A NEW LOOK AT TEST-DRIVEN DEVELOPMENT

Dave Astels

dastels@daveastels.com

The Problem

Test Driven Development (TDD) has made it to prime time. Big companies are paying big money to have their programmers trained in how to *do* TDD. It's a popular topic at conferences... agile and otherwise. There are a score of books available on TDD, mine even won a Jolt award. So it seems that everything is rosy? Everyone who's doing TDD is fully understanding it and getting the full benefit, right?

Fat Chance!

Too few people I talk to really understand what it's really about. That means that many people who practice TDD are not getting the full benefit from it. What's wrong?

The focus on testing

Well... one thing is that people think it's about testing. That's just not the case.

Sure, there are similarities, and you end up with a nice low level regression suite... but those are coincidental or happy side effects. So why have things come to this unhappy state of affairs? Why do so many not get it? Let's start with a bit of a history lesson. A few years ago I wrote an article on TDD for *The Coad Letter*[1]. In it I gave some background on TDD (from Ron Jeffries):

> "*XP originally had the rule to test everything that could possibly break. Now, however, the practice of testing in XP has evolved into Test-Driven Development.*"

So way back when, they were talking about writing tests. And that's probably why it has the testing centric vocabulary, and that is why people think it's about testing! What else would they think when they have to talk about **`TestCases`** & **`TestSuites`**, & **`Tests`**.. and have to name methods starting with "**`test`**". Granted, not all the xUnit frameworks have these requirements. The new version of jUnit (version 4) gets away from the naming conventions and reliance on subclassing **`TestCase`** but the rest of the vocabulary stays and the annotations used are test-centric

The thing is, when the evolution to TDD happened what we ended up with was a different kind of animal... not just a slight tweak on the original. The original XP folks were writing tests for everything that could break, then they started writing the tests first. Eventually we ended up at TDD where we write a small test to describe a small bit of new functionality, then code it, then

[1] http://bdn.borland.com/article/0,1410,29690,00.html "What is Test-Driven Development?"

write the next small test, and so on. But TDD is not the endpoint a lot of people seem to think it is... it's just a stepping-stone.

Units

Also, the idea of "unit" is a major problem. First of all it's a vague term, and second it implies a structural division of the code (i.e. people think that they have to test methods or classes). We shouldn't be thinking about *units*... we should be thinking about facets of behaviour.

Thinking about unit testing leads us to divide tests in a way that reflects the structural arrangement of the code. For example, having text classes and production classes in a 1-1 relationship

That's not what we want... we want behavioural divisions.. we want to work at a level of granularity much smaller than that of the typical unit test. As I've said before when talking about TDD, we should be working with very small, focused pieces of behaviour... one small aspect of a single method. Things like "*after the add() method is called with an object when the list is empty, there should be one thing in the list*". The method is being called in a very specific context, often with very specific argument, and with a very specific outcome. I even went so far as to write a blog entry titled "One Assert Per Test"[2].

So, there you have it. A fabulous idea... wrapped in packaging that causes people to think from a testing point of view.

The Result

Why is this a problem? Let's think for a minute about how people often think about *testing*.

Programmers often think "*I'm not going to write all those tests.*", "*It's really simple code, it doesn't need to be tested*", "*testing is a waste of time*", or "*I've done this (loop/data retrieval/functionalty, etc) millions of times.*".

Project managers often think "*we test after the code is done*", "*that's what we have a testing person for*", or "*we can't spend that time now*".

So with people thinking about testing, it's easy to come up with all sorts of negative reactions and reasons not to do it... especially when time gets short and the pressure's on.

[2] http://blog.daveastels.com/?p=3

So if it's not about testing, what's it about?

It's about figuring out what you are trying to do before you run off half-cocked to try to do it. You write a specification that nails down a small aspect of behaviour in a concise, unambiguous, and executable form. It's that simple. Does that mean you write tests? No. It means you write **specifications** of what your code will have to do. It means you specify the behaviour of your code ahead of time. But not far ahead of time. In fact, just before you write the code is best because that's when you have as much information at hand as you will up to that point. Like well done TDD, you work in tiny increments... specifying one small aspect of behaviour at a time, then implementing it.

When you realize that it's all about specifying behaviour and not writing tests, your point of view shifts. Suddenly the idea of having a *Test* class for each of your production classes is ridiculously limiting. And the thought of testing each of your methods with its own test method (in a 1-1 relationship) will be laughable.

Sapir-Whorf hypothesis

First some background, courtesy of Wikipedia[3]:

> The axiom that language has controlling effects upon thought can be traced to Wilhelm von Humboldt's essay "Über das vergleichende Sprachstudium", and the notion has been largely assimilated into Western thought. Karl Kerenyi began his 1976 English language translation of Dionysus with this passage:
>
> "*The interdependence of thought and speech makes it clear that languages are not so much a means of expressing truth that has already been established as means of discovering truth that was previously unknown. Their diversity is a diversity not of sounds and signs but of ways of looking at the world.*"

The hypothesis upon which I am basing my argument is named after the linguist and anthropologist Edward Sapir and his colleague and student Benjamin Whorf. It states that

> *"there is a systematic relationship between the grammatical categories of the language a person speaks and how that person both understands the world and behaves in it."*

For my purposes that means that the language you use shapes how you think... and if you want to change how you think it can help to first change your language.

---

[3]http://en.wikipedia.org/wiki/Sapir-Whorf_hypothesis

So what to do?

First stop thinking in terms of tests. As Bob Martin has been saying for a few years "Specification, not Verification". What Bob means is that verification (aka testing) is all about making sure (i.e. verifying) that your code functions correctly while specification is all about defining what it means (i.e. specifying) for your code to function correctly.

Using something like xUnit makes this hard, due to it's testing-centric language, so we need to start with a new framework for specifying behaviour. Dan North of ThoughtWorks has started the jBehave project to do just this. I and a few others are responsible for a behaviour specification framework for Ruby called rSpec, which we will explore in more detail below.

A Behaviour Specification Framework

So what does a behaviour specification look like? Well, a first pass will look and work a lot like xUnit since:

- it works quite well enough
- everyone is familiar with it

A major difference is vocabulary. Instead of subclassing **`TestCase`**, you subclass **`Context`**. Instead of writing methods that start with **`test_`**, you start them with **`should`**. It would be preferable that you don't have to worry about a naming pattern so you can choose the most appropriate name. Instead of doing verification with assertions (e.g. **`assertEquals(`*`expected`*`, `*`actual`*`)`**) you specify post conditions with something like **`shouldBeEqual(`*`actual`*`, `*`expected`*`)`**.

In Smalltalk, and Ruby, this can be even more natural if we embed the framework into the class library (a common approach in Smalltalk, by the way). You could write something like the examples shown in the following table:

JUNIT	SBSPEC	RSPEC
assertEquals(*expected*, *actual*)	*actual* shouldEqual: *expected*	*actual*.should_equal *expected*
assertNull(*result*)	*result* shouldBeNil	*result*.should_be_nil

JUNIT	SBSPEC	RSPEC

try {
  2 / 0;
  fail();
} catch (DivideByZero ex) {
}
	[2 / 0] shouldThrow: DivideByZero	{2 / 0}.should_throw DivideByZero

What Now?

As mentioned above, Dan North has started the jBehave project to create a jUnit replacement for behaviour specification that you can download and experiment with now.

If you are using Ruby, you can get our rSpec framework and start using it. At the time of this writing rSpec provides the following methods which can be called on any object... take note of that..ANY OBJECT:

- `should_equal(expected)`
- `should_not_equal(expected)`
- `should_be_same_as(expected)`
- `should_not_be_same_as(expected)`
- `should_match(expected)`
- `should_not_match(expected)`
- `should_be_nil()`
- `should_not_be_nil()`
- `should_be_empty()`
- `should_not_be_empty()`
- `should_include(sub)`
- `should_not_include(sub)`
- `should_be_true()`
- `should_be_false()`

All expectation methods will take an optional, trailing message parameter. Methods setup and teardown are available for overriding as in xUnit. They work the same and do the same thing.

So what does a behaviour specification look like? Well, here's the example from my TDD book, reworked for rSpec:

```
require 'spec'
require 'movie'
require 'movie_list'

class EmptyMovieList < Spec::Context

  def setup
    @list = MovieList.new
  end

  def should_have_size_of_0
    @list.size.should_equal 0
  end

  def should_not_include_star_wars
    @list.should_not_include "Star Wars"
  end

end

class OneMovieList < Spec::Context

  def setup
    @list = MovieList.new
    star_wars = Movie.new "Star Wars"
    @list.add star_wars
  end

  def should_have_size_of_1
    @list.size.should_equal 1
  end

  def should_include_star_wars
    @list.should_include "Star Wars"
  end

end
```

Guidelines

Name your Context classes by what they are focused on: `EmptyMovieList` and `OneMovieList`, for example. They contain only specification method directly related to that context.

Name your specification methods by the focus of that specification, e.g. `should_have_size_of_0`.

Together, the names of the Context class and expectation method should read well, to tell you exactly with is going on: `EmptyMovieList.should_have_size_of_0`. Just think of how easy it would be to extract a specification document from this.

Your specification methods should be as simple, short, and focused as you can make them. One expectation... excellent... that should be your holy grail. Simple is good. It's far better to have lots of small, simple, focused, understandable classes and methods than fewer large, bloated classes and long, complex methods.

Summary

- The problem I have with TDD is that its mindset takes us in a different direction... a wrong direction.
- We need to start thinking in terms of behavior specifications, not verification tests.
- The value of doing this will be thinking more clearly about each behaviour, relying less on testing by class or by method, and having better executable documentation.
- Since TDD is what it is, and everyone isn't about to change their meaning of that name (nor should we expect them to or want them to), we need a new name for this new way of working, which Dan North has given us... BDD: Behaviour Driven Development.

Acknowledgements

My thanks go out to Steven Baker, Gabriel Bauman, and Aslak Hellesoy for taking the first version of this article and starting work on a ruby framework to make the ideas concrete. Thanks to Kay Pentecost for her ideas on test-avoidance as well as feedback on draft versions.