



Урок 10

UI-тесты

Тестирование пользовательского интерфейса.

[Как выглядят UI-тесты?](#)

[Для чего используются UI-тесты?](#)

[Как писать UI-тесты?](#)

[Режим записи](#)

[Ручное редактирование](#)

[Использование Accessibility для поиска UI-компонентов](#)

[Работа с UIAlertController](#)

[Передача настроек приложению](#)

[Практическое задание](#)

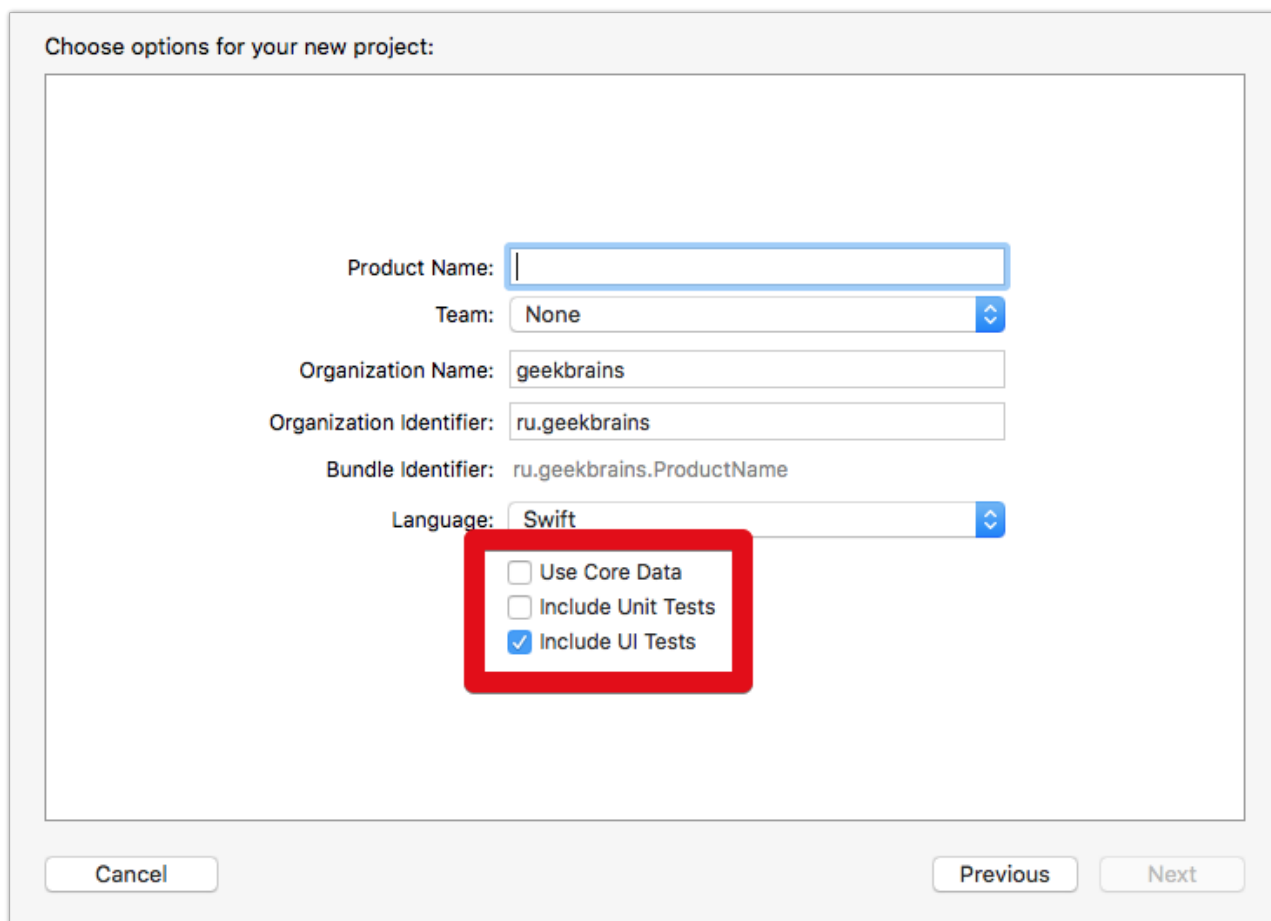
[Дополнительные материалы](#)

[Используемая литература](#)

Как выглядят UI-тесты?

UI-тестирование дает возможность проверять свойства и состояние элементов пользовательского интерфейса, эмулируя действия людей.

Тестирование пользовательского интерфейса в XCode представлено сборкой **iOS UI Testing Bundle**. Задать ей цель можно сразу при создании проекта, включив галочку **Include UI Tests**.



Choose options for your new project:

Product Name:

Team:

Organization Name:

Organization Identifier:

Bundle Identifier:

Language:

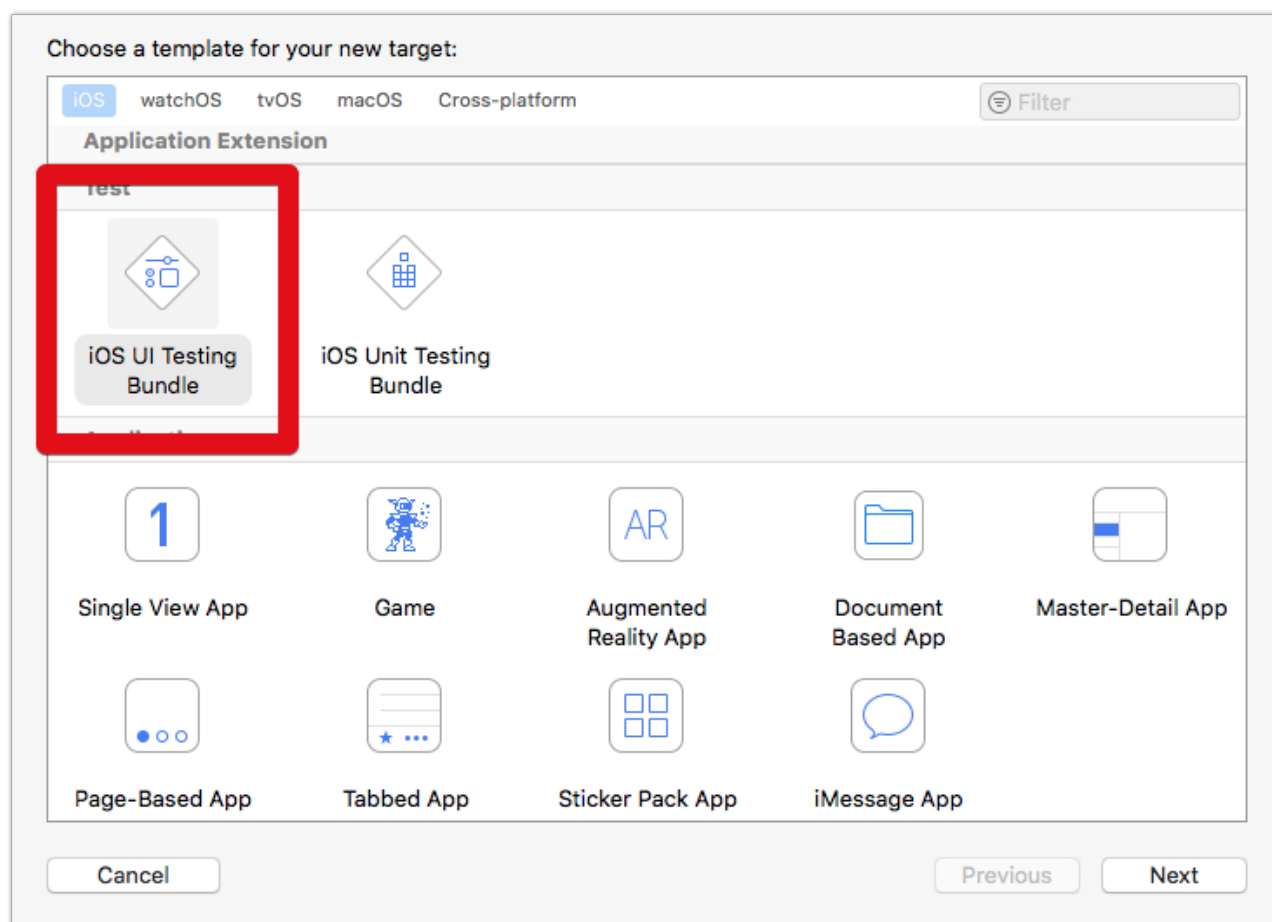
☐ Use Core Data

☐ Include Unit Tests

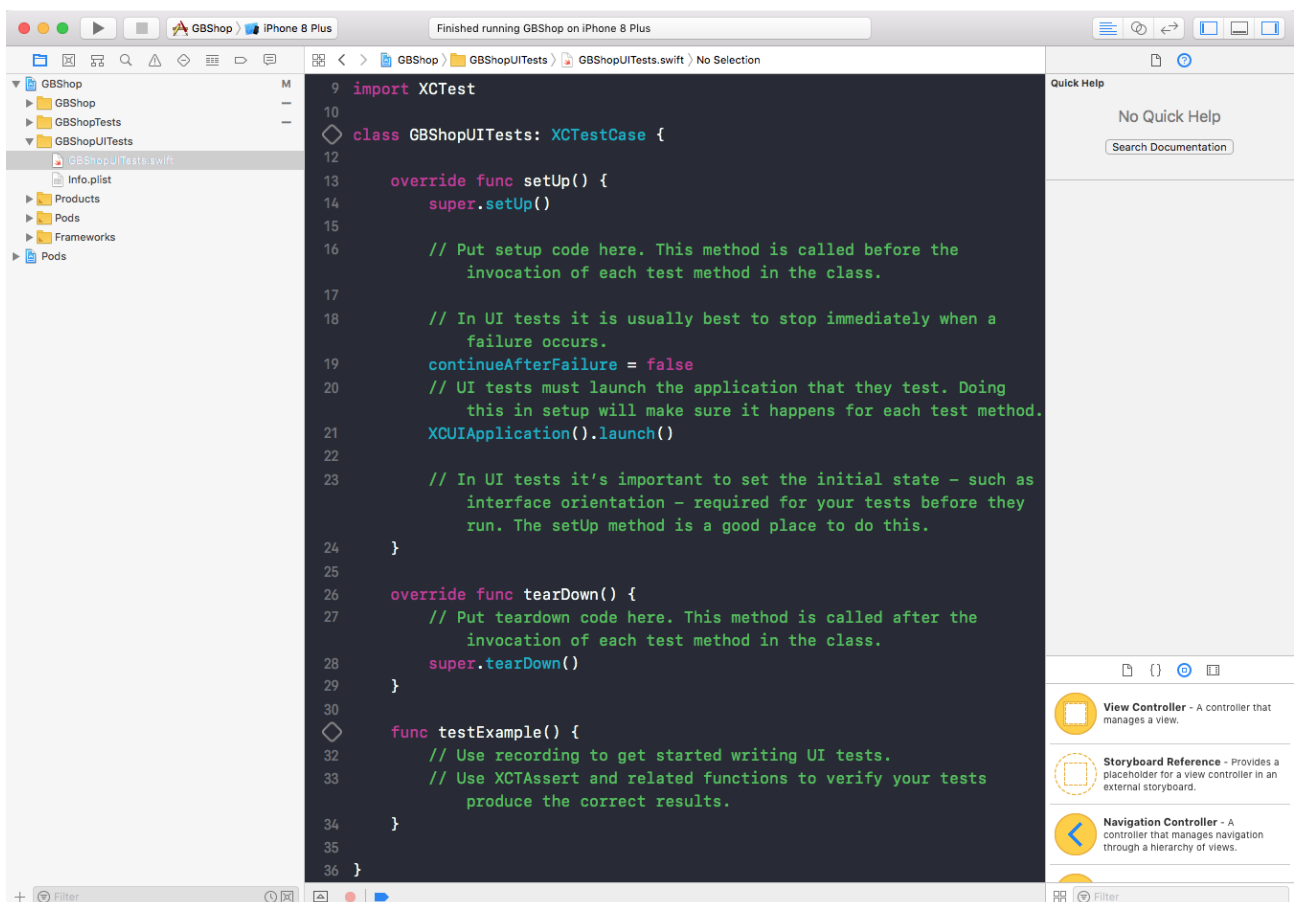
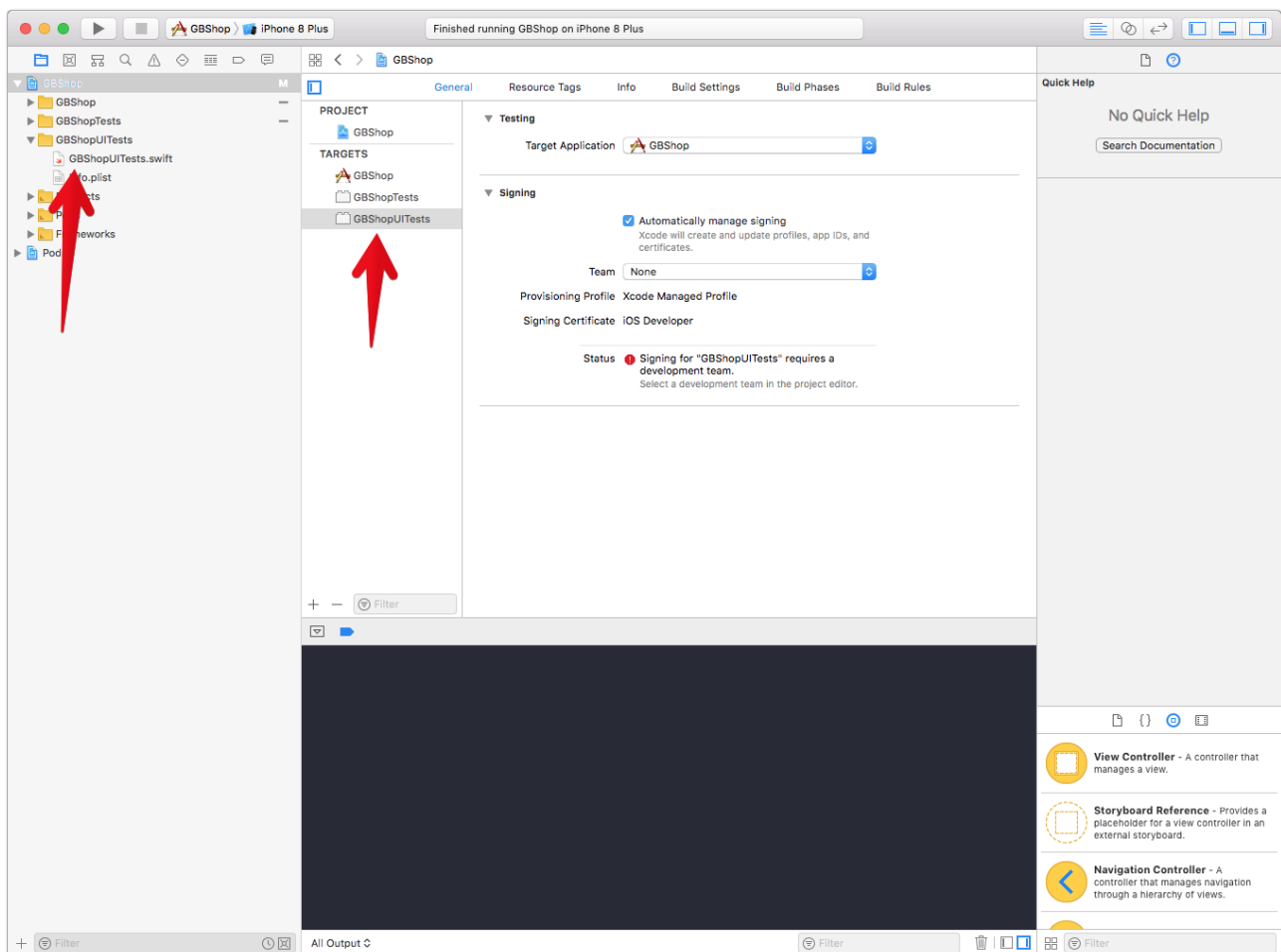
☒ Include UI Tests

Cancel Previous Next

Или просто создать цель из шаблона **iOS UI Testing Bundle**:



Будут созданы новая цель сборки проекта и каталог с примером UI-теста.



Разберемся с примером создания UI-тестов. Есть автоматически добавленный класс **GBShopUITests**, наследуемый от класса **XCTestCase**, уже знакомого нам по модульным тестам. Структура файла:

- **import XCTest** — это фреймворк тестирования, предоставляющий удобный API-интерфейс для разработки тестов;
- Метод **setUp()** — выполняется перед началом каждого теста, в нем можно инициализировать все объекты, которые понадобятся для теста. В теле этого метода есть небольшое различие с модульными тестами — наличие следующих инструкций:

```
continueAfterFailure = false
```

Назначение — остановить выполнение теста при первом сбое. В основном, это правильная конфигурация, поскольку каждый шаг в UI-тесте обычно зависит от успеха предыдущего, и если один шаг не удался, все последующие тесты также терпят неудачу.

```
XCUIApplication().launch()
```

Создание экземпляра **XCUIApplication** и его старт. UI-тесты пользовательского интерфейса должны запускать приложение, которое они тестируют, и применение **setUp()** гарантирует его запуск для каждого UI-теста.

- Метод **tearDown()** — выполняется после окончания каждого теста, в нем можно уничтожить все использованные объекты;
- Пример пустого метода формирования теста **testExample()**;
- Кнопки запуска тестов.

UI-тесты отличаются от модульных. Последние для работы получают доступ к коду (создают экземпляры классов, передают параметры в методы). UI-тесты работают с приложением, не обращаясь к коду, а используя эмуляцию пользовательских действий.

UI-тесты основаны на реализации трех новых классов:

- **XCUIApplication** — прокси для тестируемого приложения;
- **XCUIElement** — элемент пользовательского интерфейса;
- **XCUIElementQuery** — запрос для определения элементов пользовательского интерфейса.

UI-тесты в основном работают через события эмуляции пользовательских действий, совершаемых над элементами интерфейса, и ответов на запросы состояний этих элементов.

Работа UI-теста состоит из следующих этапов:

- запрашивается нахождение графического элемента;
- ожидается поведение используемого элемента;
- выбирается элемент и проверяется, соответствует ли его состояние ожидаемому результату UI-теста.

То есть с помощью запроса **XCUIElementQuery** находим **XCUIElement**, далее проводим с ним нужное событие и используем методы **XCTAssert...()** и фреймворка **XCTest**, чтобы сравнить состояние **XCUIElement** с ожидаемым эталонным состоянием.

В конце работы UI-тесты, как и модульные, предоставляют отчеты, содержащие исчерпывающую информацию о результатах тестирования UI, включая моментальные снимки состояния пользовательского интерфейса при ошибках тестирования.

Для чего используются UI-тесты?

Тестирование пользовательского интерфейса — отличный способ убедиться, что наиболее важные взаимодействия с пользовательским интерфейсом отработывают, как и задумывалось. Также позволяют проверить, что все продолжает действовать корректно после добавления новой функциональности или рефакторинга приложения.

Прежде чем писать тесты, важно понять — что нужно проверить?

В общем плане UI-тесты должны охватывать:

- Запуск приложения;
- Работоспособность UI-компонентов — правильное поведение нажатия на активные компоненты, раскрываемость ячеек, переходы;
- Целостность истории пользователя — отработка сценария пользовательской операции от начала до конца. Например, заказ товара: поиск, выбор, оплата, получение чека о покупке;
- Правильности исправления ошибок, проведения рефакторинга.

Как писать UI-тесты?

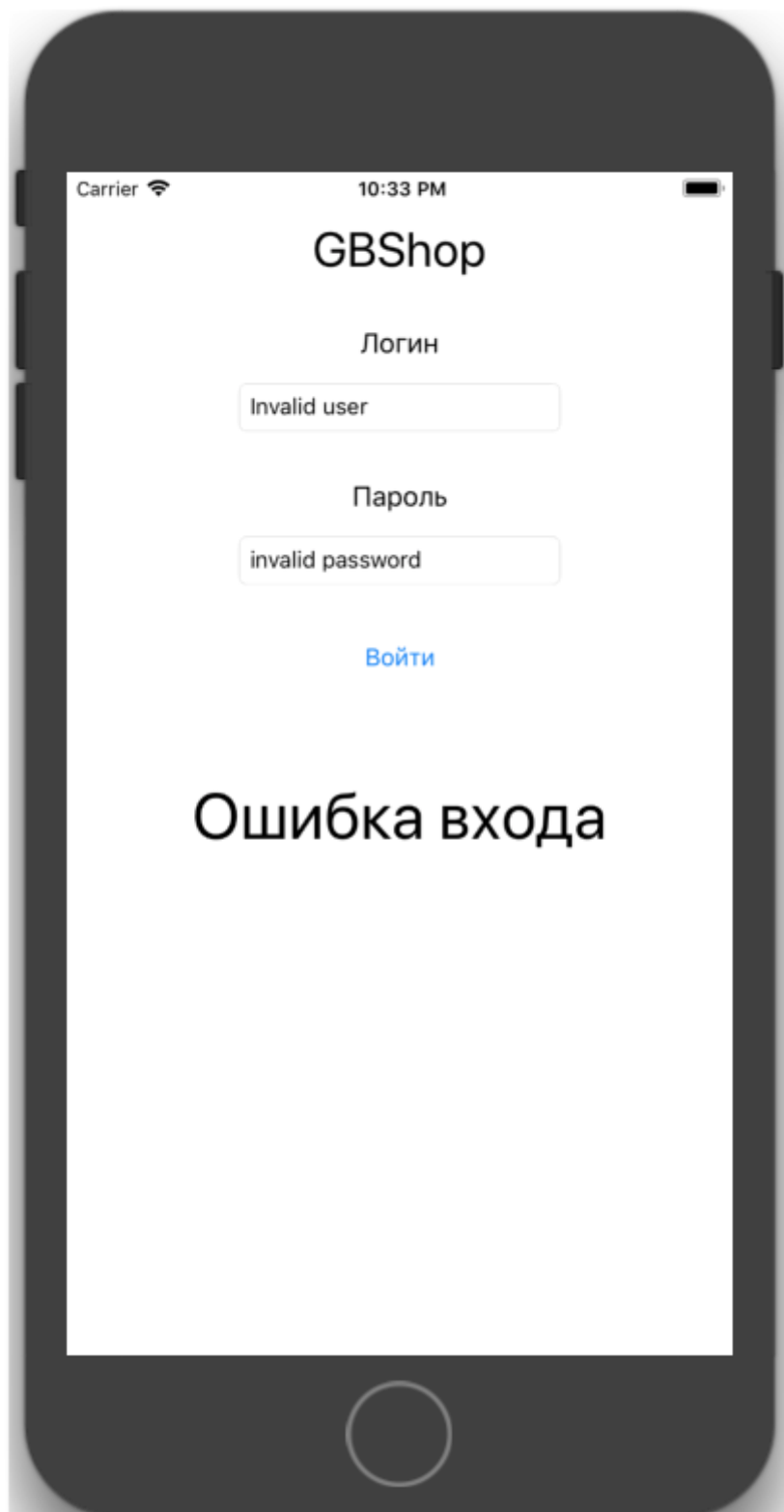
Для написания UI-теста, как и для модульного, нужно создать метод, реализующий необходимые проверки, который будет ассоциирован с нужным тестовым случаем.

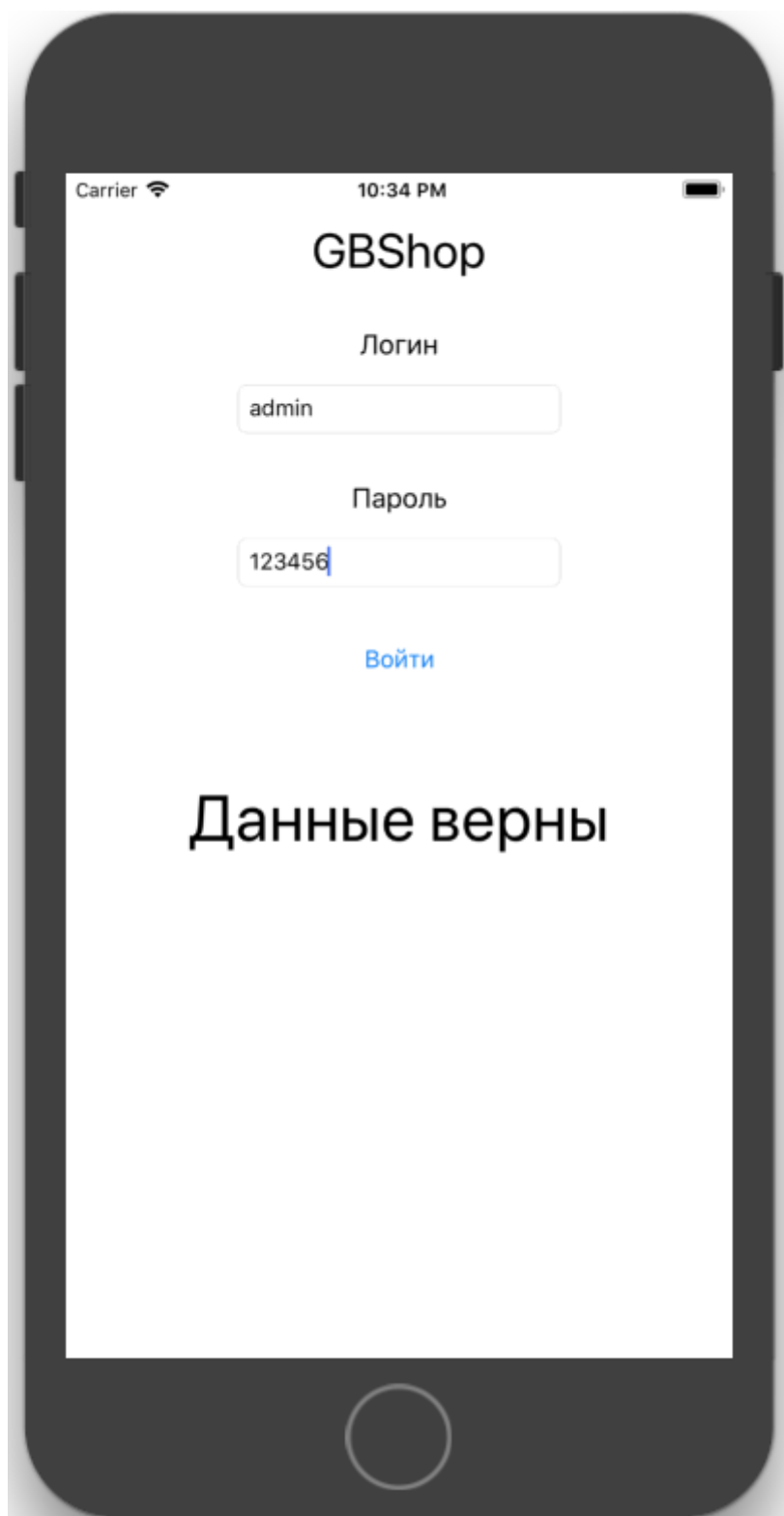
Сформировать тело этого метода можно либо с помощью ручного написания кода, либо с использованием режима записи, который добавит в нужное место метода автоматически сгенерированный код действий, совершенных вами в запущенном приложении и записанных.

Режим записи

UI-тесты включают запись действий с пользовательским интерфейсом. Это дает возможность генерировать код, который можно расширить для реализации тестов UI.

Продемонстрируем работу режима записи на примере тестирования функциональности входа в магазин. Предположим, есть форма входа с полями ввода логина/пароля, кнопкой входа и меткой под этой кнопкой, в которой будет отображаться результат входа.





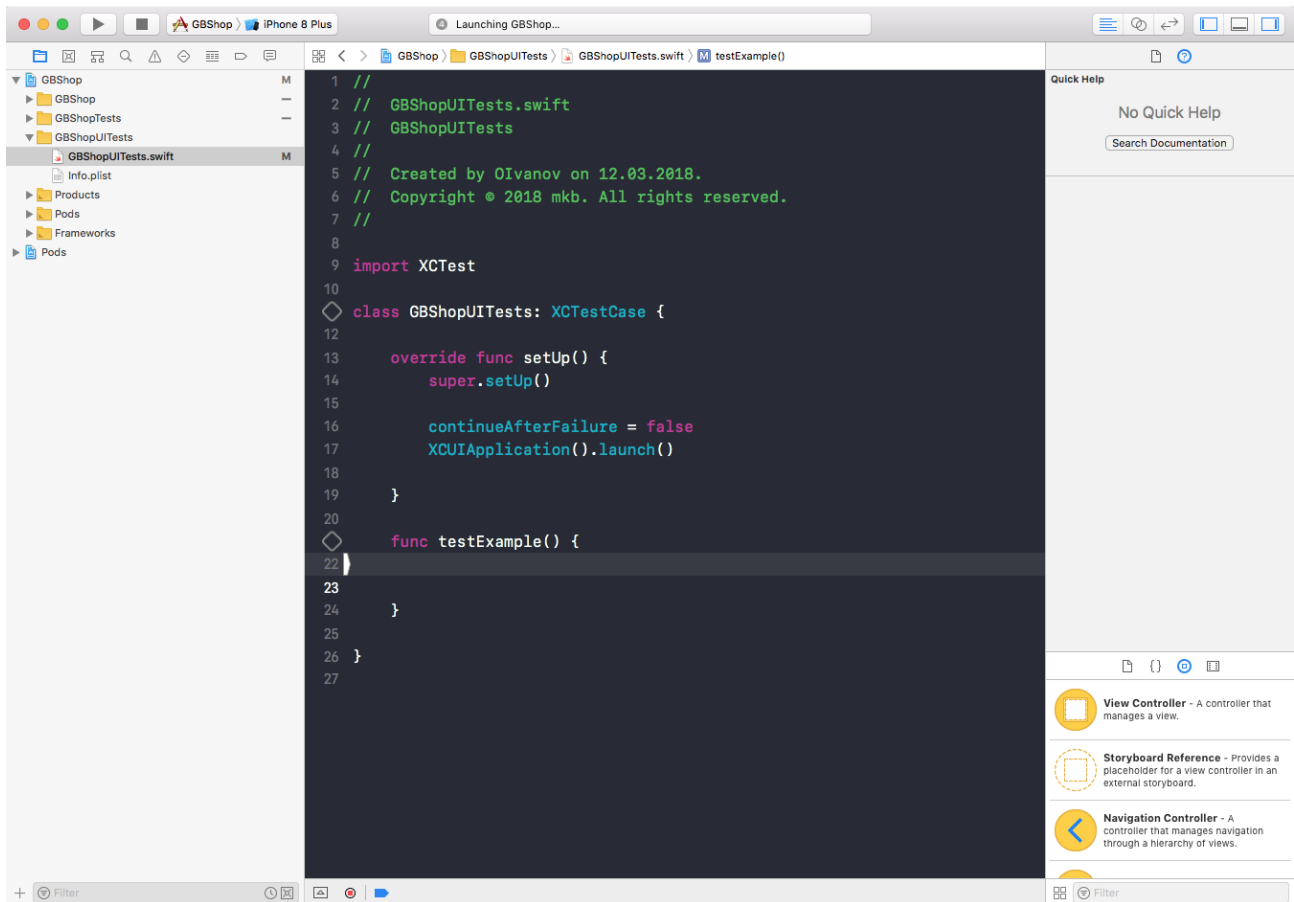
Приступим к первому тесту на успешный вход.

Шаги UI-теста:

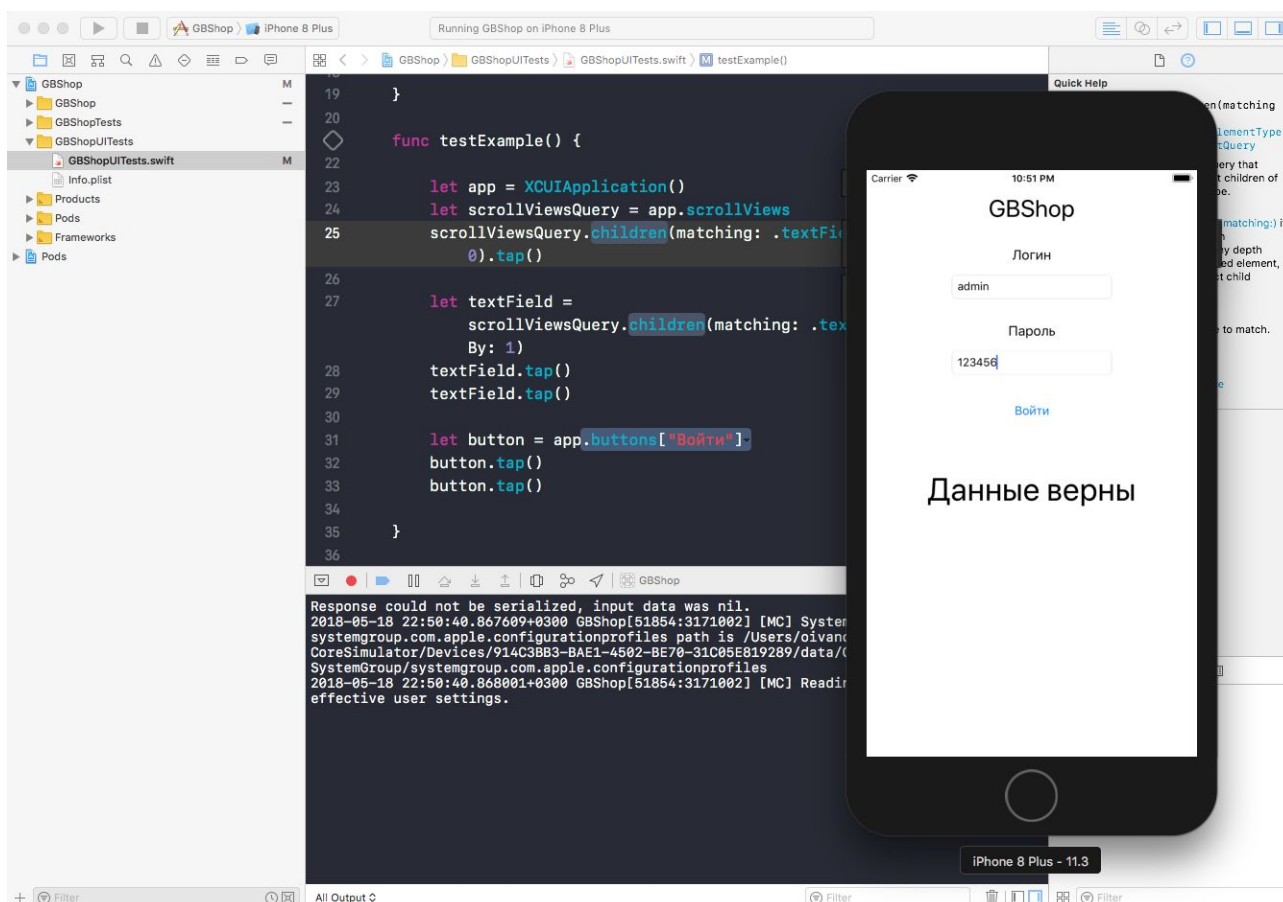
- Пользовательское действие — выбор поля ввода «Логин» и ввод верного логина «admin»;
- Пользовательское действие — выбор поля ввода «Пароль» и ввод верного пароля «123456»;
- Пользовательское действие — нажатие на кнопку «Вход»;

- Анализ ожидаемого текста «Данные верны» в результирующей метке.

Зафиксируем пользовательские действия с помощью режима записи. Для этого поставим курсор в начало тела метода теста **testSuccess()**, нажмем кнопку, как показано ниже, и совершим нужные пользовательские действия — прощелкаем мышкой элементы и введем с клавиатуры верные логин и пароль:



В результате получим автоматически созданный код наших действий.



Ручное редактирование

Данный код в большинстве случаев несовершенен и требует редактуры. Также добавим код на проверку состояния результирующей метки.

```
class UITestsUITests: XCTestCase {

    var app: XCUIApplication!

    override func setUp() {
        super.setUp()
        continueAfterFailure = false

        app = XCUIApplication()
        app.launch()
    }

    func testSuccess() {

        let scrollViewsQuery = app.scrollViews
        let loginTextField = scrollViewsQuery.children(matching:
        .textField).element(boundBy: 0)
        loginTextField.tap()
        loginTextField.typeText("admin")

        let passwordTextField = scrollViewsQuery.children(matching:
        .textField).element(boundBy: 1)
        passwordTextField.tap()
        passwordTextField.typeText("123456")
    }
}
```

```

        let button = scrollViewsQuery.buttons["Войти"]
        button.tap()

        let resultLabel = scrollViewsQuery.staticTexts["Данные верны"]
        XCTAssertNotNil(resultLabel)
    }
}

```

Добавим тест на проверку неуспешного входа с выносом общего кода:

```

import XCTest

class UITestsUITests: XCTestCase {

    var app: XCUIApplication!
    var scrollViewsQuery: XCUIElementQuery!

    override func setUp() {
        super.setUp()
        continueAfterFailure = false

        app = XCUIApplication()
        app.launch()
        scrollViewsQuery = app.scrollViews
    }

    func testSuccess() {
        enterAuthData(login: "admin", password: "123456")

        let resultLabel = scrollViewsQuery.staticTexts["Данные верны"]
        XCTAssertNotNil(resultLabel)
    }

    func testFail() {
        enterAuthData(login: "user", password: "password")

        let resultLabel = scrollViewsQuery.staticTexts["Ошибка входа"]
        XCTAssertNotNil(resultLabel)
    }

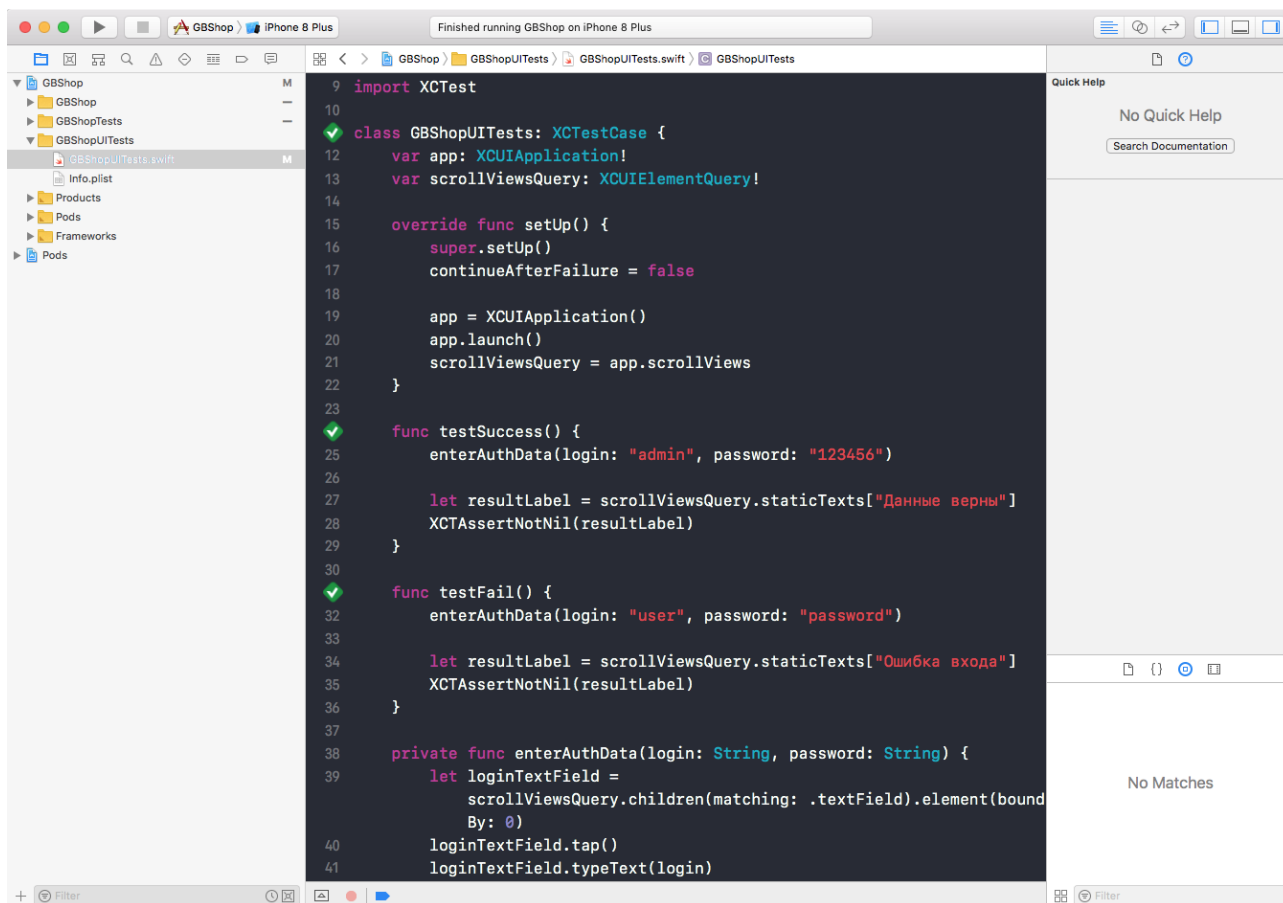
    private func enterAuthData(login: String, password: String) {
        let loginTextField = scrollViewsQuery.children(matching:
.textField).element(boundBy: 0)
        loginTextField.tap()
        loginTextField.typeText(login)

        let passwordTextField = scrollViewsQuery.children(matching:
.textField).element(boundBy: 1)
        passwordTextField.tap()
        passwordTextField.typeText(password)

        let button = scrollViewsQuery.buttons["Войти"]
        button.tap()
    }
}

```

В результате запуска теста увидим долгожданные «зеленые полоски», свидетельствующие об успешном выполнении тестов. При запуске тестов можно наблюдать интересную автоматическую работу: ввод данных, переходы по графическим элементам, выполняющиеся в запущенном приложении без нашего вмешательства.



Использование Accessibility для поиска UI-компонентов

UI-тесты основываются на двух основных технологиях: **XCTest framework** и **Accessibility**.

Accessibility — технология, позволяющая пользователям с ограниченными возможностями пользоваться iOS и macOS. Accessibility включает набор данных об интерфейсе, который используют пользователи для успешной работы с приложением. UI-тесты используют эти данные для выполнения своих функций.

Каждый **UIView** может иметь набор свойств Accessibility. Для тестирования пользовательского интерфейса больше всего интересен **accessibilityIdentifier** и **accessibilityLabel**. UI-тесты выполняют итерацию по графическим элементам и ищут Accessibility-свойства: **accessibilityIdentifier** и **accessibilityLabel**. Далее найденные компоненты используются как часть теста пользовательского интерфейса.

Используя метаданные Accessibility таким образом, можно создать более надежный UI-тест, который не зависит от содержимого текста в графических элементах.

Рекомендация: **UIView**-объекты могут быть запрошены с использованием **accessibilityIdentifier** или **accessibilityLabel**, но лучше использовать **accessibilityIdentifier**. **accessibilityLabel** — это свойство iOS Accessibility, используемое для доступа к тексту, который будет читаться для пользователя с

ограниченными возможностями. Оно может меняться во время выполнения для графических элементов.

Изменим приведенный выше пример — используем **accessibilityIdentifier**. Accessibility-свойства можно задавать как программно, так и на **storyboard** или в xib-файле.

```
class LoginController: UIViewController {

    @IBOutlet weak var scrollView: UIScrollView!
    @IBOutlet weak var loginInput: UITextField!
    @IBOutlet weak var passwordInput: UITextField!
    @IBOutlet weak var resultLabel: UILabel!
    @IBOutlet weak var enterButton: UIButton!

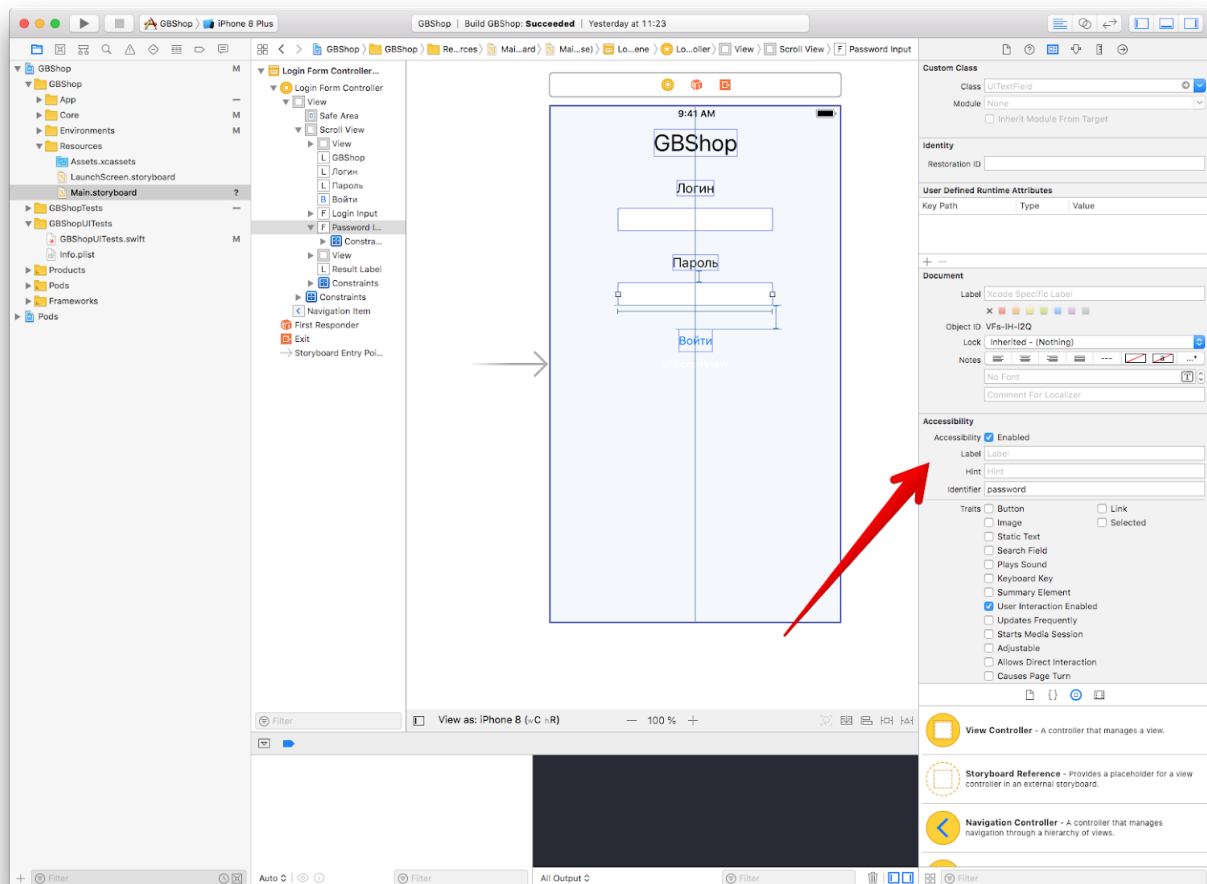
    override func viewDidLoad() {
        super.viewDidLoad()

        loginInput.isAccessibilityElement = true
        loginInput.accessibilityIdentifier = "login"

        resultLabel.isAccessibilityElement = true
        resultLabel.accessibilityIdentifier = "result"

        enterButton.isAccessibilityElement = true
        enterButton.accessibilityIdentifier = "enter"

        ...
    }
}
```



Флаг **isAccessibilityElement** показывает, что данный графический элемент доступен для Accessibility.

Перепишем тесты с использованием **accessibilityIdentifier**.

```
class GBShopUITests: XCTestCase {
    var app: XCUIApplication!
    var scrollViewsQuery: XCUIElementQuery!

    override func setUp() {
        super.setUp()
        continueAfterFailure = false

        app = XCUIApplication()
        app.launch()
        scrollViewsQuery = app.scrollViews
    }

    func testSuccess() {
        enterAuthData(login: "admin", password: "123456")

        let resultLabel = scrollViewsQuery.staticTexts["Данные верны"]
        XCTAssertNotNil(resultLabel)
    }

    func testFail() {
        enterAuthData(login: "user", password: "password")

        // Используем accessibilityIdentifier - "result"
        let resultLabel = scrollViewsQuery.staticTexts["result"]
    }
}
```

```

        XCTAssertNotNil(resultLabel)
    }

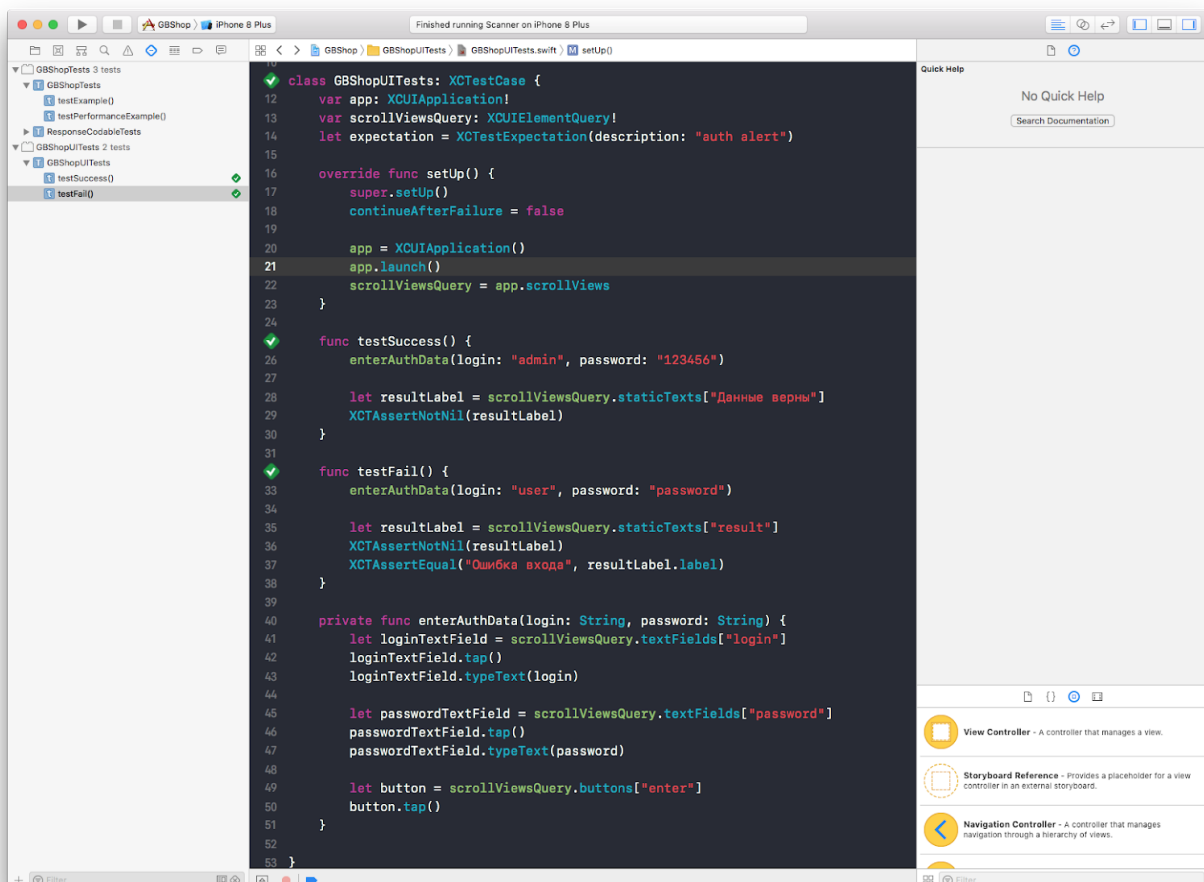
    private func enterAuthData(login: String, password: String) {
        // Используем accessibilityIdentifier - "login"
        let loginTextField = scrollViewsQuery.textFields["login"]
        loginTextField.tap()
        loginTextField.typeText(login)

        // Используем accessibilityIdentifier - "password"
        let passwordTextField = scrollViewsQuery.textFields["password"]
        passwordTextField.tap()
        passwordTextField.typeText(password)

        // Используем accessibilityIdentifier - "enter"
        let button = scrollViewsQuery.buttons["enter"]
        button.tap()
    }
}

```

Запускаем тесты и наблюдаем их успешное выполнение.



Работа с UIAlertController

Часто в приложении показываются диалоги, как системные, так и внутренние. Работу с ними организуют с помощью метода **addUIInterruptionMonitor** фреймворка **XCTest**.

Пример — показ окна сообщения с результатом аутентификации:

```
class LoginFormController: UIViewController {

    @IBAction func loginButtonPressed(_ sender: Any) {
        let title: String
        let message: String

        if LoginManager.check(login: loginInput.text, password:
passwordInput.text) {
            title = ""
            message = "Добрый день, пользователь!"
        } else {
            title = "Ошибка"
            message = "Неверный логин или пароль"
        }

        let alter = UIAlertController(title: title, message: message,
preferredStyle: .alert)
        alter.addAction(UIAlertAction(title: "Ok", style: .cancel, handler:
nil))
        present(alter, animated: true, completion: nil)

        resultLabel.text = message

        ...
    }

    ...
}
```

Доработаем тесты с использованием метода **addUIInterruptionMonitor**.

```
class GBShopUITests: XCTestCase {
    var app: XCUIApplication!
    var scrollViewsQuery: XCUIElementQuery!

    override func setUp() {
        super.setUp()
        continueAfterFailure = false

        app = XCUIApplication()
        app.launch()
        scrollViewsQuery = app.scrollViews
    }

    func testSuccess() {
        enterAuthData(login: "admin", password: "123456")
        checkAuth(message: "Добрый день, пользователь!")
    }

    func testFail() {
```

```

        enterAuthData(login: "user", password: "password")
        checkAuth(message: "Неверный логин или пароль")
    }

    private func enterAuthData(login: String, password: String) {
        let loginTextField = scrollViewsQuery.textFields["login"]
        loginTextField.tap()
        loginTextField.typeText(login)

        let passwordTextField = scrollViewsQuery.textFields["password"]
        passwordTextField.tap()
        passwordTextField.typeText(password)

        let button = scrollViewsQuery.buttons["enter"]
        button.tap()
    }

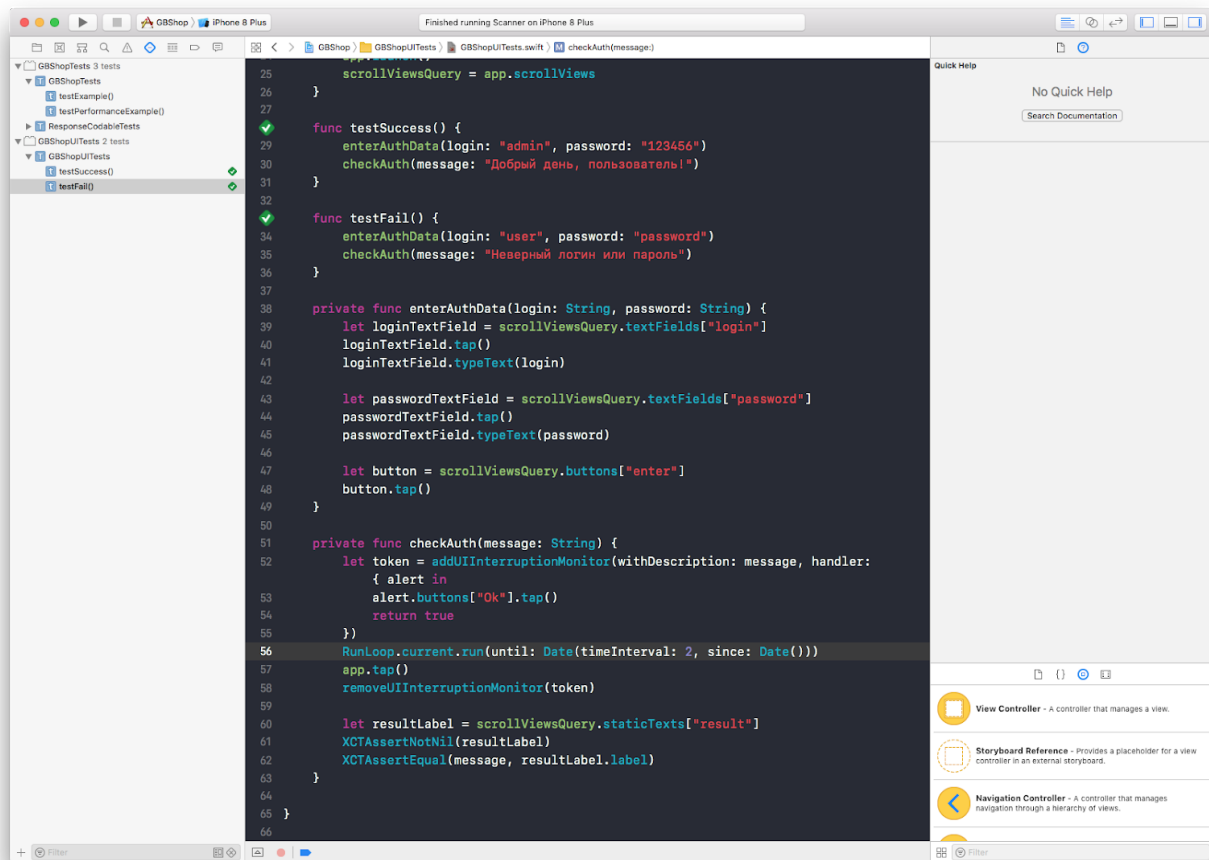
    private func checkAuth(message: String) {
        let token = addUIInterruptionMonitor(withDescription: message,
handler: { alert in
            alert.buttons["Ok"].tap()
            return true
        })
        // Диалоги находятся в другом потоке, поэтому дадим им некоторое время
        для синхронизации
        RunLoop.current.run(until: Date(timeInterval: 2, since: Date()))

        // Чтобы снова взаимодействовать с приложением
        app.tap()
        removeUIInterruptionMonitor(token)

        let resultLabel = scrollViewsQuery.staticTexts["result"]
        XCTAssertNotNil(resultLabel)
        XCTAssertEqual(message, resultLabel.label)
    }
}

```

Обратите внимание: в этом примере мы добавили вызовы методов **RunLoop.current.run()** и **app.tap()**, чтобы ожидать закрытия диалога и продолжать работать с приложением.



Передача настроек приложению

Иногда полезно передавать настройки непосредственно из теста в приложение. Для этого используется **launchArguments**.

```

override func setUp() {
    super.setUp()
    continueAfterFailure = false

    app = XCUIApplication()

    // передача "--uitesting" в приложение
    app.launchArguments.append("--uitesting")

    app.launch()
    scrollViewsQuery = app.scrollViews
}

```

Получим данную настройку в приложении. Для этого реализуем следующий код в **AppDelegate**:

```

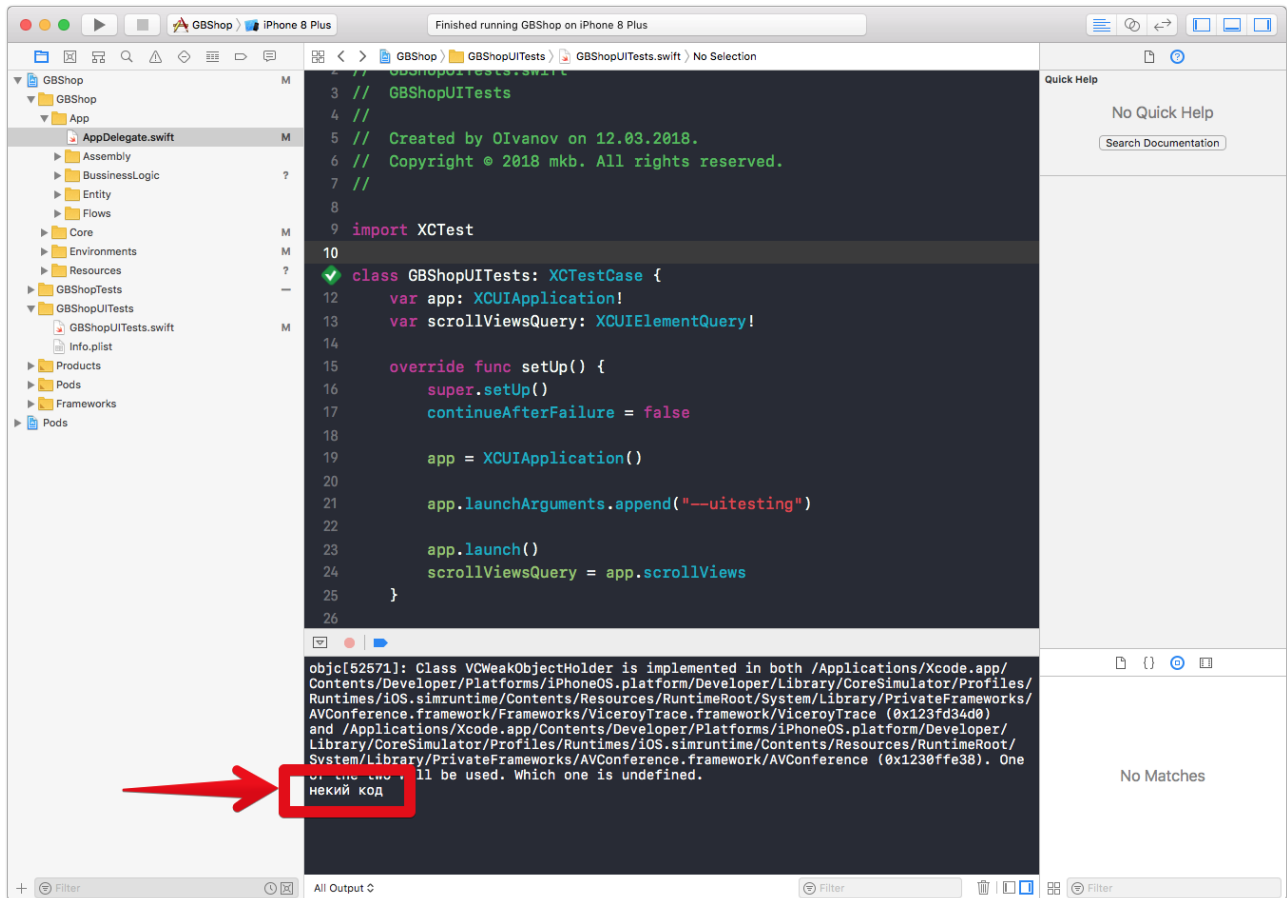
func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

    if CommandLine.arguments.contains("--uitesting") {
        print("некий код")
    }
}

```

```
    return true
}
```

После запуска теста увидим лог:



По данной настройке можем, например, удалить параметры в **UserDefaults**, очистить БД **Realm**. Можем определить состояния данных внутри приложения, делая проверку на эту настройку.

Практическое задание

1. Реализовать экраны корзины.
2. Реализовать добавление в корзину.
3. Реализовать покупку (оплату заказа).
4. Покрыть UI-тестами сценарий авторизации.

Дополнительные материалы

1. [User Interface Testing](#).

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Getting started with Xcode UI testing in Swift.](#)