The Swifty Little Mocker (тестові об'єкти у Swift)

У своєму останньому пості я хотів показати, як працює Clean Swift. Тому я не писав жодного тесту. Ви побачили, як **Clean Swift** може допомогти написати чистий код, організувати його і в майбутньому легко орієнтуватись у проекті. Ви дізналися, як перенести бізнес-логіку в **Interactor** та логіку представлення у **Presenter** згідно вимог **VIP-циклу**. І, нарешті, ви побачили, як **Swift** протоколи допомагають нам відокремлювати об'єкти.

Ця стаття присвячена протоколам, які дозволяють отримати додаткову (або, вірніше, головну) вигоду: **зробити модульне тестування простішим**. Без тестів ми не зможемо бути впевненими у тому, що внесені зміни не мають негативного впливу на існуючий код.

Але про все по порядку.

Опираючись на викладки Uncle Bob's The Little Mocker нижче наводиться короткий вступ у різні види тестових об'єктів безпосередньо у **Swift**. Я рекомендую вам прочитати його веселий освітній пост.

Java interface == Swift protocol

```
interface Authorizer {
    public Boolean authorize(String username, String
password);
}

protocol Authorizer {
    func authorize(username: String, password: String) ->
Bool
}
```

Test double (подвійний тест)

Назва подвійний тест стосується всієї сім'ї об'єктів, які використовуються у тестах.

Dummy (фіктивний)

Ви передаєте у функцію аргумент `dummy` у тому випадку, коли не знаєте, як він використовується. В рамках тесту, коли ви повинні передати аргумент, який ніколи не буде використовуватися в цьому методі, ви передаєте `dummy`. Оскільки `dummy` ніколи не використовується він може повернути пусте значення. Така поведінка цілком прийнятна і найбільш логічна.

```
public class DummyAuthorizer implements Authorizer {
     public Boolean authorize(String username, String
password) {
          return null;
     }
}

class DummyAuthorizer: Authorizer {
     func authorize(username: String, password: String) ->
Bool? {
          return nil
     }
}
```

Клас `DummyAuthorizer` є тестовим dummy, який повертає `nil`.

Приклад тесту

```
public class System {
     public System(Authorizer authorizer) {
          this.authorizer = authorizer;
     }

     public int loginCount() {
          //returns number of logged in users.
     }
}
```

```
@Test
public void newlyCreatedSystem_hasNoLoggedInUsers() {
     System system = new System(new DummyAuthorizer());
     assertThat(system.loginCount(), is(0));
}

class System {
     var authorizer: Authorizer

     init(authorizer: Authorizer) {
          self.authorizer = authorizer
     }

     func loginCount() -> Int {
          //returns number of logged in users.
          return 0
     }
}

class TheLittleMocker: XCTestCase {
     func testNewlyCreatedSystem_hasNoLoggedInUsers() {
          let system = System(authorizer: DummyAuthorizer())
          XCTAssert(system.loginCount() == 0, "Pass")
     }
}
```

Ми хочемо протестувати щойно створену систему, яка містить перелік неавторизованих користувачів. `System` є класом, у якому метод ініціалізації повинен прийняти аргумент `Authorizer` для визначення прав доступу. Але оскільки наша мета полягає в тому, щоб переконатися, що новий користувач має нульовий відлік входів у систему, ми не будемо дбати про дозволи чи вхідний аргумент `Authorizer`. Змінна `DummyAuthorizer` ідеально підходить для цього випадку, тому що вона нічого не знає про ім'я користувача та його пароль.

Stub (ціль)

`Stub` = `dummy`, але повертає певне значення, на яке спирається інша частина системи, щоб продовжити виконання тесту.

```
public class AcceptingAuthorizerStub implements Authorizer {
     public Boolean authorize(String username, String
password) {
          return true;
     }
}

class AcceptingAuthorizerStub: Authorizer {
     func authorize(username: String, password: String) ->
Bool? {
          return true
     }
}
```

Якщо продовження роботи для решти класу `System` залежить від того, яке значення повертає метод `authorize()`, ми можемо зробити так, щоб `AcceptingAuthorizerStub` завжди повертав `true`. Нас не хвилює ім'я користувача та його пароль, ми просто моделюємо процес авторизації. Наша мета полягає в тому, щоб перевірити лічильник входів після успішної авторизації у системі. Що означає "`успішна авторизація`" ми визначимо в іншому тесті.

Spy (шпигун)

Ви можете використовувати `spy`, коли потрібно переконатися, що метод викликається в тесті. Він може шпигувати, наприклад за тим, скільки разів викликається метод і які аргументи в нього передаються. Але ви повинні бути обережні. Чим більше ви шпигуєте, тим щільніше тести пов'язуються з реалізацією проекту. Таке поєднання знижує їхню надійність.

```
public class AcceptingAuthorizerSpy implements Authorizer {
     public boolean authorizeWasCalled = false;

     public Boolean authorize(String username, String
password) {
          authorizeWasCalled = true;

          return true;
     }
}
```

```
class AcceptingAuthorizerSpy: Authorizer {
    var authorizeWasCalled = false

    func authorize(username: String, password: String) ->
Bool? {
        authorizeWasCalled = true

        return true
    }
}
```

Клас `AcceptingAuthorizerSpy` містить булеву змінну `authorizeWasCalled`. Вона фіскує випадки, коли викликається метод `authorize()`. Випробовуючи тест ви можете перевірити, чи виклик насправді відбувся.

Mock (макет)

`Mock` схожий на `spy`, але робить трошечки більше. `Mock` не цікавлять значення, які повертаються з функцій. Його більше цікавить, які саме функції були викликаються, з якими аргументами, як часто і коли.

```
public class AcceptingAuthorizerVerificationMock implements
Authorizer {
     public boolean authorizeWasCalled = false;

     public Boolean authorize(String username, String
password) {
          authorizeWasCalled = true;

          return true;
     }

     public boolean verify() {
          return authorizeWasCalled;
     }
}

class AcceptingAuthorizerVerificationMock: Authorizer {
     var authorizeWasCalled = false

     func authorize(username: String, password: String) ->
Bool? {
          authorizeWasCalled = true

          return true
     }

     func verify() -> Bool {
          return authorizeWasCalled
     }
}
```

Метод `verify()` класу `AcceptingAuthorizerVerificationMock` повертає значення `true` у випадку, якщо виклик методу завершився успішно. Замість того, щоб заглядати в `spy` у твердженні, ви викликаєте метод `verify()` для перевірки істини.

Fake (фальшивка)

Досі всі об'єкти для тестів не звертали уваги на вхідні аргументи. Навіть `mock` тільки записує аргументи, він не використовує їх для створення іншого вихідного значення. Проте, `fake` має певну бізнес-логіку. Ви можете управляти його поведінкою, передаючи різні аргументи, щоб змусити повертати різні результати.

```
public class AcceptingAuthorizerFake implements Authorizer {
     public Boolean authorize(String username, String
password) {
          return username.equals("Bob");
     }
}
```

```
class AcceptingAuthorizerFake: Authorizer {
     func authorize(username: String, password: String) ->
Bool? {
          return username == "Bob"
     }
}
```

Тестуючи певне бізнес-правило (наприклад, користувач може увійти в систему тільки з правильним ім'ям), ви використовуєте AcceptingAuthorizerFake. Метод authorize() повертає true тільки якщо ім'я користувача співпадає з "Bob".

Про головне

Нижче наведено повністю функціональний тест, написаний у **Swift** за допомогою XCTest. Ви можете скопіювати і вставити наступний код в новий Xcode-файл:

```
import UIKit
import XCTest

protocol Authorizer {
    func authorize(username: String, password: String) ->
Bool?
}

class DummyAuthorizer: Authorizer {
    func authorize(username: String, password: String) ->
Bool? {
        return nil
    }
}
```

```
class AcceptingAuthorizerStub: Authorizer {
    func authorize(username: String, password: String) ->
Bool? {
        return true
    }
}

class AcceptingAuthorizerSpy: Authorizer {
    var authorizeWasCalled = false

    func authorize(username: String, password: String) ->
Bool? {
        authorizeWasCalled = true

        return true
    }
}

class AcceptingAuthorizerVerificationMock: Authorizer {
    var authorizeWasCalled = false

    func authorize(username: String, password: String) ->
Bool? {
        authorizeWasCalled = true

        return true
    }

    func verify() -> Bool {
        return authorizeWasCalled
    } }
```

```
class AcceptingAuthorizerFake: Authorizer {
     func authorize(username: String, password: String) ->
Bool? {
          return username == "Bob"
     }
}

class System {
     var authorizer: Authorizer

     init(authorizer: Authorizer) {
          self.authorizer = authorizer
     }

     func loginCount() -> Int {
          return 0
     }
}

class TheLittleMocker: XCTestCase {
     func testNewlyCreatedSystem_hasNoLoggedInUsers() {
          let system = System(authorizer: DummyAuthorizer())
          XCTAssert(system.loginCount() == 0, "Pass")
     }
}
```

У методі `testNewlyCreatedSystem_hasNoLoggedInUsers()` ви можете змінити будь-який тестовий об'єкт, але тест, як і раніше, виконається. По мірі того, як клас `System` збільшується і `Authorizer` починає по-різному використовуватись в реалізації, необхідно зробити подвійний тест більш складним, щоб переконатися, що тест продовжує виконуватись. Це означає, що вашому оригінальному `DummyAuthorizer` може знадобитися розвинутися в `stub`, `spy`, `mock` або `fake`.

Що ви можете зробити з Swifty Little Mocker?

В майбутній статті я від початку і до кінця покажу вам силу **TDD**-підходу. Або, іншими словами, як із урахуванням вимог спочатку написати тест, а потім перейти до реалізації функцій.

Крім того, ви почнете писати свої власні `mocks` і `stubs`. Вам не доведеться вивчати нові тестові бібліотеки, адже ви побачите, що писати власні тести набагато простіше. А ще ви перестанете турбуватись про своєчасне оновлення **CocoaPod**.

Найкраще те, що ваші тести стануть неймовірно швидкими, тому що зникне необхідність імпорту і завантаження зовнішніх бібліотек. Вам потрібно буде лише протестувати свої граничні, а не приватні методи. Ви зрозумієте чому і яким чином.

В наступному пості ви дізнаєтеся, що насправді означає модульне тестування. Чи дійсно вам потрібно перевіряти кожен метод кожного класу? Як переконатися, що тести покривають усі можливі шляхи коду? Як уникнути написання ненадійних тестів?