курс | development | 12 недель

iOS: разработка приложений с 0

r_d

iOS программа курса 20 занятий

1
Swift: начало

2
ООП: основы

3
Создание iOS-приложения в Xcode

4
Создание интерфейса iOS-приложения

5
Динамические интерфейсы, часть 1

6
Динамические интерфейсы, часть 2

iOS программа курса 20 занятий

7

Динамические интерфейсы, часть 3

8

Навигация в приложении, часть 1

9

Навигация в приложении, часть 2

10

Анимации в iOS

11

Работа с памятью в iOS

12

Многозадачность в iOS, часть 1

iOS программа курса 20 занятий

13
Многозадачность в iOS, часть 2

14
Дебаг iOS-приложения

15
Тестирование

16
Хранение данных в приложении

17
Работа с сетью в приложении

18
Сборка приложения

iOS программа курса 20 занятий

19

Современные архитектуры для iOS приложений

20

Защита курсовых проектов

Тестирование

- Что такое юнит-тестирование
- Зачем писать юнит-тесты
- Как писать юнит-тесты

r_d

Что такое юнит-тестирование

«Модульное тестирование (или юнит-тестирование) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.» (с)

Что такое юнит-тестирование

- Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.
- Например, обновить используемую в проекте библиотеку до актуальной версии можно в любой момент, прогнав тесты и выявив несовместимости.

Что такое юнит-тестирование

- Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

- Важно: unit тесты пишут разработчики!

Зачем писать юнит-тесты

Ради чего мы пишем юнит-тесты?

- поощрение изменений
- упращение интеграции
- документирование кода
- отделение интерфейса от реализации

Зачем писать юнит-тесты

Что говорит нам Apple о юнит-тестах? С их помощью мы находим:

- ошибки в логике работы программы
- неисправности в пользовательском интерфейсе
- проблемы со скоростью работы программы

Как писать юнит-тесты

- Для написанию любого юнит-теста сначала нам нужно определить место, в котором мы можем этот самый тест написать.
- Множество юнит-тестов, которые связаны между собой общей логикой тестирования описываются в специальных подклассах, создаваемых нами.
- Каждый подкласс — наследник класса XCTestCase.

Как писать юнит-тесты

Для написания новых юнит-тестов необходимо:

- создать новый подкласс XCTestCase и определить файл в test target
- создать один или несколько спец. методов данного подкласса
- добавить одну или более проверок внутри каждого метода

Как писать юнит-тесты

```
class TableValidationTests: XCTestCase {
    /// Tests that a new table instance has zero rows and columns.
    func testEmptyTableRowAndColumnCount() {
        let table = Table()
        XCTAssertEqual(table.rowCount, 0, "Row count was not zero.")
        XCTAssertEqual(table.columnCount, 0, "Column count was not zero.")
    }
}
```

Как писать юнит-тесты

- Все юнит-тесты и их соотвествующие тест-кейс подклассы можно найти в автогенерируемом отчёте. В нём описывается вся информация об успешности (либо не успешности) запуска каждого отдельного юнит-теста.
- Чтобы при поиске необходимого юнит-теста в коде, а также в отчёте, было легче понять, какие тесты за что отвечают, необходимо придерживаться простой логики при наименовании тест-кейс подклассов.
- Как правило XCTestCase подклассы называют общей темой того, что мы тестируем.
- Примеры: TableValidationTests, NetworkReachabilityTests, или JSONParsingTests.

Как писать юнит-тесты

- Самой важной составляющей любого юнит-теста является проверка какого-то определенного условия, подтверждающего корректность логики работы вашего кода.
- Существует специальный набор XCTAssert функций для проверки различных условий.
- Например, юнит-тест из примера содержит в себе 2 вызова функции XCTAssertEqual(_:_:_:file:line:) для проверки правильности выполнения условия, что переменные table.rowCount и table.columnCount содержат правильные значения.

Как писать юнит-тесты

Apple также рекомендует обобщенный алгоритм написания юнит-тестов.

1. Arrange — создайте все необходимые объекты и структуры данных, которые нужны вашему ключевому объекту, который вы планируете тестировать. Замените зависимости с реальных на mock-объекты (для создания mock-объектов вместо классов, как типы, используйте протоколы).
3. Act — вызовите функцию или метод у вашего объекта, который вы тестируете. В качестве аргументов передайте те, что вы создали на предыдущем шаге.
5. Assert — используйте Test Assertions для сравнения поведения написанного вами кода с ожидаемым, которые действительно должно произойти. Если функция XCTAssert вернёт false, тест считается не пройденным.

Как писать юнит-тесты

```
class MyAPITests : XCTestCase {
  func testMyAPIWorks() {
    // Arrange: create the necessary dependencies.
    // Act: call my API, using the dependencies created above.
    XCTAssertTrue(/* … */, "The result wasn't what I expected")
  }
}
```

Как писать юнит-тесты

Как называть юнит-тесты? Пара базовых рекомендаций.

- Имя теста должно отображать конкретное требование.
- Имя теста может включать ожидаемые входные параметры и состояние, а также может включать ожидаемый результат.
- Имя теста должно быть читаемо как «утверждение» (факт, что-то свершившееся).
- Имя теста может содержать название тестируемого метода и/или класса.

Demo Time 🎉🎉🎉

спасибо

задавайте вопросы