategy = ImageToJPEGDataConverter()
if let data = convertingStrategy.data(from: image) {
    UIImage(data: data)
}
convertingStrategy = ImageToJPEGLowQualityDataConverter()
if let data = convertingStrategy.data(from: image) {
    UIImage(data: data) // здесь будет видно низкое качество
}
```

Важно, что в одной и той же переменной можно хранить любую из конкретных стратегий — благодаря общему протоколу.

Реализация в проекте

Продолжаем проект «Змейка». Добавим функциональность выбора сложности игры. Но это слишком общее понятие, чтобы применять стратегию, — нужно определиться, чем именно будет регулироваться сложность игры. Во-первых, в игре появляются яблоки, и сейчас это происходит в случайных точках. Можно сделать этот элемент игры проще — заранее задать точки, в которых будут появляться яблоки. Во-вторых, у змейки есть скорость движения, и она сейчас постоянна. На легком и среднем уровне сложности можно оставить скорость постоянной, а вот на сложном сделать ее

возрастающей. Это и будут две стратегии — стратегия появления яблок и стратегия изменения скорости змейки.

Сначала добавим **enum** со сложностью игры и возможность ее выбора в главном меню. Создадим файл **Difficulty.swift** и добавим в него следующий код:

```
enum Difficulty {
    case easy, medium, hard, insane
}
```

В классе **GameViewController** добавим свойство:

```
var difficulty: Difficulty = .medium
```

Именно **GameViewController** создает сцену игры, поэтому он должен знать о ее сложности. Этот параметр будет передавать **GameViewController**'у контроллер главного меню, ведь именно в меню юзер выбирает сложность.

Откроем **Main.storyboard** и добавим в главное меню **segmented control** для выбора сложности:

Протянем соответствующий **IBOutlet** в класс **MainMenuViewController**:

```
@IBOutlet weak var difficultyControl: UISegmentedControl!
```

Также в **MainMenuViewController** добавим свойство «сложность», которое будет выбираться исходя из выделенного сегмента:

```
private var selectedDifficulty: Difficulty {
    switch self.difficultyControl.selectedSegmentIndex {
    case 0:
        return .easy
    case 1:
        return .medium
    case 2:
        return .hard
    case 3:
        return .insane
    default:
        return .medium
    }
}
```

В подготовке к **segue** передадим контроллеру игры эту сложность:

```
switch segue.identifier {
case "startGameSegue":
    guard let gameVC = segue.destination as? GameViewController else { return }
    gameVC.difficulty = self.selectedDifficulty
    gameVC.onGameEnd = { [weak self] record in
        self?.lastResultLabel.text = "Последний результат: \(record.value)"
    }
default:
    break
}
```

Теперь в **GameViewController** хранится выбранная сложность игры. Пока это ни на что не влияет, поэтому начнем создавать стратегии. Добавим стратегию создания яблок в новом файле **CreateApplesStrategy.swift**:

```
protocol CreateApplesStrategy {
    func createApples(in rect: CGRect) -> [Apple]
}
```

Желательно сразу предусматривать перспективу будущих правок. Если развивать эту игру, захочется за один ход добавлять несколько яблок на экран, придавать им разные свойства. Поэтому сразу учитываем, что стратегия может возвращать массив яблок.

Теперь добавим конкретные реализации этой стратегии. Первая — яблоки создаются в разных точках рандомно, по одному за раз — так, как работает сейчас.

```
final class RandomCreateOneAppleStrategy: CreateApplesStrategy {
    func createApples(in rect: CGRect) -> [Apple] {
        let randX = CGFloat(arc4random_uniform(UInt32(rect.maxX - 5)) + 1)
        let randY = CGFloat(arc4random_uniform(UInt32(rect.maxY - 5)) + 1)
        let apple = Apple(position: CGPoint(x: randX, y: randY))
        return [apple]
    }
}
```

Создадим еще одну стратегию, в которой заранее зададим места появления яблок массивом точек.

```
final class SequentialCreateOneAppleStrategy: CreateApplesStrategy {

    private let positions = [CGPoint(x: 210, y: 210),
                             CGPoint(x: 250, y: 250),
                             CGPoint(x: 150, y: 250),
                             CGPoint(x: 250, y: 300),
                             CGPoint(x: 210, y: 210),
                             CGPoint(x: 200, y: 210),
                             CGPoint(x: 200, y: 250),
                             CGPoint(x: 100, y: 200),
                             CGPoint(x: 150, y: 300),
                             CGPoint(x: 150, y: 250)]

    private var lastUsedPositionIndex = -1

    func createApples(in rect: CGRect) -> [Apple] {
        self.lastUsedPositionIndex += 1
        if self.lastUsedPositionIndex >= self.positions.count {
            self.lastUsedPositionIndex = 0
        }
        let position = self.positions[self.lastUsedPositionIndex]
        let apple = Apple(position: position)
        return [apple]
    }
}
```

Примечание: на каждую стратегию желательно создавать отдельный файл с кодом.

Перейдем в **GameScene.swift** и в классе **GameScene** добавим свойство, хранящее стратегию появления яблок. Запросим эту стратегию в инициализаторе сцены:

```
private let createApplesStrategy: CreateApplesStrategy
init(size: CGSize, createApplesStrategy: CreateApplesStrategy) {
    self.createApplesStrategy = createApplesStrategy
    super.init(size: size)
}
```

Изменим функцию создания яблок **createApple()** так, чтобы она использовала стратегию:

```
fileprivate func createApple() {
    guard let view = self.view, let scene = view.scene else { return }
    let apples = self.createApplesStrategy.createApples(in: scene.frame)
    guard let apple = apples.first else { return }
    self.apple = apple
    self.addChild(apple)
}
```

Теперь перейдем в **GameViewController.swift** и классу **GameViewController** добавим вычисляемое свойство:

```
private var createAppleStrategy: CreateApplesStrategy {
    switch self.difficulty {
    case .easy:
        return SequentialCreateOneAppleStrategy()
    case .medium, .hard, .insane:
        return RandomCreateOneAppleStrategy()
    }
}
```

Оно создает подходящую стратегию для каждой сложности игры. Именно эту стратегию и передадим теперь в инициализатор сцены:

```
let scene = GameScene(size: view.bounds.size, createApplesStrategy:
self.createAppleStrategy)
```

Запустим приложение и проверим его работу. Для уровней выше легкого ничего не изменилось, как мы и ожидали. Если выбрать легкий, то будет заметно, как яблоки создаются в строгой повторяющейся последовательности, которую мы сами задали. Задача выполнена!

Аналогичным образом создадим стратегию увеличения скорости змейки. Добавим протокол:

```
protocol SnakeSpeedStrategy: class {

    var snake: Snake? { get set }

    var maxSpeed: Double? { get set }

    func increaseSpeedByEatingApple()
}
```

Протокол требует задать объект змейки, у которой стратегия будет увеличивать скорость. Также можно задать максимальную скорость: чтобы на уровне с высокой сложностью не ускориться сверх меры — так, что невозможно играть.

Добавим три конкретных стратегии:

1. Не увеличивает скорость (простой и средний уровень).

2. Увеличивает скорость в арифметической прогрессии до заданного максимального значения (сложный уровень).

3. Увеличивает скорость в геометрической прогрессии, нет верхнего предела скорости (безумный уровень).

Первая стратегия простая и ничего не делает:

```
final class NotIncreaseSnakeSpeedStrategy: SnakeSpeedStrategy {

    var snake: Snake?

    var maxSpeed: Double?

    func increaseSpeedByEatingApple() { }
}
```

Обратите внимание: это не значит, что класс, который мы создали, бесполезен. Это именно стратегия, она удовлетворяет протоколу. Просто именно она ничего не делает со скоростью змейки — но это тоже стратегия! **GameScene** не будет использовать конкретно этот класс, а будет работать со стратегией, скрытой протоколом. А внутри это может быть абсолютно любая стратегия.

Теперь создадим стратегию увеличения скорости в арифметической прогрессии:

```
final class ArithmeticProgressionSnakeSpeedStrategy: SnakeSpeedStrategy {

    var snake: Snake?

    var maxSpeed: Double?

    private let diff = 10.0

    func increaseSpeedByEatingApple() {
        guard let snake = snake else { return }
        snake.moveSpeed += self.diff
        if let maxSpeed = maxSpeed {
            if snake.moveSpeed > maxSpeed {
                snake.moveSpeed = maxSpeed
            }
        }
    }
}
```

Здесь **diff** — это то, насколько будет увеличена скорость при каждом вызове метода (в **points per second**).

И, наконец, стратегия увеличения скорости в геометрической прогрессии:

```
final class GeometricProgressionSnakeSpeedStrategy: SnakeSpeedStrategy {

    var snake: Snake?

    var maxSpeed: Double?

    private let diff = 1.1

    func increaseSpeedByEatingApple() {
        guard let snake = snake else { return }
        snake.moveSpeed *= diff
        if let maxSpeed = maxSpeed {
            if snake.moveSpeed > maxSpeed {
                snake.moveSpeed = maxSpeed
            }
        }
    }
}
```

Здесь **diff** — это коэффициент, во сколько раз будет увеличена скорость при каждом вызове метода.

В **GameViewController** добавим выбор стратегии:

```
private var snakeSpeedStrategy: SnakeSpeedStrategy {
    switch self.difficulty {
    case .easy, .medium:
        return NotIncreaseSnakeSpeedStrategy()
    case .hard:
        let strategy = ArithmeticProgressionSnakeSpeedStrategy()
        strategy.maxSpeed = 350.0
        return strategy
    case .insane:
        return GeometricProgressionSnakeSpeedStrategy()
    }
}
```

Во **viewDidLoad** передадим стратегию в инициализатор сцены:

```
let scene = GameScene(size: view.bounds.size,
                      createApplesStrategy: self.createAppleStrategy,
                      snakeSpeedStrategy: self.snakeSpeedStrategy)
```

Перейдем в **GameScene**, добавим еще одно свойство и изменим инициализатор:

```
private let createApplesStrategy: CreateApplesStrategy
fileprivate let snakeSpeedStrategy: SnakeSpeedStrategy

init(size: CGSize,
     createApplesStrategy: CreateApplesStrategy,
     snakeSpeedStrategy: SnakeSpeedStrategy) {
    self.createApplesStrategy = createApplesStrategy
    self.snakeSpeedStrategy = snakeSpeedStrategy
    super.init(size: size)
}
```

В классе **GameScene** после создания объекта **Snake** надо не забыть присвоить его стратегии. Затем добавим увеличение скорости при поедании яблока:

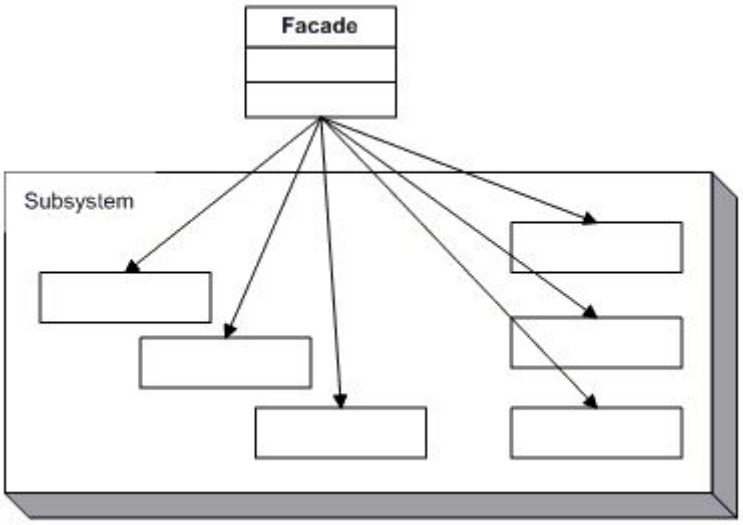
```
private func headDidCollideApple(apple: SKNode?) {
    //добавляем к змее еще одну секцию
    snake?.addBodyPart()
    //удаляем яблоко
    apple?.removeFromParent()
    self.apple = nil
    //создаем новое яблоко
    createApple()

    self.snakeSpeedStrategy.increaseSpeedByEatingApple()
}
```

Мы реализовали две стратегии. Огромный плюс этого паттерна в том, что реализация методов скрыта протоколом, и объект, использующий стратегию, знает только о протоколе. Это позволяет при работе приложения менять одну стратегию на другую при изменении условий (в нашем случае — при изменении сложности игры). Запустим проект и убедимся, что обе стратегии работают.

Паттерн Facade

Паттерн Facade (фасад) — структурный шаблон проектирования. Он упрощает сложную систему, предоставляя простой интерфейс.



Пример. Представьте, что в вашем проекте на одну цель работает много сложных классов, вызовов функций и т. д. Например, требуется загрузить конфиденциальный документ в виде файла на телефон, а этот файл еще и зашифрован для безопасности. Потребуется сходить в сеть и скачать файл (за это отвечает один класс), расшифровать его специальными алгоритмами (это делают другие несколько классов), преобразовать в данные и сохранить на диск (этим занимается **caretaker** из паттерна **Memento**). При этом на любом этапе может возникнуть ошибка, которую нужно обработать. Это сложно, хотелось бы просто вызвать функцию **downloadFile()** — и все. Решение есть: создать отдельный класс «фасад», который и будет скрывать порядок обращения к другим классам (скачивание, расшифровка, сохранение, обработку ошибок), а наружу вынести только простой метод **downloadFile()**.

У этого паттерна нет строгих предписаний о том, как реализовывать фасад, какие у него должны быть методы. К каждой задаче фасад делается так, чтобы максимально удобно структурировать сложную работу нескольких систем. Поэтому сразу перейдем к реализации паттерна в проекте.

Реализация в проекте

Только что мы задавали стратегии скорости змейки и появления яблок, влияющие на сложность игры, и уже получается сложновато. Фасад будет идеальным решением, чтобы скрыть стратегии. Фасад должен, зная сложность игры, создавать яблоки, управлять скоростью змеи и делать все то, что мы еще можем добавить в игру, чтобы варьировать сложность.

Итак, создадим файл **DifficultySettingsFacade.swift** и добавим туда следующий код:

```
import UIKit
import SpriteKit

final class DifficultySettingsFacade {

    let difficulty: Difficulty

    weak var snake: Snake? {
        didSet {
            snakeSpeedStrategy.snake = snake
        }
    }
```

```
    // MARK: - Strategies

    private lazy var createApplesStrategy: CreateApplesStrategy = {
        switch self.difficulty {
        case .easy:
            return SequentialCreateOneAppleStrategy()
        case .medium, .hard, .insane:
            return RandomCreateOneAppleStrategy()
        }
    }()

    private lazy var snakeSpeedStrategy: SnakeSpeedStrategy = {
        switch self.difficulty {
        case .easy, .medium:
            return NotIncreaseSnakeSpeedStrategy()
        case .hard:
            let strategy = ArithmeticProgressionSnakeSpeedStrategy()
            strategy.maxSpeed = 350.0
            return strategy
        case .insane:
            return GeometricProgressionSnakeSpeedStrategy()
        }
    }()

    // MARK: - Init

    init(difficulty: Difficulty) {
        self.difficulty = difficulty
    }

    // MARK: - Methods

    func createApples(in scene: SKScene) -> [Apple] {
        return createApplesStrategy.createApples(in: scene.frame)
    }

    func increaseSnakeSpeed() {
        self.snakeSpeedStrategy.increaseSpeedByEatingApple()
    }
}
```

Фасад инициализируется сложностью игры. Чтобы фасад работал со змейкой, ей необходимо передать экземпляр змеи. И затем мы просто вызываем функции **createApples** и **increaseSnakeSpeed**, больше не задумываясь о том, какую стратегию надо выбрать для определенной сложности.

Из **GameScene** убираем стратегии и делаем инициализацию с вариантом сложности:

```
fileprivate let difficultyFacade: DifficultySettingsFacade

init(size: CGSize, difficulty: Difficulty) {
    self.difficultyFacade = DifficultySettingsFacade(difficulty: difficulty)
    super.init(size: size)
}
```

При столкновении с яблоком теперь обращаемся к фасаду:

```
private func headDidCollideApple(apple: SKNode?) {
    //добавляем к змее еще одну секцию
    snake?.addBodyPart()
    //удаляем яблоко
    apple?.removeFromParent()
    self.apple = nil
    //создаем новое яблоко
    createApple()

    self.difficultyFacade.increaseSnakeSpeed()
}
```

И для создания яблока тоже обращаемся к фасаду:

```
fileprivate func createApple() {
    let apple = self.difficultyFacade.createApples(in: self).first!
    self.apple = apple
    self.addChild(apple)
}
```

Из **GameViewController** осталось убрать свойства, рассчитывающие выбор стратегий, и инициализировать сцену со сложностью:

```
let scene = GameScene(size: view.bounds.size, difficulty: self.difficulty)
```

Мы упростили интерфейс и грамотно структурировали всё, что связано со сложностью игры, создав один дополнительный объект — фасад.

Паттерн Observer

Паттерн **Observer** (наблюдатель) — поведенческий шаблон проектирования. Он используется, когда одни объекты должны узнавать об изменениях состояния других.

Есть четыре основных способа применить этот паттерн:

1. **NotificationCenter**. Это самый простой, но наименее желательный способ реализации. Дело в том, что он нарушает абстракцию, усложняет код для чтения и поддержки. Частично вы с ним должны быть уже знакомы из предыдущих курсов.

2. **KVO (key-value observing).** Это механизм для реализации **observer**, пришедший из **Objective-C**. **KVO** использует **objc-runtime**, поэтому нам нужно помечать классы и свойства маркером **@objc**, а классы наследовать от **NSObject**. При разработке на чистом **Swift** используется крайне редко (да и на Objective-C, как правило, для специфических задач).

3. **RxSwift.** Это стороннаяя библиотека (https://github.com/ReactiveX/RxSwift), многими командами она используется в продакшене. Если вы проходили курс «Современные средства iOS-разработчика», то должны быть вкратце знакомы с этим инструментом. Его изучение и использование выходит за рамки нашего курса.

4. **Реализация своей обертки для обсервинга (наблюдения) свойств.** Если не хочется тянуть библиотеку **RxSwift** или вы с командой разработки решили, что не будете ее использовать (а первые два способа, как правило, не рассматриваются вообще из за их минусов), то это отличный способ реализации паттерна. Рассмотрим именно его и применим в проекте.

Мы применим этот паттерн для того, чтобы отображать на экране игры актуальную скорость змейки (которая может теперь меняться).

Пример в Playground

Суть применения паттерна с помощью четвертого способа в том, чтобы свойство класса делать, например, не типа **Double**, а **Observable<Double>**, где **Observable** — класс-обертка.

Чтобы сэкономить время, мы не будем писать ее с нуля, а просто скопируем готовое решение. Добавим в плейграунд следующий код:

```
public struct ObservableOptions: OptionSet, CustomStringConvertible {

    public static let initial = ObservableOptions(rawValue: 1 << 0)
    public static let old = ObservableOptions(rawValue: 1 << 1)
    public static let new = ObservableOptions(rawValue: 1 << 2)

    public var rawValue: Int

    public init(rawValue: Int) {
        self.rawValue = rawValue
    }

    public var description: String {
        switch self {
        case .initial:
            return "initial"
        case .old:
            return "old"
        case .new:
            return "new"
        default:
            return "ObservableOptions(rawValue: \(rawValue))"
        }
    }
```

```
}

public class Observable<Type> {

    fileprivate class Callback {
        fileprivate weak var observer: AnyObject?
        fileprivate let options: [ObservableOptions]
        fileprivate let closure: (Type, ObservableOptions) -> Void

        fileprivate init(observer: AnyObject,
                         options: [ObservableOptions],
                         closure: @escaping (Type, ObservableOptions) -> Void) {
            self.observer = observer
            self.options = options
            self.closure = closure
        }
    }

    // MARK: - Properties
    public var value: Type {
        didSet {
            removeNilObserverCallbacks()
            notifyCallbacks(value: oldValue, option: .old)
            notifyCallbacks(value: value, option: .new)
        }
    }

    // MARK: - Object Lifecycle
    public init(_ value: Type) {
        self.value = value
    }

    // MARK: - Managing Observers
    private var callbacks: [Callback] = []

    public func addObserver(_ observer: AnyObject,
                            removeIfExists: Bool = true,
                            options: [ObservableOptions] = [.new],
                            closure: @escaping (Type, ObservableOptions) ->
Void) {
        if removeIfExists {
            removeObserver(observer)
        }

        let callback = Callback(observer: observer,
                                options: options,
                                closure: closure)
        callbacks.append(callback)

        if options.contains(.initial) {
            closure(value, .initial)
        }
    }
```

```
    public func removeObserver(_ observer: AnyObject) {
        callbacks = callbacks.filter { $0.observer !== observer }
    }

    // MARK: - Private

    private func removeNilObserverCallbacks() {
        callbacks = callbacks.filter { $0.observer != nil }
    }

    private func notifyCallbacks(value: Type,
                                 option: ObservableOptions) {
        let callbacksToNotify = callbacks.filter {
            $0.options.contains(option)
        }
        callbacksToNotify.forEach { $0.closure(value, option) }
    }
}
```

Класс **Observable** позволяет наблюдать за изменениями его значения, подписываться на них и удалять наблюдателей. Не будем вдаваться в его реализацию. Важно то, что можно вызвать функцию **addObserver**, чтобы подписаться на изменения этого класса.

Применим эту обертку для реализации паттерна. В плейграунде создадим класс **User**, и его имя сделаем observable-свойством, то есть свойством, за изменением которого можно наблюдать:

```
class User {
    public let name: Observable<String>
    public init(name: String) {
        self.name = Observable(name)
    }
}
```

Нам нужен объект, который будет наблюдать за изменениями имени юзера. Это может быть любой объект. В данном случае создадим пустой класс **Observer** и один его экземпляр:

```
class Observer { }
let observer = Observer()
```

Далее создадим юзера и подпишемся обсервером на изменения его имени:

```
let user = User(name: "name1")
user.name.addObserver(observer, options: [.initial, .new, .old]) { name, change
in
    print("name changed. change = \(change), name = \(name)")
}
user.name.value = "name2"
```

При изменении имени в консоль будет выводиться информация об этом изменении. Сначала юзер инициализируется с именем **name1**, потом оно меняется на **name2**. Вот что будет выведено в консоль:

```
name changed. change = initial, name = name1
name changed. change = old, name = name1
name changed. change = new, name = name2
```

Реализация в проекте

В проекте «Змейка» будем выводить текущую скорость змеи на экран. Для этого воспользуемся оберткой **Observable**, рассмотренной только что в плейграунде.

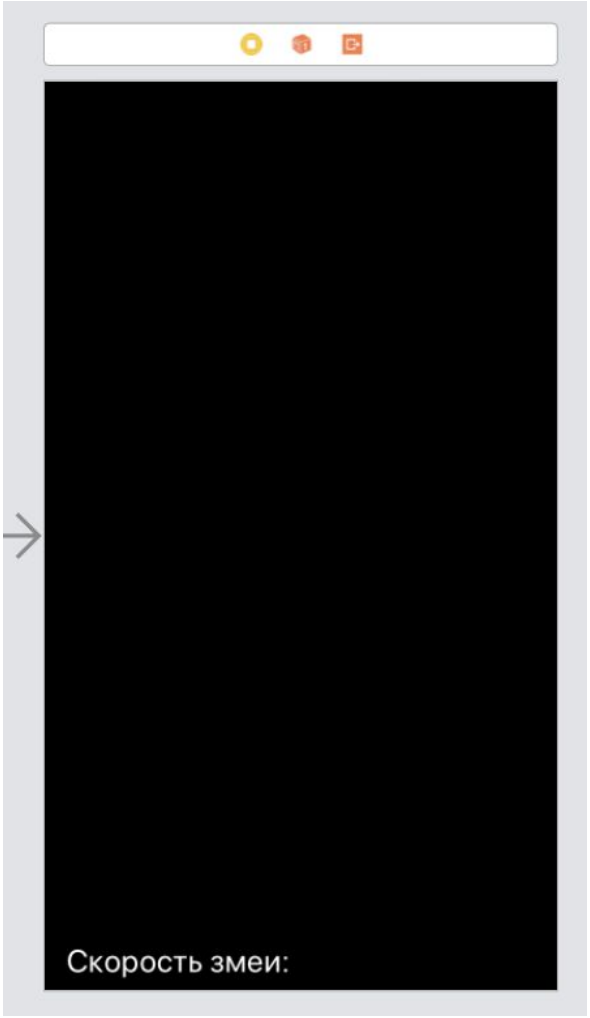
Создадим файл **Observable.swift** и скопируем в него весь код обертки (структура **ObsevableOptions** и класс **Observable**).

Зайдем в файл **Snake.swift** и изменим тип переменной **moveSpeed** с **Double** на **Observable<Double>**:

```
/// Скорость перемещения
var moveSpeed = Observable<Double>(125.0)
```

После этого компилятор будет ругаться, что свойство **moveSpeed** не является числом. Везде, где оно использовалось, нужно **moveSpeed** заменить на **moveSpeed.value**. Теперь мы готовы наблюдать за этим свойством.

Добавим лейбл на вью-контроллер игры.



И создадим соответствующий **IBOutlet** у класса **GameViewController**:

```
@IBOutlet weak var speedLabel: UILabel!
```

В методе **viewDidLoad** создается сцена. Сцена, в свою очередь, создает змейку и хранит ее. Мы можем обратиться к этой змейке, чтобы наблюдать за ее observable-свойством. Добавим этот код в конец метода **viewDidLoad** перед закрывающей скобкой:

```
scene.snake?.moveSpeed.addObserver(self, options: [.new, .initial], closure: {
[weak self] (moveSpeed, _) in
    self?.speedLabel.text = "Скорость змеи: \(moveSpeed)"
})
```

Все будет работать. Запустим проект и убедимся, что на экране выводится правильная скорость змеи — для легкого и среднего уровня сложности она будет постоянная, для сложного и безумного будет увеличиваться с каждым съеденным яблоком.

Паттерн Builder

Паттерн **Builder** (строитель) — порождающий шаблон проектирования. Он применяется, когда сложный процесс создания объекта можно разбить на шаги, чтобы упростить.

Пример в Playground

Как пример рассмотрим приготовление бургера. Нужны будут мясо и хлеб, а также другие ингредиенты (добавки):

```
enum Ingredient {
    case egg
    case tomato
    case chili
    case potato
    case mustard
    case vasabi
    case cheese
    case onion
    case salad
}
enum Meat {
    case chicken
    case beef
    case pork
    case vegeterianTofu
}

enum Bread {
    case wheat
    case rye
}
```

Также в бургер можно добавить соусы, которые готовят из ингредиентов:

```
struct Sauce {
    var ingredients: [Ingredient] = []
}
```

Бургер, который содержит все выше перечисленное:

```
struct Hamburger {
    var meat: Meat
    var bread: Bread
    var sauces: [Sauce]
    var additionalIngredients: [Ingredient]
}
```

Создадим классический бургер с двумя соусами, салатом, сыром и луком:

```
let classicBurger = Hamburger(meat: .beef, bread: .wheat, sauces:
[Sauce(ingredients: [.egg, .mustard]), Sauce(ingredients: [.tomato])],
additionalIngredients: [.salad, .onion, .cheese])
```

Инициализация объекта выглядит сложно. А еще представьте, что бургер создается пошагово: например, один объект (условно — работник кухни) добавляет в него мясо, другой (соус-шеф) создает композицию соусов и так далее. В этом случае инициализация объекта еще более осложняется, и нужно придумывать решение. Такой задаче идеально подходит паттерн **builder** — объект (бургер) строится пошагово, инициализация разделяется на несколько шагов и за счет этого упрощается.

Создадим класс **HamburgerBuilder**:

```
class HamburgerBuilder {
    private(set) var meat: Meat = .beef
    private(set) var bread: Bread = .wheat
    private(set) var sauces: [Sauce] = []
    private(set) var additionalIngredients: [Ingredient] = []

    func build() -> Hamburger {
        return Hamburger(meat: meat, bread: bread, sauces: sauces,
additionalIngredients: additionalIngredients)
    }

    func setMeat(_ meat: Meat) {
        self.meat = meat
    }

    func setBread(_ bread: Bread) {
        self.bread = bread
    }

    func addSauce(_ sauce: Sauce) {
        sauces.append(sauce)
    }

    func addAdditionalIngredient(_ ingredient: Ingredient) {
        additionalIngredients.append(ingredient)
    }
}
```

Настройка свойств для бургера происходит в отдельных методах. Когда всё настроено, нужно вызвать функцию **build()**, которая вернет готовый объект бургера.

Итак, реализуем теперь создание различных бургеров. Пусть за это отвечает объект **KitchenEmployee** (работник кухни):

```
class KitchenEmployee {

    func createClassicBurger() -> Hamburger {
        let builder = HamburgerBuilder()
        builder.setMeat(.beef)
        builder.setBread(.wheat)
        builder.addSauce(Sauce(ingredients: [.egg, .mustard]))
        builder.addSauce(Sauce(ingredients: [.tomato]))
        builder.addAdditionalIngredient(.salad)
        builder.addAdditionalIngredient(.onion)
        builder.addAdditionalIngredient(.cheese)
        return builder.build()
    }

    func createChickenChiliSpecial() -> Hamburger {
        let builder = HamburgerBuilder()
        builder.setMeat(.chicken)
        builder.setBread(.rye)
        builder.addSauce(Sauce(ingredients: [.chili, .tomato]))
        builder.addAdditionalIngredient(.vasabi)
        builder.addAdditionalIngredient(.potato)
        return builder.build()
    }
}
```

Сейчас мы решили задачу и создали не самый простой объект. Важно, что при различной настройке параметров у нас получались разные по смыслу объекты.

Но не стоит использовать паттерн **builder** везде, лишь чтобы избавиться от длинного инициализатора, заменив его вызовами функций **builder.setProperty(...)**. С созданием объектов в большинстве случаев прекрасно справляется инициализатор (а также другой порождающий паттерн — «фабрика», который мы изучим на следующем уроке). Повторим: builder стоит использовать, если создание объекта занимает несколько шагов, и нужно иметь несколько входных точек для конфигурации объекта. Этот паттерн строит объект постепенно, как бы по кирпичикам. А если объект должен быть проинициализирован мгновенно, **builder** использовать не стоит.

Мы не будем применять **builder** к проекту «Змейка» как раз по этой причине — сейчас там нет сложных объектов, которые нуждаются в пошаговой инициализации. Всегда оценивайте целесообразность применения паттерна к вашей задаче!

Практическое задание

1. Ранее в вашей игре вопросы должны были идти последовательно, каждую новую игру в одном и том же порядке. Теперь сделайте два варианта —когда вопросы идут последовательно и когда перемешиваются в случайном порядке при каждой игре. Используйте паттерн **strategy**.

2. Добавьте экран настроек игры (для перехода в него в меню должна быть отдельная кнопка). В настройках сделайте возможность выбрать, в каком порядке идут вопросы — в случайном или последовательном. Выбранный вариант должен сохраниться в синглтоне **Game** и использоваться в самой игре.

3. Добавьте на экран игры лейбл, в котором будет указан номер текущего вопроса и сколько процентов от общего числа вопросов уже получили правильный ответ. С помощью **observer** (используйте обертку **Observable<Type>**; если сложно, воспользуйтесь **NotificationCenter**), следите за этими данными (в вашей архитектуре они должны быть в объекте **GameSession**) и отображайте на экране в лейбле.

4. Реализуйте в приложении возможность самому добавить новый вопрос. На экран меню поместите еще одну кнопку — «Добавить вопрос». При нажатии на нее открывается новый экран с формой ввода вопроса. Пользователь заполняет поля, нажимает «Добавить», и этот вопрос добавляется в игру в дополнение к уже существующим (дополнительно: если делали подсказки в модели вопроса, то их поведение сгенерируйте случайным образом — то есть юзер не должен задавать эти поля). Используя паттерн **memento**, добавьте еще один **Caretaker** для того, чтобы сохранять созданные юзером вопросы на диск и использовать их при последующих запусках приложения.

5. * **Задание на паттерн Builder.** Экран добавления вопроса реализуйте в виде вью-контроллера с **table view**. Форма заполнения вопроса — это ячейка **UITableView** с текстовыми полями. Внизу **table view** добавьте кнопку в виде плюса, при нажатии на которую к **table view** добавится еще одна ячейка с формой ввода вопроса. Кнопка «Добавить» должна добавлять в приложение не один вопрос, а массив вопросов. Создание массива новых вопросов из введенных на экране данных реализуйте с помощью **builder**. Билдер при вызове функции **build()** должен вернуть именно массив вопросов. Сам билдер можно хранить и использовать прямо во время работы программы — когда юзер что-то ввел, у билдера вызвали соответствующий метод.

6. ** **Задание на паттерн Facade**. Реализуйте возможность использовать подсказки «Звонок другу», «Помощь зала» и «50/50», если еще не сделали этого (в скобках с пометкой «дополнительно» в предыдущих заданиях были описаны шаги по созданию этой функциональности). В самой игре юзер при ответе на вопрос может нажать на одну из трех кнопок с подсказками. При этом нужно обратиться к модели текущего вопроса и вызвать у него одно из поведений при подсказке. Закройте доступ к этим подсказкам фасадом, применив паттерн **facade**. У **GameSession** добавьте свойство с фасадом **HintUsageFacade**. Этот фасад должен хранить свойство «текущий вопрос», и у него должны быть методы **callFriend()**, **useAuditoryHelp()**, **use50to50Hint()**. Подумайте, как лучше архитектурно реализовать использование этого фасада и как он будет передавать результат вызова метода наружу, во вью-контроллер. Это задание с двумя звездами — здесь вам нужно самостоятельно решить архитектурную задачу!

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».