



RETRO GUIDE TO iOS UNIT TESTING



Copyright © 2021 by Kristijan Kralj

All rights reserved. No part of this book may be reproduced or used in any manner without written permission of the copyright owner except for the use of quotations in a book review. For more information, address:

kristijan@blueinvader.com

FIRST EDITION

Table Of Contents

How to Start With Unit Testing When You Don't Have Time for Unit Testing?	3
Unit Testing: A Suprisingly Simple Way to Write Better Apps	9
The Lost Art of the Test-Driven Development	18
Improve the Quality of Your SwiftUI App With UI Tests	28
References	36

How to Start With Unit Testing When You Don't Have Time for Unit Testing?

“Yesterday is gone. Tomorrow has not yet come. We have only today. Let us begin.”

- Mother Teresa

(Let's start a book about unit testing with a quote from Mother Teresa. Said no one ever.)

Most people think that unit testing is time-consuming. That you have to spend days and days writing unit tests once you are done with developing a new feature.

The same response gave 300 developers from UK and US in the [survey](#) conducted in 2019. 270 out of 300 agree that software quality is improved with unit tests, but 42% of them skip writing unit tests in order to speed up new feature development.

The unit tests reality

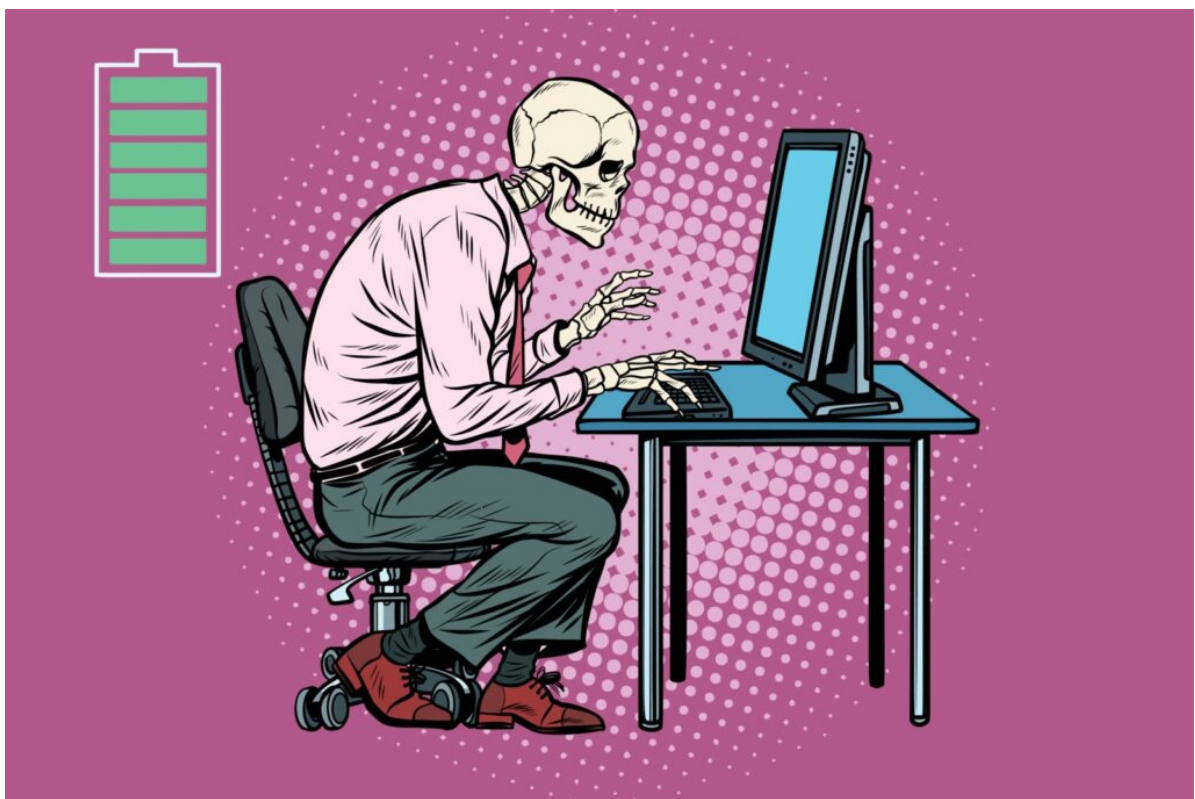
Here's the truth: When performing continuously, unit testing doesn't take too much of your time. But rather, the time you invest (yes, invest, since you'll reap the rewards in the future) in unit testing is the same time you would spend fixing a bug that proper set of unit tests would have prevented.

That being said, it still takes some time to write unit tests. And we as developers think that we don't have the time to write unit tests. There are deadlines to be met, right? (right??)

Skipping unit tests does not guarantee that project will finish on time

Here's the thing, though, skipping unit tests is not a guarantee that the feature will be shipped on time. Further more, without unit tests, it is far more likely that more bugs will be discovered during the QA phase. This means the same code goes back to you to debug it and fix it.

Hopefully, a fix for one bug won't introduce a new bug in some other place in the system.



99 little bugs in the code,
99 little bugs in the code.
Take one down, patch it around. 127 little bugs in the code...

..but only unit tests are not enough

One sidenote: the unit tests are not silver bullet, and won't catch all bugs. You also need to have integration and UI tests. But unit tests are pretty good place to start with automated testing.

“Time you enjoy wasting is not wasted time.”

- Marthe Trolly-Curtin

(Thanks, Marthe, for letting me know that watching Netflix is ok.)

Back to the focus of this post. Unit testing. Let me fix that quote for you:

*“Time you spend **testing** is not wasted time.”*

- Kristijan Kralj

Many big companies like Facebook, Google or LinkedIn rely heavily on automated, unit tests.

How big companies like Facebook, Google or LinkedIn develop apps?

Facebook – In (1), Feitelson and team report on the development and deployment practices in Facebook. The report is focused on the main facebook.com web application which experiences high traffic and needs to remain running even as new software is being deployed. Facebook works in a development mode where the software product is never finished and new versions are pushed on a daily basis. In this environment, Facebook relies on testing done by developers who write automated tests which can be run before every release.

Google – In (2), Copeland and team describe the development and deployment process at Google. The focus is on providing fast feedback on the quality of the code changes. This is achieved by running automated tests on every check-in and providing feedback to the engineers on the quality of their code.

LinkedIn – In (3) Nunduri describes the role that unit test play in certifying the LinkedIn mobile application. LinkedIn depends heavily on unit tests to catch issues faster and earlier in the development process.

What's wrong if I don't write unit tests?

The bug fixing phase will take longer than it should. You'll constantly deal with the same, repetitive bugs, instead of working on a new features.

And the software quality will suffer as well.

In his [talk](#) "Preventing the Collapse of Civilization" from 2019, [Jonathan Blow](#) talks how software quality in general is declining. There is a part in his presentation, starting at [22:57](#), where he is actually counting how many times any software he is using produces a bug. **And it happened ALL THE TIME for him!**

I'm sure it happens for you as well. Almost every day you see some kind of bug or crash in applications you use, some of them are really funny and weird.



My Macbook Pro every time I have something urgent to finish: OMG, a source code! Let me compile it carefully and slooowly....

How can I start if I don't know anything about unit testing?

Actually, it's very easy to start with unit testing. Every unit test consists of 3 parts:

1. **Arrange objects** – create the class(es) which you will be testing and set them up as necessary.
2. **Act on an object** – Call the method you want to test.
3. **Assert (check)** – that something works as expected.

In the Arrange part of the test, you initialize the class you want to test. Next, in the Act part of the test, you call the method which is being tested, and in the Assert part of the test, you check that the result is as expected.

You will learn more about how to start with unit testing in the next chapter.

When and how to introduce unit testing to my daily work?

The first step is to create a unit testing project. Simply add a unit testing project to your solution and commit that. That concludes the first step. Great job so far!

Next, you don't have to go all in on unit testing immediately, but rather, the next time you have to fix a bug, write a unit test to cover that part of the code. Just one small, tiny unit test. This should take you no more than 5-10 minutes.

As a result of this exercise, you will feel more confident that the fix you made actually works. Because, now you have a proof, your unit test, that you made a better code!

Oh, I see, unit tests are great, but how can I convince my boss?

“Change will not come if we wait for some other person, or if we wait for some other time. We are the ones we’ve been waiting for. We are the change that we seek.”

Barack Obama

If you got this far in this book, the chances are that you are interested in unit testing and you see the benefits of unit testing. **Good job!**

If your main question in this case is: I would like to start with unit testing with my current app, but my company doesn’t practice unit testing? How can I convince them?

Simply, start with unit testing on your own and see the benefits. Once you have evidence that unit tests work and help, use that as an argument to suggest to your boss to go with unit testing.

Unit Testing: A Suprisingly Simple Way to Write Better Apps

Question – What do:

- Alamofire (35.1k Github stars ★)
- RxSwift (19.5k stars ★)
- Kingfisher (18k ★)
- SwiftLint (14.3k ★)
- Snapkit (17.2k ★)
- SDWebImage (23.3k ★)
- Realm (14.3k ★)

all have in common?

Answer - They are

...all free open-sourced libraries

...used by millions of iOS developers around the world

...extensively covered with hundreds of unit tests written for each of them.

If I have seen further it is by standing on the shoulders of Giants.

- Isaac Newton

Just like our buddy Isaac stood on the shoulders of giants, aka used the help of others to help him progress further and faster, we also rely on the libraries mentioned above to develop our apps in the shortest amount of time.

Since so many developers rely on them, they have to be stable and have as few bugs as possible. And they use unit tests to achieve that goal.

One of the easiest ways to minimize the number of bugs in any app is with unit testing.

What is unit testing?

Unit testing is a type of testing you test small, isolated pieces of your code.

The purpose of unit testing is to confirm that each small piece, that is unit, works as expected on its own. A unit is usually a single method in a single struct/class, but it can be the struct/class itself.

Here's the deal. If you want to write better apps then keep reading...

Unit tests has many benefits:

- It increases confidence in changing code. If good unit tests are written and if they are run every time any code is changed, we will be able to quickly catch any new bug introduced with the latest change.
- Development is faster. How? Without unit tests, we usually have to check that the code works through the app UI. With unit testing in place, we write the test, write the code and run the test. Writing tests takes time but the time is compensated by the less amount of time it takes to run the tests.
- The cost of fixing a bug detected during unit testing is smaller than the cost of fixing the same bug later. It's less expensive to fix a bug during development, than later when your changes are already deployed to the App Store.
- Debugging is easy. When a test fails, only the latest changes need to be debugged.

Will unit tests magically solve all your problems?

No. Unit testing is not a silver bullet.

Unit tests only tests the functionality of the small, independent pieces of your code. They will not catch integration errors, such as features that need multiple units to be executed to work properly.

That's why it's recommended to also have integration and UI tests.

The other thing is that developers need to invest (yes, invest, since that time is repaid in the long term) extra time to write and maintain unit tests. This is not an easy task, sometimes. We all have deadlines to meet, code reviews to perform, meetings to attend.



Are you lonely?
Tired of working on your own?
Do you hate making decisions?
Hold a meeting!

You can see people, show charts, feel important...
All on company time!

Here's how to get started with iOS unit testing

How do we test our code? A unit test usually consists of three main actions:

1. Arrange objects, creating and setting them up as necessary.
2. Act on an object.
3. Assert that something is as expected.

Let's go through one example to see these main parts in a single unit test. We have a simple `Calculator` struct, which has one method, `addTwoNumbers`.

```
struct Calculator {  
    func addTwoNumbers(first: Int, second: Int) -> Int {  
        return first + second  
    }  
}
```

(At this point you are probably wondering why I'm showing you this simple code, since the code is never that simple. But, I'm showing you how to test this simple code so we can fully focus on the mechanics of writing unit tests, and put the actual code in the second place.

Plus, you never know, maybe will have to test the exact same function in the future. In that case, lucky you...)

One unit test to check that `addTwoNumbers` returns the sum of numbers passed in as parameters might look like:


```
func test_addTwoNumbers_returns_sum_of_numbers(){
    //Arrange
    let calculator = Calculator()
    //Act
    let result = calculator.addTwoNumbers(5,6)
    //Assert
    XCTAssertEqual(11, result)
}
```

1. In the Arrange part of the test, we initialize the struct we want to test.
2. Next, in the Act part of the test, we call the method which is being tested.
3. In the Assert part of the test, we check that the result is as expected. We use `XCTAssertEqual` method to do that. That method is a part of a `XCTest` framework, which provides us with the methods to make writing tests easier and faster.

How to name tests? Use

test_UnitOfWork_ExpectedBehaviour_ScenarioUnderTest naming convention.

Every test method in Xcode needs to start with `tests_`, in order to be recognized as test.

***UnitOfWork** name could be as simple as a method name (if that's the whole unit of work) or more abstract, if it's a use case that encompasses multiple methods or classes such as `UserLogin` or `RemoveUser`.*

***ExpectedBehaviour** is the result we expect to occur after executing the unit of work.*

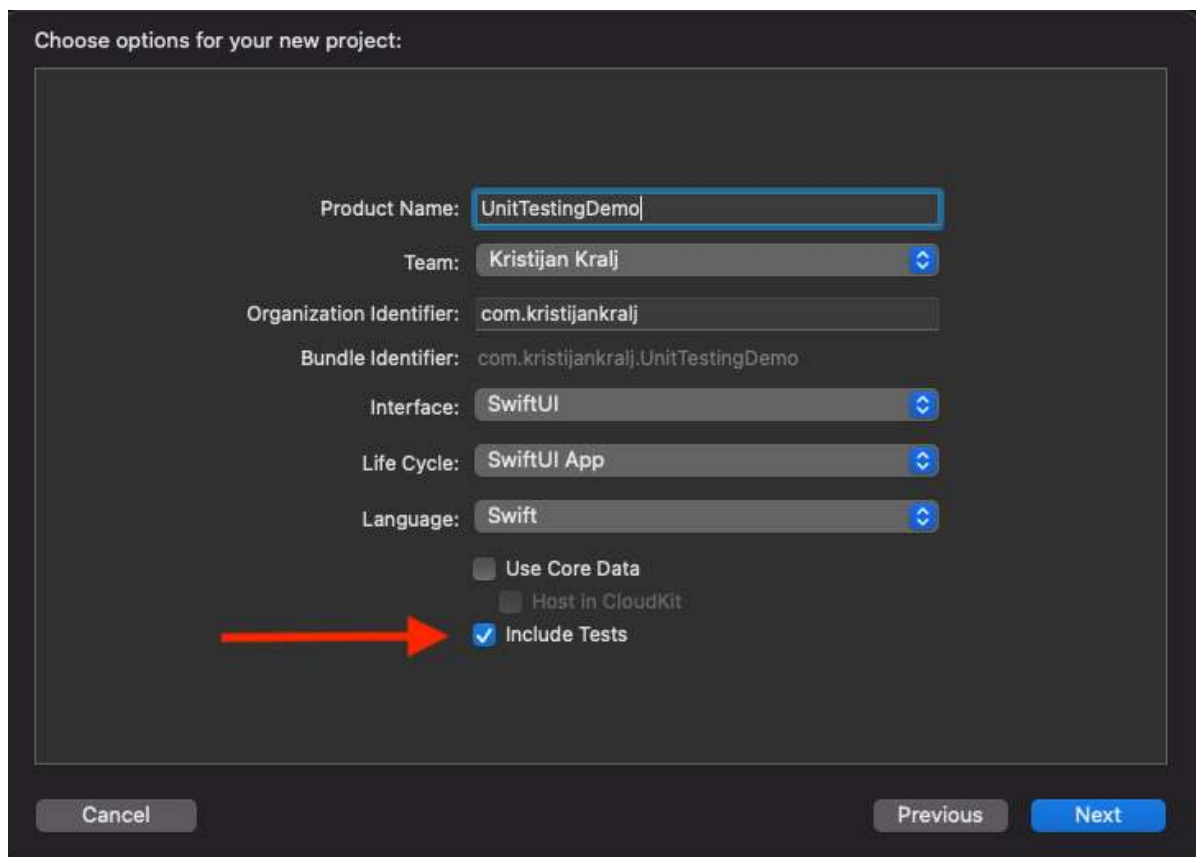
***ScenarioUnderTest** represents a scenario, or circumstances under which the unit of work is being executed.*

How to add tests to the Xcode project

Ok, the previous example was a simple way to show the elements of a unit test.

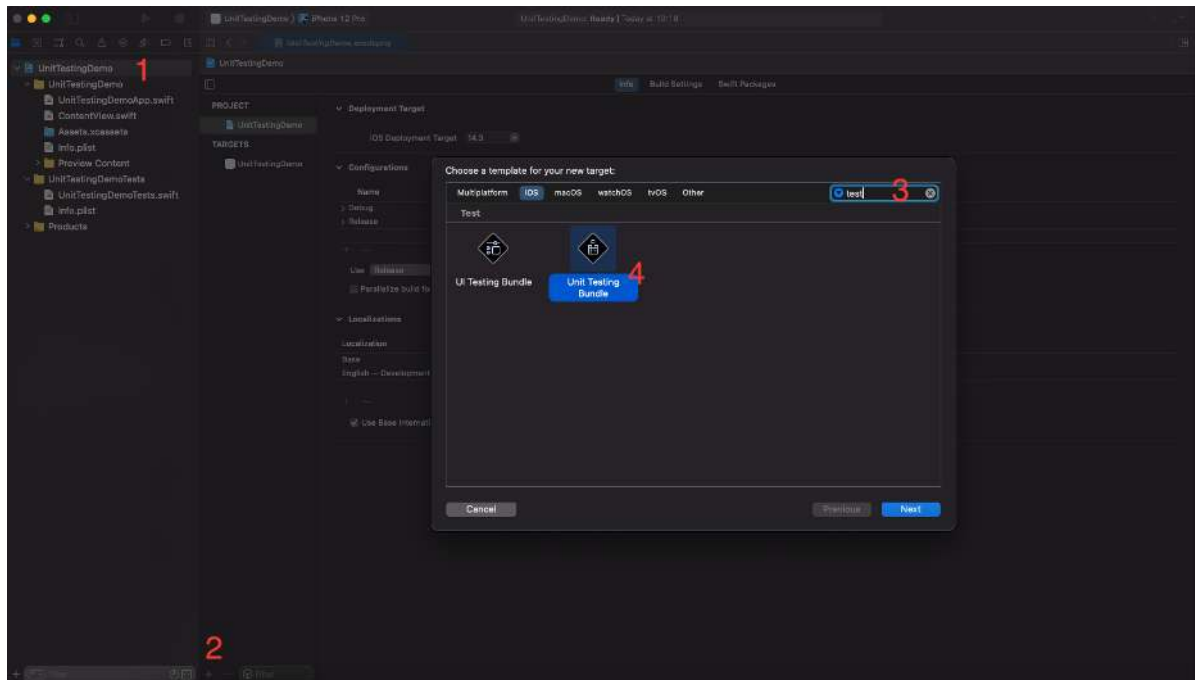
Next, let's go a bit further and see how to test more advanced objects.

Let's create a new project. While creating it, we need to make sure to tick the box that says "Include Tests". This will create a unit testing target.



Include tests while creating a new project

If you want to add unit testing target to the existing project, follow the next 4 steps:



Add unit tests to the existing project

1. Click on your project.
2. Tap on the plus sign, to add another target.
3. Filter by entering “test” search term.
4. Choose the **Unit Testing Bundle**.

By default, the unit testing target comes with a test class that has the following methods:

- `setUpWithError` – this method is called before the invocation of each test method in the class
- `tearDownWithError` – this method is called after the invocation of each test method in the class
- `testExample` – an example of a functional test case
- `testPerformanceExample` – an example of a performance test case

To the Moon and back

We create a class called Rocket, which we will test:

```

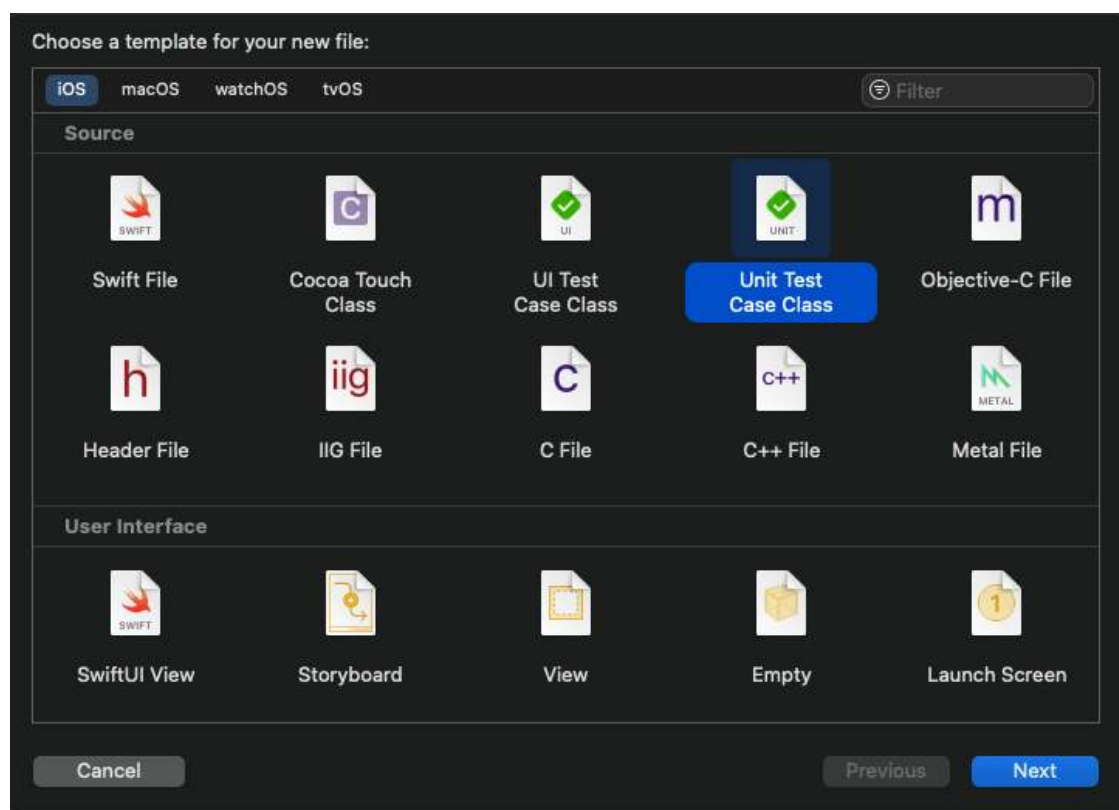
class Rocket {
    var speed: Int
    var enginesAreWorking: Bool
    var direction: String

    init() {
        speed = 0
        enginesAreWorking = false
        direction = "No direction..."
    }

    func flyToTheMoon() {
        speed = 29000
        enginesAreWorking = true
        direction = "To the Moon!"
    }
}

```

Now we create a unit test to check that, when we launch the rocket to the Moon, that it reaches the desired speed, engines are working and the direction is correct. To create new class, right click on the unit tests folder [NewFile...-> UnitTestCaseClass](#) :



Add new unit test class

We name our test class **RocketTests**. The naming convention is *ClassName + Tests*(ending with plural Tests, since every test class usually contains more than one test).

In order for the Rocket class to be visible in the unit testing target, you need to add the following line after `import XCTest` :

```
@testable import UnitTestingDemo
```

The test for the `flyToTheMoon` function looks like:

```
func test_flyToTheMoon_goes_to_the_moon() {  
    let rocket = Rocket()  
  
    rocket.flyToTheMoon()  
  
    XCTAssertEqual(29000, rocket.speed)  
    XCTAssertTrue(rocket.enginesAreWorking)  
    XCTAssertEqual("To the Moon!", rocket.direction)  
}
```

In the Arrange step of the test, we initialize the `Rocket` class, then we execute the `flyToTheMoon` function in the Act step. Finally, in the Assert step, we check that the properties `speed` , `enginesAreWorking` and `direction` are set to the expected values.

We run our tests by navigating to `Product-> Test` , or Cmd+U.

Conclusion

Testing is a very important part of every application project. It makes sure the application behaves as expected, minimizes the number of bugs and improves our confidence while refactoring.

That's why it should be part of every developer's toolbox.

The code from this chapter is available [on the Github](#).

The Lost Art of the Test-Driven Development

I don't play the lottery. I don't care what my horoscope says.

I think most things about the world could be improved if people thought more about what they're doing.

When someone gets upset with their computer, I tend to side with the computer. I think art is overrated, and tests are underrated. In fact, I don't understand why tests aren't art.

- Max Berry + my edit in underscores

When it comes to coding, there are a ton of ways to get the job done. Some developers prefer to dive head-first into their code, getting as much done as possible as fast as possible. Others prefer to take a more measured approach, where they start with the big picture and work out smaller tasks as they go. But one of the most popular approaches today is one that sits in between these two extremes: test-driven development.

We can summarize test-driven development as a red, green, refactor programming process. Write a failing test (**red**), make it pass by writing a missing implementation (**green**), and make changes to the code so it's production-ready (**refactor**). It looks so easy, like a painting made by some famous painter. *So easy, almost like an art.*

Therefore, tests are written before the code, before the actual implementation. The tests guide the developer to the path of the actual implementation. They drive the implementation, hence the name test-driven development (TDD).

TDD cycle

The process of test-driven development follows the following phases, as described in [Kent Beck's book Test-Driven Development](#):

1. For new, missing functionality, add a small test that indicates that there is a missing implementation. A compile error also counts as a failing test.
2. Run all tests to be sure that the new test fails, and all the existing tests pass. This is a very important step. You want to be 100% sure that all the existing tests are fine. In case you add a missing implementation and the new test passes, but one of the existing tests fails, you know for sure that the new code has broken the test.
3. Make the change. The implementation at this stage should be a quick and dirty solution, code copy/pasted from stack overflow, or from the existing codebase.

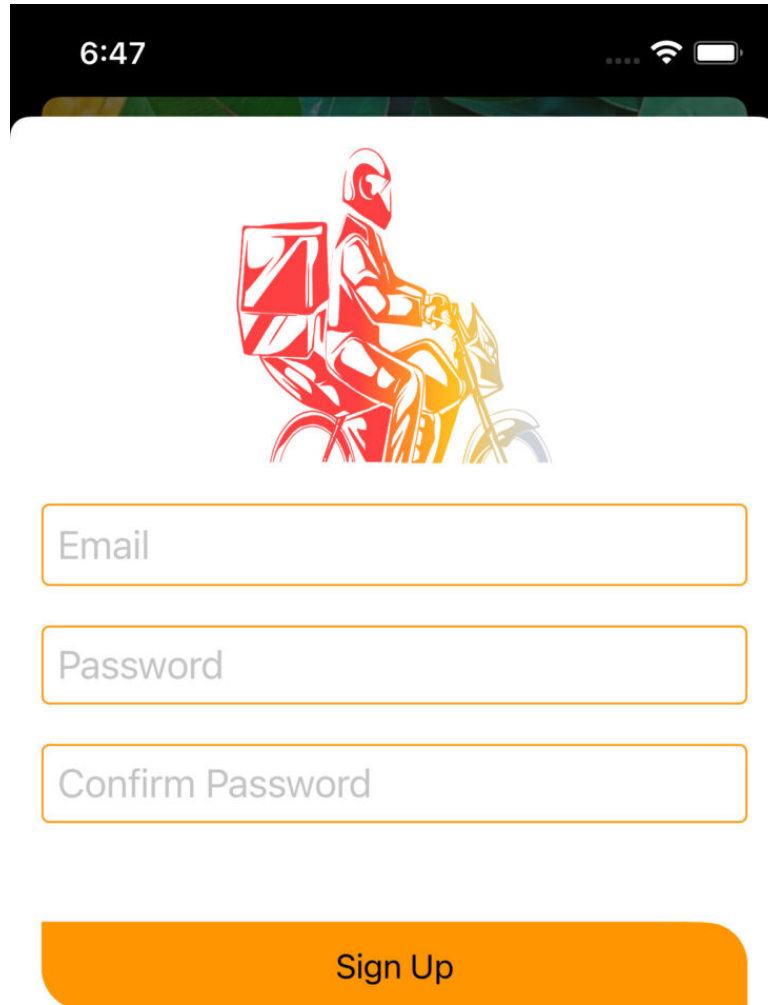
Whoah, whoah, wait a minute. What kind of unholy word are you preaching here, I hear you ask? Shouldn't I always write clean code, and avoid duplication, I hear you ask? Well, you will get to it. This step is not about production-ready code, it's about the code that works, and turns the test to green. After you make a solution, any solution that works, you will make the production-ready code in step 5.

4. All tests should pass now. If that's the case, then great, you can move to the last step. But if some tests fail now, you should go back to step 3, and make changes to our solution, so that all tests pass.
5. The last step, refactoring, is very important and it shouldn't be skipped. At this point, when your tests pass, and you have a working solution, remove any duplication introduced in step 3, and make all necessary refactorings. Some useful resources to be better at this step are:

- [Martin Fowler: Refactoring](#)
- [Joshua Kerievsky: Refactoring to patterns](#)

Excuse me, sir, may I check your password?

Let's go through one example, to see how TDD works in practice. Suppose you were given a task of making a new registration form. The next image shows a simple registration form.



The image shows a mobile application interface for a registration form. At the top, there is a status bar with the time 6:47, signal strength, Wi-Fi, and battery icons. Below the status bar is a header image of a delivery person on a motorcycle. The form consists of three text input fields labeled 'Email', 'Password', and 'Confirm Password'. Below these fields is a large orange button labeled 'Sign Up'.

And the first task you need to do is to make sure the password your users type in is secure enough.



Sorry but your password must contain:
an uppercase letter,
a number,
a haiku,
a gang sign,
a hieroglyph,
and the blood of a virgin!

The password in the app has to:

1. be at least 8 characters
2. contain an upper-case character
3. contain a digit

(we won't ask for the blood of a virgin in our app.)

You will place this logic in the `isPasswordSecure` function of the `RegisterViewModel`.

Start by creating a failing test to check the first requirement.

```
func test_isPasswordSecure_returns_false_if_password_has_less_than_8_characters() {

    let registerViewModel = RegisterViewModel()

    let result = registerViewModel.isPasswordSecure("1234567")

    XCTAssertFalse(result)
}
```

If you build and run the tests, you'll get the following error:



✖ Cannot find type 'RegisterViewModel' in scope
RegisterViewModelTests.swift

The error marks start of the journey

This can be fixed by adding the following class:

```
class RegisterViewModel {
    func isPasswordSecure(_ password:String) -> Bool {
        return true
    }
}
```

Run the tests, and it fails again, this time with `XCTAssertFalse failed` error message.

Let's make the test pass by implementing the following in the `isPasswordSecure`:

```
guard password.count >= 8 else { return false }
```

Now the test passes. A quick look at the code, nothing to refactor for now.

Banging out more tests

The second requirement is next. First, the failing test.

```
func test_isPasswordSecure_returns_false_if_password_does_not_contain_uppercase_character() {
    let registerViewModel = RegisterViewModel()

    let result = registerViewModel.isPasswordSecure("12345678")

    XCTAssertFalse(result)
}
```

The implementation, put this after the first guard clause:

```
guard let _ = password.rangeOfCharacter(from: CharacterSet(charactersIn: "ABCDEFGHIJKLMNOPQRSTUVWXYZ")) else { return false }
```

The second test now passes. The last step is refactoring. By looking at the tests, the initialization of the `RegisterViewModel` is duplicated in both tests. This can be refactored by applying the extract method refactoring. `RegistrationViewModelTests` after refactoring looks like this:

```

import XCTest
@testable import YouHungry

class RegisterViewModelTests: XCTestCase {

    func test_isPasswordSecure_returns_false_if_password_has_less_than_8_characters() {
        let registerViewModel = createRegisterViewModel()

        let result = registerViewModel.isPasswordSecure("1234567")

        XCTAssertFalse(result)
    }

    func test_isPasswordSecure_returns_false_if_password_does_not_contain_uppercase_character() {
        let registerViewModel = createRegisterViewModel()

        let result = registerViewModel.isPasswordSecure("12345678")

        XCTAssertFalse(result)
    }

    fileprivate func createRegisterViewModel() -> RegisterViewModel {
        return RegisterViewModel()
    }
}

```

The last requirement is that a password needs to contain a digit. The failing test:

```

func test_isPasswordSecure_returns_false_if_password_does_not_contain_digit()
{
    let registerViewModel = createRegisterViewModel()

    let result = registerViewModel.isPasswordSecure("ABCDEFGH")

    XCTAssertFalse(result)
}

```

And the implementation:

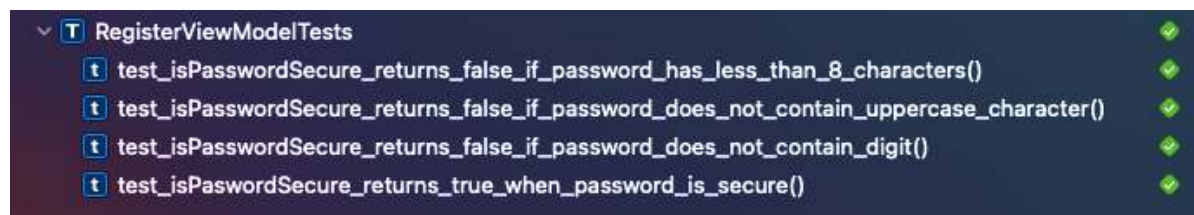
```
guard let _ = password.rangeOfCharacter(from: CharacterSet(charactersIn: "1234567890")) else { return false }
```

The test passes now.

The last test we can write is to check that `isPasswordSecure` returns true when the password is ok:

```
func test_isPaswordSecure_returns_true_when_password_is_secure() {  
    let registerViewModel = createRegisterViewModel()  
  
    let result = registerViewModel.isPasswordSecure("ABCDEFGH112")  
  
    XCTAssertTrue(result)  
}
```

All tests pass now, which means we are done.



All good, chief!

The final implementation of `isPasswordSecure` function looks like this:

```
func isPasswordSecure(_ password: String) -> Bool {  
    guard password.count >= 8 else { return false }  
  
    guard let _ = password.rangeOfCharacter(from: CharacterSet(charactersIn: "ABCDEFGHIJKLMN  
OPQRSTUVWXYZ")) else { return false }  
  
    guard let _ = password.rangeOfCharacter(from: CharacterSet(charactersIn: "1234567890"))  
    else { return false }  
  
    return true  
}
```

You are done with the implementation, and you have tests covering the new code.

Great!

The guaranteed method for avoiding fixing the same bug over and over

Test-driven development is a great way to produce high-quality code, with high test coverage. The example shown in this post goes through implementing a new feature using TDD, but TDD can be used for fixing bugs as well. The process when fixing a bug is:

1. Write a failing test that indicates the code is broken.
2. Find a fix the bug
3. Run all tests, all should pass now
4. Refactor if necessary

Conclusion

As a software engineer, it is not hard to see why so many people these days are against the idea of Test-Driven Development (TDD). Having to create tests for something that is not yet built seems weird, some say, which is why they choose to either skip TDD altogether or do it only occasionally. The truth is TDD is not about how it feels when you code, but it's about mastering a skill that will make you a better engineer.

The code from this chapter is available [on the Github](#).

Improve the Quality of Your SwiftUI App With UI Tests

How do you know whether or not you have a fever?

Sure, you kinda know since you feel sick, but the only way to be sure is by testing your body's temperature.

How does your girl/woman (or you, if you are female) know that she is pregnant?

She may have symptoms, but the only way to confirm her doubts is with the pregnancy test.



Honey, I've got some positive news.
We're going to need
...a bigger car,
...a bigger house,
...aaand you are going to need a bigger paycheck.

Then how then do you know that your code works? You might check it by manually running your app and check that things work. But the only fool-proof way to be sure that the code works is by writing tests.

Tests don't lie and don't have emotions.

How to copy and paste your app users' moves

Apple defines UI tests as a way to *"Make sure that your app's user interface behaves correctly when expected actions are performed."*

With UI testing we can find and interact with UI elements, and validate UI properties and state.

Three main classes for UI testing are:

- XCUIApplication
- XCUIElement
- XCUIQuery

Example:

```
//application
let app = XCUIApplication()
app.launch()

//element and query
let addButton = app.buttons["Add"]
addButton.tap()

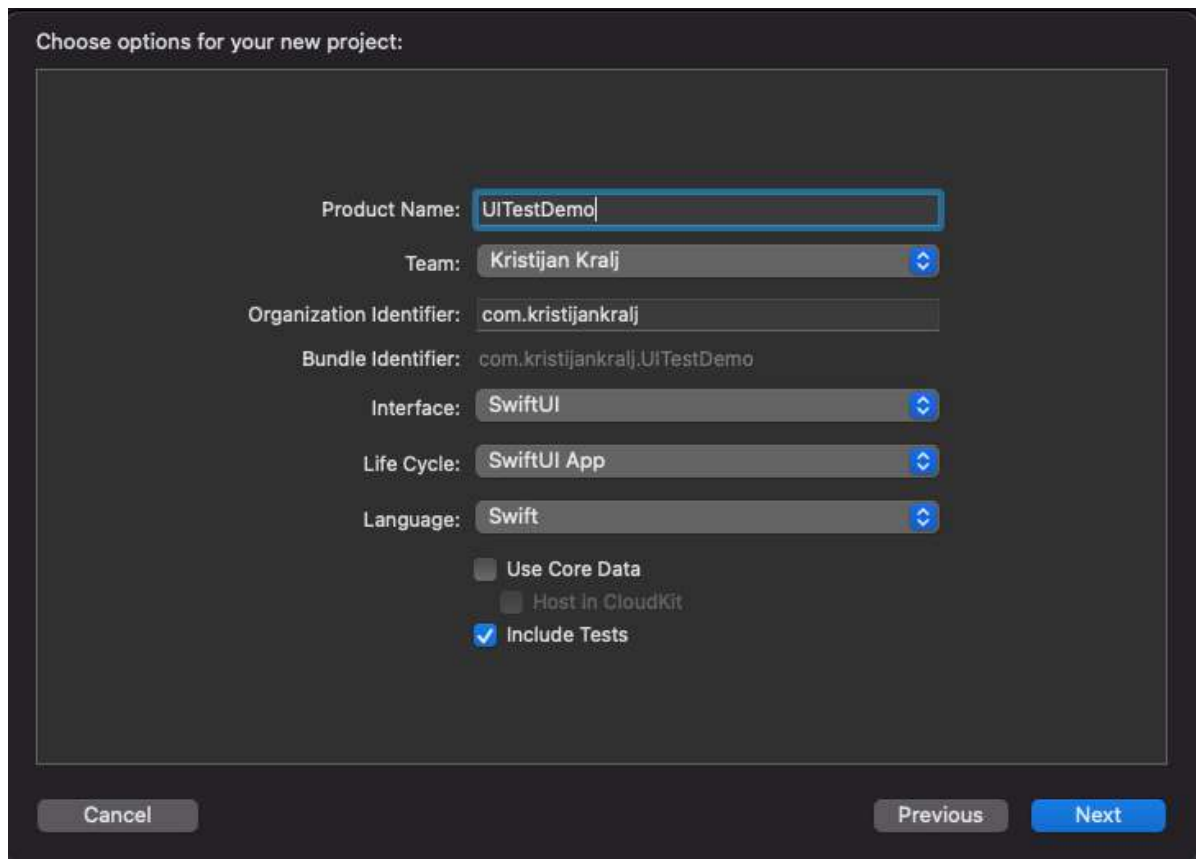
//assertion
XCTAssertEqual(app.tables.cells.count, 1)
```

We launch the app using `launch` method of the `XCUIApplication` class, use `buttons["Add"]` query to find `XCUIElement`, and `XCTAssertEqual` to validate app UI state.

How to add UI tests to a project

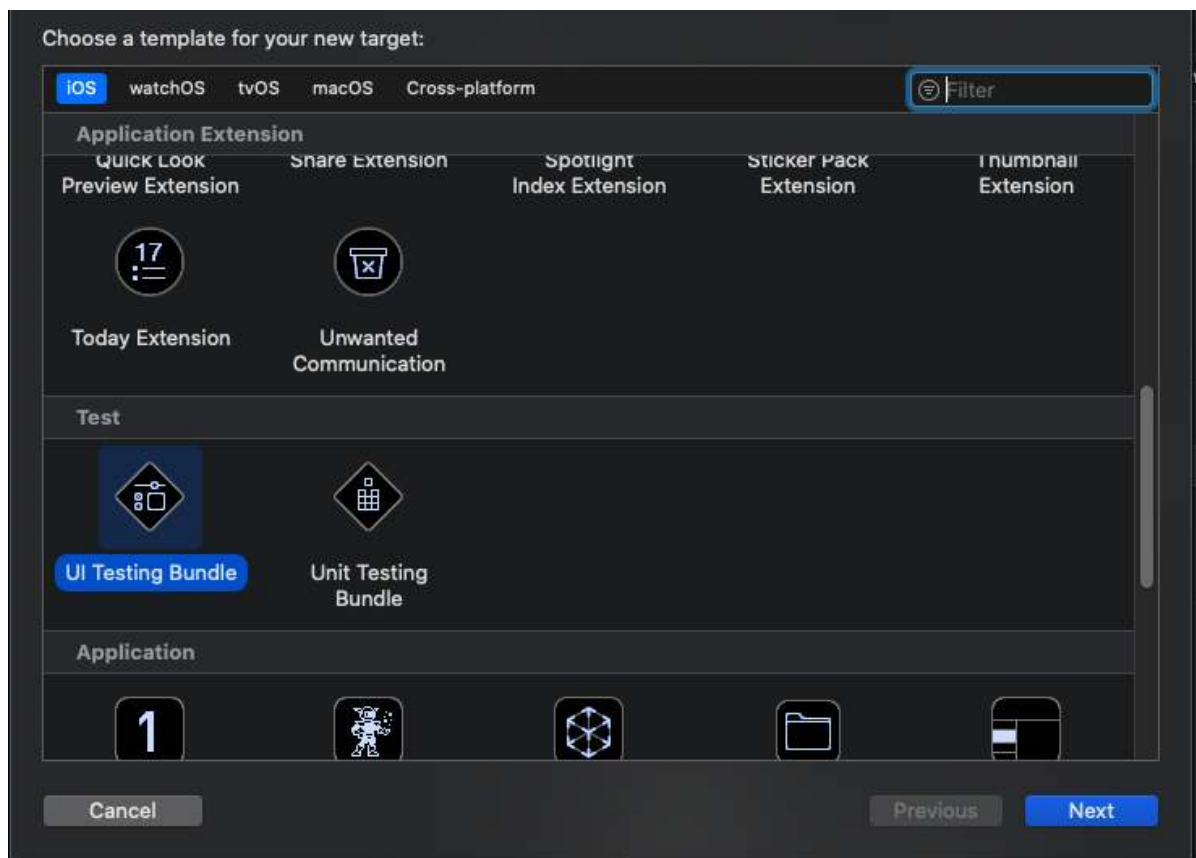
The UI tests require a separate target within your Xcode project. There are two ways to add the UI Tests target:

1. While creating a new project, make sure the `Include Tests` is checked.



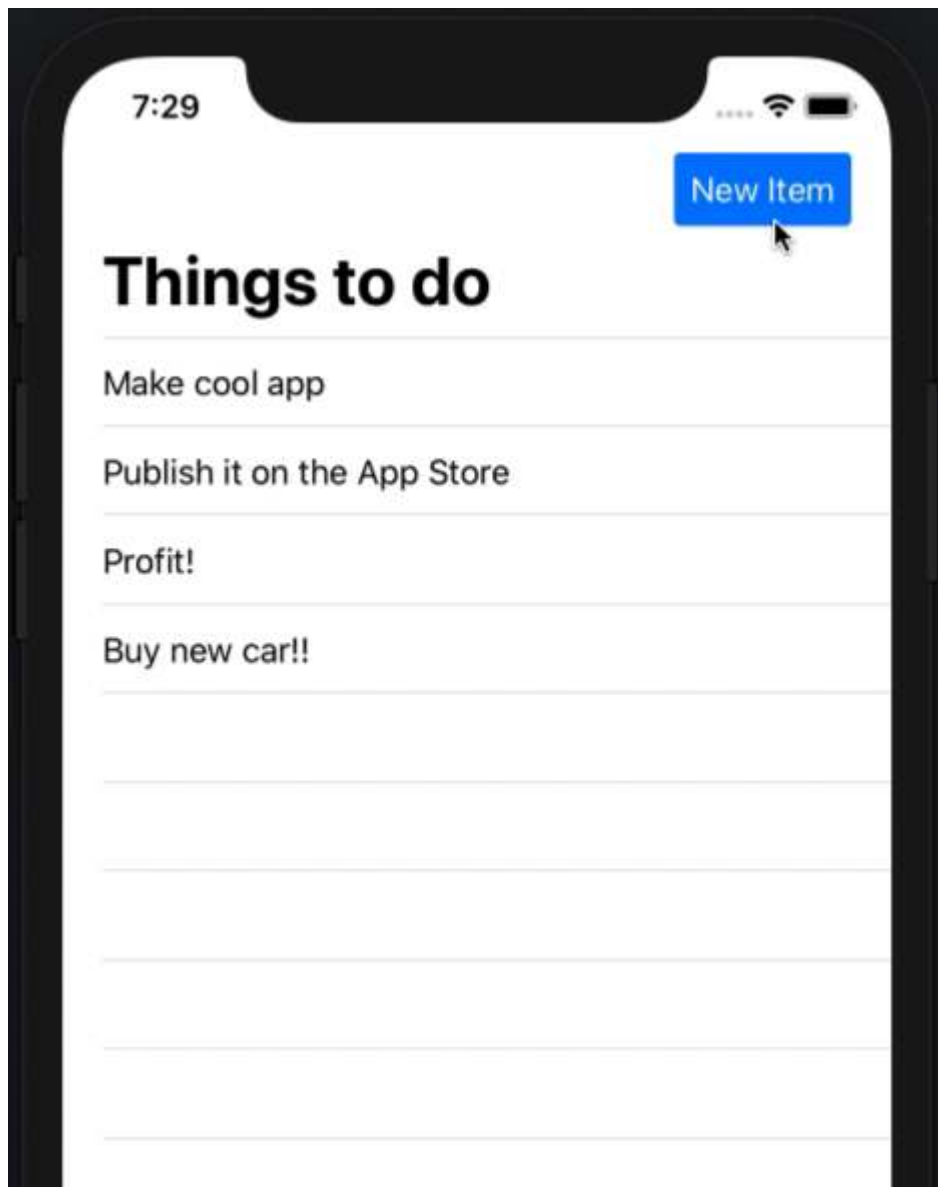
This will create both unit testing and UI testing targets.

2. In an existing project, click on `File->New->Target...` and select `UI TestingBundle` under `Test` section.



Create your first UI test

You are going to test the simple To-Do application written in SwiftUI. The app displays a list of tasks, and it's possible to delete a task by left swiping on it.



You can download the project from the [Github](#) to follow along.

The first test you are going to create will test that a new item can be added to the list.

Open the `UITestDemoUITests` file, and put a new breakpoint in the `test_item_can_be_added_to_the_list`, below the line:

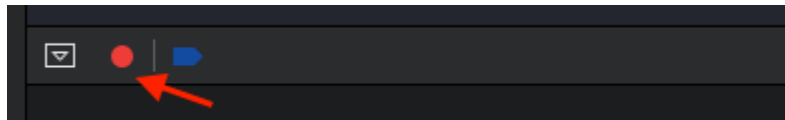
```
app.launch()
```

We are going to use the Test Recording option in Xcode to record our test. This will automatically add the code necessary for testing that the item can be added to the list. Nice!



It's time to record some code!

The record button is in the left part of the Debug area.



Run the first test, and it will stop at the breakpoint. Hit the record button and add the new item in the application. The code will be automatically generated for every action you take.

The test recorder will generate similar code to this:

```
app.navigationBars["Things to do"].buttons["New Item"].tap()

let todoItem = app.textFields["Enter name"]

todoItem.tap()
todoItem.typeText("Buy a house")
app.buttons["Add"].tap()
```

The last thing you need to do is to add the assert that the newly created item exists:

```
XCTAssertTrue(app.tables.staticTexts["Buy a house"].exists)
```

Add one more UI test

Let's add one more test. This test is going to check that the item can be deleted.

In the `test_item_can_be_removed_from_the_list` test, just below the `app.launch()` line, create a breakpoint and run the test.

Once the execution stops at the breakpoint, delete the first item in the list.

The completed test looks like:

```
func test_item_can_be_removed_from_the_list() {
    let app = XCUIApplication()
    app.launch()
    let tablesQuery = XCUIApplication().tables

    let firstItem = tablesQuery.cells["Make cool app"].children(matching: .other).element(boundBy:0)
    firstItem.swipeLeft()
    tablesQuery.buttons["Delete"].tap()

    XCTAssertFalse(app.tables.staticTexts["Make cool app"].exists)
}
```

Useful WWDC videos for UITests

Here is the list of WWDC videos related to the UI testing:

- [WWDC 2015 – UI Testing in Xcode](#)
- [WWDC 2016 – Advanced Testing and Continuous Integration](#)
- [WWDC 2017 – What's New in Testing](#)
- [WWDC 2017 – Engineering for Testability](#)
- [WWDC 2018 – What's New in Testing](#)
- [WWDC 2019 – Testing in Xcode](#)
- [WWDC 2020 - Handle interruptions and alerts in UI tests](#)

Conclusion

When to use unit tests, when to use UI tests? Use unit tests to test business logic and smaller pieces of your app. A unit test failure will show more precisely where the issue is.

On the other hand, UI testing covers broader aspects of functionality. You should use both in your code.

UI Testing candidates:

- common workflows
- custom views
- interaction with documents, opening and saving

The code from this post is available [on the Github](#).

References

1. Feitelson, Dror G., Eitan Frachtenberg, and Kent L. Beck. "Development and deployment at Facebook." IEEE Internet Computing 4 (2013): 8-17.
2. Copeland, Patrick. "Google's Innovation Factory: Testing, Culture, and Infrastructure." Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. IEEE, 2010.
3. Nundri. "iOS Testing Pyramid."
<https://engineering.linkedin.com/blog/2016/11/ios-test-pyramid>
4. Avtenied, Nikolai. "Unit testing and software quality, empirical research results." <https://www.linkedin.com/pulse/unit-testing-software-quality-empirical-research-results-avteniev/>