This ebook is still in progress, so I'd really appreciate your feedback via email rafael.oshiro@gmail.com or Twitter @roshiro

Enjoy the reading!

Rafael Oshiro

1. Introduction

About Unit Testing Software

When I first heard of unit testing it was back when the testing culture of TDD was not well spread and testing had no space in the developers conversations. I didn't have experience working with it and didn't really know how important it was, it was just boring to me. Actually I just realized how important unit testing was a few years ago, when I started working with a team with a strong TDD culture. Today I can say you will probably find unit testing more fun after you understand what you actually need to test, and this is what I want to achieve with this book.

"I just realized how important unit testing was a few years ago"

If you have no experience with testing at all or have already tried writing some tests before and still struggle when you need to unit test your code, this book will be very useful for you. **The objective of this book is to teach you the fundamental concepts of unit testing that you**

must know before start writing your tests in any language. Also, I tried to put here some examples of real world scenarios that you'll most likely find at work, some scenarios that were really tricky for me when I was starting with testing.

One problem I always see with newcomers to unit testing is that they do have the knowledge of testing framework, but they lack knowledge of the concepts of unit testing and I strongly believe all developers should learn these them before diving into the specifics of testing frameworks. Even good developer struggle to write tests when they don't know what and how to test their classes. **If you learn the concepts of unit testing, you'll have no problems implementing tests in Ruby, Java, Javascript, .NET or any other language, because all testing frameworks follow the same idea**.

For example, if you take a student of computer science with no knowledge of Object-Oriented programming and try to teach him Java, how do you think he is going to perform? Surely he will have big problems with the basic jargon like Classes, Objects, Inheritance, etc. Learning the foundations on which the language is based on is essential for the good understanding of the technology. The same applies to testing, if you don't understand the base of unit testing, you'll have problems understanding the technology.

Is this book for me?

In a nutshell this book is a good fit for developers who have little or no experience at all with unit testing, regardless of the language they work on.

I find this book is a great fit for anyone who is completely new to software testing because it gives them clear direction on where to start. And this book is also a good fit for developers who have already written a few tests in jUnit, RSpec, Jasmine or any other testing framework out there but are still insecure about why and when to use all the fundamental concepts of testing.

How the book is organized

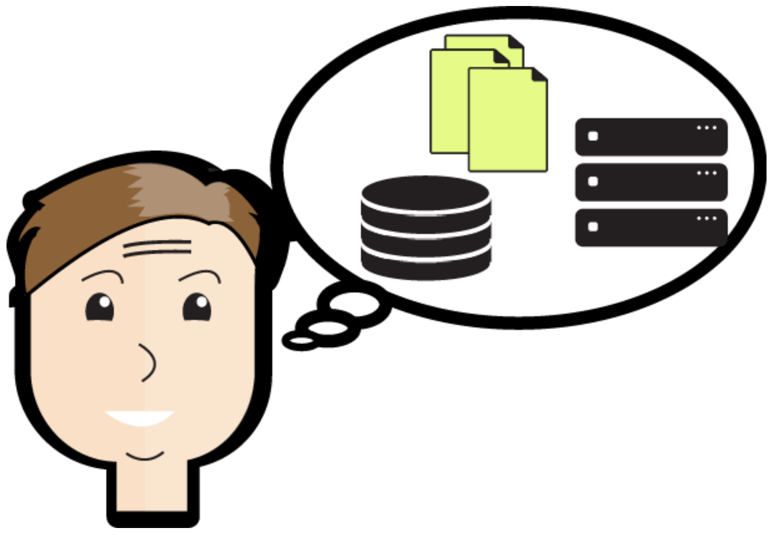
This book is organized in 7 parts: Introduction, What unit testing is and what's not, What makes a good unit test, Unit Testing, The challenges of testing legacy code, Code coverage, Continuous Integration, Testing frameworks and Mind Map. All of them are full of important information and I would recommend not to skip them. In the last chapter, Mind Maps, I attached a mind map which is in my opinion a great way to hack your brain to store everything you learned throughout this book. You can revisit it whenever you need to refresh your mind.

2. What unit is testing and what's not

Unit testing is as the name suggests is the test we write for a single unit of code in isolation. By isolation I mean it cannot rely/depend on other pieces of code, third party libraries, database, file system, network, etc.

There are strong reasons why you should write tests in isolation without relying on external resources and in my opinion this is the most important concept to always keep in mind. Following this concept will make the learning process smoother and more enjoyable, also it will allow you to write better tests.

"... whenever I need to write tests for my classes I always try to identify first what are the external resources"



Personally, whenever I need to write tests for my classes I always try to identify first what are the external resources my class depends on and then I can focus on applying the correct techniques to break these dependencies. There's an entire section in Chapter 2 dedicated to breaking dependencies of unit tests.

Imagine you are developing an iPhone app to allow users to upload photos from their phones to a server in the cloud. Once the photo is uploaded, the server returns a boolean response indicating it received the photo successfully or not. In case the upload process fails for some reason, such as loss of connection, the app has to catch this exception and inform the user about that unexpected error.

Even without looking at the code of this app in question, we can easily identify some external resources being used by the app:

- Internet Connection;
- Photo to be uploaded;
- Server on the cloud where the photo will be uploaded to;

Based on what you read at the beginning of this topic, when writing unit tests we should not rely on external resources because a **unit test should be written in a way that it can be run at any time, as many times as we want and it should be fast**. It doesn't any make sense to set up the whole environment the app needs in order to just run its unit tests. It means for example, if the machine where the unit tests are running doesn't have internet connection, the tests should not fail.

And that probably leads to another question:

Does it mean unit tests don't ensure the app works as a whole?

That's an interesting question and the answer is yes, unit testing does not ensure the app works as a group (your code + external resources). The responsibility of making sure all the parts that compose the app works perfectly integrated is from the **Integration Test**.

"The goal of unit tests is to ensure the logic and the link/the interaction between our code and the third party components are done correctly"

The goal of unit tests is to ensure **the logic and the link/the interaction** between our code and the third party components are done correctly and the code being tested produces the expected output for all different supported scenarios. So if a developer is fixing a bug and wrongly changes the way the code interact with an external resource, the tests should fail.

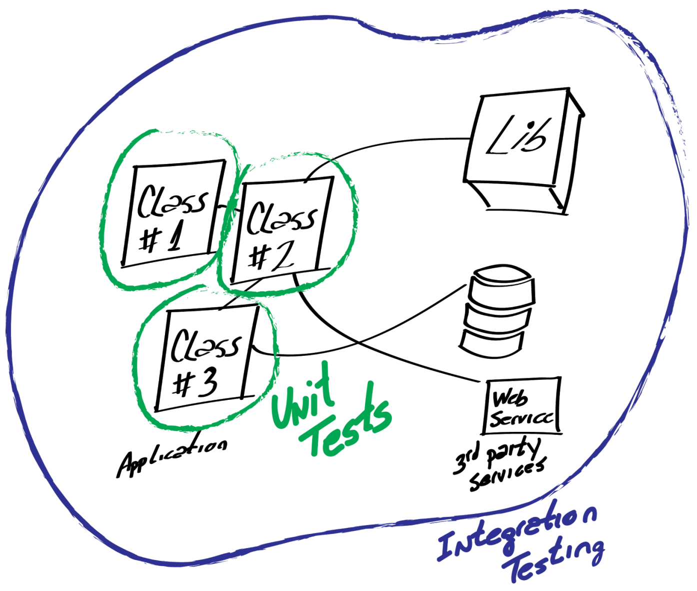
Doing that we are confident that the app does work as expected in isolation. In the future, when a bug fix or a refactoring is done, we can run all the unit tests and they all should pass. If it doesn't, it means a bug has been introduced and it can be tracked down and fixed before the deploy in production.

Integration Testing is not Unit Testing

In summary integration testing ensures all the parts that compose your application works together, including third party resources like databases, email delivery, file system, etc and these tests are usually performed by dedicated testers or developers, or both together it all varies from company to company.

"***Different from unit testing, in Integration Testing everything is real."***

Integration testing takes place after unit testing is done. And it is performed following a **Test Case**, which consists of a series of steps and the expected result. A test case ensures a specific functionality in the application produces the expected result.



Unit testing cover each single class and their link with external resources. Integration testing covers the whole system.

Different from unit testing, in integration testing everything is real, nothing is stubbed or mocked (these concepts will be covered in part 2). It means it will ensure all parts of the application works well together. Integration testing is a complement of unit testing.

An interesting scenario where integration testing complements unit testing is for example when the credentials for the SMTP server in the configuration file of the application is wrong. Unit tests won't be able to catch it since the unit test will not try to connect to the SMTP server and send a test email and check if it was successful or not. Now, when **integration test** the testers will notice something is wrong in the application when the email he or she should receive hasn't arrived. That would probably indicate a problem in link between the application and the SMTP server.

Another example to show the importance of integration test is to ensure all the use cases/scenario were implemented as described in the requirements. When the developer does not implement a specific scenario the unit tests will pass, but the integration test will probably fail since the test cases should cover the missing scenario.

Integration test suites should be run whenever bug fixes are introduced, new functionality are added, refactorings are done or any other changes in the codebase are put in place, to ensure the system works as expected.

User Acceptance Testing is not Unit Testing

Different from unit testing and integration testing, user acceptance testing (UAT) is performed by real users of the application, because they have the knowledge of the business requirements and since they will be the end users, they will know if the system works for them. For these tests, the application runs in a proper UAT environment, with their own database setup with a set of data very similar to the real world in order to reproduce real scenarios as close as possible, sometimes the data is a copy from the production.

"UAT is performed by the real users of the application..."

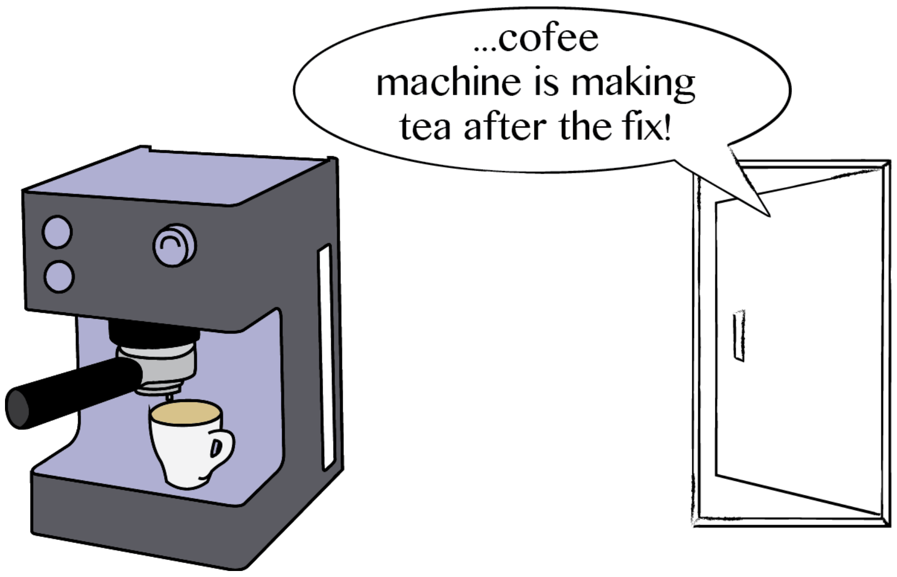
The goal of this test is to ensure the functionalities, business logic and workflow are implemented correctly according to the requirements.

Regression Testing is not Unit Testing

Have you had this experience where one bug is fixed and another one introduced? Or even worse, after fixing one bug you re-introduce others that had already been fixed ages ago?

Everybody knows this sometimes happens, and that's when regression testing comes into play.

"...it ensure what was working before the bug fix still works after the fix"



Regression testing is carried out after bugs are fixed, it ensure what was working before the bug fix still works after the fix. The objective of this type of test is retest the application even the areas that were not changed to ensure the bug fixes didn't break any existing functionality and also to ensure the previous behaviour still produces the same desired behaviour.

As regression testing is laborious and time consuming, it's a good idea to **automate them** using proper tools, this reduces the amount of manual re-test.