Урок 3

Тестирование

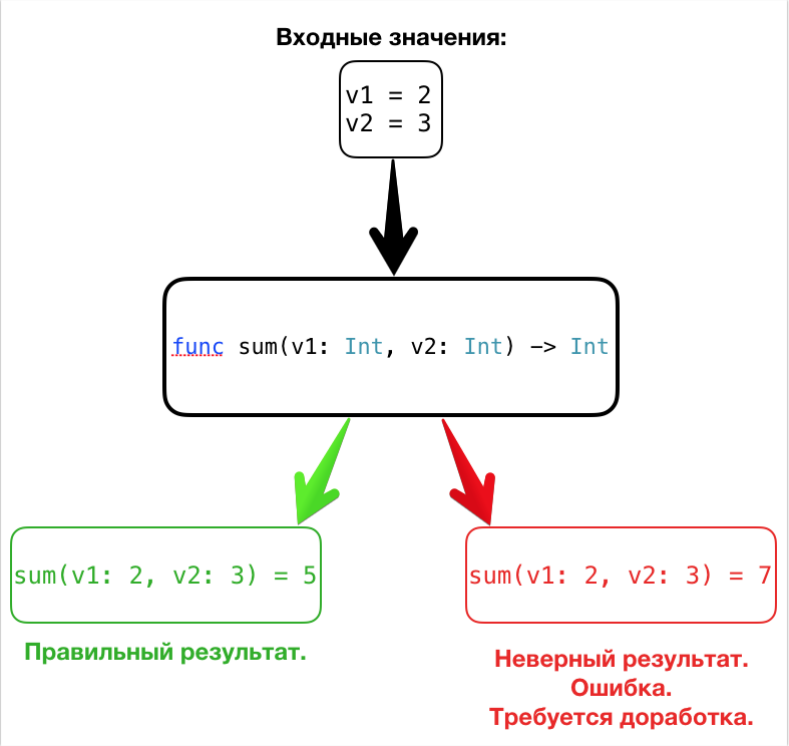
Уровни тестирования и подходы к нему. Пишем хорошие тесты.

Тесты

Писать программный код — непросто, а работающий и качественный — еще сложнее. Даже опытные программисты не в состоянии предусмотреть все возможные пути использования написанного кода и вероятные ошибки. Помогают тесты. Сделать их хороший набор не всегда легко, но уже в процессе написания тестов зачастую выявляется много проблем для первой реализации кода.

Что такое тесты?

В самом простом понимании тесты — это код, который запускает другой код и анализирует его поведение. Например, есть функция сложения с двумя входными параметрами (слагаемыми) и одним выходным (суммой). Пишем тест, который будет проверять правильность реализации функции сложения. На вход функция получит 2 и 3 — результатом должно быть 5. Если это так — функция в большинстве случаев работает корректно, если же она вернет другое число, можно с уверенностью сказать, что требуется доработка.



Назначение тестов

Тесты нужны, чтобы понять, что код написан правильно и решает поставленные задачи. С помощью набора тестов, выбранного определенным образом, проверяем соответствие между реальным и ожидаемым поведением кода. В более широком смысле — контролируем качество кода.

Также тесты позволяют зафиксировать правильность написанного кода. При внесении изменений, правок или новой зависящей функциональности тесты позволяют убедиться, что мы ничего не сломали и ранее написанный код работает правильно.

И все же наличие тестов не может гарантировать корректную работу приложения. Они только показывают, что часть кода работает правильно (и то — не всегда). Вернемся к нашему примеру: тесту на проверку сложения чисел 2 и 3. Функция отработала правильно и вернула 5. Но при проверке

на входных данных 2 и 4 получаем неверный результат 8. Ошибка! Тестирование зависит еще и от выбора обширности тестовых случаев. Но углубляться в написание всевозможных случаев тоже не стоит. Так не останется времени для самого главного — написания самого кода тестируемой функциональности.

Как понять, что тестировать, а что — нет? Одни говорят о необходимости покрытия кода на 100%, другие считают это лишней тратой ресурсов. Хороший набор тестов — это баланс между легкостью написания, поддержки и проверки функциональности. Не стоит бояться не учесть какой-нибудь тестовый случай и в результате получить ошибку. Она может быть выявлена на других этапах тестирования приложения, и далее этот случай попадает в набор тестов, который со временем будет только расширяться и становиться полнее.

Виды тестов

Unit-тесты — это тестирование маленького участка кода, реализующего определенное поведение, который часто (но не всегда) является классом. **Unit** — это или весь класс, или его фрагмент. С помощью Unit-тестов легко обнаружить и исправить ошибки. Unit-тесты основаны на изолированной проверке каждого отдельного элемента.

Интеграционные тесты проверяют взаимодействие нескольких компонентов системы: как соприкасаются друг с другом отдельные **unit**`ы. Данные тесты сложно писать, но они полезны и важны при проверке качества проекта в целом. По статистике, большинство ошибок возникает именно в местах стыковок компонентов системы. Рассмотрим пример использования интеграционных тестов. Представим, что в нашем проекте есть компоненты по работе с сетью и базами данных, и после успешного входа в приложение происходит сохранение информации в БД. Тест, который будет производить вход и проверять сохраненные данные, и будет интеграционным.

Системное тестирование — тестирование приложения в целом. Основная задача — проверка функциональности приложения в целом. При этом выявляются такие дефекты, как неверное использование ресурсов, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и подобные. Для небольших проектов это тестирование, как правило, выполняется вручную.

Распространенные подвиды тестов

UI-тесты проверяют взаимодействие компонентов системы друг с другом, имитируя действия пользователя (нажатия, переходы). UI-тесты помогают убедиться, что все обработчики правильно связаны с нужными графическими компонентами, осуществляется верный стек переходов (например, правильно работают сегвеи) и т.д. Но UI-тесты позволяют не только проверить работу графических компонентов, но и могут выступить в роли интеграционных тестов. Ведь под тестируемым контроллером может скрываться и работа с сетевым слоем, и с БД, и с другими компонентами системы.

Ручное тестирование выполняют люди, чтобы выявить дефекты в приложении. Оно проводится тестировщиками или обычными пользователями, проходящими всевозможные сценарии действий.

Виды ручного тестирования:

- **Smoke (дымовое)** — серия коротких и, как правило, бессистемных тестов. Цель — определить, что приложение запускается и не содержит явных дефектов. В большинстве случаев программисты используют этот вид ручного тестирования, запуская приложения и проверяя проделанную работу непосредственно на устройстве. Такое smoke-тестирование включает небольшое количество сценариев и предназначено для выявления явных

ошибок функциональности. Обычно smoke-тесты проводятся после обновления ПО и проверяют, что базовый функционал приложения работает;

- **Регрессионное тестирование (регресс)** — полное тестирование приложения по заранее написанным тест-кейсам (сценариям). Цель — убедиться, что внесенные изменения не повредили ранее существовавшие возможности. Например, мы выпустили новый функционал — работу с картами. Регрессионное тестирование проверит ее и протестирует всю функциональность приложения (вход, сохранение данных и так далее). Кроме того, произведет проверку исправления всех ошибок, найденных до ввода новой функциональности. Регресс — очень долгий и дорогой вид тестирования;

- **Тестирование сборки** — аналог smoke-теста, выполняемый для определенной сборки;

- **Санитарное тестирование** — подмножество регресса, тесты которого направлены на обнаружение дефектов в конкретной функции программы.

Unit-тесты

Это тот вид тестирования, который мы будем выполнять. Разберемся, какие критерии важны для теста.

Характеристики unit-тестов

Быстрота — тесты должны выполняться быстро, чтобы не влиять негативно на скорость разработки программного кода. Тесты должны оперативно предоставлять уверенность, что все работает, как надо. К тому же, если тест будет медленным, его просто никто не будет запускать, что может привести к плачевным последствиям.

Независимость — тесты должны не зависеть от того, что запускалось перед или после них, а без проблем выполняться в произвольном порядке. Например, тесты авторизированных действий требуют успешного входа в приложение. Но не надо каждый раз при написании такого теста выполнять вход. Нужно воспользоваться приемами **mock-object** (мок-объект) и **stub** (заглушка):

- **Stub** подменяет внешнюю зависимость и позволяет получить из нее данные. При этом игнорирует те, которые могут поступать из тестируемого объекта в **stub**. Это один из самых популярных видов тестовых объектов;

- **Mock-object** не только подменяет внешнюю зависимость, но и содержит логику. Он может определенным образом реагировать на некорректно переданные данные, проверяет правильность поведения тестируемого объекта.

Повторяемость — результат тестов не должен зависеть от контекста. Не важно, на каком устройстве, при каком состоянии интернет-соединения, с какого сервера принимаем данные — результаты тестирования должны быть объективными и повторяемыми. (2 + 2) всегда должно равняться 4.

Очевидность — тест должен иметь недвусмысленный результат: прошел или нет.

Пример очевидного теста:

2 + 3 выдает 5. Правильно.

2 + 3 выдает 6. Неправильно.

Пример неочевидного теста:

Тест на вход в приложение. Подаем на вход логин и пароль, на выходе имеем ошибку — «Ошибка входа».

Что произошло? Ошибка чтения данных? Истекло время ожидания ответа от сервера? Неправильно введен логин или пароль?

В подобных случаях тест должен разбиваться на более мелкие unit`ы (части). Отдельно выделяем тест чтения данных, и выполняем только его. Старайтесь не писать тесты с большим количеством вызовов из разных слоев системы. Очень сложно определить, какой именно шаг теста завершился ошибкой.

Своевременность — тесты должны быть написаны «к месту» и вовремя — когда необходимы, а не когда «дойдут руки». Данная характеристика по сути пропагандирует **TDD** (но об этом — позже). К примеру, мы закончили код, отдали его в релиз. Тестировщики его проверили, нашли ошибки, мы их исправили. Допустим, этот код не подвергнется модификации. И тут мы решили написать тесты. Зачем? Код уже протестирован и отдан в релиз. Лучше это время потратить на написание следующей части кода.

Важно понимать:

- *Тесты должны писаться тогда, когда они нужны.*
- *Тесты должны писаться тогда, когда меняется код.*
- *Тесты — это часть написания кода.*

Подходы к тестированию

Помимо обычных Unit-тестов часто применяют подходы **TDD** и **BDD**. В основном они схожи, но есть и отличия.

TDD — разработка через тестирование

TDD (test-driven development) — это разработка через тестирование. В соответствии с этим подходом вся разработка программного кода разбивается на множество небольших циклов: сначала пишутся тесты, которые покрывают изменение, затем — код, который эти тесты проходит. После этого производится модификация кода, при необходимости пишутся новые тесты. Если какие-то тесты не проходят, этот участок кода исправляется.

Рассмотрим в качестве примера нашу функцию сложения. Сначала пишем тест. В нем определяем название и сигнатуру функции. Запускаем — ошибка, не проходит сборка (функции-то нет). Пишем функцию с простым кодом, реализующим сложение. Запускаем тесты. Добиваемся их прохождения. Производим модификацию кода (может понадобиться вынести куски кода или избавиться от лишнего — производим рефакторинг кода). Снова запускаем тесты. Добиваемся их прохождения.

TDD — это лишь рекомендация. Может показаться, что писать как обычно — гораздо быстрее. Да, на первых порах применение TDD будет отнимать дополнительное время. Но вскоре привыкаешь писать сначала тесты, а затем код, и делаешь это так же быстро, как без тестов. К тому же, потратив время на работу с этим подходом, вы выиграете, облегчая поддержку продукта.

BDD

BDD (behaviour-driven development) — это разработка, основанная на описании поведения, разновидность **TDD**. Разница только в словах — в использовании таких выражений, как «мой класс должен вести себя так-то и так-то» и «мой метод должен делать то-то и то-то». Именно эта установка, по замыслу авторов **BDD**, делает подход более удобочитаемым, а написание тестов — естественным.

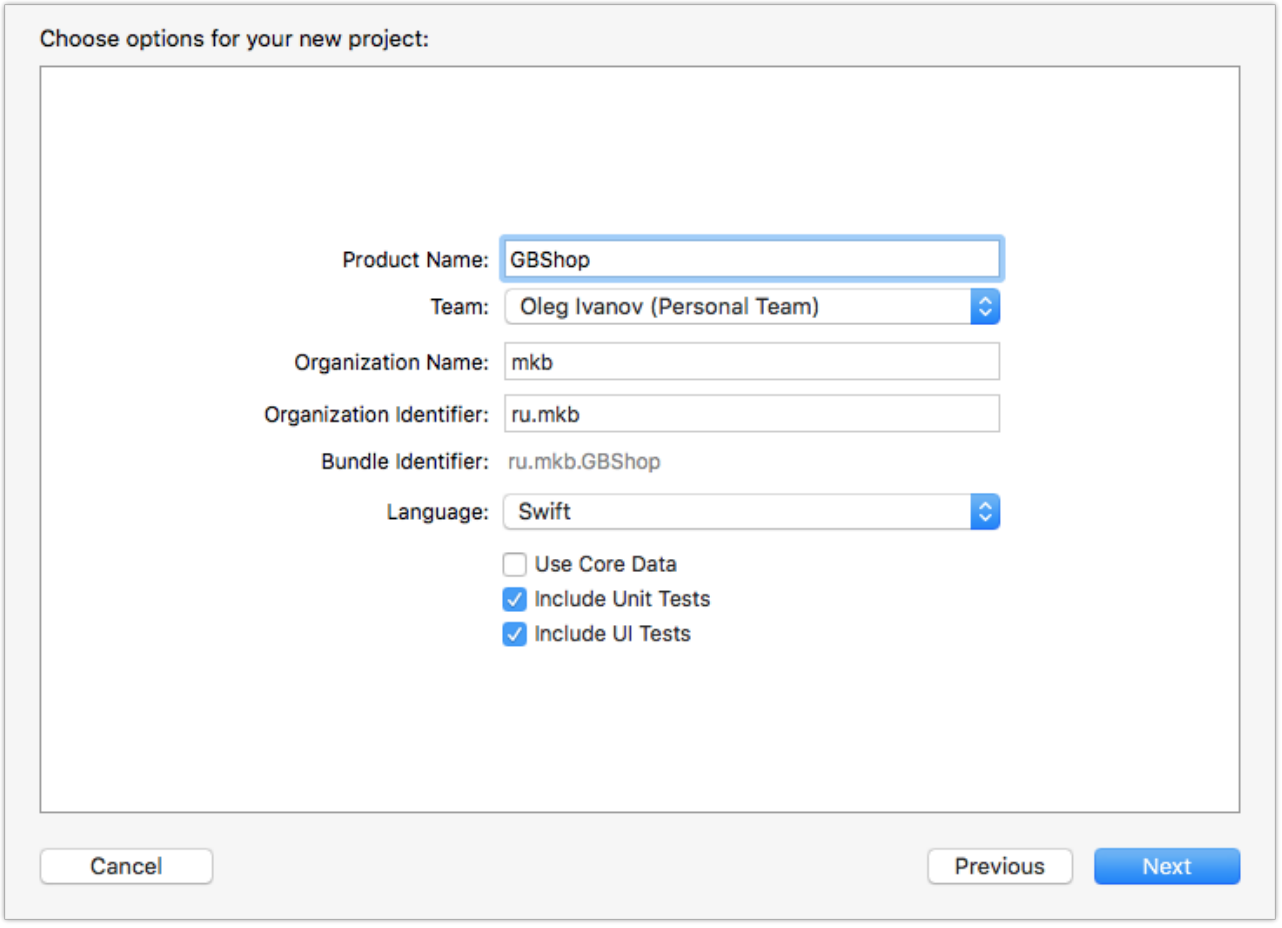
GBShop

Применим знания на практике: напишем первый Unit-тест в нашем проекте.

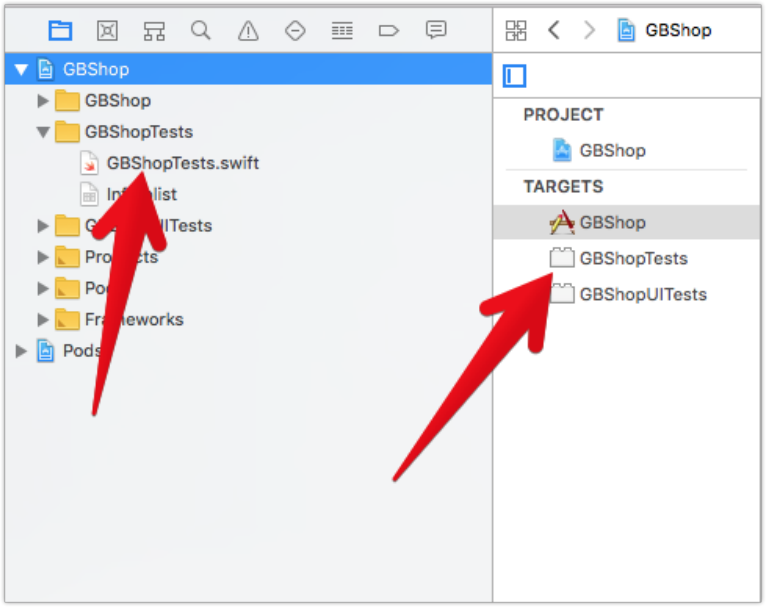
Написание Unit-теста

Прежде чем писать код Unit-теста, разберем инструментарий **Xcode**, который будет нам помогать.

Помним, что при создании проекта **GBShop** в настройках мы включили 2 галочки: **Include Unit Tests** и **Include UI Tests**.



Include Unit Tests предоставляет нам новую цель сборки (**Unit Test Target**) для проведения модульного (**Unit-**) тестирования. Также нам доступны автоматически сгенерированные файлы (классы) для примера модульного тестирования. **Include UI Tests** производит аналогичные действия только для графического (**UI-**) тестирования.



Рассмотрим структуру файла **GBShopTests.swift**. В самом начале предоставляется включение фреймворка **XCTest**. Именно с его помощью будем писать Unit-тест и производить валидацию данных.

```
import XCTest
```

Далее предлагают воспользоваться ключевым словом **@testable import**, которое дает доступ к данным модуля **GBShop**. Без этого включения, в соответствии с моделью управления доступом **Swift**, данные модуля **GBShop** в нашем тестовом коде недоступны.

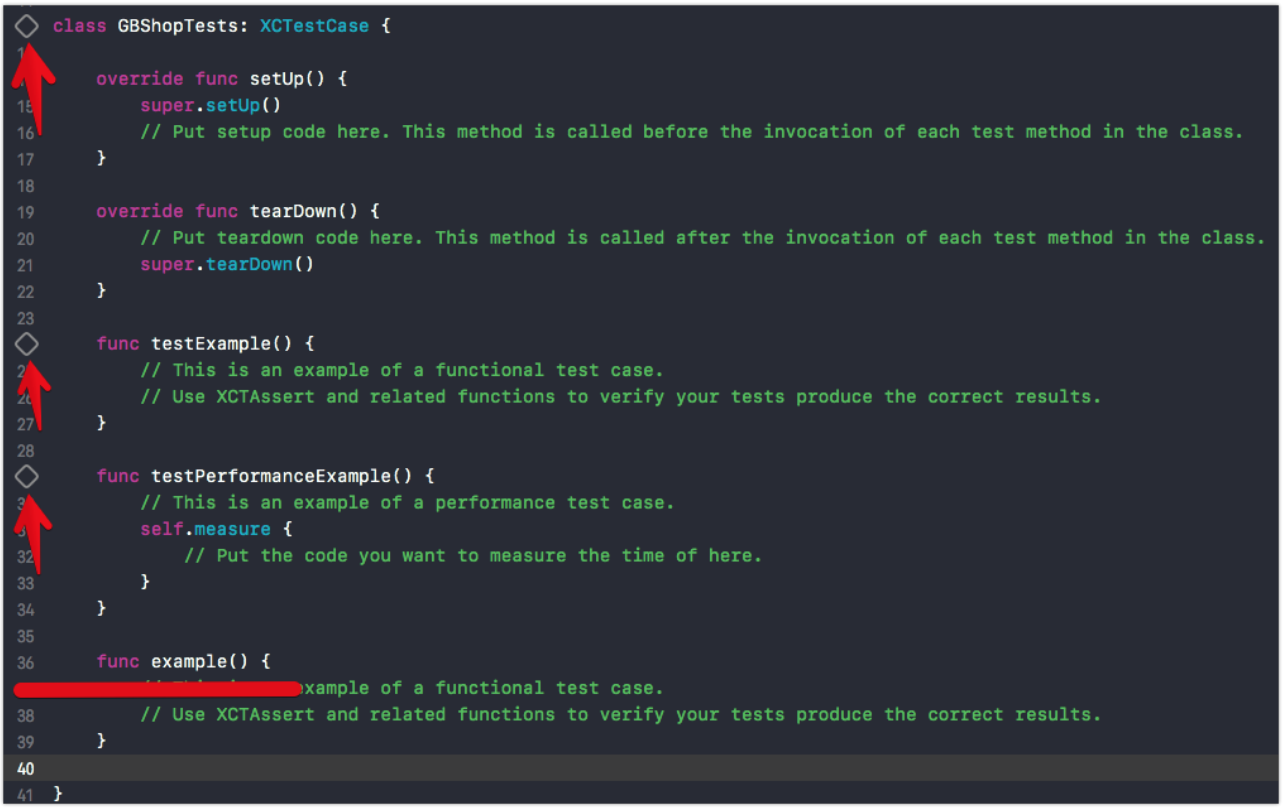
```
@testable import GBShop
```

Далее предоставляют класс **GBShopTests**, наследуемый от класса **XCTestCase** фреймворка **XCTest** с набором методов.

```
class GBShopTests: XCTestCase {
    override func setUp() {
        super.setUp()
        // Put setup code here. This method is called before the invocation of
each test method in the class.
    }
    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of
each test method in the class.
        super.tearDown()
    }
    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the
correct results.
    }
    func testPerformanceExample() {
        // This is an example of a performance test case.
        self.measure {
            // Put the code you want to measure the time of here.
        }
    }
 }
```

Как видим из описания, первые два метода — **setUp()** и **tearDown()** — позволяют выполнять тесты «с чистого листа». Возьмем за правило: в методе **setUp()**, который вызывается перед стартом каждого теста, создаем и инициализируем нужные переменные и освобождаем их в методе **tearDown()**, который вызывается после завершения каждого теста.

Оставшиеся два метода — **testExample()** и **testPerformanceExample()** — примеры полноценных тестов. Обратите внимание: без приставки «test» тест считается внутренним методом класса и не выполняется. Напротив названия каждого теста есть элемент его запуска. Подобный есть и напротив класса с набором тестов — для отдельного запуска только тестов этого класса.



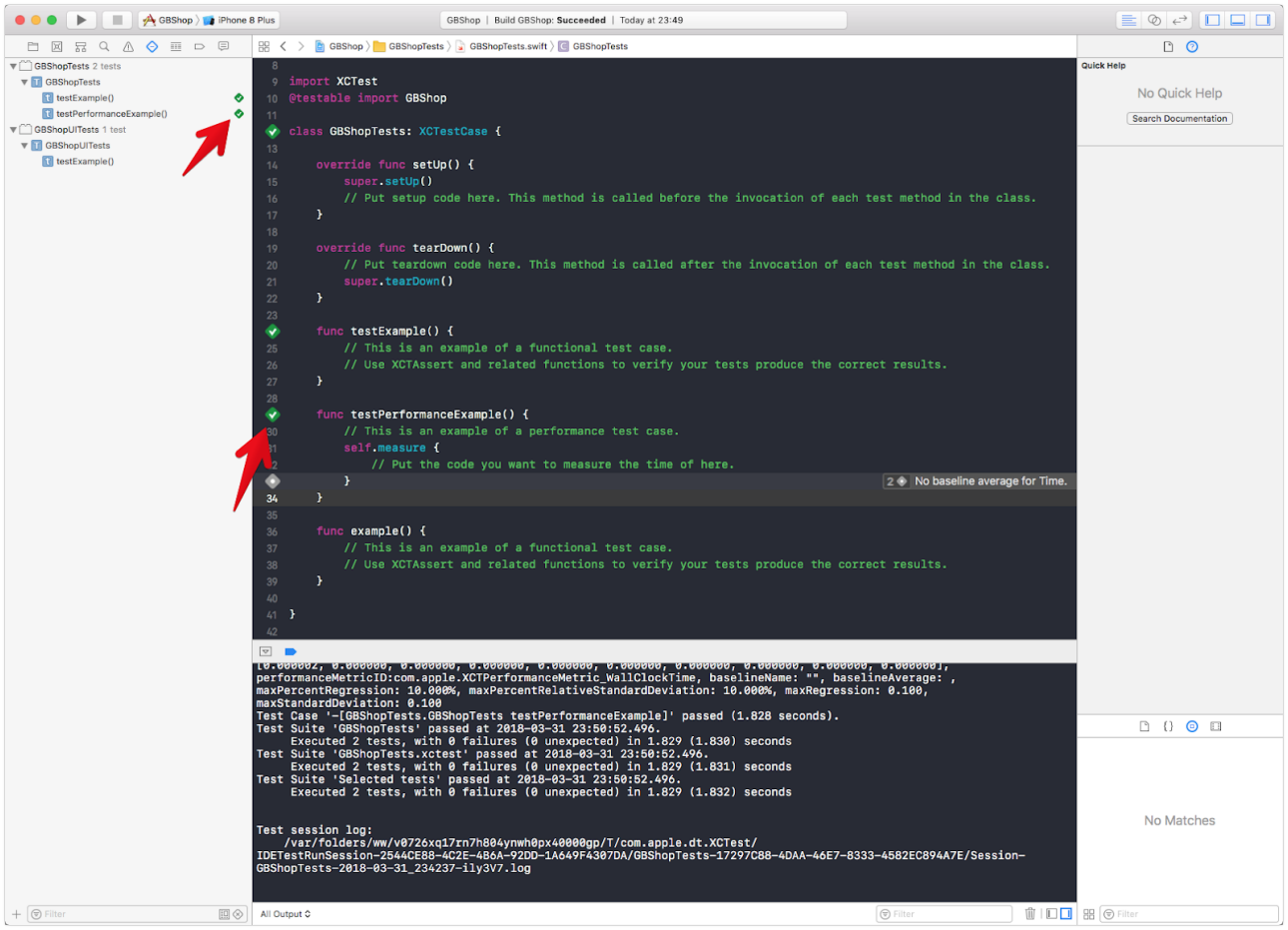
```
class GBShopTests: XCTestCase {

    override func setUp() {
        super.setUp()
        // Put setup code here. This method is called before the invocation of each test method in the class.
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method in the class.
        super.tearDown()
    }

    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the correct results.
    }

    func testPerformanceExample() {
        // This is an example of a performance test case.
        self.measure {
            // Put the code you want to measure the time of here.
        }
    }

    func example() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the correct results.
    }

}
```

Данный класс уже готов к запуску модульных тестов (**testExample** и **testPerformanceExample**).

Тесты можно запускать несколькими способами:

- по одному — с помощью кнопки запуска напротив метода теста;
- все тесты класса — с помощью кнопки запуска напротив имени класса;
- все тесты целиком — с помощью комбинации клавиш **CMD+U** или кнопки запуска напротив названия цели на вкладке **Test Navigator**.

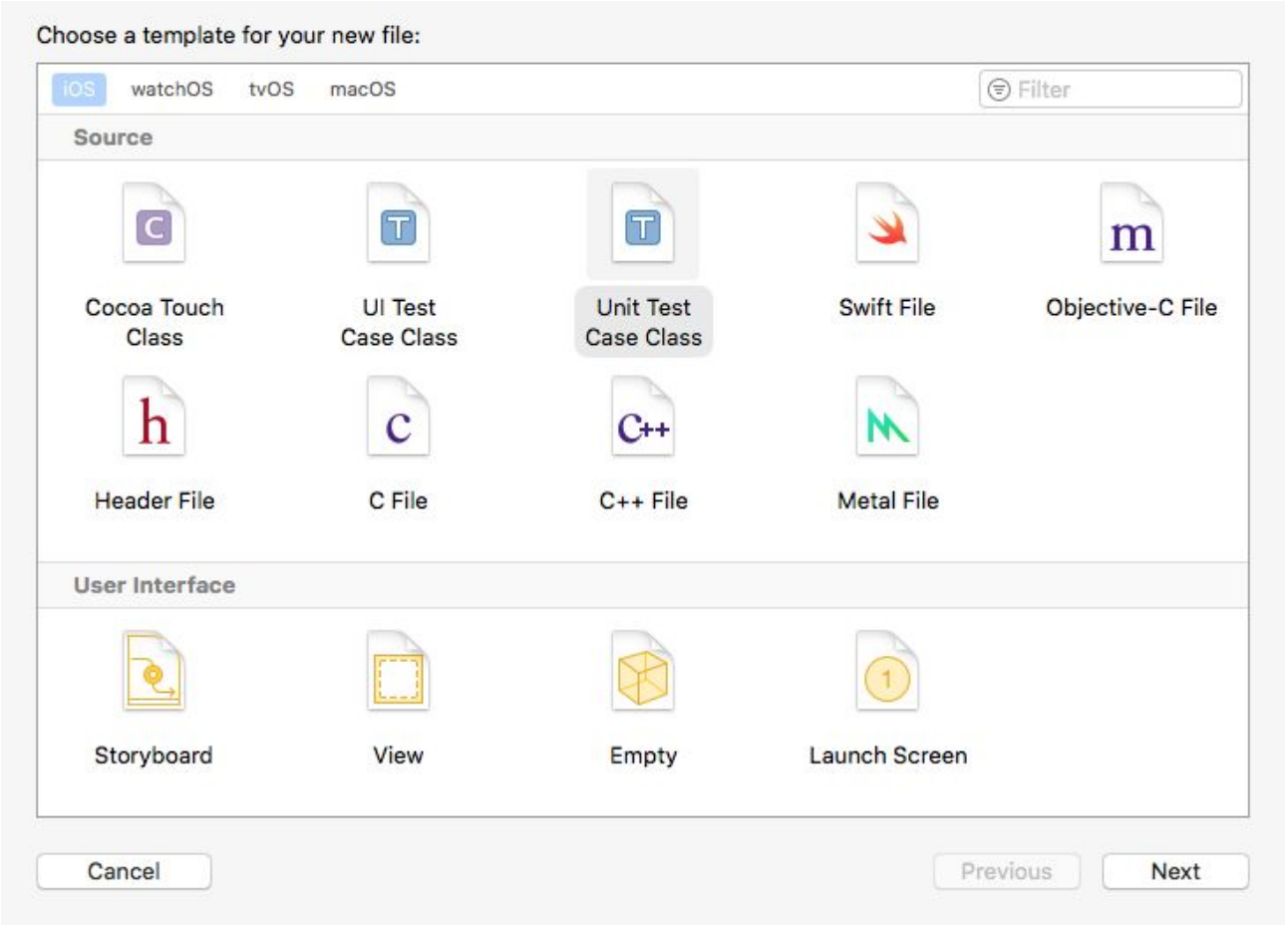
Воспользуемся кнопкой запуска тестов класса:



В результате сборки и выполнения этих тестов получаем маркировку зеленого цвета на кнопках запуска тестов — напротив каждого успешно выполненного. Все они пустые, поэтому и прошли. Если бы тест не прошел, «покраснела» бы маркировка и строка с ошибкой. В окне вывода видим лог отчета о пройденных тестах.

Для навигации по всем тестам проекта представлена вкладка **Show the Test navigator**. Здесь удобно работать с результатами выполнения и элементами управления тестами.

Создадим новый класс (файл) для тестирования — **ResponseCodable**. Возьмем за правило для новых тестов создавать отдельный класс тестов — только для тестирования этой функциональности. При создании класса воспользуемся шаблоном и в конце имени класса добавим **‘Tests’** - **‘ResponseCodableTests’**. Данная приставка относится к «магии» организации модульного тестирования в **Xcode**.



Дадим название этому методу теста и добавим код проверки с использованием метода фреймворка **XCTest - XCTFail()**. Разберем ситуацию ошибочного **Url**. Посмотрим на результат тестирования. Для написания теста воспользуемся **stub** (заглушкой), добавив классы **PostStub**, **ApiErrorStub**, **ErrorParserStub**.

```
struct PostStub: Codable {
    let userId: Int
    let id: Int
    let title: String
    let body: String
}

enum ApiErrorStub: Error {
    case fatalError
}

struct ErrorParserStub: AbstractErrorParser {
    func parse(_ result: Error) -> Error {
        return ApiErrorStub.fatalError
    }

    func parse(response: HTTPURLResponse?, data: Data?, error: Error?) -> Error?
{
        return error
    }
}
```

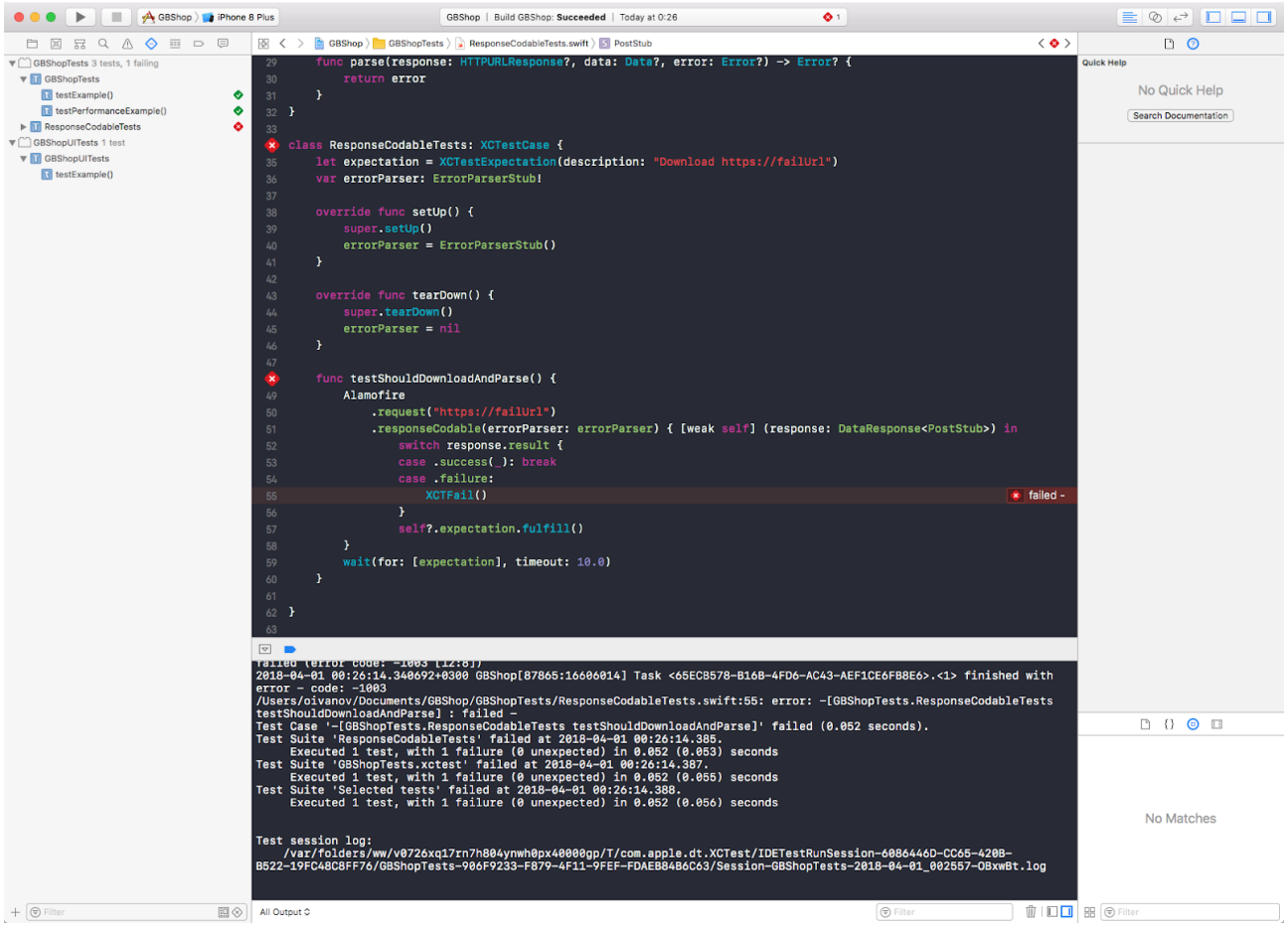
Код метода теста преобразуется:

```
func testShouldDownloadAndParse() {
    let errorParser = ErrorParserStub()

    Alamofire
        .request("https://failUrl")
        .responseCodable(errorParser: errorParser) {(response:
DataResponse<PostStub>) in
            switch response.result {
            case .success(_): break
            case .failure:
                XCTFail()
            }
    }
}
```

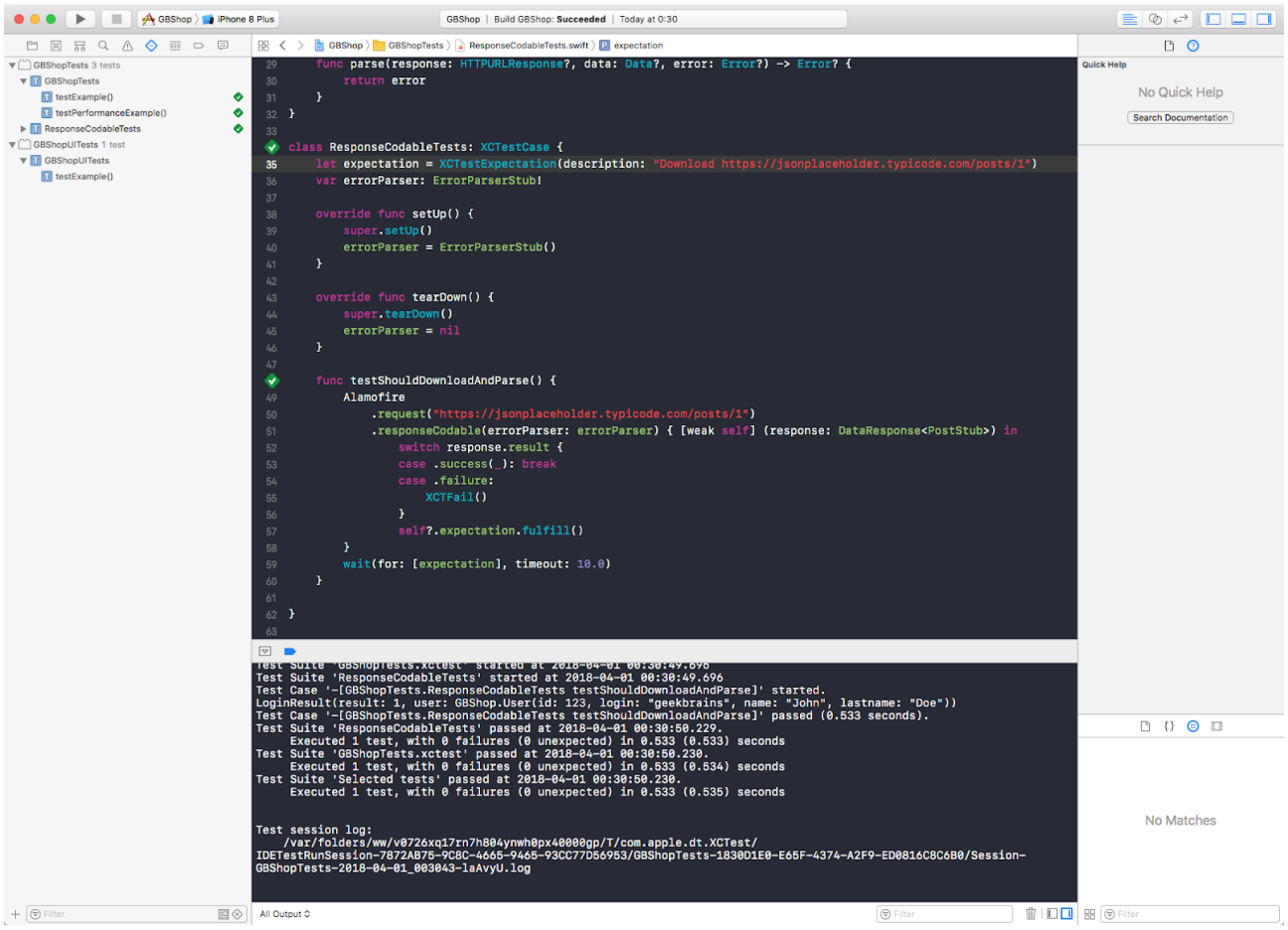
Но при запуске теста увидим его успешное прохождение, хотя должна быть ошибка (из-за невалидного URL). Дело в том, что тест не дождался ответа от сервера и сразу завершился. Для ожидания в тестах используется класс **XCTestExpectation**. Применим его:

```
class ResponseCodableTests: XCTestCase {
    let expectation = XCTestExpectation(description: "Download https://failUrl")
    var errorParser: ErrorParserStub!

    override func setUp() {
        super.setUp()
        errorParser = ErrorParserStub()
    }

    override func tearDown() {
        super.tearDown()
        errorParser = nil
    }

    func testShouldDownloadAndParse() {
        Alamofire
            .request("https://failUrl")
            .responseCodable(errorParser: errorParser) { [weak self] (response:
DataResponse<PostStub>) in
                switch response.result {
                case .success(_): break
                case .failure:
                    XCTFail()
                }
                self?.expectation.fulfill()
        }
        wait(for: [expectation], timeout: 10.0)
    }

}
```

Теперь видим, что тест завершился с ошибкой:



Рассмотрим пример валидного URL. В качестве удаленного сервера воспользуемся ресурсом **jsonplaceholder.typicode.com**, который предоставляет удобное **api** для тестирования. Заменим **URL** с **https://failUrl** на **https://jsonplaceholder.typicode.com/posts/1**.



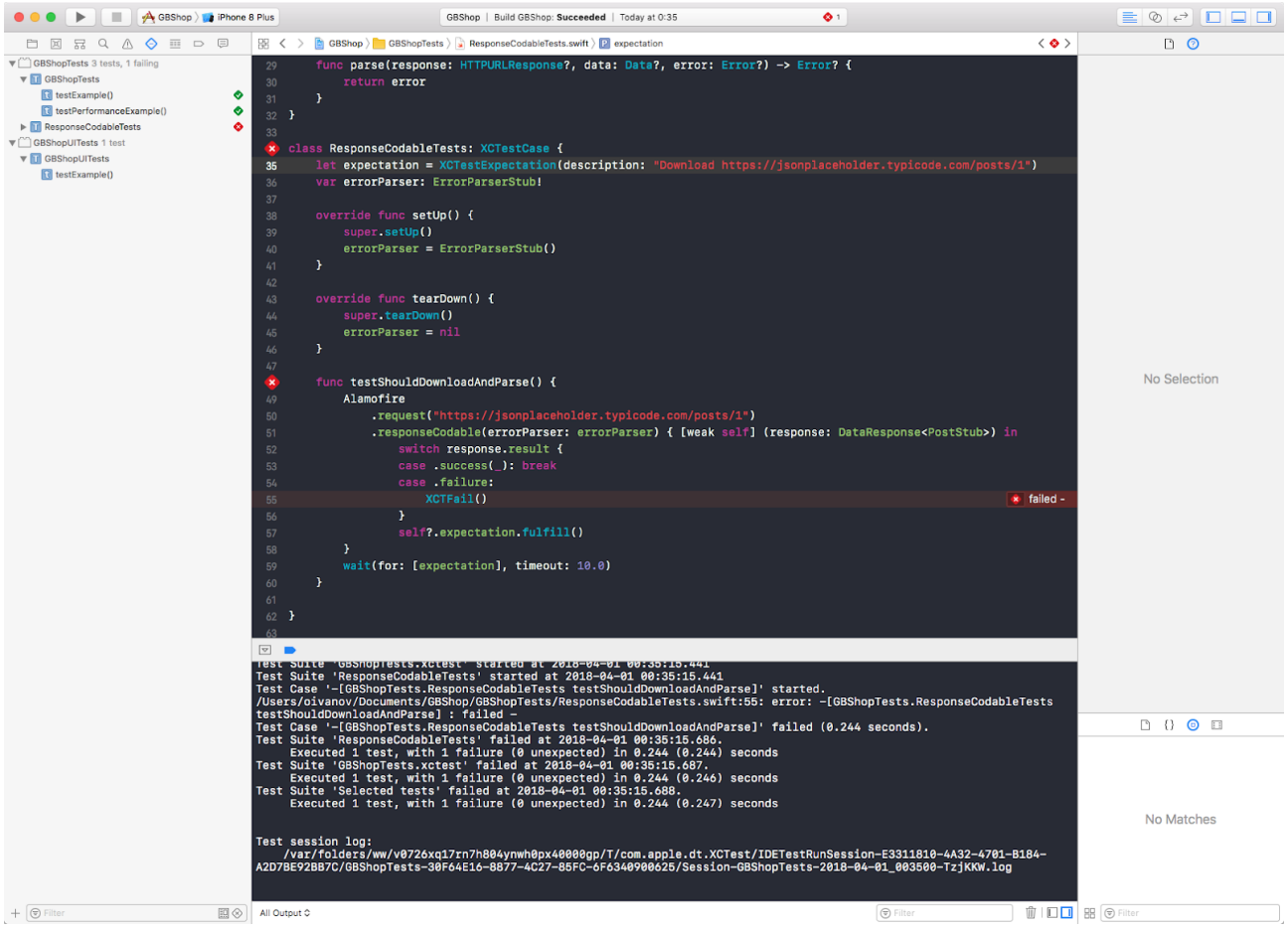
Тест успешно завершился.

Рассмотрим последний пример. Допустим, мы задумали изменить код метода **responseCodable** в расширении класса **DataRequest** и ошиблись: случайно заменили входной параметр **data** метода **Request.serializeResponseData** на **nil**.

```
extension DataRequest {
    @discardableResult
    func responseCodable<T: Decodable>(
        errorParser: AbstractErrorParser,
        queue: DispatchQueue? = nil,
        completionHandler: @escaping (DataResponse<T>) -> Void)
        -> Self {
            let responseSerializer = DataResponseSerializer<T> { request,
response, data, error in
                if let error = errorParser.parse(response: response, data: data,
error: error) {
                    return .failure(error)
                }
                let result = Request.serializeResponseData(response: response,
data: nil, error: nil)
                switch result {
                case .success(let data):
                    do {
                        let value = try JSONDecoder().decode(T.self, from: data)
                        return .success(value)
                    } catch {
                        let customError = errorParser.parse(error)
                        return .failure(customError)
```

```
                    }
                case .failure(let error):
                    let customError = errorParser.parse(error)
                    return .failure(customError)
                }
            }
            return response(queue: queue, responseSerializer:
responseSerializer, completionHandler: completionHandler)
    }
}
```

Запускаем тест и смотрим, как он отреагирует на изменение:



Все правильно: тест показал, что внесенные изменения вызвали ошибки, требуется исправить их. После восстановления кода метода **responseCodable** в расширении класса **DataRequest** тест снова пройдет без ошибок.

Практическое задание

1. Покрыть ваш код тестами.
2. Все дальнейшие изменения тоже покрываются тестами.
3. Реализовать получение списка товаров и отдельного товара.

Не забывайте о том, что проходили на прошлых уроках: **DI**, работу с **git**. Это и последующие домашние задания должны быть выполнены с учетом всех методик и практик, что мы изучаем на курсе.

Дополнительные материалы

1. BDD vs TDD.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. jsonplaceholder.typicode.com.