Clean Swift TDD Частина 3 – Worker

У частині 1 і частині 2 ми тестували ListOrdersViewController і ListOrdersInteractor відповідно, але використовували несправжній **OrdersWorker**. Сьогодні ми тестуватимемо **OrdersWorker** для того, щоб переконатися, що він повертає правильний перелік Замовлень.

Але спочатку давайте абстрагуємося від деталей отримання списку Замовлень з бази даних. В процесі роботи ми будемо працювати з такими сховищами даних:

- пам'ять
- Core Data
- API

Файл **OrdersWorker** може використовувати будь-яке з них. Він не залежить від типу задіяного сховища даних.

Основна робота по збереженню даних

Як правило, Замовлення можуть зберігатися у локальній **Core Data** або на сервері. Проте, під час розробки додатку і його тестування ми не хочемо отримувати замовлення ні з локальної **Core Data**, ні з сервера, бо це займає деякий час. З метою пришвидчення цих процесів буде зручніше створити сховище даних у пам'яті. Воно найкраще підходить для організації швидкого доступу до збережених даних.

Тому давайте спочатку спроектуємо **OrdersWorker** з підтримкою **API**, яке забезпечить нас зручним методом доступу до сховища даних.

У файлі OrdersWorker.swift:

- **Додамо** OrdersStoreProtocol **і оголосимо метод** fetchOrders().
- **В клас OrdersWorker додамо змінну** ordersStore **типу OrdersWorker.**
- **Створимо ініціалізатор з** ordersStore **у якості вхідного аргумента.**

Наш файл OrdersWorker.swift **тепер повинен виглядати так:**

```
protocol OrdersStoreProtocol {
    func fetchOrders(completionHandler: (orders: [Order]) ->
Void)
}

class OrdersWorker {
    var ordersStore: OrdersStoreProtocol

    init(ordersStore: OrdersStoreProtocol) {
        self.ordersStore = ordersStore
    }

    func fetchOrders(completionHandler: (orders: [Order]) ->
Void) {
        completionHandler(orders: [])
    } }
```

Далі ми створимо всі три сховища даних і приведемо їх у відповідність з `OrdersStoreProtocol`**, який щойно визначили. Для цього необхідно просто додати порожній метод** `fetchOrders()`.

Створимо новий файл `OrdersMemStore.swift`:

```
class OrdersMemStore: OrdersStoreProtocol {
    func fetchOrders(completionHandler: (orders: [Order]) ->
Void) {
    }
}
```

Створимо новий файл *OrdersCoreDataStore.swift*:

```
class OrdersCoreDataStore: OrdersStoreProtocol {
    func fetchOrders(completionHandler: (orders: [Order]) ->
Void) {
    }
}
```

Створимо новий файл *OrdersAPI.swift*:

```
class OrdersAPI: OrdersStoreProtocol {
    func fetchOrders(completionHandler: (orders: [Order]) ->
Void) {
    }
}
```

В майбутньому нам буде зручно і легко перемикатися між цими трьома сховищами у наступний спосіб:

```
OrdersWorker(ordersStore: OrdersMemStore())
OrdersWorker(ordersStore: OrdersCoreDataStore())
OrdersWorker(ordersStore: OrdersAPI())
```

Ця техніка проходження різних об'єктів, що відповідають одному і тому ж самому протоколу, заданому у **OrdersWorker** ініціалізаторі (або конструкторі) називається **впровадження залежностей у Constructor**.

Детальна інформація про те, як насправді відбувається вибірка `Замовлень` в кожному конкретному сховищі є специфічною і внутрішньою по відношенню до реалізації самих сховищ даних. Файл **OrdersWorker** є тільки споживачем `OrdersStoreProtocol API`.

Деякі пояснення перед TDD

Давайте змінимо спосіб утворення об'єкта **OrdersWorker**. Для цього необхідно повернутися назад і внести зміни до `ListOrdersInteractor`. З метою проведення тестування скоригуємо `ordersWorker` для взаємодії зі сховищем `OrdersMemStore`:

```
var ordersWorker = OrdersWorker(ordersStore:
OrdersMemStore())
```

I ListOrdersInteractorTests:

```
let ordersWorkerSpy = OrdersWorkerSpy(ordersStore:
OrdersMemStore())
```

Тепер знову зберемо наш проект.

Ізолювання залежностей

Як правило, це дуже корисно, щоб спочатку визначити і виділити якусь залежність. Ми перевіряємо **OrdersWorker** і його ordersStore для зовнішнішнього сховища даних. Будь то пам'ять, **Core Data** або сервер — це **залежність**. Давайте "скопіюємо" її:

```
class OrdersMemStoreSpy: OrdersMemStore {
    // MARK: Method call expectations
    var fetchedOrdersCalled = false

    // MARK: Spied methods
    override func fetchOrders(completionHandler: (orders:
[Order]) -> Void) {
        fetchedOrdersCalled = true
        let oneSecond =
dispatch_time(dispatch_time_t(DISPATCH_TIME_NOW), 1 *
Int64(NSEC_PER_SEC))
     dispatch_after(oneSecond, dispatch_get_main_queue(), {
            completionHandler(orders: [Order(), Order()])
        })
    } }
```

Щоб зімітувати **асинхронну** (у фоні і незаблоковану) вибірку `Замовлень` використаємо **GCD** для затримки виклику `completionHandler` на 1 секунду.

Метод `fetchOrders()` поверне значення миттєво. Через 1 секунду `completionHandler` виконається в основному потоці. Ми просто створимо декілька `Замовлень` і отримаємо їх у вигляді масиву.

`OrdersMemStoreSpy` виконається всередині класу `OrdersWorkerTests`, який ми розглянемо пізніше.

Наш перший тест

В [останньому пості] ми "скопіювали" поведінку **OrdersWorker**, щоб перевірити роботу `ListOrdersInteractor`. Тепер настав час для **TDD** у **OrdersWorker**, щоб зробити декілька запитів на отримання реальних замовлень зі сховища даних.

```
import XCTest

class OrdersWorkerTests: XCTestCase {
    // MARK: Subject under test
    var sut: OrdersWorker!

    // MARK: Test lifecycle
    override func setUp() {
        super.setUp()
        setupOrdersWorker() }
```

```
    override func tearDown() {
        super.tearDown()
    }

    // MARK: Test setup
    func setupOrdersWorker() {
        sut = OrdersWorker(ordersStore: OrdersMemStoreSpy())
    }

    // MARK: Test doubles
    class OrdersMemStoreSpy: OrdersMemStore {
        // MARK: Method call expectations
        var fetchedOrdersCalled = false

        // MARK: Spied methods
        override func fetchOrders(completionHandler:
(orders: [Order]) -> Void) {
            fetchedOrdersCalled = true
            let oneSecond =
dispatch_time(dispatch_time_t(DISPATCH_TIME_NOW), 1 *
Int64(NSEC_PER_SEC))
            dispatch_after(oneSecond,
dispatch_get_main_queue(), {
            completionHandler(orders: [Order(), Order()])
            })
        }
    }
```

```
    // MARK: Tests
    func testFetchOrdersShouldReturnListOfOrders() {
        // Given
        let ordersMemStoreSpy = sut.ordersStore as!
OrdersMemStoreSpy

        // When
        let expectation = expectationWithDescription("Wait
for fetched orders result")
        sut.fetchOrders { (orders: [Order]) -> Void in
     expectation.fulfill()
        }

        // Then
        XCTAssert(ordersMemStoreSpy.fetchedOrdersCalled,
"Calling fetchOrders() should ask the data store for a list
of orders")
    waitForExpectationsWithTimeout(1.1) { (error: NSError?)
-> Void in
        XCTAssert(true, "Calling fetchOrders() should result
in the completion handler being called with the fetched
orders result")
        }
    }
}
```

У методі `setupOrdersWorker()` **ми встановили досліджуваний об'єкт як екземпляр** **`OrdersWorker`**, **щоб задіяти** `OrdersMemStoreSpy`.

У секції `Given` ми повинні спочатку отримали посилання на цього шпигуна (`spy`), щоб пізніше перевірити його під час створення наших тверджень.

У секції `When` ми будемо використовувати **`XCTest`** для підтримки **асинхронного тестування**. Для цього спочатку треба зробити виклик методу `expectationWithDescription()`. Потім виконаємо метод `fetchOrders()`. Усередині блоку запустимо метод `fulfill()`, щоб організувати затримку.

У секції `Then` перевіримо, чи метод `fetchOrders()` був викликаний. Можна також викликати `waitForExpectationsWithTimeout()` із затримкою `1,1` секунди (тільки трохи довше, ніж `dispatch_after` у `OrdersMemStoreSpy`). Якщо математичне очікування виконується, це означає, що обробник завершення виконується правильно. Саме те, що нам потрібно.

Реалізація логіки

Зараз наші старання можуть зазнати краху, тому **`OrdersWorker`** повинен негайно викликати обробник завершення і повернути порожній масив `Замовлень`. Без жодного `Замовлення`!

Давайте це зробимо.

Внесемо зміни у метод `fetchOrders()` файла **`OrdersWorker`** з метою задіяти нову змінну `ordersStore` для вибірки замовлень зі сховища даних наступним чином:

```
func fetchOrders(completionHandler: (orders: [Order]) ->
Void) {
    ordersStore.fetchOrders { (orders: [Order]) -> Void in
      completionHandler(orders: orders)
    }
}
```

Замість того, щоб негайно викликати `completionHandler` і повернути порожній масив `Замовлень`, ми виконаємо метод `fetchOrders()` у `ordersStore`. Після завершення `fetchOrders()` необхідно виконати `completionHandler` всередині блоку методу `fetchOrders()`.

Крім того, необхідно змінити виклик `completionHandler` для повернення `переліку Замовлень` для `ordersStore`.

Виконайте тест зараз — він повинен пройти успішно. Весь код і тест для цього прикладу **TDD** знаходяться на GitHub.

Висновки

Ми трохи відійшли від **VIP-циклу** і розглянули **Worker**. Після цього використали **TDD** для вивчення процесу розробки `OrdersWorker`.

- Спершу ми визначили простий **API** — `OrdersStoreProtocol` з одиночним методом `fetchOrders()`.
- Далі, ми створили три сховища даних: `(1) OrdersMemStore`, `(2) OrdersCoreDataStore` і `(3) OrdersAPI`. Всі вони відповідають `OrdersStoreProtocol`.

- **OrdersWorker** може вільно вибрати одне з цих трьох сховищ даних для роботи за допомогою **впровадження залежностей Constructor**.
- Для того, щоб виділити залежність **OrdersWorker** від ordersStore, ми створили OrdersMemStoreSpy. Ви можете використовувати **GCD** для імітації **асинхронного** повернення списку Замовлень.
- Ми написали тест testFetchOrdersShouldReturnListOfOrders() для перевірки факту виклику метода fetchOrders().
- Ми також використали **підтримку асинхронного тестування** у **XCTest** для створення і перевірки виконання очікувань. Коли вони виконані, ми знаємо, що метод fetchOrders() повернув декілька Замовлень.

В наступному пості ми повернемося до **VIP-циклу**, щоб познайомитися з **TDD** для **Presenter**.