Принципы Объектно-ориентированного программирования

ООП это методология программирования в основе которой лежит понятие объекта. Под объектом понимается экземпляр некоторого класса, отвечающий за определенный функционал программы. Работа всей программы осуществляется путем взаимодействия ее объектов между собой.

Рассмотрим основные принципы ООП на примере небольшой программы. Приложение будет без какого-либо UI, чтобы сконцентрироваться именно на принципах ООП. Приложение будет некоторым клиентом (интерфейс, получающий все данные из сети и служащий для их отображения и взаимодействия с данными) для администрирования пользователей.

Основной сущностью нашего приложения, которой мы будем оперировать, будет пользователь (User). Создадим структуру User. Можно использовать класс, но так как конкретный экземпляр у нас будет просто хранить информацию о пользователе и не производить никаких действий (ни с кем не взаимодействовать), то подойдет структура (к тому же программа будет работать чуточку быстрее, так как структуры лежат в Стеке, по прямому доступу).

```
struct User: CustomStringConvertible {

    let id: Int
    let name: String
    var description: String {
        "User \(name) with id - \(id)"
    }
}
```

Добавим также протокол CustomStringConvertible для корректного вывода информации о пользователе в консоль. Пока что мы лишь подготавливаем основную структуру нашего приложения, так что говорить про какие-то принципы пока рано.

Следующим шагом подготовим для нашего приложения то, где мы будем хранить наших пользователей. Все хранилища у нас будут иметь одинаковый интерфейс, различающийся лишь принципом хранения и загрузки пользователей. Создадим такой интерфейс в качестве протокола.

```
protocol StorageProviderProtocol {
    static func load() -> [User]
    static func save(_ users: [User])
}
```

Все, что нам достаточно знать про такой интерфейс это то, что мы можем сохранить пользователей в наше хранилище или загрузить их оттуда, детали реализации будут зависеть от самого хранилища. Далее создадим дефолтное хранилище, которое не будет уметь что-либо хранить и загружать, а будет являться лишь заглушкой.

```
enum Default: StorageProviderProtocol {

    static func load() -> [User] {
        print("I don't know anything about users!")
        return []
    }

    static func save(_ users: [User]) {
        print("I don't know what to do with these users!")
    }
}
```

Используем перечисление и статические методы для того, чтобы не нужно было создавать экземпляры данных хранилищ, а они были доступны глобально.

Первое осмысленное хранилище которые мы создадим далее будет Сервер на котором будут храниться пользователи нашего приложения.

```
enum Server: StorageProviderProtocol {

    private static var users = [User]()

    static func load() -> [User] {
        return users
    }

    static func save(_ users: [User]) {
        self.users = users
    }
}
```

Здесь все просто: создаем приватное свойство для хранения списка пользователей и реализуем два метода, описанных в требованиях протокола, где указываем логику их работы. Не будем требовать от сервера каких-то специфических возможностей, поэтому нам будет достаточно данной реализации.

Следующее хранилище, которые мы опишем будет база данных на стороне клиентского приложения. Прежде чем описывать хранилище создадим отдельную модель данных для хранения пользователя в базе.

```
private struct UserModel {

    private let dataSeparator: Character = "|"

    var data: Data

    init(user: User) {
        let components = ["\(user.id)", user.name]

        data = components
            .joined(separator: String(dataSeparator))
            .data(using: .utf8) ?? Data()
    }
```

```
    func toUser() -> User? {
        guard let string = String(data: data, encoding: .utf8) else {
            return nil
        }

        let components = string
            .split(separator: dataSeparator)
            .map(String.init)

        guard components.count == 2 else { return nil }

        guard let idString = components.first,
              let id = Int(idString),
              let name = components.last else { return nil }

        return User(id: id, name: name)
    }
}
```

Остановимся подробнее на данной структуре. Данные в нашей базе будут храниться виде бинарных данных Data. Прежде чем переводить информацию о нашем пользователе в тип Data при создании модели базы данных, мы склеим все свойства пользователя в одну строку (используя метод joined(separator:) у массива строк) через разделитель dataSeparator. После чего можно вызвать метод data(using:) у строки, указав кодировку (в нашем случае UTF-8) для превращения строки в бинарные данные. Также укажем значение по умолчанию (??) в виде пустых данных, так как результат преобразования String в Data опциональный.

Также у модели будет метод toUser, который будет конвертировать бинарные данные обратно в экземпляр структуры User. Для этого мы сначала преобразуем бинарные данные в строку, далее разбиваем полученную строку на компоненты, используя тот же разделитель, который использовали при объединении свойств пользователя в одну строку. Проверяем что компоненты действительно две и что это число и строка. Если все хорошо, возвращаем из данного метода нашего пользователя.

Теперь опишем хранилище, указав структуру для моделей базы данных в качестве приватной структуры нашего хранилища.

```
enum Database: StorageProviderProtocol {

    private struct UserModel {...}

    private static var users = [UserModel]()

    static func load() -> [User] {
        return users.compactMap { $0.toUser() }
    }

    static func save(_ users: [User]) {
        self.users = users.map(UserModel.init)
    }
}
```

Не просто так мы модель данных UserModel сделали приватной внутри хранилище. Это сделано для того, чтобы данная модель не была доступна извне, потому как вне нашей базы она не нужна, и не должна использоваться в основном коде. Все что мы делаем в методах загрузки и сохранения это переводим одни модели данных в другие, используя соответствующие методы массивов.

Теперь, когда у нас есть хранилища, нам нужно создать менеджеров для взаимодействия с ними, чтобы не делать это напрямую из нашего приложения.

Как обычно начнём с объявления интерфейса данных менеджеров виде протокола.

```
protocol SourceManagementProtocol {
    var source: StorageProviderProtocol.Type { get }
}
```

Здесь мы всего лишь указываем на то, что менеджер должен иметь источник данных виде хранилища. Далее укажем ещё один интерфейс позволяющий получить загруженные данные от нашего менеджера, либо редактировать их.

```
protocol UserCollectionable {
    var collection: [User] { get set }
}
```

Остановимся на этом моменте подробнее. Зачем каждый раз описывать протокол вместо создания конкретных реализаций? На самом деле помимо основных трех принципов ООП выделяют дополнительный (нулевой) принцип называемый "Абстракция", идея которого в том, что мы не должны завязываться на конкретную реализацию, а взаимодействовать лишь с тем интерфейсом, который необходим в данный момент времени. Если в какой-то момент времени нам достаточно лишь знать, что менеджер взаимодействует с некоторым хранилищем мы будем использовать первый протокол. Если в другой момент времени мы будем работать с менеджером как с хранилищем закешированных данных пользователей нам нужно знать только то, что эти данные есть и не нужно знать откуда и как они взялись. Используем второй протокол.

Теперь, когда у нас есть весь необходимый интерфэйс для реализации класса менеджера мы можем приступить к его созданию.

```
class UsersCollectionManager: SourceManagementProtocol,
UserCollectionable {

    var source: StorageProviderProtocol.Type { Default.self }

    private var isSavingNeeded = false

    var collection = [User]() {
        didSet {
            guard isSavingNeeded else { return }
            saveData()
        }
    }

    init() {
        print("\(source) connected")
```

```
        loadData()
    }

    deinit {
        print("\(source) disconnected")
    }

    private func saveData() {
        source.save(collection)
        print("\(source)| Saved  collection with \(collection.count)
elements")
    }

    private func loadData() {
        defer {
            isSavingNeeded = true
        }
        isSavingNeeded = false

        collection = source.load()
        print("\(source)| Loaded collection with \(collection.count)
elements")
    }
}
```

Разберём реализацию данного класса. Во-первых мы указываем тип хранящихся данных для наших менеджеров (User) и тип хранилища в качестве источника этих данных (source). По умолчанию будем использовать дефолтное хранилище. Далее в классе мы описываем флаг отвечающие за то нужно ли сохранять закешированные данные в хранилище и саму коллекцию хранящую список пользователей (требуемую одним из протоколов). В методе init будем выводить в консоль информацию о том, что менеджер подключился к конкретному хранилищу после чего будем производить загрузку данных из этого хранилища. В методе deinit будем выводить в консоль информацию о том, что мы завершили работу с хранилищем.

Далее опишем два приватных метода, осуществляющих взаимодействие с хранилищем с дополнительным логированием в консоль. Обратите внимание на метод загрузки в котором мы устанавливаем флаг сохранения в значение false. Это нужно для того, чтобы после загрузки данных в свойство collection в наблюдателе свойства не был вызван метод сохранения. Блог defer сработает в конце работы метода и просто вернет флаг обратно в значение true. Последним шагом мы описываем методы загрузки и сохранения, требуемые в протоколе.

Методы сохранения и загрузки не случайно описаны приватно. Это первый принцип ООП, который называется "Инкапсуляция". Идея данного принципа в том, что мы должны скрывать часть реализации нашего класса из внешнего кода для того, чтобы не было возможности нарушить работу экземпляра класса. В нашем случае загрузка и сохранение данных будет происходить в автоматическом режиме, при создании менеджера или при изменении данных. И из внешнего кода не нужно иметь доступ к этим методам, чтобы редактировать или получать список пользователей. Нам даже не нужно знать, как это работает. Вся логика должна быть сокрыта внутри класса менеджера.

Теперь, когда у нас есть основной класс менеджера мы можем воспользоваться вторым принципом ООП называемым "Наследование". Идея этого

принципа в том, что мы можем не дублировать функционал схожих классов, а использовать механизм наследования (иначе говоря мы можем перенять уже готовые методы и свойства родительского класса от которого мы создаем наследника). Таким образом мы можем создать двух менеджеров для работы с Сервером и с Базой данных, используя все наработки с редактированием и получением списка пользователей, которые есть в родительском классе менеджера.

```
final class NetworkManager: UsersCollectionManager {

    override var source: StorageProviderProtocol.Type { Server.self }
}

final class DatabaseManager: UsersCollectionManager {

    override var source: StorageProviderProtocol.Type { Database.self }
}
```

Все что нам нужно переопределить в классах наследниках это источник получения данных. Также объявим для данных классов модификатор final, указывающий на то, что от этих классов мы не будем создавать наследников. Это добавит нашей программе небольшую оптимизацию производительности.

Теперь приступим к описанию основного класса нашего приложения.

```
final class AdminApp {

    private var currentManager: UserCollectionable =
UsersCollectionManager()
    var isNetworkAvailable: Bool = false {
        didSet {
            if isNetworkAvailable, oldValue == false {
                synchronizeNetwork()
            }

            setupManager()
        }
    }

    var users: [User] {
        get {
            currentManager.collection
        }
        set {
            currentManager.collection = newValue
            if isNetworkAvailable {
                synchronizeDatabase()
            }
        }
    }

    init(isNetworkAvailable: Bool = true) {
        self.isNetworkAvailable = isNetworkAvailable
        setupManager()
```

```
    }

    private func setupManager() {
        if isNetworkAvailable {
            currentManager = NetworkManager()
        } else {
            currentManager = DatabaseManager()
        }
    }

    private func synchronizeNetwork() {
        NetworkManager().collection = users
    }

    private func synchronizeDatabase() {
        DatabaseManager().collection = users
    }
}
```

Для корректной работы приложения нам нужно описать свойство, хранящее менеджера (по умолчанию будем использовать родительский класс работающий с дефолтом хранилищем). Здесь мы наглядно видим третий принцип ООП, называемый "Полиморфизм". Идея этого принципа в том, что мы можем объединить разные реализации под некоторым общим интерфейсом или типом данных. Мы намеренно указываем тип менеджера в виде интерфейса для доступа к закешированным данным пользователей, потому как иное от менеджер нам не нужно. Теперь, используя полиморфизм, мы можем сохранить в это свойство любого менеджера который будет обладать данным интерфейсом, не важно будет ли он работать сетью, базой данных, дефолтным хранилищем, или вообще будет какой-то другой менеджер.

Далее опишем свойство для хранения состояния "соединения с сетью" для переключения между сетевым менеджером и менеджером базы данных. И последнее вычисляемое свойство users со списком пользователей, получаемых от менеджера, либо сохраняемых в его свойство collection при редактировании. Каждый раз когда мы будем редактировать пользователей, если есть соединение сетью мы будем также дополнительно сохранять изменённые данные в локальную базу данных, чтобы иметь возможность работать с актуальными данными, когда соединения с сетью не будет.

В конструкторе класса приложения мы будем устанавливать значение состояние сети (будто при старте приложения проверяем сеть) и создавать соответствующего менеджера. Далее опишем три приватных метода для создания менеджера и сохранения отредактированных пользователей на сервере или в базе данных.

Основная работа над приложением закончена, создадим дополнительно глобальный метод печати, который позволит нам протестировать работу приложения наглядно.

```
func printStorageInfo(after action: String) {
    print()
    print("--------ACTION--------")
    print(action)
    print("--------SERVER--------")
    print(Server.load())
```

```
    print("-------DATABASE-------")
    print(Database.load())
    print()
}
```

Здесь мы будем выводить в консоль действие производимое с приложением и выводить актуальную информацию о пользователях на сервере и в базе данных.

Начнем эксперимент. Запустим приложение с доступом в сеть, добавим двух пользователей, после чего сымитируем отключение от сети. Добавим ещё одного пользователя и закроем приложение.

```
// Launch App
var myApp: AdminApp? = AdminApp(isNetworkAvailable: true)

myApp?.users.append(User(id: 1, name: "Bob"))
myApp?.users.append(User(id: 2, name: "Bill"))

myApp?.isNetworkAvailable = false

myApp?.users.append(User(id: 3, name: "Jack"))

// Close App
myApp = nil
```

Теперь воспользуемся методом тестирования приложение чтобы посмотреть состояние пользователей на сервере в базе данных.

```
printStorageInfo(after: "Added 2 users with network and 1 without")
```

Получим следующий вывод в консоль.

```
Default connected
I don't know anything about users!
Default| Loaded collection with 0 elements
Server connected
Server| Loaded collection with 0 elements
Default disconnected
Server| Saved  collection with 1 elements
Database connected
Database| Loaded collection with 0 elements
Database| Saved  collection with 1 elements
Database disconnected
Server| Saved  collection with 2 elements
Database connected
Database| Loaded collection with 1 elements
Database| Saved  collection with 2 elements
Database disconnected
Database connected
Database| Loaded collection with 2 elements
Server disconnected
Database| Saved  collection with 3 elements
Database disconnected
```

```
--------ACTION--------
Added 2 users with network and 1 without
--------SERVER--------
[User Bob with id - 1, User Bill with id - 2]
-------DATABASE-------
[User Bob with id - 1, User Bill with id - 2, User Jack with id - 3]
```

Во-первых здесь поэтапно отображается информация о всех подключениях и отключениях менеджеров нашего приложения, начиная с Дефолтного, при старте приложения и заканчивая менеджером Базы данных, который работал, когда приложение не имело доступ в сеть. Далее мы видим, что на сервере у нас хранится два пользователя, а в базе данных три, потому что при добавлении последнего пользователя сеть была недоступна.

Ещё раз запустим приложение без доступа в сеть, добавим четвёртого пользователя и выведем информацию о состоянии пользователей на Сервере и в Базе данных.

```
// Launch App
myApp = AdminApp(isNetworkAvailable: false)

myApp?.users.append(User(id: 4, name: "Jess"))

printStorageInfo(after: "Added user without network")
```

```
Default connected
I don't know anything about users!
Default| Loaded collection with 0 elements
Database connected
Database| Loaded collection with 3 elements
Default disconnected
Database| Saved  collection with 4 elements

--------ACTION--------
Added user without network
--------SERVER--------
[User Bob with id - 1, User Bill with id - 2]
-------DATABASE-------
[User Bob with id - 1, User Bill with id - 2, User Jack with id - 3,
User Jess with id - 4]
```

Как мы видим все работает корректно: в Базе данных пользователи прибавляются, а без доступа к сети на сервер они не попадают.

```
myApp?.isNetworkAvailable = true

printStorageInfo(after: "Connect to network")
```

Если теперь появится соединение к сети, База данных и Сервер синхронизируются и будут хранить всех пользователей.

```
Server connected
```

```
Server| Loaded collection with 2 elements
Server| Saved  collection with 4 elements
Server disconnected
Server connected
Server| Loaded collection with 4 elements
Database disconnected

--------ACTION--------
Connect to network
--------SERVER--------
[User Bob with id - 1, User Bill with id - 2, User Jack with id - 3,
User Jess with id - 4]
-------DATABASE-------
[User Bob with id - 1, User Bill with id - 2, User Jack with id - 3,
User Jess with id - 4]
```

Если сейчас добавить пятого пользователя, он будет добавлен как на Сервер так и в Базу данных.

```
myApp?.users.append(User(id: 5, name: "Mike"))

// Close App
myApp = nil

printStorageInfo(after: "Added user with network")
```

```
Server| Saved  collection with 5 elements
Database connected
Database| Loaded collection with 4 elements
Database| Saved  collection with 5 elements
Database disconnected
Server disconnected

--------ACTION--------
Added user with network
--------SERVER--------
[User Bob with id - 1, User Bill with id - 2, User Jack with id - 3,
User Jess with id - 4, User Mike with id - 5]
-------DATABASE-------
[User Bob with id - 1, User Bill with id - 2, User Jack with id - 3,
User Jess with id - 4, User Mike with id - 5]
```

Нам удалось создать корректно работающее приложение, которое позволяет взаимодействовать со списком пользователей как с доступом в сеть так и без доступа, синхронизируя информацию в различных хранилищах. К тому же мы опирались на основные принципы ООП, которые позволили написать нам качественный, поддерживаемый код. Любой, кто будет взаимодействовать с нашим приложением сможет лишь редактировать список пользователей, либо менять состояние соединения с сетью и никак не сможет поломать основную логику нашего приложения.

SOLID принципы

Следующие принципы которые мы рассмотрим неразрывно связаны с принципами ООП. Аббревиатура SOLID является акронимом и каждая буква здесь расшифровывается в отдельный принцип. Рассмотрим принципы на другом примере кода, где у нас будет некоторая панель для управления устройствами, позволяющая включать и выключать их. Управление устройствами будем осуществлять через наборы команд.

Так как SOLID принципы являются продолжением принципов ООП, мы должны придерживаться тех же правил, что и выше. Начнём с того, что объявим основные протоколы, требуемые для работы с устройствами и командами.

```
protocol TurningOn {
    func turnOn()
}

protocol TurningOff {
    func turnOff()
}

protocol Executable {
    func execute()
}
```

Придерживаясь принципа абстракция мы также реализуем здесь один из принципов SOLID скрывающийся под буковкой I. Это принцип разделения интерфейсов (The Interface Segregation Principle). Идея этого принципа в том, чтобы не писать большие интерфейсы, которые сложно будет использовать если хотя бы один метод или свойства, описанные в протоколе будут для нас лишними. Поэтому лучше описать несколько небольших протоколов и использовать их композицию в конкретной реализации.

Следующим шагом опишем основные классы устройств, которые будем использовать в нашем коде.

```
class Device {

    private enum State {
        case on, off
    }

    private var state: State = .off

    func turnOn() {
        state = .on
    }

    func turnOff() {
        state = .off
    }
}
```

За основу возьмём некоторый абстрактный родительский класс Device с состоянием и двумя методами для изменение этого состояния.

```
class Radio: Device, TurningOn, TurningOff {

    override func turnOn() {
        super.turnOn()
        print("Radio turned on")
        print("On wave: Pshhhhhh")
    }

    override func turnOff() {
        super.turnOff()
        print("Radio turned off")
    }
}

class Computer: Device, TurningOn, TurningOff {

    override func turnOn() {
        super.turnOn()
        print("Computer turned on")
        print("Motherboard: Pip")
    }

    override func turnOff() {
        super.turnOff()
        print("Computer turned off")
    }
}
```

Используя родительский класс и те протоколы, через который мы хотим управлять устройствами, создадим два класса конкретных устройств.

```
class Supercomputer: Computer {

    override func turnOn() {
        print("Start cooling system: pshhhh")
        print("Supercomputer as", terminator: " ")
        super.turnOn()
    }
}
```

Мы также можем использовать наследование от классов уже реализующих необходимый интерфейс для того, чтобы получить другие классы с возможностью управления ими.

```
class PerpetualEngine: Device, TurningOn {

    override func turnOn() {
        super.turnOn()
        print("Perpetual engine turned on")
        print("Engine: Vrrrrrr")
    }

    override func turnOff() {
```

```
        print("Can't stop!")
    }
}
```

Более того, мы не обязаны использовать оба протокола, так как мы уже говорили о том, что протоколы используются в композиции. Следовательно мы можем создать такое устройство, которое будет иметь только один интерфейс (вечный двигатель, который нельзя выключить).

Здесь можно остановиться и поговорить про ещё один принцип скрываешься под буквой L. Это принцип постановки Барбары Лисков (The Liskov Substitution Principle). Идея принципа в том, что у нас должна быть возможность использовать различные реализации, объединённые одним интерфейсом, также как это было при полиморфизме, либо использовать родительские и дочерние реализации классов в едином ключе. Иначе говоря, если у нас есть какой-то метод или объект, взаимодействующий с некоторым устройством (не важно каким), и мы передадим ему любое устройство, он должен отработать корректно. Либо, если у нас есть метод для работы с суперкомпьютером мы без проблем должны иметь возможность заменить его на экземпляр класса родителя (Computer) и при этом программа должна отработать без ошибок. Если посмотреть на реализацию вечного двигателя, то несмотря на то, что мы унаследовали от класса Device, мы сломали работу метода отключения данного устройства. Такая реализация имеет место быть только в том случае, если мы используем протоколы для взаимодействия между устройствами и в данном случае мы указали лишь протокол позволяющий включать устройство, но не выключать его. Однако стоит избегать в коде подобного переопределение методов родителя, когда реализация не дополняется, а полностью меняется.

Теперь создадим два класса, описывающих команды включения и выключения для командной панели.

```
final class OnSwitcher: Executable {

    private let device: TurningOn

    init(device: TurningOn) {
        self.device = device
    }

    func execute() {
        device.turnOn()
    }
}

final class OffSwitcher: Executable {

    private let device: TurningOff

    init(device: TurningOff) {
        self.device = device
    }

    func execute() {
        device.turnOff()
    }
```

```
}
```

Мы описываем эти классы с модификатором final, так как мы предполагаем наследования от них и указываем, что они должны имплементировать протокол Executable для того, чтобы можно было выполнить данную команду. При создании каждой из команд мы будем указывать то устройство на которое она направлена для включения или выключения его. Здесь мы можем наглядно увидеть использование ещё двух принципов. Первый принцип, скрывающийся под буквой S, принцип единый ответственности (The Single Responsibility Principle). Идея данного принципа в том, что мы не должны реализовывать классы, отвечающие за множественный функционал. Каждый класс должен быть создан только для одной конкретной задачи и должен решать только её. В нашем примере каждая из команд знает лишь про конкретное устройство, которым она управляет, и совершает лишь одно дейтвие над ним. Эти классы намеренно не объединены в один для того, чтобы не перегружать команду параметрами при создании. Иначе, помимо хранения конкретного устройства наша команда должна была бы хранить также то действие ,которое необходимо произвести над данным устройством.

Второй принцип, который здесь реализуются скрывается за последней буквы D, принцип инверсии зависимостей (The Dependency Inversion Principle). Этот принцип очень схож с принципом Абстракции из ООП и заключается, опять же, в том, чтобы не завязываться на конкретную реализацию конкретного устройства. В каждом из классов команд мы указываем тип устройства, как некоторую сущность, которую можно либо включить либо выключить. Остальная информация о том, на кого направлена команда нам не нужна.

Следующим шагам реализуем класс командной панели в котором рассмотрим последний из оставшейся принципов.

```
class CommandPanel {

    var commands: [Executable]

    init(commands: [Executable]) {
        self.commands = commands
    }

    func execute() {
        commands.forEach { $0.execute() }
    }
}
```

Мы также используем здесь принцип инверсии зависимостей для того, чтобы хранить разные команды в одном наборе. Мы можем как передать эти команды при создании командной панели, так и изменять их извне, так как свойство commands является открытым. Последний принцип, который мы рассмотрим скрывается под буквой O, принцип открытости и закрытости (The Open Closed Principle). Идея этого принципа в том, что наш код должен быть закрыт от изменений и открыт для расширения. Некоторые неверно понимают этот принцип, используя в его трактовке принцип Инкапсуляции из ООП. Этот принцип не говорит нам о том, что мы должны скрывать часть реализации внутри класса, вместо этого он несёт в себе идею того, что мы не должны изменять наш исходный класс, если захотим добавить ему дополнительные возможности. Конечно, мы можем добавить некоторую реализацию в расширении класса, не трогая исходное описание класса,

но идея здесь несколько глубже. Мы намеренно используем инверсию зависимостей для команд, чтобы, если в будущем мы захотим обработать совершенно другую команду, имплементирующую протокол Executable, то мы сможем это сделать просто добавишь реализацию такой команды в коде и передав её в командную панель, не изменяя исходный код класса.

Проведём несколько экспериментов с нашим кодом, чтобы посмотреть насколько гибкой получилась наша реализация. Мы можем, как создавать различные наборы команд для различных устройств, так можем работать с этими устройствами, как с некоторым абстрактным интерфейсом.

```
CommandPanel(commands: [
    OnSwitcher(device: Computer()),
    OnSwitcher(device: Radio()),
    OnSwitcher(device: Supercomputer()),
    OnSwitcher(device: PerpetualEngine())
]).execute()

print("---------")

CommandPanel(commands: [
    OnSwitcher(device: Computer()),
    OffSwitcher(device: Computer()),
    OnSwitcher(device: PerpetualEngine())
]).execute()

print("---------")

let devicesForLaunch: [TurningOn] = [Computer(), Radio(),
Supercomputer(), PerpetualEngine()]

devicesForLaunch.forEach { $0.turnOn() }
```

Вот что мы увидим в консоли после выполнения программы.

```
Computer turned on
Motherboard: Pip
Radio turned on
On wave: Pshhhhhh
Start cooling system: pshhhh
Supercomputer as Computer turned on
Motherboard: Pip
Perpetual engine turned on
Engine: Vrrrrrr
---------
Computer turned on
Motherboard: Pip
Computer turned off
Perpetual engine turned on
Engine: Vrrrrrr
---------
Computer turned on
Motherboard: Pip
Radio turned on
On wave: Pshhhhhh
```

```
Start cooling system: pshhhh
Supercomputer as Computer turned on
Motherboard: Pip
Perpetual engine turned on
Engine: Vrrrrrr
```

Как видите, использование различных принципов на практике позволяет писать гибки легко масштабируемых код. Помимо рассмотренных принципов есть также другие которые повсеместно применяются каждый день, возможно даже неосознанно. Такие принципы как DRY (не повторяйся), KISS (не усложняй), YAGNI (не делай лишнего функционала), и многие другие. Все принципы являются лишь набором рекомендаций и не являются истиной, которой нужно придерживаться при реализации кода. Бывают ситуации, когда слепое следование принципам только усложнит код. Например, вы реализуете какое-то приложение максимально следую всем принципам которые знаете. Тогда ваше приложение будет состоять из множества файлов, оно будет легко масштабируемым и будет грамотно описано, но в тоже время будет состоять из огромного количества мелких разрозненных кусочков собрать которые воедино новому разработчику (или вам через длительный период времени) будет очень сложно. Вместо одной конкретной реализации некоторого экрана в вашем приложении у вас может получиться куча классов и протоколов так или иначе описывающих реализацию данного экрана. Поэтому не стоит в простых приложениях, либо в некоторых частных случаях стараться сделать все максимально универсальным и поделенным на фрагмент. Понимание того в какой мере и где стоит использовать принципы приходит лишь с опытом и нельзя описать некоторый набор правил для их реализации. Все очень субъективно и зависит от конкретной задачи. Тем не менее всегда следует помнить про основные принципы и, когда вы пишите код следует заранее продумывать его реализацию, чтобы не переписывать и не городить костыли.

Примеры кода доступны по ссылке https://github.com/SemyonovE/Swift.Course.UIKit