GeekBrains

Архитектуры и шаблоны проектирования на Swift

Продвинутые паттерны. Часть 1

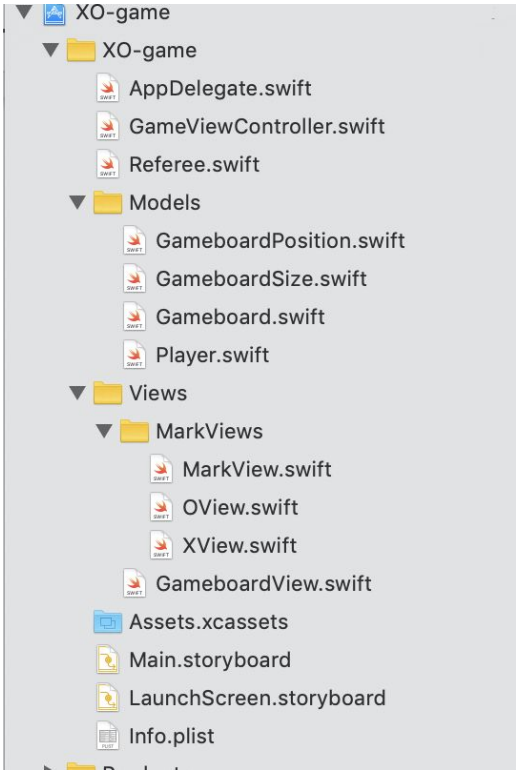
Паттерны state, prototype, command.

Оглавление

Проект

На этом уроке будем писать игру крестики-нолики. Ее прототип доступен по ссылке: https://drive.google.com/open?id=1veNXF8lsVapSvWVHLCgRcNSlQ0-RYRmB.

Что сейчас есть в проекте? Взглянем на иерархию файлов:



- **GameViewController** — единственный вью-контроллер в игре — экран, на котором происходит все действие. Он содержит IBOutlet'ы всех UI-элементов.
- **Referee** — класс, который умеет определять победителя игры в крестики-нолики. Пока он нигде не применяется. Мы воспользуемся им позднее.
- **GameboardPosition**, **GameboardSize** — простые модели, определяющие размер и позицию на игровой доске.
- **Gameboard** — класс, объект которого будет хранить текущее состояние игровой доски: в каких позициях выставлены крестики и нолики.
- **Player** — модель игрока. Это простая энамка с двумя вариантами, так как мы будем поддерживать в игре двух игроков.
- **MarkView** — базовый класс view крестика и view нолика. Не содержит логики.
- **OView** — view нолика. Будем добавлять эту view на доску для отображения. Наследник MarkView.
- **XView** — view крестика. Будем добавлять ее на доску для отображения. Наследник **MarkView**.
- **GameboardView** — view с доской, содержит отображение игрового поля с разделителями. Не содержит логики. На нее можно добавлять **MarkView**.

Сейчас игра фактически не содержит логики — она только отображает игровое поле. В **GameViewController** сейчас есть код, который выставляет крестики в соответствующие ячейки поля при нажатии на них. Этот код находится там просто для примера.

Перейдем к изучению первого паттерна. Он понадобится нам для того, чтобы организовать пошаговое выполнение игры — сначала ходит первый игрок, потом второй и так далее. В итоге будет определяться победитель.

Паттерн State

Паттерн **State** (состояние) — поведенческий шаблон проектирования. По его названию понятно, что этот паттерн работает с состояниями объекта. Их может быть несколько, и в зависимости от них будет реализовываться поведение.

Сразу попробуем осознать этот паттерн в нашем примере с игрой в крестики-нолики. Сначала ходит первый игрок — при нажатии на ячейку игрового поля в ней должен появиться крестик. Состояние игры — ход первого игрока. После него игра переходит в следующее состояние — ход второго игрока, и на поле будет располагаться уже нолик. Также будет отличаться сама логика игры — в первом состоянии ходы будут записываться первому игроку, во втором — второму. Когда игра закончена (либо один из игроков собрал выигрышную комбинацию, либо все ячейки оказались заняты), она переходит в последнее состояние — состояние выигрыша (либо ничьей). Обратите внимание, что в разных состояниях приложение делает похожие действия: обрабатывает нажатие на ячейку, выводит информацию. Но реализации этих действий отличаются: в зависимости от состояния отображается разная информация, и нажатие обрабатывается по-своему. В этом суть паттерна **State**.

Можно сказать, что паттерн **State** похож на **Strategy**. Вспомните структуру паттерна «стратегия». У всех стратегий есть общий интерфейс — то есть общий набор методов и свойств, объявленных в протоколе. И есть несколько конкретных стратегий, которые имеют одни и те же методы, но с разной реализацией. В паттерне **State** все то же самое — все состояния имеют общий интерфейс (либо протокол, либо базовый класс), но разную реализацию. Отличие состоит в том, что состояние хранится приложением, и обязательно произойдет переход из одного состояния в другое. Также в стратегии, как правило, наша цель — не вдаваясь в реализацию, выполнить одно действие и получить результат. Состояние же влияет на множество компонентов — в нашем примере и на UI-приложения, и на логику работы.

Пример в проекте

Применим паттерн State. Нам будут нужны три состояния — ход первого игрока, ход второго игрока и «игра окончена». Начнем с общего интерфейса. В отдельном файле **GameState.swift** создадим протокол **GameState**:

```
public protocol GameState {

    var isCompleted: Bool { get }

    func begin()

    func addMark(at position: GameboardPosition)
}
```

Свойство **isCompleted** должно сказать, «выполнено» ли состояние, то есть можно ли приложению перейти к следующему состоянию. Функция **begin()** должна реализовать состояние — правильно настроить внешний вид и логику (подготовиться). Функция **addMark(at position: GameboardPosition)** должна добавлять крестик либо нолик на игровое поле.

*Важно: написанные нами свойства и функции в протоколе не являются общими для паттерна **State**. Мы просто написали тот интерфейс, который будет нам удобен для управления состояниями.*

Теперь напишем конкретное состояние — состояние хода игрока. Добавим класс **PlayerInputState**, который будет поддерживать протокол **GameState**. Тут будет довольно много кода, дальше мы его разберем:

```
public class PlayerInputState: GameState {

    public private(set) var isCompleted = false

    public let player: Player
    private(set) weak var gameViewController: GameViewController?
    private(set) weak var gameboard: Gameboard?
    private(set) weak var gameboardView: GameboardView?

    init(player: Player, gameViewController: GameViewController, gameboard:
Gameboard, gameboardView: GameboardView) {
        self.player = player
        self.gameViewController = gameViewController
        self.gameboard = gameboard
        self.gameboardView = gameboardView
    }

    public func begin() {
        switch self.player {
        case .first:
            self.gameViewController?.firstPlayerTurnLabel.isHidden = false
            self.gameViewController?.secondPlayerTurnLabel.isHidden = true
        case .second:
            self.gameViewController?.firstPlayerTurnLabel.isHidden = true
            self.gameViewController?.secondPlayerTurnLabel.isHidden = false
        }
        self.gameViewController?.winnerLabel.isHidden = true
    }

    public func addMark(at position: GameboardPosition) {
        guard let gameboardView = self.gameboardView
            , gameboardView.canPlaceMarkView(at: position)
            else { return }

        let markView: MarkView
        switch self.player {
        case .first:
            markView = XView()
        case .second:
            markView = OView()
        }
        self.gameboard?.setPlayer(self.player, at: position)
        self.gameboardView?.placeMarkView(markView, at: position)
        self.isCompleted = true
    }
}
```

Рассмотрим код по шагам:

1. По умолчанию **isCompleted = false**. То есть при инициализации состояния оно еще не завершено, что логично. Состояние будет завершаться после удачной установки крестика или нолика.

2. Мы хотим, чтобы состояние внутри себя настраивало UI приложения. Для этого состоянию **PlayerInputState** передаем следующие объекты:

    a. **gameViewController: GameViewController?**

    b. **gameboardView: GameboardView?**

    Обращаясь к ним, состояние будет изменять UI экрана и добавлять на поле крестики и нолики.

3. Также состоянию передается объект **gameboard: Gameboard?**, чтобы не забывать и про логику игры. Ведь этот объект будет запоминать позиции размещенных отметок.

4. Делаем их все **weak**, поскольку состояние не должно хранить другие объекты приложения. Как раз наоборот, объекты приложения (в нашем случае — вью-контроллер) хранят состояние.

5. И передаем в состояние номер игрока. Он будет храниться в свойстве **let player: Player**.

6. Поговорим про реализацию общего протокола **GameState**. Функция **begin()** настраивает **UI**. Если это ход первого игрока, то скрываются лейблы, относящиеся ко второму, и наоборот. Также скрывается лейбл, который должен показать победителя игры.

7. Функция **addMark(at position: GameboardPosition)** выставляет соответствующую отметку на игровом поле — крестик для первого игрока и нолик для второго. Также реализация этой функции добавляет соответствующую пометку о ходе игрока в класс **Gameboard**. После этого состояние переходит в режим готовности.

Самое время воспользоваться кодом этого состояния. Перейдем в **GameViewController.swift**. Добавим следующие свойства классу **GameViewController**:

```
private let gameboard = Gameboard()
private var currentState: GameState! {
    didSet {
        self.currentState.begin()
    }
}
```

Как уже упоминалось, объект **gameboard** будет хранить текущее размещение меток на игровом поле. Главное, что мы добавили свойство **currentState** — «текущее состояние». Мы сделали это свойство **force unwrapped** (то есть с !), потому что хотим проинициализировать его начальным состоянием во **viewDidLoad**. Также мы указали, что, когда будет устанавливать новое состояние, у него сразу будет вызываться **begin()**.

Далее добавим следующий код в функцию **viewDidLoad**:

```
override public func viewDidLoad() {
    super.viewDidLoad()
    self.goToFirstState()

    gameboardView.onSelectPosition = { [weak self] position in
        guard let self = self else { return }
        self.currentState.addMark(at: position)
        if self.currentState.isCompleted {
            self.goToNextState()
        }
    }
}
```

Вызовом **self.goToFirstState()** устанавливаем начальное состояние. Код этой функции будет далее. Также добавляем обработку нажатия юзера на ячейку поля. По такому нажатию у текущего состояния вызываем **addMark**, и далее, если состояние «завершено», переходим к следующему.

Приватные функции **goToFirstState()** и **goToNextState()**:

```
private func goToFirstState() {
    self.currentState = PlayerInputState(player: .first,
                                         gameViewController: self,
                                         gameboard: gameboard,
                                         gameboardView: gameboardView)
}

private func goToNextState() {
    if let playerInputState = currentState as? PlayerInputState {
        self.currentState = PlayerInputState(player:
playerInputState.player.next,
                                             gameViewController: self,
                                             gameboard: gameboard,
                                             gameboardView: gameboardView)
    }
}
```

Итак, установка первого состояния — это установка **PlayerInputState** с первым игроком. Переход к следующему состоянию — это установка **PlayerInputState** со следующим игроком (если был первый, то будет второй, и наоборот).

Теперь игра будет работать, состояния будут переключаться. Чего мы уже добились с помощью паттерна **State**? Во-первых, вынесли много кода из вью-контроллера в объект состояния. Решение о том, как обрабатывать начало нового состояния и размещение отметки, целиком лежит на состоянии (точнее на конкретной его реализации). Мы написали код, который позволяет легко добавить сколько угодно новых состояний. Можно, например, добавить состояние третьего, четвертого игрока, состояние паузы игры, какое-нибудь интересное состояние врага, который будет вредить игрокам. Главное, что этот код легко изменяем.

Кстати, можно было бы для первого и второго игрока сделать два разных состояния — **FirstPlayerState** и **SecondPlayerState**. Мы решили объединить их в одни, просто чтобы меньше дублировать код. Но вариант с двумя отдельными реализациями тоже возможен.

Идем дальше. Пока в нашей игре нельзя выиграть или проиграть. Исправим это. Добавим новое состояние в отдельный файл **GameEndedState.swift**:

```
public class GameEndedState: GameState {

    public let isCompleted = false

    public let winner: Player?
    private(set) weak var gameViewController: GameViewController?

    public init(winner: Player?, gameViewController: GameViewController) {
        self.winner = winner
        self.gameViewController = gameViewController
    }

    public func begin() {
        self.gameViewController?.winnerLabel.isHidden = false
        if let winner = winner {
            self.gameViewController?.winnerLabel.text = self.winnerName(from:
winner) + " win"
        } else {
            self.gameViewController?.winnerLabel.text = "No winner"
        }
        self.gameViewController?.firstPlayerTurnLabel.isHidden = true
        self.gameViewController?.secondPlayerTurnLabel.isHidden = true
    }

    public func addMark(at position: GameboardPosition) { }

    private func winnerName(from winner: Player) -> String {
        switch winner {
        case .first: return "1st player"
        case .second: return "2nd player"
        }
    }
}
```

Реализация этого класса состояния довольно проста. Добавление метки не должно происходить, поэтому мы оставили функцию **addMark(at position: GameboardPosition)** пустой. Когда состояние приходит в действие (то есть извне вызывается функция **begin()**), то состояние **GameEndedState** настраивает UI в зависимости от того, какой игрок выиграл, или же была ничья. Это состояние никогда не перейдет в **isCompleted = true**, так как по смыслу это последнее, завершающее состояние в игре, после которого можно только начать все заново. Но это только пока, ведь паттерн **State** позволяет легко изменить это — придумать новое состояние, которое идет после того, как определился победитель (но мы такое не придумали).

Чтобы воспользоваться новым состоянием, перейдем обратно в **GameViewController** и добавим ему свойство:

```
private lazy var referee = Referee(gameboard: self.gameboard)
```

В функции **goToNextState()** будем определять победителя. То есть каждый раз, когда ход может перейти другому игроку, мы проверим, не закончилась ли игра уже. Изменим эту функцию следующим образом:

```
private func goToNextState() {
    if let winner = self.referee.determineWinner() {
        self.currentState = GameEndedState(winner: winner, gameViewController:
self)
        return
    }
    if let playerInputState = currentState as? PlayerInputState {
        self.currentState = PlayerInputState(player:
playerInputState.player.next,
                                             gameViewController: self,
                                             gameboard: gameboard,
                                             gameboardView: gameboardView)
    }
}
```

Теперь у нас автоматически переключаются состояния приложения — из хода первого игрока в ход второго, потом снова первого — и так до тех пор, пока кто-то из них не победит и крестики-нолики перейдут в состояние конца игры. Правда, пока есть один минус — если все поля доски заполнены, но никто не победил, игра не переходит в состояние «конец игры, ничья». Это легко исправить — подумайте самостоятельно, как это сделать.

Паттерн Prototype

Паттерн **Prototype** (прототип) — порождающий шаблон проектирования. Его суть в том, чтобы создавать объект не через инициализатор (как обычно это делается), а путем полного копирования уже существующего объекта. При этом должен быть предоставлен максимально удобный интерфейс для такого копирования.

Пример в Playground

Зачем может понадобиться полное копирование объекта? Для примера рассмотрим класс **Monster**:

```
class Monster {
    var health: Int
    var damage: Int

    init(health: Int, damage: Int) {
        self.health = health
        self.damage = damage
    }
}

func hitMonster(_ monster: Monster, damage: Int) {
    monster.health -= damage
}
```

Создадим одного монстра:

```
let monster = Monster(health: 10, damage: 10)
```

Есть обстоятельства, при которых нужно создать точно такого же монстра. Например, в игре он появляется каждые 10 секунд, или мы просто хотим создать армию клонов (у Саурона была большая армия орков — возможно, он просто разумно использовал паттерн «прототип»).

Monster — это класс, и если написать такой код:

```
let monster2 = monster
```

… то переменная **monster2** будет содержать ссылку на того же монстра. Это легко проверяется:

```
hitMonster(monster, damage: 5)
print(monster.health)
print(monster2.health)
```

Урон 5 был нанесен монстру с переменной **monster**, но вывод в плейграунде покажет, что и у **monster**, и у **monster2** здоровье окажется на уровне 5. Данное поведение объясняется тем, что объекты классов копируются по ссылке (**reference type**).

Но мы отвлеклись от главного — как скопировать существующего монстра, но при этом получить новый объект? В Objective-C для этой цели активно использовался протокол **NSCopying**. Им же можно воспользоваться и в Swift. Добавим классу монстра поддержку этого протокола:

```
class Monster: NSCopying {
    var health: Int
    var damage: Int

    init(health: Int, damage: Int) {
        self.health = health
        self.damage = damage
    }

    func copy(with zone: NSZone? = nil) -> Any {
        return Monster(health: self.health, damage: self.damage)
    }
}
```

Для поддержки протокола необходимо было реализовать функцию **copy(with zone: NSZone? = nil) -> Any.** Обсудим ее чуть позже. Пока важно, что можно применить **copy()** у объекта монстра и получить нового с такими же характеристиками. Сделаем это:

```
let monster = Monster(health: 10, damage: 10)
let monster2 = monster.copy() as! Monster

hitMonster(monster, damage: 5)
print(monster.health)
print(monster2.health)
```

Теперь вывод в консоль покажет здоровье первого монстра — 5, а второго — 10, как мы и хотели. Потому что при вызове **monster.copy()** произошло полное копирование и создался новый объект. Но наследие Objective-C дает о себе знать:

1. **copy(with zone: NSZone? = nil)** возвращает **Any**. Поэтому при копировании пришлось принудительно приводить скопированный объект к нужному типу: **as! Monster**.

2. Мы применили **monster.copy()**, не передав параметр **zone**, потому что по умолчанию он **nil**. В принципе, так и нужно делать, однако все еще остается возможность передать **NSZone**. Это тоже класс objective-c. Раньше он использовался для низкоуровневого управления памятью освобождаемых объектов. С появлением ARC и тем более в языке Swift не нужно пользоваться этим объектом. Поэтому для нас это ничего не значащий параметр, засоряющий код.

Исправить эти минусы поможет собственная реализация протокола **Copying** (да, нативного решения на чистом Swift, к сожалению, нет).

Создадим протокол **Copying**, требованием которого будет обеспечить инициализатор, принимающий на вход объект такого же типа:

```
protocol Copying {
    init(_ prototype: Self)
}
```

Мы хотим получить простой интерфейс для копирования объектов. Поэтому в этом же протоколе объявим функцию **copy()** и сразу же дадим ей дефолтную реализацию:

```
extension Copying {
    func copy() -> Self {
        return type(of: self).init(self)
    }
}
```

Вызывая **copy()** на объекте, поддерживающем **Copying**, будет создавать новый объект через инициализатор **init(_ prototype: Self)**. Это удобно — нужно только написать реализацию этого инициализатора. Сделаем это для класса **Monster**. Удалим поддержку **NSCopying** и добавим ее для нашего нового протокола **Copying**:

```
class Monster: Copying {
    var health: Int
    var damage: Int

    init(health: Int, damage: Int) {
        self.health = health
        self.damage = damage
    }

    required init(_ prototype: Monster) {
        self.health = prototype.health
        self.damage = prototype.damage
    }
}
let monster = Monster(health: 10, damage: 10)
let monster2 = monster.copy()
```

Теперь компилятор понимает, что **monster2** имеет тип **Monster**. Все остальное будет работать точно так же, монстр будет успешно копироваться. Зато мы избавились от недостатков протокола **NSCopying**.

Пример в проекте

В игре есть отличное место для паттерна «прототип». Когда на поле ставится крестик или нолик, для этого значка каждый раз создается новая view. Вместо этого можно состоянию **PlayerInputState** при инициализации передать прототип view, которая будет копироваться, если нужно разместить новую метку на поле.

Сначала добавим в проект протокол **Copying** (реализацию писали выше).

Затем реализуем этот протокол у класса **MarkView**:

```
public class MarkView: UIView, Copying {
    ...
    required init(_ prototype: MarkView) {
        super.init(frame: prototype.frame)
        self.lineColor = prototype.lineColor
        self.lineWidth = prototype.lineWidth
        self.textColor = prototype.textColor
    }
}
```

Теперь зайдем в класс **PlayerInputState**. Добавим ему свойство:

```
public let markViewPrototype: MarkView
```

И будем передавать его в инициализаторе:

```
init(player: Player, markViewPrototype: MarkView, gameViewController:
GameViewController, gameboard: Gameboard, gameboardView: GameboardView) {
    self.player = player
    self.markViewPrototype = markViewPrototype
    self.gameViewController = gameViewController
    self.gameboard = gameboard
    self.gameboardView = gameboardView
}
```

В функции **addMark** уберем код, отвечавший за создание view отметки. Вместо этого будем копировать view из прототипа:

```
public func addMark(at position: GameboardPosition) {
    guard let gameboardView = self.gameboardView
        , gameboardView.canPlaceMarkView(at: position)
        else { return }

    self.gameboard?.setPlayer(self.player, at: position)
    self.gameboardView?.placeMarkView(self.markViewPrototype.copy(), at:
position)
    self.isCompleted = true
}
```

Модели Player добавим код, который должен возвращать соответствующую view:

```
var markViewPrototype: MarkView {
    switch self {
    case .first: return XView()
    case .second: return OView()
    }
}
```

При создании состояния просто передадим этот прототип в инициализатор. Для этого в **GameViewController** изменим методы **goToFirstState** и **goToNextState** на следующую реализацию:

```
private func goToFirstState() {
    let player = Player.first
    self.currentState = PlayerInputState(player: player,
                                         markViewPrototype:
player.markViewPrototype,
                                         gameViewController: self,
                                         gameboard: gameboard,
                                         gameboardView: gameboardView)
}

private func goToNextState() {
    if let winner = self.referee.determineWinner() {
        self.currentState = GameEndedState(winner: winner, gameViewController:
self)
        return
    }
    if let playerInputState = currentState as? PlayerInputState {
        let player = playerInputState.player.next
        self.currentState = PlayerInputState(player: player,
                                             markViewPrototype:
player.markViewPrototype,
                                             gameViewController: self,
                                             gameboard: gameboard,
                                             gameboardView: gameboardView)
    }
}
```

Паттерн Command

Паттерн **Command** (команда) — поведенческий шаблон проектирования. В этом паттерне центральную роль играют команды. Можно сказать, что примеры команд мы видели и раньше (но не применительно к этому паттерну). Зашли на экран в приложении — вью-контроллер отдал команду view отрисоваться и команду сервису на выполнение сетевого запроса. Когда игра завершилась, вью-контроллер отдал команду сохранить результаты на диск.

Но паттерн **Command** — это немного другое. В нем команды становятся отдельными объектами. Объект команды — это то, что должно быть выполнено, но обычно через какое-то время. До этого

момента объект команды хранится в памяти и даже может изменяться. Паттерн «команда» служит для того, чтобы описать и сохранить действие, которое должно быть выполнено позже.

Рассмотрим структуру паттерна **Command**. Она состоит из трех объектов:

```
Invoker -> Command -> Receiver
```

Invoker («вызывающий») — объект, который хранит команды и ставит их на исполнение. Это контроллер для команд.

Command — непосредственно объект команды. Он инкапсулирует действие, которое должно быть выполнено позднее.

Receiver («получатель») — объект, который получает команды. Именно он делает всю основную работу. Он получает команды, но они содержат лишь описание действия. Само действие делает **receiver**.

Пример в Playground

Сначала рассмотрим простой пример, чтобы понять роли всех действующих объектов в паттерне. Есть дверь, которая может открываться и закрываться. Открытие и закрытие двери — это команды. Дверь — это **receiver**, именно в этом классе будет реализована функциональность того, что команда выполняет. В нашем случае это будет просто смена флага **isOpen** с **false** на **true**, если дверь открывается, и с **true** на **false**, если закрывается. Также мы добавим сюда еще один объект, **invoker**, который будет управлять открытием и закрытием двери, создавать и вызывать команды. Начнем с **receiver**, то есть с двери:

```
// MARK: - Receiver

class Door {

    private(set) var isOpen = false {
        didSet {
            print(self.isOpen ? "door was opened" : "door was closed")
        }
    }

    func open() {
        guard !self.isOpen else {
            return
        }
        sleep(1)
        self.isOpen = true
    }

    func close() {
        guard self.isOpen else {
            return
        }
        sleep(1)
        self.isOpen = false
    }
}
```

Создадим две команды — на открытие и закрытие двери, и общий протокол для них. В принципе, можно было бы обойтись и одной командой, но мы используем разделение для примера:

```
// MARK: - Command

protocol Command {

    func execute()
}

class OpenDoorCommand: Command {

    weak var door: Door?

    func execute() {
        self.door?.open()
    }
}

class CloseDoorCommand: Command {

    weak var door: Door?

    func execute() {
        self.door?.close()
    }
}
```

И, наконец, **invoker**:

```
// MARK: - Invoker

class DoorInvoker {

    let door = Door()
    var commands: [Command] = []

    func work() {
        let openCommand = OpenDoorCommand()
        let closeCommand = CloseDoorCommand()
        openCommand.door = self.door
        closeCommand.door = self.door
        self.commands = [openCommand, closeCommand, openCommand, closeCommand]
        self.commands.forEach { $0.execute() }
    }
}

let invoker = DoorInvoker()
invoker.work()
```

DoorInvoker имитирует сложную логику по созданию, хранению и вызову команд. В нашем примере, правда, эта логика очень проста, но обычно **invoker** инкапсулирует управление командами.

В выводе консоли последовательно увидим распечатывание «door was opened» и «door was closed».

Применение на практике

Поговорим о более жизненных примерах применения этого паттерна в iOS-разработке.

Чаще всего его используют для логирования и отправки запросов пачками.

Логирование. Это запись действий приложения в лог — отдельный файл, который можно хранить на клиенте и при необходимости достать, чтобы отправить на сервер (а там уже разработчики будут смотреть логи и выяснять, что привело к ошибке в приложении). Запись большого количества текста в файл — не самая дешевая операция, и делать ее на каждый чих приложения затратно. Поэтому лучше складировать команды на запись логов в отдельном классе (роль **invoker**), а затем, когда логов накопится достаточно много (или приложение вот-вот выгрузится из памяти), все накопленные записи разом сохранить в файл.

Отправка запросов пачками. Аналогичная ситуация. На каждое важное действие юзера приложение отправляет запрос на наш сервер. Это может делаться для подсчета статистики, конверсии и т. д. И опять же — если на каждый скролл, каждый тап, каждый свайп приложение будет отправлять запрос, то это может быть накладно в плане производительности и нагрузки на сеть. Вместо этого можно складировать команды на отправку запросов, а затем, когда их накопится достаточно много, отправить все пачкой (batch-запросом) и начать складировать снова.

Еще пара примеров, более редких и специфичных:

- Есть сложное действие, которое юзер должен выполнить, и оно разбито на несколько экранов. Например, покупка. На первом экране юзер вводит контактные данные, на втором — адрес доставки, на третьем — данные банковской карты для оплаты, затем нажимает кнопку «Оплатить». По сути это все одно действие, одна команда. Она разбита на шаги, и объект, который будет выступать в роли **invoker**, должен будет на каждом шаге дополнять команду покупки, а затем вызвать ее после того, как юзер нажмет «Оплатить».

- Иногда загрузку приложения можно разбить на шаги. Например, сначала инициализируется один сервис, потом идут какие-то запросы, потом инициализируется другой сервис, проверяется валидность сессии и т. д. Каждый из этих шагов можно представить в виде команды. Invoker’ом будет являться класс-загрузчик (скорее всего, **AppDelegate**), receiver’ами —различные сервисы, которые нужно проинициализировать в приложении. Если примерно известно, сколько времени занимает выполнение каждой команды, то можно вывести индикатор загрузки.

Пример в проекте

Применим паттерн **Command** для сохранения записей в логи. Когда юзер тапает по игровому полю, нужно сделать запись об этом в лог, указав игрока и позицию на поле. Когда игра завершается — либо победой, либо ничьей, — надо записать это в лог. И когда в приложении нажимается кнопка перезапуска игры, про это тоже будем писать в лог. Получается три действия. В реальном приложении их может быть намного больше.

Добавим **enum**, перечисляющий эти действия. Это не является важной частью паттерна, просто нам так будет удобнее и код будет красивее. Создадим файл **LogAction.swift** и добавим в него следующий код:

```
public enum LogAction {

    case playerInput(player: Player, position: GameboardPosition)

    case gameFinished(winner: Player?)

    case restartGame
}
```

Теперь перейдем к реализации паттерна. Есть три объекта: **Invoker**, **Command**, **Receiver**. Начнем с команды. Добавим файл **LogCommand.swift** и напишем в него следующий код:

```
// MARK: - Command

final class LogCommand {

    let action: LogAction

    init(action: LogAction) {
        self.action = action
    }

    var logMessage: String {
        switch self.action {
        case .playerInput(let player, let position):
            return "\(player) placed mark at \(position)"
        case .gameFinished(let winner):
            if let winner = winner {
                return "\(winner) win game"
            } else {
                return "game finished with no winner"
            }
        case .restartGame:
            return "game restarted"
        }
    }
}
```

Инициализируем команду с одним из **LogAction**. В самой команде есть свойство, возвращающее строку, которую мы должны записать в лог.

Теперь создадим **Receiver**. Напомним — это тот объект, который делает саму работу. Команда содержит лишь описание действия (в нашем случае — то сообщение, которое необходимо записать). А **Receiver** непосредственно будет писать, то есть он и будет логером. Мы не будем писать реализацию логера, так как это не относится к паттерну и выходит за рамки курса, да и реализация может сильно отличаться в зависимости от конкретных целей. Поэтому просто будем печатать лог в консоль. Создадим файл **Logger.swift** и добавим в него следующее:

```
// MARK: - Receiver

```

```
final class Logger {

    func writeMessageToLog(_ message: String) {
        /// Здесь должна быть реализация записи сообщения в лог.
        /// Для простоты примера паттерна не вдаемся в эту реализацию,
        /// а просто печатаем текст в консоль.
        print(message)
    }
}
```

Дальше займемся имплементацией последнего объекта из паттерна — **Invoker**. Мы перешли к нему в самом конце, потому что он является связующим звеном между двумя остальными. Он управляет командами и ставит их на выполнение, то есть передает receiver'у.

Реализация будет следующая (в файле **LoggerInvoker.swift**):

```
// MARK: - Invoker

internal final class LoggerInvoker {

    // MARK: Singleton

    internal static let shared = LoggerInvoker()

    // MARK: Private properties

    private let logger = Logger()

    private let batchSize = 10

    private var commands: [LogCommand] = []

    // MARK: Internal

    internal func addLogCommand(_ command: LogCommand) {
        self.commands.append(command)
        self.executeCommandsIfNeeded()
    }

    // MARK: Private

    private func executeCommandsIfNeeded() {
        guard self.commands.count >= batchSize else {
            return
        }
        self.commands.forEach { self.logger.writeMessageToLog($0.logMessage) }
        self.commands = []
    }
}
```

Для простоты примера будем использовать синглтон для инвокера. В принципе, и в реальной задаче это может быть синглтон. Этот объект хранит массив команд в свойстве **commands**. У него есть метод **addLogCommand**, который добавляет команду на исполнение. Она не будет выполнена сразу же, вместо этого просто добавится в массив, затем произойдет проверка. Если команд на исполнение

уже достаточно (их количество равно константе **batchSize** или превышает ее), то все эти команды будут выполнены и очищены. Если нет, будем ждать дальше.

И создадим public-функцию, которая позволит добавлять логи из клиентского кода, используя удобную энамку **LogAction**. Вернемся в файл **LogAction.swift** и добавим в конце:

```
public func Log(_ action: LogAction) {
    let command = LogCommand(action: action)
    LoggerInvoker.shared.addLogCommand(command)
}
```

Логер готов к использованию. В **PlayerInputState** при добавлении метки добавим вызов записи в лог:

```
public func addMark(at position: GameboardPosition) {
    Log(.playerInput(player: self.player, position: position))
    ...
}
```

В **GameEndedState** добавим запись в лог об окончании игры в вызове **begin()**:

```
public func begin() {
    Log(.gameFinished(winner: self.winner))
    ...
}
```

В **GameViewController**, где ловится нажатие на кнопку **Restart**, добавим вызов логирования в соответствующий **IBAction**:

```
@IBAction func restartButtonTapped(_ sender: UIButton) {
    Log(.restartGame)
    ...
}
```

Можно проверить, что происходит вывод в консоль, — но не сразу, а когда накопится от 10 сообщений.

Практическое задание

1. Задание на паттерн **State**.

   В этом практическом задании вы будете продолжать проект с игрой в крестики-нолики, который делали на уроке. Задача в том, чтобы добавить еще один режим — игра против компьютера. Режим можно выбрать из главного меню, на экране которого должно быть две кнопки: «Игра против компьютера» и «Игра с двумя игроками». Напишите алгоритм, по которому компьютер будет делать ходы. Самый простой вариант — компьютер рандомно ставит отметку в любой свободной клетке.

(* Дополнительно: можете усложнить этот алгоритм — в интернете есть готовые решения и советы.)

Добавьте новый **state** — ход компьютера. Аналогично примерам из урока сделайте переключение состояния со **state** игрока на **state** компьютера при выполнении шагов в игре.

2. Второе задание на паттерн **State**.

   Примечание: возможно, задания 2 и 3 вы решите делать вместе, поэтому прочтите их оба, прежде чем приступать к выполнению.

   Изменим правила игры: теперь игроки ходят не по очереди, а сначала первый игрок указывает 5 клеток, в которые он ставит крестик, затем второй игрок указывает 5 клеток, куда он ставит нолик. Затем игра должна показать расстановку крестиков и ноликов в последовательности крестик — нолик — крестик — нолик — ... Иными словами, правила игры такие же, как раньше, только теперь каждый игрок должен вслепую заранее указать все свои 5 ходов. Если отметка одного игрока при этом накладывается на отметку другого (они же не знали ходы друг друга, так что такое возможно), то она сначала стирает предыдущую отметку и затем ставится в эту клетку. Когда пройдена последовательность ходов, игра определяет победителя.

   Реализуйте такое поведение для игры между двумя живыми игроками, добавив еще одно состояние — состояние хода игрока, в котором игрок ставит все 5 отметок. Затем добавьте еще состояние исполнения игры. В нем игра должна последовательно показать ходы игроков, как если бы они шли в обычной игре в крестики-нолики. Затем должен определиться победитель (для этого можно использовать написанный на уроке код).

3. Задание на паттерн **Command**.

   Ход игрока представьте в виде команды. **Recievers** (получатели) у нас уже есть — это объект доски и ее view. Подумайте, кто будет являться invoker'ом. Подсказка: для него лучше создать отдельный объект. Команду инициализируйте с информацией об игроке, позиции и доске. В команду добавьте метод **execute()**, в котором будет обращение к receiver'ам: добавить отметку на доску, отрисовать ее. **PlayerInputState** измените так, чтобы когда юзер тапает на клетку, он вызывал добавление команды invoker'у. Когда все ходы расставлены, **state** исполнения игры сообщает invoker'у, что нужно исполнить все команды, и таким образом партия игры будет разыграна.

4. * Необязательное задание на паттерн **Prototype**. Исследуйте свои имеющиеся проекты — где можно применить паттерн «прототип». Если можно вместо повторной инициализации копии объекта использовать прототип, то добавьте этому объекту поддержку протокола **Copying** и реализуйте паттерн.

5. * Дополнительное (необязательное) задание на паттерн **State**. Сделайте приложение, которое будет отображать светофор с тремя состояниями: «горит зеленый», «горит красный» и «горит желтый свет». Все три состояния сделайте разными объектами, поддерживающими один протокол **State**. Состояния должны сменять друг друга с задержкой.

6. ** Дополнительное (необязательное) задание на паттерн **Command**. Реализуйте паттерн «команда» для одной из задач, описанных в этой методичке в разделе «Паттерн Command. Применение на практике».

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».