нетология

Тестирование, Unit тесты (TDD)

Алексей
Жембловский

Алексей Жембловский

iOS developer EPAM

План занятия

1. Зачем нужны тесты и какие они бывают
2. Создание unit-тестов при помощи XCTest
3. Методологии TDD и BDD
4. Создание UI-тестов при помощи XCTest

Зачем нужны тесты и какие они бывают

Основная задача тестирования — это проверка корректности работы приложения на этапе разработки.

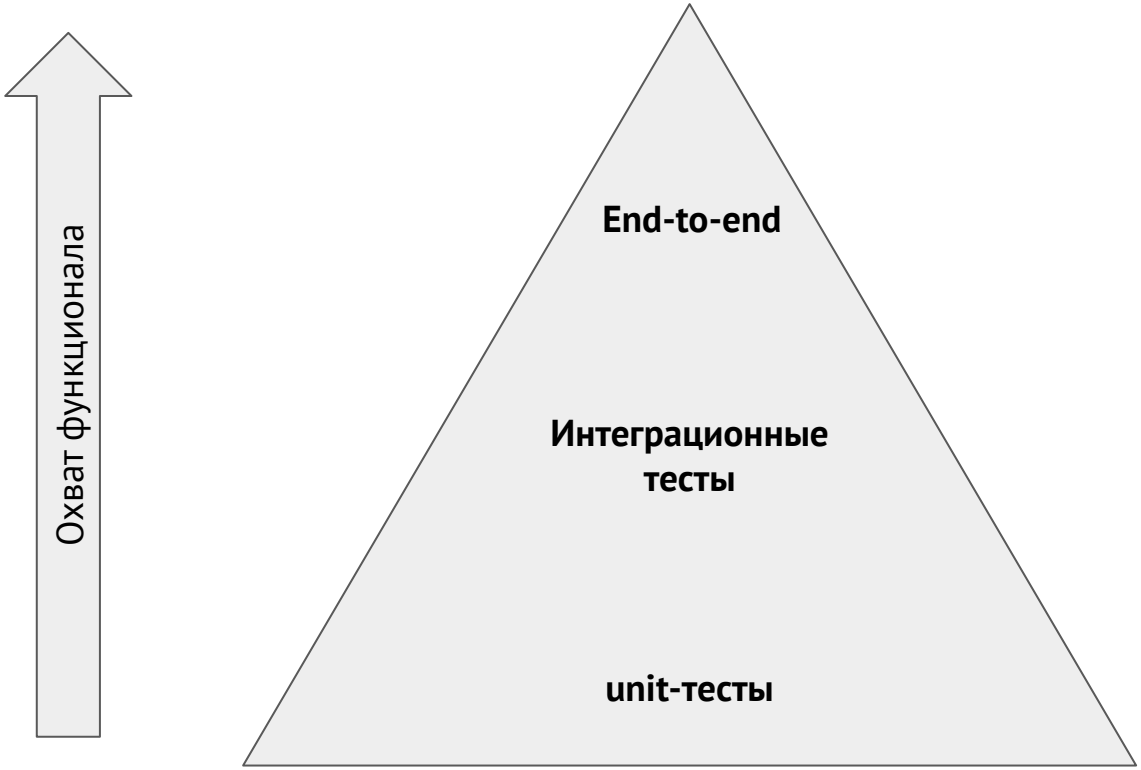
В контексте мобильной разработки различают следующие типы тестов:

- **Unit-тесты** — тестирование отдельных объектов приложения, например классов.
- **Интеграционные тесты** — тестирование взаимодействия логической группы объектов.
- **End-to-end тесты** — тестирование приложения целиком, в том же виде, в котором его будет видеть конечный пользователь.

Распределение тестов

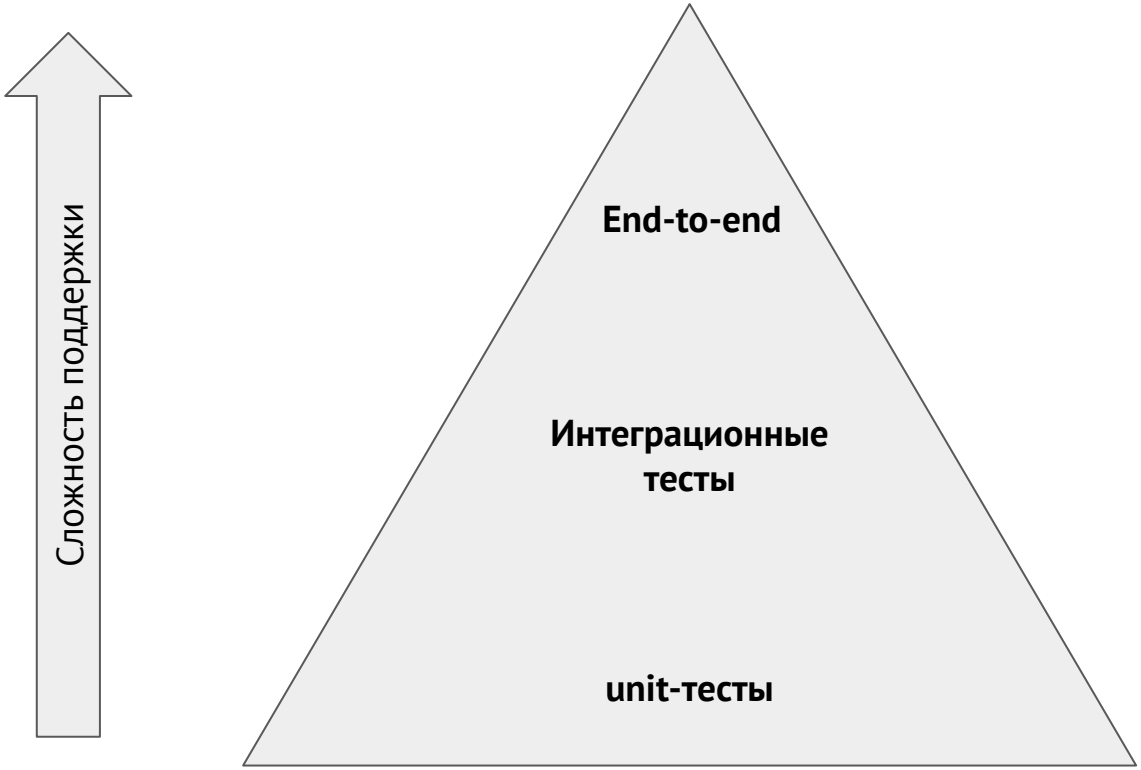
Объем тестов падает от основания пирамиды к ее верхушке. Происходит это ввиду представленных критериев.

Охват функционала растет от основания к верхушке.



Распределение тестов

Сложность поддержки тестов также растет от основания к верхушке.



Распределение тестов

Скорость работы растет от основания к верхушке.

unit-тесты

Совокупность требований к unit-тестам можно описать акронимом FIRST:

- **Fast** — тесты должны быть быстрыми (десятки-сотни миллисекунд).
- **Independent** — тесты не должны влиять друг на друга.
- **Repeatable** — результат теста должен быть одинаковым всегда.
- **Self-validating** — тест должен быть полностью автоматизирован и не предполагать дополнительных действий со стороны разработчика.
- **Timely** — написанными своевременно.

Создание unit-тестов при помощи XCTest

Рассмотрим тестовый проект, который представляет собой два экрана: со списком песен и с информацией о каждой из них.

На экране информации также располагается кнопка “Share”, ее функционал мы и протестируем.

Структура, описывающая песню:

```
struct Song {
    let id: Int
    let artist: String
    let title: String
    let artwork: UIImage
}
```

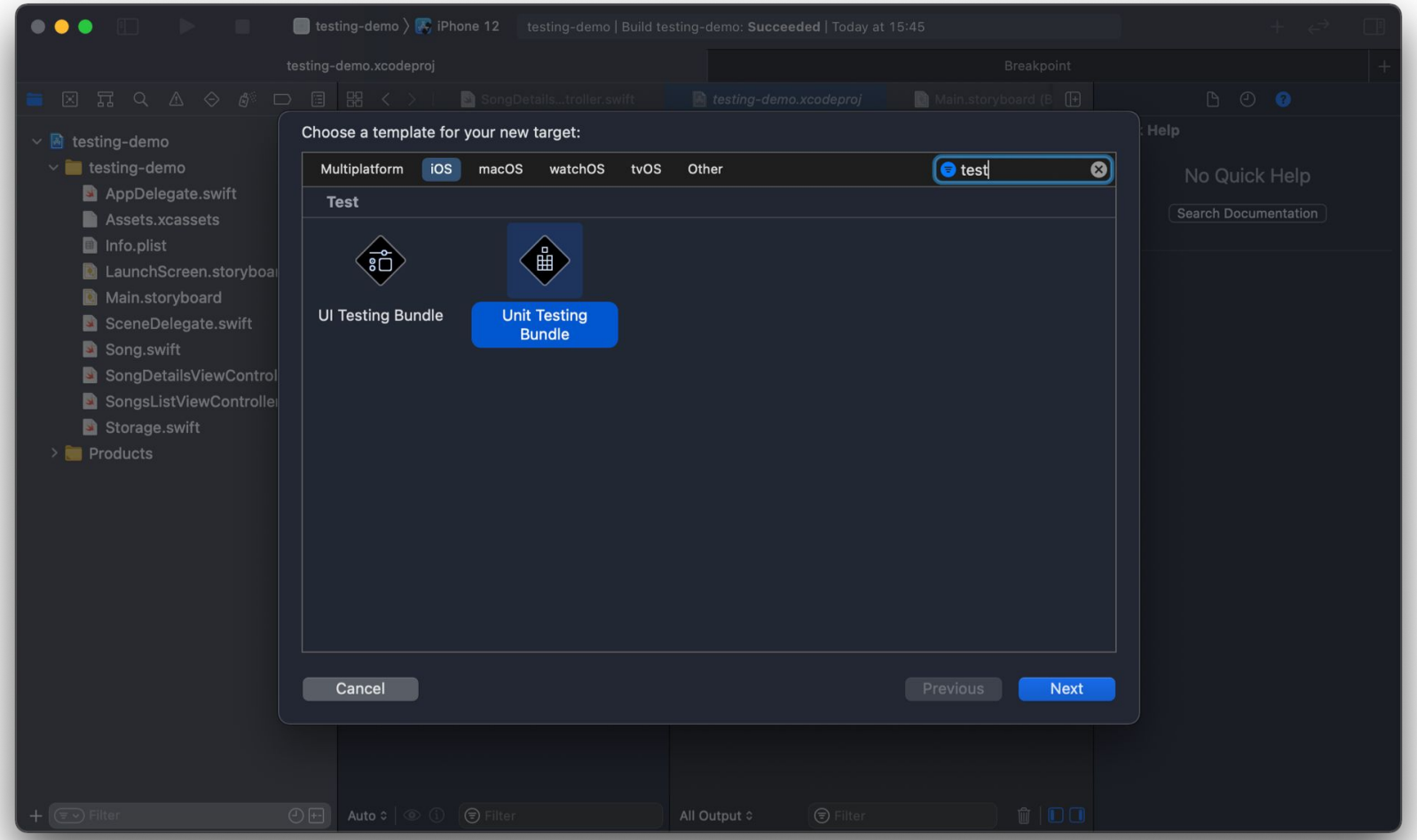
Создание unit-тестов при помощи XCTest

Экран деталей и обработка нажатия на Share:

```
class SongDetailsViewController: UIViewController {
    // ...
    @IBAction func actionShare(_ sender: Any) {
        let songUrl = URL(string: "songsappscheme://opensong?id=\(song.id)")!
        let attachments: [Any] = ["Check the song … \(songUrl)"]
        let activityViewController = UIActivityViewController(...)
        activityViewController.excludedActivityTypes = [...]
        activityViewController.completionWithItemsHandler = { _, success, _, error in
            if success {
                self.displaySuccessAlert()
            } else {
                if error != nil {
                    self.displayErrorAlert()
                } else {
                    // action cancelled
                }
            }
        }
        present(activityViewController, animated: true, completion: nil)
    }
```

Создание unit-тестов при помощи XCTest

Для начала добавим таргет unit-тестов:



Создание unit-тестов при помощи XCTest

В добавленном таргете автоматически создается сабкласс `XCTestCase`, рассмотрим его методы:

- **testXxxxx()** — эти методы представляют собой тестируемые сценарии.
- **setUp()/setUpWithError()** — методы инициализации, вызываются перед каждым сценарием.
- **tearDown()/tearDownWithError()** — метод деинициализации, которые вызываются после выполнения каждого сценария.
- Различные вариации функции **XCTAssert** — используются в тестовых сценариях для осуществления проверок.
- Асинхронные проверки осуществляются при помощи объекта **XCTTestExpectation**.

Создание unit-тестов при помощи XCTest

Вынесем в отдельный класс логику, которую мы хотим протестировать:

```
class ShareHelper {

    let detailsViewController: SongDetailsViewController

    // ...

    func shareSong(_ song: Song) {
        let songUrl = URL(string: "songsappscheme://opensong?id=\(song.id)")!
        let attachments: [Any] = ["Check the song … \(songUrl)"]
        let activityViewController = UIActivityViewController(...)
        activityViewController.excludedActivityTypes = [...]
        activityViewController.completionWithItemsHandler = { _, success, _, error
in
            if success {
                self.detailsViewController.displaySuccessAlert()
            } else {
                // ...
            }
        }
        detailsViewController.present(activityViewController ...)
    }
```

Создание unit-тестов при помощи XCTest

Попробуем написать тест, проверяющий, что метод `shareSong()` генерирует правильную URL и вызывает `presenter`:

```
class ShareHelperTests: XCTestCase {
    var shareHelper: ShareHelper!
    var detailsViewController: SongDetailsViewController!

    override func setUp() {
            super.setUp()
        detailsViewController = SongDetailsViewController()
        shareHelper = ShareHelper(detailsViewController: detailsViewController)
    }

    func testShare_veriryUrl_verirySuccessAlert() {
        let dummySong = Song(id: 1, artist: "artist", title: "title", artwork:
    nil)
        shareHelper.share(dummySong)
        // ???
    }
```

Вывод: с нынешней реализацией проверку совершить нельзя, нужно рефакторить.

Создание unit-тестов при помощи XCTest

Создадим протокол `SharePresenter` и используем его в `ShareHelper`:

```
protocol SharePresenter {
    func showShareScreen(content: String, completion: @escaping (ShareResult) -> ())
    func displaySuccessAlert()
    func displayErrorAlert()
}

class ShareHelper {

    // ...

    func shareSong(_ song: Song) {
        let songUrl = URL(string: "songsappscheme://opensong?id=\(song.id)")!
        let content = "Check the song \(song.artist) - \(song.title) in my app: \(songUrl)"
        presenter.showShareScreen(content: content) { [weak self] result in
            guard let self = self else { return }
            switch result {
            case .success:
                self.presenter.displaySuccessAlert()
            case .error:
                self.presenter.displayErrorAlert()
            case .cancelled:
                break
            }
        }
```

Создание unit-тестов при помощи XCTest

Протокол `SharePresenter` реализуется классом `SongDetailsViewController`. Но помимо этого мы создадим макет для тестов:

```
class SharePresenterMock: SharePresenter {

    var showShareScreenHandler: ((String, @escaping (ShareResult) -> ()) -> ())?
    var showShareScreenCounter: Int = 0
    var displaySuccessAlertHandler: (() -> ())?
    var displaySuccessAlertCounter: Int = 0
    var displayErrorAlertHandler: (() -> ())?
    var displayErrorAlertCounter: Int = 0

    func showShareScreen(content: String, completion: @escaping (ShareResult) -> ()) {
        showShareScreenCounter += 1
        showShareScreenHandler?(content, completion)
    }

    func displaySuccessAlert() {
        // ...
    }

    func displayErrorAlert() {
        // ...
    }
}
```

Создание unit-тестов при помощи XCTest

Напишем тест еще раз, используя макет презентера:

```
func testShare_verifyUrl_verifySuccessAlert() {
    var assignedContent: String?
    sharePresenter.showShareScreenHandler = { content, completion in
        assignedContent = content
        completion(.success)
    }

    XCTAssertEqual(sharePresenter.displaySuccessAlertCounter, 0)
    XCTAssertEqual(sharePresenter.displayErrorAlertCounter, 0)

    let dummySong = Song(id: 1, artist: "some artist", title: "some title", artwork: nil)
    shareHelper.shareSong(dummySong)

    XCTAssertEqual(assignedContent, "Check the song \(dummySong.artist) -
\(dummySong.title) in my app: songsappscheme://opensong?id=\(dummySong.id)")
    XCTAssertEqual(sharePresenter.displaySuccessAlertCounter, 1)
    XCTAssertEqual(sharePresenter.displayErrorAlertCounter, 0)
}
```

Создание unit-тестов при помощи XCTest

Теперь напишем тест, проверяющий асинхронное взаимодействие при помощи expectation:

```
func testAsyncShare_verifyAlert() {
    sharePresenter.showShareScreenHandler = { _, completion in
        DispatchQueue.main.asyncAfter(deadline: .now() + .milliseconds(100)) {
            completion(.success)
        }
    }

    let expectation = expectation(description: "Alert displayed")
    sharePresenter.displaySuccessAlertHandler = {
        expectation.fulfill()
    }

    let dummySong = Song(id: 1, artist: "some artist", title: "some title", artwork: nil)
    shareHelper.shareSong(dummySong)

    waitForExpectations(timeout: 1.0)
}
```

Создание unit-тестов при помощи XCTest

Написание и поддержка макетов — это трудоемкий, но необходимый процесс. Благодаря тому, что макеты делаются по шаблону, эту задачу можно автоматизировать при помощи кодогенерации.

Mockolo (https://github.com/uber/mockolo) — это инструмент командной строки, генерирующий макеты протоколов при помощи аннотаций.

```
/// @mockable
protocol SharePresenter {
    func showShareScreen(content: String, completion: @escaping (ShareResult)
-> ())
    func displaySuccessAlert()
    func displayErrorAlert()
}
```

Пример команды:

```
mockolo
     --sourcedirs "~/demo/testing-demo"
     --destination "~/demo/.../MockoloGenerated.swift"
     --header "@testable import testing-demo"
```

Неявные плюсы тестирования

- **написание более качественного кода** — чтобы тестирование было эффективным, объекты должны иметь четкие задачи, явные зависимости и явные интерфейсы взаимодействия.
- **документирование кода** — тесты документируют ожидаемое поведение кода, а также определяют критичные сценарии.
- **упрощение рефакторинга** — наличие тестов позволяет проверить правильность работы оптимизированного кода.

Test Driven Development

Суть методологии TDD — это написание тестов **до** написания рабочего кода. Преимущества такого подхода:

- **опять же документирование** — разработчик определяет ожидаемое поведение объекта заранее.
- **высокое покрытие кода тестами** — написание тестов невозможно отложить, потому что мы их пишем в первую очередь.
- **возможность применить парное программирование** — один разработчик пишет тесты, другой делает так, чтобы они проходили. Как следствие, реализации становятся более надежными и гибкими.

Написание тестов при помощи Quick и Nimble

Тесты в Quick наследуются от объекта `QuickSpec`, который содержит только один метод `spec()`. В нем происходит инициализация объектов и описание всех сценариев. Пользовательские истории описываются при помощи методов `describe`, `context` и `it`. Выглядит это следующим образом:

```
override func spec() {
    describe("displaying songs list") {
        beforeEach {
             // инициализация
        }
        context("when the app starts to load the list of songs") {
            it("then app should display activity indicator and hide placeholder") {
                // первый тестовый сценарий
            }
        }
        context("when loading finished successfully") {
            it("then loaded songs should be displayed in alphabetic order, activity
indicator and placeholder should be hidden") {
                // второй тестовый сценарий
            }
        }
```

Behavior Driven Development

BDD идет в вопросе документирования еще дальше. В этом подходе акцент смещается от того, **как** работает код, на то, **что** он делает.

“Что” формализуется в виде пользовательских историй, представленных в виде:

GIVEN <условие>
WHEN <действие>
THEN <результат>

Тесты, отвечающие таким требованиям, требуют поддержку дополнительных синтаксических конструкций. Здесь на помощь приходит тандем фреймворков:

- Quick (https://github.com/Quick/Quick)
- Nimble (https://github.com/Quick/Nimble).

Написание тестов при помощи Quick и Nimble

Давайте напишем тесты для списка песен, используя этот подход. Проверим, что экран списка загружает запрашивает и отображает песни из хранилища, а также показывает activity indicator во время загрузки.

За хранение песен отвечает объект SongStorage со следующим интерфейсом:

```
protocol SongStorage: AnyObject {
    func fetchSongs(completion: @escaping (Result<[Song], StorageError>) -> ())
}
```

Написание тестов при помощи Quick и Nimble

Как и в предыдущем случае вынесем тестируемую логику в отдельный объект. Назовем его `SongListViewModel`:

```
protocol SongListViewModelDelegate: AnyObject {
    func songListViewModelDidUpdate(_ viewModel: SongListViewModel)
}

class SongListViewModel {
    // ...
    func loadData() {
        isLoading = true
        shouldShowPlaceholder = false
        delegate?.songListViewModelDidUpdate(self)

        songStorage.fetchSongs { [weak self] result in
            guard let self = self else { return }
            if case .success(let songs) = result {
                self.songs = songs.sorted { $0.artist < $1.artist }
            }
            self.isLoading = false
            self.shouldShowPlaceholder = self.songs.isEmpty
            self.delegate?.songListViewModelDidUpdate(self)
        }
    }
```

Написание тестов при помощи Quick и Nimble

Конкретная реализация будет выглядеть так:

```
// ...

var storage: SongStorageMock!
var viewModel: SongListViewModel!
var viewModelDelegate: SongListViewModelDelegateMock!

beforeEach {
    storage = SongStorageMock()
    viewModelDelegate = SongListViewModelDelegateMock()
    viewModel = SongListViewModel(storage: storage)
    viewModel.delegate = viewModelDelegate
}

context("when the app starts to load the list of songs") {
    it("then app should display activity indicator and hide placeholder") {
        viewModel.loadData()
        expect(viewModelDelegate.songListViewModelDidUpdateCounter).to(equal(1))
        expect(viewModel.isLoading).to(equal(true))
        expect(viewModel.shouldShowPlaceholder).to(equal(false))
     }
}

// ...
```

UI-тестирование при помощи XCTest

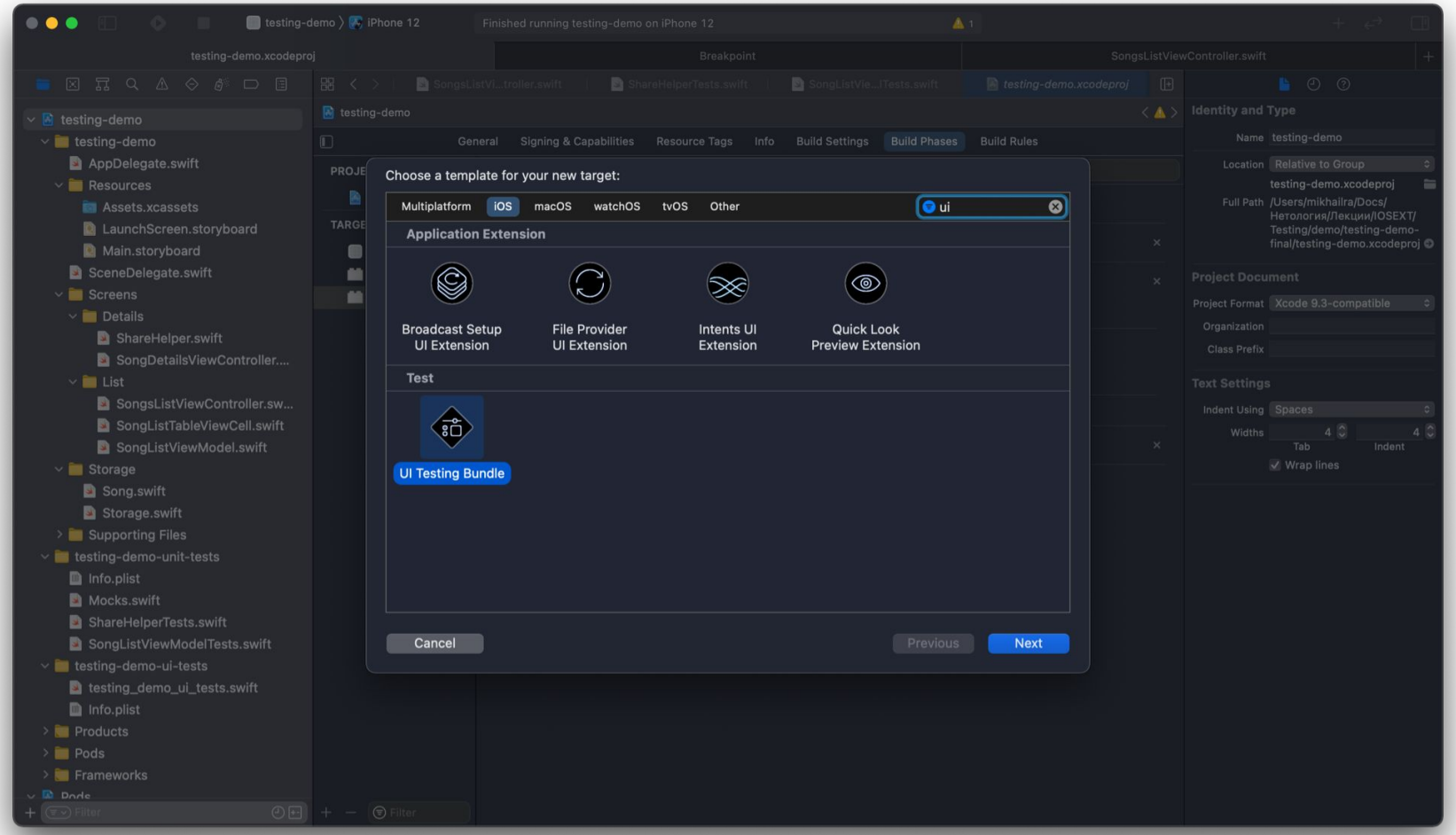
UI-тестирование позволяет нам проверить правильность работы приложения при взаимодействии с пользовательским интерфейсом. Это работает путем нахождения элементов интерфейса по их имени, генерации событий и проверки изменений их состояния.

UI-тестирование — это наиболее медленный и дорогой способ проверки работоспособности приложения. Применяйте его только для критичных сценариев.

Создадим тест для проверки сценария шэринга в демо-проекте.

UI-тестирование при помощи XCTest

Для начала добавим таргет UI-тестов:



UI-тестирование при помощи XCTest

Как и в случае с unit-тестами, Xcode автоматически создаст сабкласс `XCTestCase`. Рассмотрим реализованный сценарий:

```
func testNavigationFromListToDetailsAndShare() {
    // 1. Тест начинается с запуска приложения
    let app = XCUIApplication()
    app.launch()

    // 2. Ищем ячейку с песней в списке
    let songCell = app.staticTexts["Hiatus Kaiyote"]
    XCTAssert(songCell.waitForExistence(timeout: 5.0))

    // 3. Тапаем на нее и проверяем переход на экран деталей находя кнопку Share
    songCell.tap()
    let shareButton = app.staticTexts["Share"]
    XCTAssert(shareButton.exists)

    // 4. Тапаем на Share и ждем открытие activity list view
    shareButton.tap()
    let activityList = app.otherElements["ActivityListView"]
    XCTAssert(activityList.waitForExistence(timeout: 5.0))

    // ...
}
```

Итоги

- Мы поговорили о тестах и о том, как они могут быть полезны в процессе разработки. Тесты не являются обязательными, но расширение команды и масштабирование проекта без них значительно усложняется.
- Если вы нацелены использовать тесты, то необходимо строить архитектуру вашего проекта с учетом этого факта. Как следствие, с внедрением тестов ваш код становится чище.
- Необходимо соблюдать правильный баланс между различными типами тестов, чтобы этап тестирования был быстрым, и его можно было интегрировать в рабочий процесс.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** учебной группы.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Алексей Жембловский