Mocks, Fakes & Stubs

... and other techniques
in Test Driven Development,
illustrated with Code Kata Examples

Emily Bache

Mocks, Fakes and Stubs

and other techniques in Test Driven Development, illustrated with Code Katas

Emily Bache

This book is for sale at http://leanpub.com/mocks-fakes-stubs

This version was published on 2013-12-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 Emily Bache

Tweet This Book!

Please help Emily Bache by spreading the word about this book on Twitter!

The suggested hashtag for this book is #mocksfakesstubs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#mocksfakesstubs

Also By Emily Bache

The Coding Dojo Handbook

Contents

Introduction

If you're interested in improving your practice at programming in general, and Test Driven Development in particular, I'd suggest doing code katas is a good way to learn. There's a host of programmers around the world using code katas to hone their skills and learn from one another, in a coding dojo, at a code retreat, in a pair on a lunchbreak, or just by themselves. This book uses Code Katas to illustrate some modern coding techniques, so you can learn about, and practice them.

A code kata is a simple coding problem that may take a little effort to solve the first time, but the point is you don't only do it once. Start over from scratch and try it again, but think more carefully about *how* you're solving it. Can you work in small steps, refactoring often, and keep your design simple and clean? Can you write a good suite of automated tests that support your development and provide good regression protection for future maintenance work? The challenge in a Code Kata is to solve the problem well, and to demonstrate good practices in *how* you reach your solution. If you're using Code Katas like this, you'll find you build up a repertoire of katas that you know well, and can use to polish your coding technique.

Code Katas are also useful when you're trying out something new - a tool, technique or coding environment. Keep one variable the same - the problem to solve - and see how your coding experience, design, and tests are affected by the new element. It's this way of using Code Katas that is the focus of this book. I'm assuming you are familiar with each code kata, and can use it to show you a new technique.

In this book I'll show you examples including, but not limited to, Mocks Fakes and Stubs. I'll also discuss techniques like London School TDD, Outside-In development, and Approval testing. I'll begin each topic with a discussion of the theory, and follow it up with some code listings and sample tests. My aim is that it'll be a little like pair programming with someone experienced in the technique, who can explain what they mean both in theory and example, and give you some commentary about when you might choose to use it. You'll definitely be expected to put in some work yourself, learning the Code Katas, and trying the techniques out for yourself. It'll certainly be more fun if you can get a group together and start a Coding Dojo.

My first book, "The Coding Dojo Handbook"[1], is full of practical advice about how to set up this kind of a space for learning. While I was working on that book, I found myself writing long passages about particular techniques, and how you could learn about them using some of the code katas listed in the catalogue in that book. In the end, I decided that material didn't belong in "The Coding Dojo Handbook", and that it would be better to start a new book.

The techniques in "Mocks, Fakes and Stubs" are all useful to practice in a Coding Dojo environment, but not exclusively. I've included examples taken from a range of programming languages, including

[1] http://leanpub.com/codingdojohandbook

Java, Python, C++ and C#. You can do a Code Kata in pretty much any language you feel like, and you'll find and starting code is available in many popular languages on my Github page[2]. Almost all of the code katas used in this book are listed in the Kata Catalogue from "The Coding Dojo Handbook".

[2] http://github.com/emilybache

Acknowledgments

By publishing this work-in-progress on LeanPub, I hope to receive feedback in the form of comments from the wider community. If you are kind enough to do so, I will be very happy to write your name here in the finished book.

I would like to thank Llewellyn Falco and Geoff Bache, who both reviewed the section on Approval Testing, and Steve Freeman, Jon Jagger, Brian Marick and Seb Rose, who all helped me to understand London School TDD better.

My sincere thanks to other early readers who provided me with feedback, including Olof Bjarnason, Javier Gutierrez, Nat Pryce.

How to Read This Book

I suggest you read the introduction to each section, to get a feel for whether you're interested in the technique. If you are, then I suggest you put the book down at that point, and go and try out the code kata in question for yourself. Then when you think you know the kata a little, read the rest of the section. As you read it, follow along in your editor, and do the kata again, following the advice in the text.

I hope the experience will be a little like pair programming with someone who knows the technique well.

Code Kata

In Karate, a Kata is a kind of exercise you use during practice, to learn techniques. A Kata comprises a very detailed sequence of moves, that a practitioner will repeat over and over again as they practice. The idea is that eventually the Kata moves become imprinted in your "muscle memory", and you can do them without any particular concious thought. Then when you need the moves and techniques in an unscripted fight or sparring situation, they come totally naturally.

Dave Thomas, (co-author of "The Pragmatic Programmer"), suggested that as programmers, we could benefit from similarly practicing "Code Kata" exercises. He wrote a blog post[3] where he presented the idea, and also some sample Kata exercises. Since then many others have contributed new Code Katas, and much practicing has happened!

Two kinds of practice

Actually, there are two distinct kinds of practice you can do. The one you're probably familiar with is "Incidental Practice". This is when you repeat the same exercise over and over again, so that the moves become ingrained and unconcious. It's a way to develop your *habits*, the behaviours that you fall into when you're not really thinking about what you're doing.

> Kent Beck famously said:
>
> *"I'm not a great programmer. I'm just a good programmer with great habits"*
>
> Well, clearly Kent Beck *is* a great programmer, but it's rather interesting that he thinks habits are so important. When you're under pressure, you're tired, there's a manager standing over you stressing about a deadline… do you still write tests? Do you still refactor? Is that behaviour ingrained enough - is it your habit?

The second kind of practice is "Deliberate Practice". This is when you try to do something you *can't comfortably do*. You deliberately seek out experiences that will stretch you - just enough - so you have to think carefully about what's happening, and conciously force yourself to choose to do things in a particular way.

Another aspect of Deliberate Practice is what you do when you find something too hard. If, even with concious effort, you find the technique too difficult - you break it down into components that you can practice separately.

[3] http://codekata.pragprog.com/

> Musicians do a lot of practice, including Deliberate Practice. Perhaps you're playing a piece and it sounds good until you reach a tricky section - for instance there are many notes to fit into a short space of time, or particular notes that are hard to reach on your instrument. You pause, and just play the tricky section by itself, slowly, several times. Once you can succeed in playing that part slowly, you play it again, a little faster, and continue repeating it a little faster each time, until you can play the whole piece at the correct speed.
>
> What you're doing here is breaking down the technique into parts - first playing the notes in the correct order, then the problem of playing them fast enough.

I think you can do either kind of practice with a Code Kata, and it's even more fun if you can arrange to do the practice together with others, in the Coding Dojo for example. That's what my first book, "The Coding Dojo Handbook"[4] is all about.

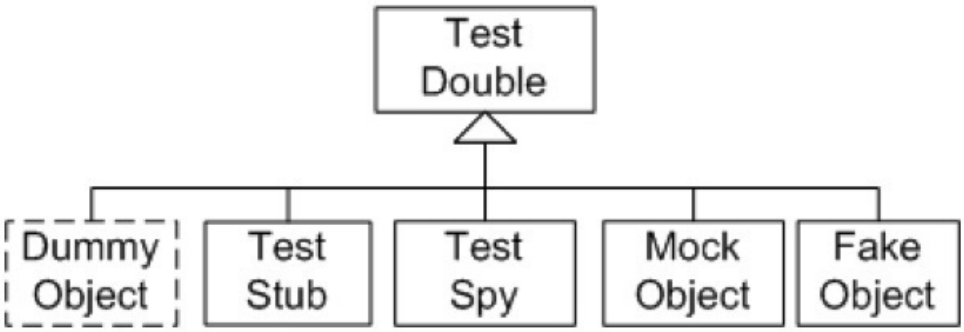
In this book we'll be looking more closely at how to use Code Katas to learn techniques. What you'll find is that most of the time I'm doing deliberate practice - using a Kata to expand my repertoire of techniques. If you already know the Kata, and have done it before using familiar techniques, it ought to be easier to see the new technique and how it relates to the ones you already know. So I do recommend that you try out the katas for yourself as well as read the book. Then you can use the katas as a basis for your own deliberate practice.

[4] https://leanpub.com/codingdojohandbook

Test Doubles

Many classes need collaborators in order to do their work. When you're unit testing a particular class, you might be able to use its real collaborators, but in some cases that's not convenient. You can replace the real collaborators with various kinds of "test double". Think of a "test double" as being like a "stunt double". When you're filming a movie you don't want the real actor jumping out of aeroplanes or taming wild horses, you get someone more expendable!

Gerard Meszaros came up with the umbrella term "test double" in his book "xUnit Test Patterns"[5] to try to clear up the terminology. Many people say "mock" when they mean "stub" or "fake", and it can be quite confusing. It doesn't help that you use a "Mocking Framework" to create all sorts. I'm preferring the term "Isolation Framework" because that's the effect using them has - to isolate an object for testing purposes.



Mocks, Fakes and Stubs are all kinds of Test Double

In addition to the three basic types - Mocks, Fakes and Stubs - you have a "Test Spy" which is any kind of test double that you can interrogate afterwards about what the class under test did. There's also a "Dummy Object" which isn't really a test double since it doesn't interact with the class under test, but it's often referred to as one anyway.

In the rest of this chapter I'd like to illustrate the distinction between the various kinds of test double with some examples. Before I do, you might want to have a go at these Code Katas for yourself. I think you'll understand things better if you do. The katas in question are Tyre Pressure, Unicode To Html Converter, (These two are in my github repo called "Racing-Car-Katas"[6]), Single Sign On[7], and FizzBuzz.

[5] http://xunitpatterns.com

[6] https://github.com/emilybache/Racing-Car-Katas

[7] https://github.com/emilybache/Single-Sign-On-Kata

Approval Testing

Approval Testing is a simple idea, and in fact many people do it already without using a special testing framework. Often when you write a test, the assert statement will be comparing two strings. One way to write the test is to start out with just an empty string in the "expected" part of the assert statement, even though that's not the correct behaviour. When the production code works, you just copy the actual output and paste over the empty expected string. It's often a little less work, particularly if calculating the exact expected string up front is tricky. This is what I'd call *approving a result*, rather than making an assertion up front, hence the name "Approval Testing".

Classic TDD tells you to completely define the test up front, before you work on the implementation to make the test pass. Approval Testing tells you to still fully define the "Arrange" and "Act" parts of the test up front, but only partially define the "Assert" part. You decide what behaviour you're going to assert on, but not exactly what that behaviour should be. You then go through an iterative process of developing the functionality and checking the actual output by hand, until you decide it's correct. Then you "Approve" the result and save it in the test. The outcome from both Classic and Approval testing approaches is similar - a repeatable regression test that you can run at any time to check your program still works.

Minesweeper example

Note: Approval Testing approaches normally make use of a tool, which I'll explain about later. For this example I'm going to keep it very simple and show you how you'd do it in plain JUnit.

If you were implementing Minesweeper with an Approval testing approach, you'd start by writing a failing test:

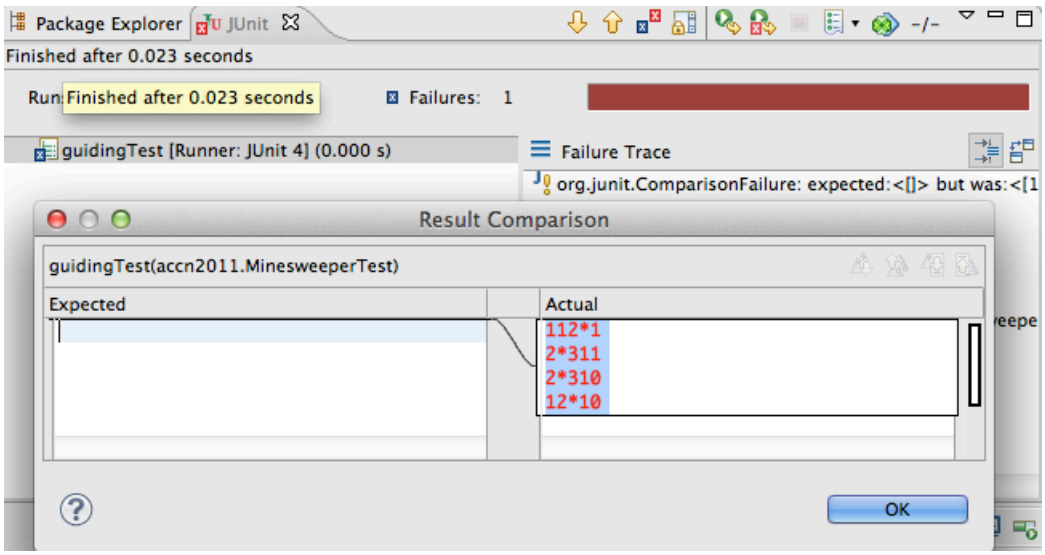
```
@Test
public void guidingTest() {
    String minefield = "...*." + "\n" +
                       ".*..." + "\n" +
                       ".*..." + "\n" +
                       "..*.." + "\n";

    String expectedClues = "";
    assertEquals(expectedClues, new Minefield(minefield).clues());
}
```

Partially defined approval test for Minesweeper

Writing this much of the test has let you make some important design decisions. You've decided to construct a Minefield object using a string, and that this object has a "clues" method that also

returns a string. Whether those were good design decisions or not is up to debate! The way the kata is defined does require you to read in a minefield as a string, and return clues as a string - so even if you decide on different classes and methods than this, I think your guiding test will be somewhat similar. I don't think this design is disastrously bad, anyway, and it's enough for this example! What I want you to see is that this is an Approval Test rather than a Classic TDD test. You haven't calculated exactly what the expected clues should be at this point.

You can use this test to get started on implementing functionality, and you may choose to write some finer-grained intermediate tests as well. For example, tests that check you can parse the input string, and tests that check you can count mines neighbouring a particular square. At some you should have written enough production code that you'll be able to run the guiding test and see a failure:



Test failure information when functionality is in place, but test is still incomplete

By carefully looking at this failure information, you decide your program actually has the correct behaviour now. You "approve" it by copying and pasting the actual output string into your test. You now have a fully defined regression test:

```
@Test
public void guidingTest() {
    String minefield = "...*." + "\n" +
                       ".*..." + "\n" +
                       ".*..." + "\n" +
                       "..*.." + "\n";

    String expectedClues = "112*1" + "\n" +
                           "2*311" + "\n" +
                           "2*310" + "\n" +
                           "12*10" + "\n";
    assertEquals(expectedClues, new Minefield(minefield).clues());
}
```

a regression test for Minesweeper

In this case we're not using any special approval testing framework, and pasting the string is a little more work that it might be. Java doesn't handle multi-line strings particularly elegantly! In general, I wouldn't recommend using an Approval Testing approach without tool support, whichever programming language you're using. I'll talk more about that later in this chapter.

The Golden Master

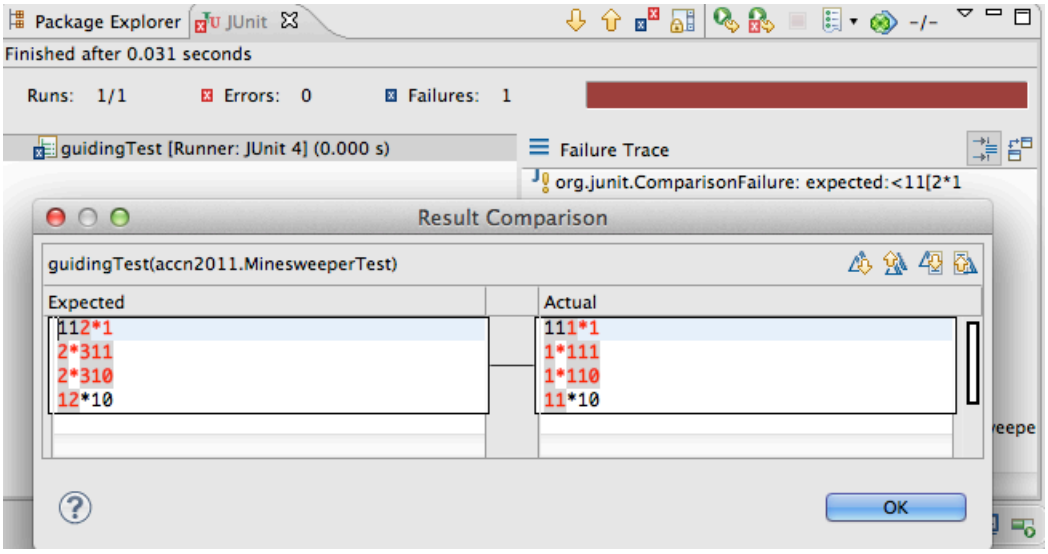
When you approve a result, what you're actually doing is saving a result which is the gold standard you'll refer to and compare against in future test runs. We call this approved version of the output the "Golden Master". In a unit test like this, it's just a string hard coded in a source file, but more usually, you'll store your "Golden Master" in a separate file.

The "Golden Master" could be any kind of output from your program that can be compared against new output from subsequent runs. That could be for example a screenshot, the contents of a database table or a binary format message that will be sent through a socket. Having said that, by far the easiest kind of Golden Master to work with uses plain text. For example, the result of "toString" on an object, a JSON document, a log file, an ASCII art "screenshot"…

There are countless tools for manipulating, storing, comparing and updating plain text. That's why you sometimes hear this approach referred to as "Text-Based Testing" - most of the time, your Golden Master is a text file.

Detecting regressions

Going back to our Minesweeper example, imagine you make a refactoring mistake. You see an unexpected test failure like this:



a failing regression test for Minesweeper

Looking at the diff, you can see you've probably introduced an error in the part of the code that counts how many mines are next to each square, or in the part that stores this information. The error doesn't look like being in the part that formats the output into a string. This may be enough of a clue (!) for you to identify your mistake and correct it straight away.

If you've written other tests covering these parts of the code, hopefully one of them will now be failing too. If not, you may want to write some more fine-grained unit tests to further pin down exactly where the error is. The point is, the Approval test works well as a regression test. I think it gives you more than that though, the approval test helps you to smoothly handle requirements changes.

Managing behaviour

Another possible scenario for the test failure above, would be that the change was deliberate. The requirements changed, the customer decided a boolean grid answering the question "am I next to a mine" is all that's required, a heat map is overkill. So one possible outcome from this situation is that you decide to update the test. You "approve" the new output, in this example by pasting the new output over the old expected output in the test definition. In other words, you update your "Golden Master".

```
@Test
public void guidingTest() {
    String minefield = "...*." + "\n" +
                       ".*..." + "\n" +
                       ".*..." + "\n" +
                       "..*.." + "\n";

    String expectedClues = "111*1" + "\n" +
                           "1*111" + "\n" +
                           "1*110" + "\n" +
                           "11*10" + "\n";
    assertEquals(expectedClues, new Minefield(minefield).clues());
}
```

updated Golden Master for Minesweeper

With proper tool support, approving a new Golden Master becomes quick and easy. In fact, you might find yourself doing it rather often. While developing a new piece of functionality you might save intermediate results, for example, when the output improves from an empty string to a string of the right length, you could update the Golden Master. You can then continue to develop, updating the Golden Master several times as you iterate your way towards the complete feature.

Working this way with tests enables a change of view compared with Classic TDD. You start to see your tests as a way to define the current behaviour of your system, and to help you manage changes. They are not assertions of universal, eternal truth, they are a snapshot of the current truth. I find these kinds of tests to be really agile, treating changes in functionality as the norm rather than the exception.

Tools for Approval Testing

In the minesweeper example above, the fully defined Approval test is pretty similar to a test you might have defined up front in a Classic TDD approach. That's not always the case. Usually the string you're asserting on will be longer - too long to be kept in the source file. Because of this, and the way you want to be able to update the Golden Master easily, using a approval testing tool becomes essential.

There are two main tools I know of that support Approval Testing. One is "ApprovalTests"[8], and the other is "TextTest"[9][10]. Both work with a wide variety of programming languages, and help you to manage a test suite with many Golden Master files. The main difference is that ApprovalTests integrates with a unit testing framework, and TextTest accesses your program via a command line interface. There is also an add-on tool for TextTest called StoryText that allows you to test via a GUI.

Anything I write here about these tools is likely to soon become out of date, so I suggest you go and look at the online documentation.

[8] http://approvaltests.com

[9] http://texttest.org

[10] Full Disclosure: I am married to Geoff Bache, the creator of TextTest.

Legacy Code

In the minesweeper example, I've described a TDD process where you're developing the test cases alongside the production code. Of course, if the system is already built and the functionality works, it can be quite straightforward to create a regression suite using an Approval Testing approach. It should be a case of identifying input from actual usage, and recording the corresponding output in Golden Master files.

If the internal design of a system is poor, it can be very difficult to start with a unit testing approach. You get a chicken and egg situation where you can't add unit tests because there are no units, and you can't refactor to create units, because there are no unit tests to lean on! In this situation, you could add Approval tests for the whole system, to put a safety net in place for refactoring.

If the system already produces an output that is suitable to use as a Golden Master, such as a plain text message, receipt, or batch run report, that's ideal. An alternative, is to have the system create a log of what it's doing, and use that. In most systems, it's cheap and risk-free to add log statements, no matter how horribly gnarly the design is. You can add targeted log statements in places where the system makes important decisions, or changes important data. When you come to refactor, you move the code around, the log statements move too, and your tests should stay green. It can give you enough leverage to get your legacy code under control.

You could use Approval tests as a kind of "scaffolding" that you put in place for a short time while you do some major remodeling, then discard once you've got a better structure that you can unit test. Or maybe you'll find you can get good enough regression protection and fast enough feedback with just Approval tests, and never get around to writing any unit tests. I've known it go either way, depending on the codebase and the preferences of the programmers working on it.

Guru Checks Output

In Classic TDD you fully define your tests up front. One criticism that's often leveled at Approval Testing is that if you don't do this, you could easily make a mistake and save the wrong result as your Golden Master.

> *"This is an anti-pattern - a bad idea - called "Guru Checks Output". Checking the output of a computer is very error-prone. It's easy to look at the numbers and decide they look correct. It's also easy to be wrong when you do that. Far better to have the expected answers up front, even if they have to be computed by hand."*
>
> – Ron Jeffries et al, "Extreme Programming Installed", p33.

Ron is concerned that if you don't actually really know what the correct output ought to look like, you can deceive yourself into thinking your implementation is correct when it's not. It's the double-

entry bookkeeping aspect of TDD - you calculate the answer twice, once to define the tests and once when you write the production code. With Approval Testing you only do the calculation once, in the production code. When you define the tests though, you do have to decide on "Arrange" and "Act", and think about whether you're going to be capable of recognizing the correct behaviour in the "Assert". If you're worried it'll be unclear, perhaps you should use a different testing approach.

Having practiced an Approval Testing approach for some years now, I really haven't found "Guru Checks Output" to be a big problem. The tools can warn you if you're trying to save a golden master with obvious errors in it, such as an empty file or a huge stack trace. If you do make a mistake and save the wrong thing, it doesn't take long for you to notice that tests aren't failing when they're supposed to. You've always got previous Golden Masters in your version control system to go back to, or you can just approve a new version.

Where Approval Testing shines and where it doesn't

As you saw in the minesweeper example, it can be a little less work to define an Approval test than a Classic TDD test, and this is particularly advantageous if calculating the "correct" result in advance is difficult. With a unit testing approach, you'll often restrict yourself to using examples where you can easily calculate the expected result.

In extreme cases, it might be impossible to decide in advance what the "correct" answer is - for example if you're implementing an optimizer that will use a heuristic to search an infinite set of possible answers, and return the one that it thinks has the lowest cost.[11] Defining the test up front would be as much work as implementing the software!

Writing good Approval tests is definitely a skill though. You can get into trouble with maintainability, as with any automated testing approach. It's important to define tests at the right level of abstraction. If the Golden Master contains low level details that are prone to change and you don't really care if they do change, your test will fail when you don't want it to. Ideally you should design tests that use a public, stable API, and check output that is part of the public contract for your application.

Another problem can be test isolation. In the minesweeper example, you might have several tests that check different minefields, covering different edge cases. (Yes, *real* edge cases!). If you decide to change the output format so a square not next to any mines changes from a "0" to a ".", all your tests will fail. What you need in this case is a way to bulk-update all your Golden Masters, but only with this one change. This is where a good tool is essential. It can help you spot all the tests that have changed in exactly the same way, and update them together. It can help you write a Regular Expression to search and replace in all the relevant Golden Master files.

[11] This is the actual situation that Geoff Bache faced that led him to design TextTest in the first place!

Further Reading

To learn more about the techniques in this book, it's good to go back to the people who originated them, and see what they've written.

- "Growing Object Oriented Software, Guided by Tests", Steve Freeman and Nat Price
- Approval Tests website - http://approvaltests.com
- TextTest website - http://texttest.org