GeekBrains

Архитектуры и шаблоны проектирования на Swift

Введение. Базовые паттерны. Часть 1

Паттерны delegate, singleton, memento.

Оглавление

Введение в паттерны проектирования

Что такое паттерны и зачем они нужны

Проектирование приложения — это один из самых трудозатратных и трудоемких этапов при создании программного продукта. Решается главный вопрос — как сделать приложение правильно и быстро, чтобы потом не возникло трудностей. Большинство задач, которые встают перед разработчиками, не уникальны — значит, есть готовые решения. Удачные находки можно использовать многократно. Проектировщик-эксперт способен как разрабатывать собственные решения, так и уместно применять известные и хорошо зарекомендовавшие себя подходы — паттерны проектирования.

Их мы и будем изучать на курсе. Сконцентрируемся на тех паттернах, которые широко применяются в разработке приложений под iOS. С некоторыми из них разработчики сталкиваются чуть ли не каждый день, другие решают специфические задачи.

Чтобы понять, для чего нужны и почему важны паттерны, проведем аналогию. Представьте, что вы хотите сделать новый автомобиль, но никогда этим не занимались. Какой он будет формы? Сколько у него будет колес? Как водитель будет управлять им? Сейчас вы представили, как четырехколесное средство передвижения обтекаемой формы управляется сидящим водителем с помощью руля. Почему именно так? Потому что практикой использования уже было выяснено, что так лучше всего — удобнее, экономнее, надежнее. Именно по такому принципу применяются паттерны в ООП — и вы не столкнетесь с ними в разработке до тех пор, пока вам не потребуется «сделать автомобиль». Но иногда случается так, что вы создаете «трицикл» с рулем на крыше, — и только потом, набив несколько шишек на неудобном «монстре», узнаете, что существует паттерн «автомобиль». И он значительно упростил бы вам жизнь, если бы вы знали про него раньше.

Паттерны проектирования не привязаны к конкретному языку программирования. В некоторых областях находят более широкое применение одни паттерны и почти не используются остальные, в других направлениях картина может отличаться. Во многих современных языках программирования предусмотрены упрощенные синтаксические конструкции, позволяющие легко использовать некоторые паттерны. В Swift немало такого синтаксического сахара.

Виды паттернов

Паттерны проектирования классифицируются по области применения. Выделяют следующие категории:

1. **Порождающие**. В основе этих паттернов всегда лежит создание объектов.
2. **Структурные**. Позволяют лучше структурировать код, чтобы его было легче понимать.
3. **Поведенческие**. Самая большая категория. Сюда входят все паттерны, которые так или иначе определяют поведение объектов и их взаимодействие друг с другом.

Отдельная категория — **архитектурные** паттерны. Это шаблоны организации кода во всей программе или ее значительной части. Они определяют структуру программы в целом, в то время как дизайн-паттерны решают специальную задачу внутри выбранной архитектуры.

Существуют также **антипаттерны** — примеры того, как не надо писать код. Но они зачастую не так четко определены, и среди программистов часто возникают споры, что считать антипаттерном, а что нет. Мы рассмотрим их кратко на курсе.

Как будем изучать паттерны

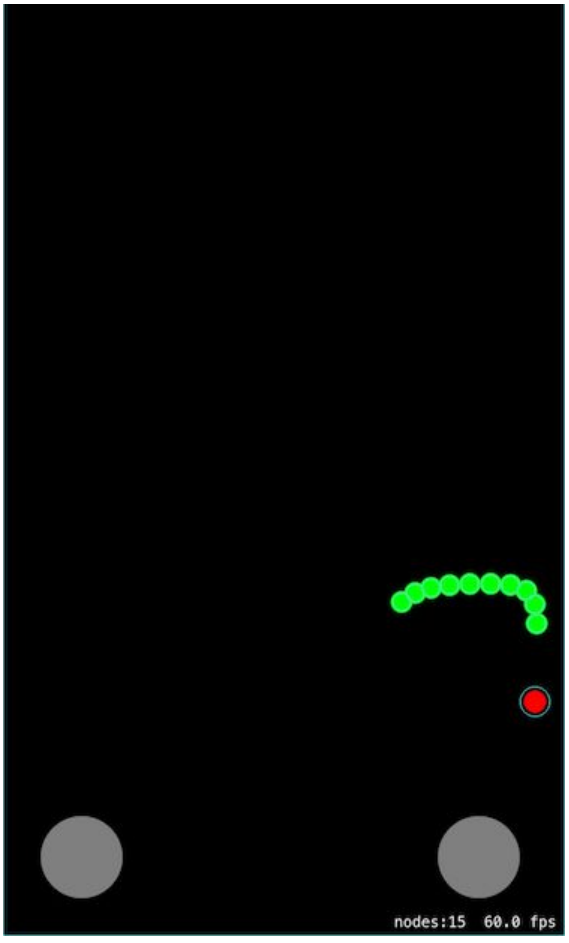
С первого по шестой уроки будем изучать порождающие, структурные и поведенческие шаблоны, которые чаще всего встречаются в iOS-разработке: от самых общих и часто используемых до наиболее специализированных и востребованных далеко не в каждом проекте. На шестом уроке мы также поговорим об антипаттернах. На седьмом и восьмом уроках погрузимся в тему архитектурных паттернов и изучим 4 самых востребованных и известных архитектуры — **MVC**, **MVP**, **MVVM**, **VIPER**.

Чтобы понять, как применяются паттерны, будем задействовать их в боевых условиях — то есть в проектах, приближенных к реальным. Но писать их с нуля, конечно, не будем. Вместо этого возьмем заготовленные проекты и в них применим паттерны, решая задачи, которые могут возникнуть в работе. Поэтому к каждому уроку следует приходить подготовленными — это будет частью самостоятельного практического задания: знакомиться с кодом проекта, который мы будем использовать на уроке. Не нужно вникать в каждую строчку — достаточно в общих чертах знать, что делает приложение, какие есть экраны, классы, структуры и как они используются.

Знакомство с проектом змейка

На первом уроке изучим три паттерна — **delegate**, **singleton**, **memento**.

Для этого воспользуемся проектом известной игры «Змейка». Ученики пишут ее в конце курса по основам Swift, и этот проект мы будем использовать на первых двух уроках. Здесь кодовая база простая, и мы по ней сейчас пройдемся. А наибольшую часть кода, которая использует SpriteKit (фреймворк для создания 2D-игр под устройства Apple), затрагивать не будем.



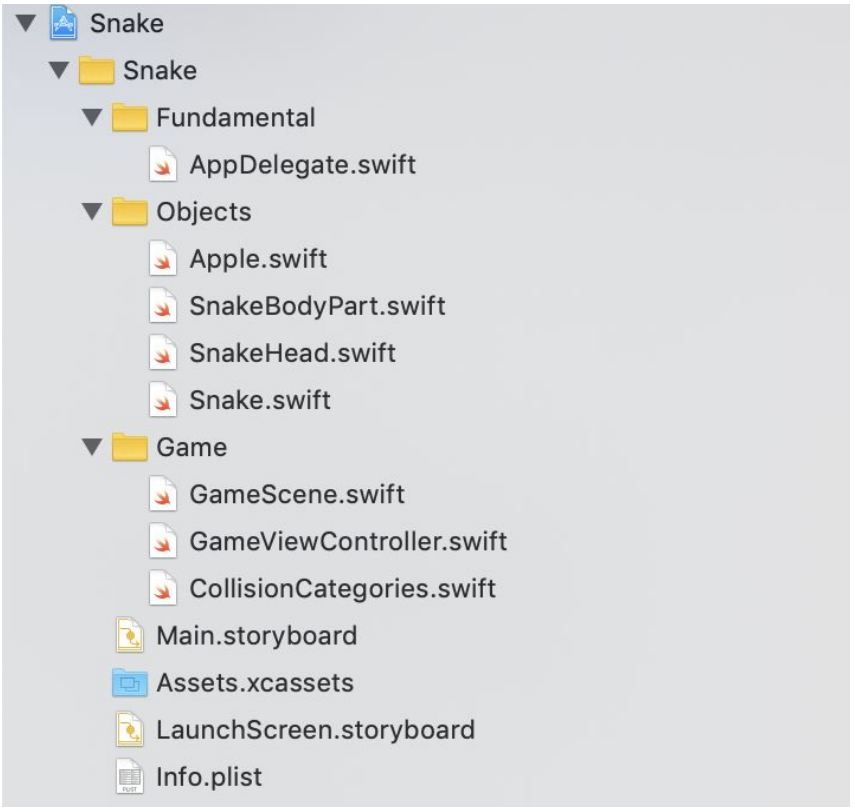
Стартовый шаблон проекта можно скачать по ссылке: https://drive.google.com/open?id=1GkQFcLu98SYA1Wts_zTICjQBmtwgg3Un.

Открываем и запускаем проект. На черном фоне двигается зеленая змея, поедающая красные яблоки. Поворот змеи выполняется с помощью кнопок внизу экрана. Реализация передвижения содержит логику, по которой следующая (начиная с головы) часть змейки постоянно движется к предыдущей, — поэтому змея поворачивает плавно, без острых углов. У игры сейчас всего один экран.

Рассмотрим структуру кода, над которым нам предстоит работать.

На следующем скриншоте вы видите структуру проекта. Пробежимся по ней:

- **AppDelegate** — точка входа в приложение;
- **Apple** — объект яблока, которое добавляется на экран;
- **SnakeBodyPart** — объект, описывающий часть тела змейки (то есть одну зеленую точку);
- **SnakeHead** — наследник **SnakeBodyPart**, описывает голову змеи. Это особенная часть тела: именно она испытывает столкновения и поедает яблоки, поэтому для нее предусмотрен отдельный класс;
- **Snake** — объект, описывающий всю змею целиком;
- **GameScene** — содержит логику игры;
- **GameViewController** — соответствует единственному экрану, на котором происходит игра;
- **CollisionCategories** — структура для описания видов столкновений игровых объектов (змейки со стеной, змейки с яблоком, змейки с самой собой).



Пройдемся по наиболее важным частям кода. Зайдем в класс **GameScene**. Функция **headDidCollideApple(apple: SKNode?)** обрабатывает столкновение головы змеи и яблока — змейка вырастает еще на одну часть, съеденное яблоко удаляется, затем генерируется новое.

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

Вот функция создания нового яблока:

```
fileprivate func createApple() {
    guard let view = self.view, let scene = view.scene else { return }
    let randX = CGFloat(arc4random_uniform(UInt32(scene.frame.maxX - 5)) + 1)
    let randY = CGFloat(arc4random_uniform(UInt32(scene.frame.maxY - 5)) + 1)
    let apple = Apple(position: CGPoint(x: randX, y: randY))
    self.apple = apple
    self.addChild(apple)
}
```

В ней случайно генерируется точка в пределах экрана, в которую яблоко будет помещено, а затем оно добавляется на экран.

Обрабатывается также столкновение змеи со стеной:

```
private func headDidCollideWall(_ contact: SKPhysicsContact) {
    self.restartGame()
}
```

При столкновении змейки со стеной игра перезапускается — удаляются все элементы, затем добавляется новая змея и генерируется новое яблоко. В общем, все начинается с начала.

В классе **Snake**, описывающем змею целиком, обратите внимание на два свойства — **moveSpeed** и **body**:

```
class Snake: SKShapeNode {

    /// Скорость перемещения
    var moveSpeed = 125.0

    /// Массив где хранятся части змеи
    var body = [SnakeBodyPart]()

    ...
}
```

Паттерн Delegate

Один из самых часто применяемых паттернов в iOS разработке. Суть его в том, что один объект для выполнения определенных действий передает управление другому объекту - делегирует ему некоторую часть обязанностей.

Примеры в iOS SDK

В **UIKit** из **iOS SDK** паттерн **delegate** используется очень часто.

- Элемент **UITextField** — это текстовое поле для пользовательского ввода. У него есть свойство **delegate** типа **UITextFieldDelegate?**. Например, метод **textFieldDidBeginEditing** этого делегата вызывается, когда юзер тапнул на поле для ввода текста. Так что делегат может обработать это событие, если необходимо. А метод **textFieldShouldClear(_ textField: UITextField) -> Bool**, который тоже реализуется делегатом, возвращает **Bool**, означающий, нужно ли очистить поле для ввода при нажатии на крестик. Так элемент делегирует часть обязанностей другому объекту — в первом примере для обработки события, во втором — для определения правильного поведения.

- У **UITableView** и **UICollectionView** два делегата. Один — это непосредственно свойство **delegate**, второй — свойство **dataSource**. Название второго просто более точно отражает, для чего используется этот делегат, но по сути это тот же паттерн.

Реализация в проекте

В iOS-разработке паттерн **delegate** часто применяют, чтобы прокинуть данные с одного экрана на другой. Например, мы находимся на экране, который отображает дату рождения; затем переходим на экран, где их можно изменить, делаем это и возвращаемся — данные на первом экране должны обновиться.

Добавим в проект «змейки» экран с главным меню — именно с него будет стартовать приложение. На этом экране будет кнопка «Играть», открывающая экран с игрой. При проигрыше игра должна возвращаться в главное меню, при этом должен будет отобразиться результат последнего раунда (количество съеденных яблок). Откуда возьмем данные, если экрана игры уже нет? Прокинем данные с него с помощью делегата.

Результат игры известен в игровой сцене **GameScene**, потому что она хранит экземпляр змейки. Также именно в **GameScene** обрабатывается столкновение со стеной. Нам нужно оповестить об окончании игры вью-контроллер **GameViewController**, чтобы он мог уйти с экрана. В **GameScene.swift** добавим протокол для этого:

```
protocol GameSceneDelegate: class {
    func didEndGame(withResult result: Int)
}
```

Классу **GameScene** добавим свойство, которое будет хранить ссылку на делегат:

```
weak var gameDelegate: GameSceneDelegate?
```

В функции **headDidCollideWall(_ contact: SKPhysicsContact)** оповестим делегат о том, что игра окончена:

```
private func headDidCollideWall(_ contact: SKPhysicsContact) {
    guard let snake = self.snake else { return }
    self.gameDelegate?.didEndGame(withResult: snake.body.count - 1)
}
```

Перейдем в **GameViewController.swift**. Он будет делегатом **GameScene**, чтобы ловить оповещение об окончании игры. Создавая сцену, зададим ей делегат:

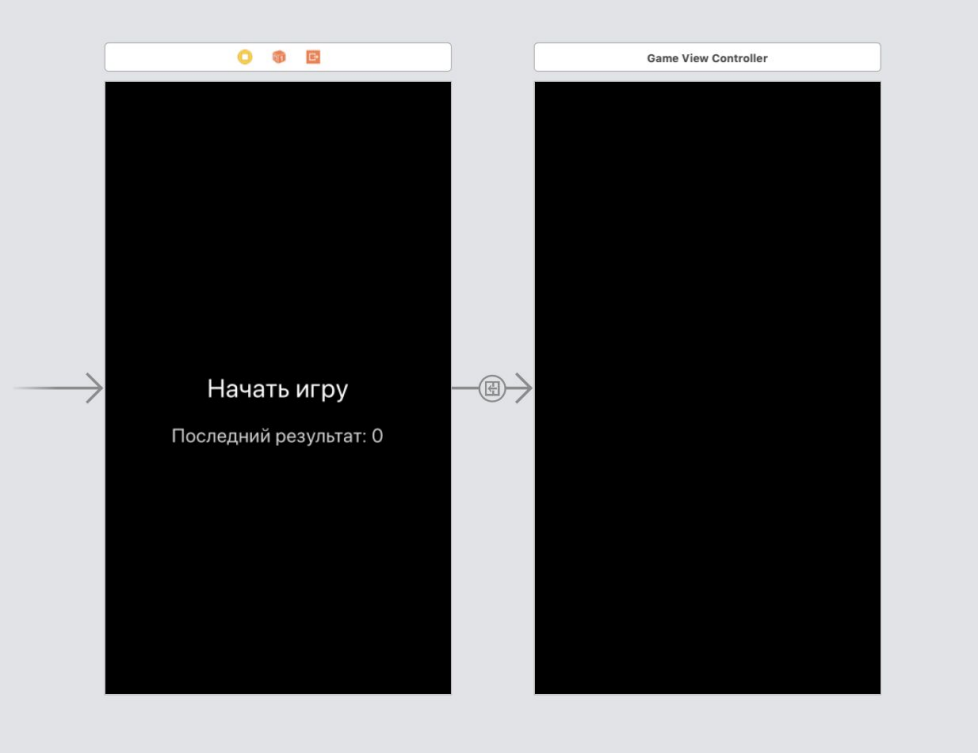
```
override func viewDidLoad() {
    super.viewDidLoad()
    let scene = GameScene(size: view.bounds.size)
    scene.gameDelegate = self
    ...
}
```

И реализуем протокол:

```
extension GameViewController: GameSceneDelegate {
    func didEndGame(withResult result: Int) {
        self.dismiss(animated: true, completion: nil)
    }
}
```

Нужно добавить экран главного меню, куда и будет уходить приложение при окончании игры и вызове метода **self.dismiss(animated: true, completion: nil)** выше.

Перейдем в **Main.storyboard** и добавим экран главного меню. Сделаем его точкой входа. Кнопка «Начать игру» с помощью **segue** ведет на экран с игрой. Также добавим лейбл, отображающий результат последней игры (см. скриншот).



Теперь столкновение со стеной завершает игру и возвращает нас в главное меню.

Теперь задача состоит в том, чтобы отобразить в главном меню результат игры. Для этого добавим еще один протокол в файле **GameViewController.swift**:

```
protocol GameViewControllerDelegate: class {
    func didEndGame(withResult result: Int)
}
```

В класс **GameViewController** добавим свойство:

```
weak var delegate: GameViewControllerDelegate?
```

По окончании игры будем не только закрывать экран, но и дальше прокидывать результат уже делегату вью-контроллера:

```
extension GameViewController: GameSceneDelegate {
    func didEndGame(withResult result: Int) {
        self.delegate?.didEndGame(withResult: result)
        self.dismiss(animated: true, completion: nil)
    }
}
```

Добавим класс **MainMenuViewController**, соответствующий вью-контроллеру главного меню. Из сториборда надо расставить **IBOutlet**'ы, а также присвоить идентификатор **segue**, например **startGameSegue**. Вью-контроллер главного меню будет выглядеть в коде следующим образом:

```
final class MainMenuViewController: UIViewController {
    @IBOutlet weak var startGameButton: UIButton!
    @IBOutlet weak var lastResultLabel: UILabel!

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        switch segue.identifier {
        case "startGameSegue":
            guard let destination = segue.destination as? GameViewController
else { return }
            destination.delegate = self
        default:
            break
        }
    }
}

extension MainMenuViewController: GameViewControllerDelegate {
    func didEndGame(withResult result: Int) {
        self.lastResultLabel.text = "Последний результат: \(result)"
    }
}
```

Будет происходить следующее: **GameScene** при столкновении с игрой будет делегировать обработку окончания игры **GameViewController**'у и передавать ему результат игры. **GameViewController** выполняет свою часть функциональности — закрывает себя и передает управление дальше **MainMenuViewController**'у. Он использует результат игры для того, чтобы отобразить его на UI.

Использование замыканий вместо делегатов

Вы могли заметить, что использование паттерна **delegate** увеличивает количество сущностей (как минимум нужно создать протокол, его реализацию и хранить **delegate** свойством). Иногда это не оправдано. Swift позволяет писать в функциональном стиле, что во многих случаях улучшает читаемость кода и уменьшает его количество. Можно поступить следующим образом: вместо обращения к **delegate** вызывать замыкание, которое будет задаваться другим объектом извне. Перейдем в код в файле **GameScene.swift**. Удалим протокол **GameSceneDelegate** и свойство **weak var gameDelegate: GameSceneDelegate?**. Вместо него добавим другое свойство:

```
var onGameEnd: ((Int) -> Void)?
```

Заменим реализацию функции **headDidCollideWall** на следующую:

```
private func headDidCollideWall(_ contact: SKPhysicsContact) {
    guard let snake = self.snake else { return }
    self.onGameEnd?(snake.body.count - 1)
}
```

Если замыкание есть, то оно будет вызвано и в него передастся результат игры.

Далее в **GameViewController.swift** вместо строчки **scene.gameDelegate = self** и реализации протокола **GameSceneDelegate** (их надо удалить) напишем следующее (в **viewDidLoad**, где раньше устанавливался делегат):

```
scene.onGameEnd = { [weak self] result in
    self?.onGameEnd?(result)
    self?.dismiss(animated: true, completion: nil)
}
```

Именно это замыкание отработает при столкновении змейки со стеной. Не забывайте писать **[weak self]**, так как замыкания захватывают сильной ссылкой объекты, которые в них передаются.

Аналогичным образом можно удалить протокол **GameViewControllerDelegate** и все, что с ним связано, и заменить на замыкание (код в **MainMenuViewController**):

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

Количество сущностей уменьшилось, читаемость кода не ухудшилась. Во многих случаях все же используются делегаты, особенно когда протоколы абстрагируют множество методов и свойств (как те же **UITableViewDelegate** и **UITableViewDataSource**). И это во многом вопрос стиля кода, так что в разработке используются и тот и другой подход.

Паттерн Singleton

Singleton — это паттерн, который предписывает определенному классу иметь только один экземпляр и предоставляет глобальный доступ к этому экземпляру.

Большой плюс этого паттерна — простота использования. Во многих случаях он может существенно упростить логику взаимодействия между объектами, когда по смыслу задачи объект определенного класса может быть только один в программе. Например, когда в приложении есть объект пользователя, хранящий данные о нем. Кажется логичным, что этот объект будет синглтоном, так как приложение единовременно использует только один юзер. Соответственно, доступ к его данным будет осуществляться максимально просто из любого места кода.

Но этот паттерн обладает двумя существенными минусами:

1. Необдуманное использование синглтонов может привести к тому, что проект станет невозможно масштабировать. А с его ростом может потребоваться поддерживать в приложении несколько юзеров одновременно. Тогда придется избавляться от синглтона, который уже успели использовать по всему приложению, и такая задача рефакторинга вполне может затянуться на долгий срок или даже что-то сломать.

2. Паттерн затрудняет юнит-тестирование. Нельзя создать мок объекта (фейковый объект, имитирующий поведение настоящего, — часто используется для модульных тестов), так как паттерн предписывает обращение к объекту только через синглтон.

Из-за этих двух минусов в больших системах часто намеренно избегают этого паттерна (и даже считают антипаттерном). Но он не плох и не хорош, как и все паттерны, а только обладает своими плюсами и минусами и подходит для конкретных задач.

Пример в Playground

Откроем **Playground** в XCode. Создадим класс **User**, пусть у него будет имя и дата рождения.

```
class User {
    var name: String?
    var birthDate: Date?
}
```

Допустим, по физическому смыслу объект-юзер в приложении должен быть только один. К этому объекту может обратиться любая часть программы, которой доступна сущность **User**, и нельзя создать более одного юзера. В Objective-C эта логика писалась вручную: для класса создавался геттер, который инициализировал объект, если его еще нет, и возвращал уже существующий объект, если он ранее был создан. Инициализация оборачивалась в конструкцию, обеспечивающую потокобезопасность (чтобы не было ситуации, когда объект инициализируется с двух разных потоков и в итоге получается два разных объекта). В Swift все стало намного проще. Чтобы создать синглтон, надо написать одну строчку внутри класса **User**:

```
static let shared = User()
```

Теперь к синглтону можно обращаться из любого места программы:

```
let user = User.shared
print(user.name ?? "noname")
```

Но все еще можно инициализировать объект извне. Чтобы запретить такую возможность, следует сделать инициализатор приватным. Класс **User**, реализованный по паттерну Singleton, будет выглядеть так:

```
class User {
    static let shared = User()

    var name: String?
    var birthDate: Date?

    private init() { }
}
```

Как это можно использовать? Теперь в одном месте программы, где собираются данные юзера (например, пользователь вводит данные в форму, или данные достаются из базы, или приходят с сервера), можно установить свойства **name** и **birthDate**:

```
User.shared.name = "some name"
```

А в другом месте программы можно взять эти данные и использовать, чтобы, например, отобразить их на экране:

```
self.nameLabel.text = User.shared.name
```

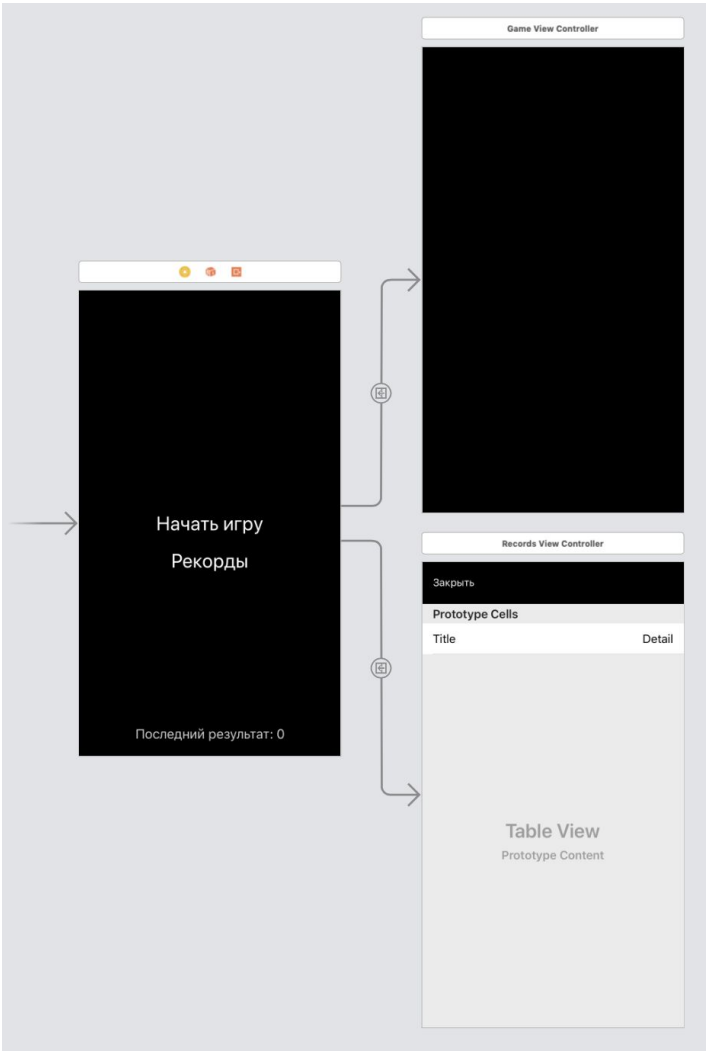
Примеры в iOS SDK

Как и делегат, синглтон часто используется в нативных библиотеках Apple.

- **UIApplication.shared** предоставляет доступ к экземпляру приложения. Этот объект всегда один в приложении (это очевидно, поэтому тут применение синглтона абсолютно оправдано).
- У приложения есть делегат, и он всегда один. Это как раз тот класс, который всегда по дефолту добавляется под именем **AppDelegate** при создании нового проекта в XCode. Обратиться к нему можно с помощью **UIApplication.shared.delegate**.
- **FileManager.default** предоставляет доступ к объекту-синглтону, с помощью которого можно управлять файловой системой. Удалить системные файлы не получится, но можно, например, сохранить данные на диск, создать файлы и папки внутри sandbox-приложения.
- **UserDefault.standard** — объект-синглтон хранилища данных, которое можно использовать для сохранения простых и небольших данных на диск.

Реализация в проекте

Стоит новая задача: все набранные очки в ходе серий игр в «змейку» отображать в отдельном экране рекордов. Экраны будут выглядеть примерно так:



Для экрана рекордов создаем **RecordsViewController**, который содержит **UITableView** и кнопку «Закрыть». В подробности реализации UI вдаваться не будем, тем более что на применение паттернов это сейчас не влияет.

Для модели игрового рекорда создадим структуру:

```
struct Record {
    let date: Date
    let score: Int
}
```

Где-то эти рекорды нужно хранить. Можно сказать, что рекорды распространяются на всю игру — это массив, который может изменяться. С новыми играми будут добавляться новые рекорды, но все же этот массив один на все приложение. Похоже на синглтон.

Напомним, что использование паттерна Singleton в больших проектах должно быть хорошо обдумано. Следует оценить, насколько синглтон повлияет на масштабируемость и тестируемость, и при необходимости воспользоваться более подходящим решением.

А в нашей игре полезно будет создать синглтон, соответствующей ей в целом. В этом объекте как раз можно будет хранить массив рекордов, и в любом месте иметь к ним удобный доступ.

```
final class Game {

    static let shared = Game()

    private(set) var records: [Record] = []

    private init() { }

    func addRecord(_ record: Record) {
        self.records.append(record)
    }

    func clearRecords() {
        self.records = []
    }
}
```

Теперь, в момент соприкосновения змейки со стеной, мы будем не просто завершать игру, но и добавлять рекорд:

```
private func headDidCollideWall(_ contact: SKPhysicsContact) {
    guard let snake = self.snake else { return }
    let score = snake.body.count - 1
    let record = Record(date: Date(), score: score)
    Game.shared.addRecord(record)
    self.onGameEnd?(score)
}
```

А на экран рекордов необходимо подтягивать данные из синглтона **Game.shared**. В нашем случае рекорды отображаются в **table view** — ее **data source** можно реализовать в **RecordsViewController** следующим образом:

```
extension RecordsViewController: UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
        return Game.shared.records.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "RecordCell",
for: indexPath)
        let record = Game.shared.records[indexPath.row]
        let dateFormatter = DateFormatter()
        dateFormatter.dateStyle = .short
        cell.textLabel?.text = dateFormatter.string(from: record.date)
        cell.detailTextLabel?.text = "\(record.score)"
        return cell
    }
}
```

Паттерн Memento

Этот паттерн (в переводе — хранитель) используется для того, чтобы при работе с объектом была возможность его сохранить и впоследствии восстановить. Чаще всего нужно сохранять состояние приложения в определенный момент — например, чтобы после завершения программы и повторного открытия начать с предыдущего состояния. Если в программе есть фича **undo/redo** (отменить изменения и вернуть изменения), то можно использовать массив хранителей, чтобы восстанавливать все предыдущие состояния.

Пример в Playground

В паттерне **Memento** есть три участника:

```
// 1. Originator — объект, который требуется сохранить и впоследствии
восстановить.
// 2. Memento — сохраненные данные.
// При сохранении Originator эндкодится в Memento, при восстановлении Memento
декодится в Originator.
// 3. Caretaker — объект, который занимается сохранением и восстановлением.
```

Будем сохранять и восстанавливать состояние игры, в которой есть количество здоровья и очки. Это будет класс **Game**. Делаем его **Codable**, чтобы можно было энкодить и декодить его в **Data** из коробки:

```
// 1. Originator
class Game: Codable {
    var score = 0
    var health = 100

    func getDamage() {
        health -= 10
    }

    func killMonster() {
        score += 10
    }
}
```

Сохранять объект будем в сырые данные, то есть в **Data**:

```
// 2. Memento
typealias Memento = Data
```

Главное — реализовать сохранение и восстановление игры. Этим должен заниматься объект **GameCaretaker**:

```
// 3. Caretaker
class GameCaretaker {
    private let decoder = JSONDecoder()
    private let encoder = JSONEncoder()
    private let key = "game"

    func saveGame(_ game: Game) throws {
        let data: Memento = try encoder.encode(game)
        UserDefaults.standard.set(data, forKey: key)
    }

    func loadGame() throws -> Game {
        guard let data = UserDefaults.standard.value(forKey: key) as? Memento
            , let game = try? decoder.decode(Game.self, from: data) else {
                throw Error.gameNotFound
        }
        return game
    }

    public enum Error: Swift.Error {
        case gameNotFound
    }
}
```

При сохранении игры он использует **JSONEncoder** и сохраняет полученные данные в **UserDefaults**. Для загрузки он берет данные из **UserDefaults** и с помощью **JSONDecoder** превращает их в объект игры. Если что-то не получилось, то выбрасываем из функции ошибку. Теперь посмотрим, как это будет работать. Создадим игру и заложим в ней несколько действий:

```
let game = Game()
game.killMonster()
game.getDamage()
game.getDamage()
game.killMonster()
game.killMonster()
print(game.health)
print(game.score)
```

Печатается 80 здоровья и 30 очков.

Теперь создадим хранитель и сохраним игру. Затем восстановим ее:

```
let caretaker = GameCaretaker()
try caretaker.saveGame(game)
let restoredGame = try caretaker.loadGame()
print(restoredGame.health)
print(restoredGame.score)
```

Печатаются те же 80 здоровья и 30 очков, но это уже другой объект, восстановленный из сохраненного состояния.

Реализация в проекте

Сейчас при перезапуске игры «змейка» все полученные рекорды стираются, так как они не записываются на диск. Исправим это с помощью паттерна **Memento**. Будем хранить массив рекордов **[Record]** в виде **Data** и восстанавливать их при перезапуске.

Сначала изменим структуру **Record**, которую мы собираемся сохранять. Она должна поддерживать **Codable**:

```
struct Record: Codable {

    let date: Date
    let value: Int
}
```

Поскольку типы всех полей этой структуры из коробки поддерживают **Codable**, то и сама она теперь будет это поддерживать — ничего дополнительно писать не надо.

Создадим файл **RecordsCaretaker.swift** и запишем в него следующее:

```
final class RecordsCaretaker {

    private let encoder = JSONEncoder()
    private let decoder = JSONDecoder()

    private let key = "records"

    func save(records: [Record]) {
        do {
            let data = try self.encoder.encode(records)
            UserDefaults.standard.set(data, forKey: key)
        } catch {
            print(error)
        }
    }

    func retrieveRecords() -> [Record] {
        guard let data = UserDefaults.standard.data(forKey: key) else {
            return []
        }
        do {
            return try self.decoder.decode([Record].self, from: data)
        } catch {
            print(error)
            return []
        }
    }
}
```

Это практически полностью повторяет то, что мы сделали в плейграунде, только в случае ошибки мы возвращаем пустой массив (чтобы было проще). Конечно, лучше не использовать **UserDefaults** — это самый элементарный вариант. Часто в проекте у вас используется база данных, например **CoreData** или **Realm**. Или можно сохранять данные на диск с помощью **FileManager**. И преобразовывать объекты в данные необязательно через **JSONDecoder** / **JSONEncoder**, но это этот вариант используют чаще всего.

Осталось воспользоваться этой функциональностью. Зайдем в **Game.swift** и добавим классу **Game** приватное свойство:

```
private let recordsCaretaker = RecordsCaretaker()
```

Изменим приватный инициализатор — мы хотим, чтобы при инициализации синглтона **Game** сразу подтягивались рекорды, сохраненные на диске:

```
private init() {
    self.records = self.recordsCaretaker.retrieveRecords()
}
```

Чтобы каждый раз, когда записывается или удаляется рекорд, на диске было соответствующее сохранение, добавим к свойству **records** это:

```
private(set) var records: [Record] {
    didSet {
        recordsCaretaker.save(records: self.records)
    }
}
```

Теперь каждый раз при изменении массива рекордов на диск будет сохраняться новое состояние. Можно проверить, что при выгрузке приложения и повторной загрузке все будет подтягиваться и рекорды никуда не денутся.

Практическое задание

1. В качестве практического задания к урокам 1 и 2 вы будете создавать игру в стиле «Кто хочет стать миллионером». **Не нужно** использовать SpriteKit. Все экраны делайте в сториборде с обычными UI-компонентами (UIButton, UILabel и другими). Придумайте или нагуглите вопросы к игре (достаточно 5–10). Создайте структуру **Question**, содержащую сам вопрос, варианты ответа и правильный ответ.

   (* Дополнительно: можете добавить в структуру поля, отображающие, что будет, если взять подсказку «Звонок другу» или «Помощь зала».)

   Добавьте в проект эти вопросы.

2. Создайте экраны главного меню и игры. На экране главного меню должны быть кнопки «Играть» и «Результаты».

3. При нажатии на кнопку «Играть» должен открываться экран, на котором юзеру предлагается ответить на вопрос, выбрав ответ из четырех вариантов. При правильном ответе должен появляться следующий вопрос, при неправильном игра заканчивается. Верстку экрана с игрой придумайте самостоятельно.

   (* Дополнительно: можете добавить в игру возможность использовать подсказку — «Звонок другу», «Помощь зала», «Убрать два неправильных ответа».)

4. Создайте синглтон **Game** и класс **GameSession** (не синглтон). У **Game** создайте опциональное свойство типа **GameSession**. Когда нажимается кнопка «Играть» и начинается игра, создавайте новый **GameSession** и передайте его синглтону **Game**. В **GameSession** храните всю информацию о ходе текущей игры: сколько вопросов было решено правильно, сколько их всего (дополнительно: какие доступные подсказки остались; сколько юзер выиграет денег, если закончит игру). Всю эту информацию **GameSession** должен получать через **delegate** от **GameViewController** — вью-контроллера, на котором происходит игра и нажимаются кнопки ответов.

5. Когда игра заканчивается, в синглтоне **Game** посчитайте результат — какой процент от общего числа вопросов получили правильные ответы. Сохраните этот результат в синглтоне (подсказка: нужно отдельное свойство, массив результатов игры) и обнулите у него свойство **GameSession**, так как игра была завершена.

6. Используя паттерн **Memento**, сохраняйте результаты на диск, как это было сделано на уроке.

Дополнительные материалы

1. Design Patterns on iOS using Swift — Part 1/2.
2. Design Patterns on iOS using Swift — Part 2/2.

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»