Behavior Driven Development

Christopher C. Lamb

Department of Electrical and Computer Engineering
University of New Mexico

June 13, 2011

THE UNIVERSITY of NEW MEXICO

Outline

What is Behavior Driven Development?

Behavior Driven Development (BDD) is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.
— *Dan North, London, 2009*

Basically, this means that it is driven by specific, quantifiable value associated by produced code. The code is produced with an eye toward *verifiable quality*, and as much of the quality strategy as possible is automated. This implies that we have automated tests and automated builds. BDD builds on *Test Driven Development* (TDD), which in turn builds on disciplined unit testing. We'll cover all of this today.

So what is this testing of units?

Unit testing has been around for 40 years or so. Basically, you test the smallest units of code that are assembled into your system. People used to do it with mainframes. Automated testing's been around that long too, but people used to write their own test frameworks for this kind of thing. About 15 years ago, Things changed.

Kent Beck and Erich Gamma created JUnit, a unit testing framework to support the development methodology they were evangelizing (e.g. eXtreme Programming). Both of them are still around, and both were and are very influential. Now we test classes and methods.

This established a de-facto standard for unit test frameworks now referred to as *xUnit*.

Why do we do it?

Developing software is hard. It's getting harder and harder as time goes by. Software systems are becoming:

- *More Complex*
  Ten years ago distributed systems were hard. Mobile computing was just starting. Now everything's distributed, cloud computing is widely used by developers, mobile computing ubiquitous. This all depends on layers upon layers of usually third-party libraries.
- *Less Trustworthy*
  So we use all these new libraries to create software, how do we know that it's trustworthy? that we're using it correctly? When we need to change our code base, do we know we can do it without causing other problems?

Why do we do it?

This is much better than the way we used to develop and test software.

Back in the old days, most projects had dedicated testing teams, with legions of testers. These testers would test the system through some kind of user interface, something they built or one of the developers built, and then they would test the system according to some huge test plan that outlined thousands of steps they needed to go through to test the software.

This cost a fortune, and didn't work.

Why do we do it?

Just writing the test plan was a horrible, expensive experience. These things were huge, and took lots of thought and time. Then you needed an army to execute it, and it took forever, all of which cost $$$. And how can you test error conditions in your communication layer when it is two libraries removed from the user interface?

Answer: you can't, not through the UI.

Enter unit tests.

Why do we do it?

Unit testing solves many of these problems. Unit tests are:

- *Automated* — Huge test suites run in seconds, and they can be automated to run whenever you want them to. They can be configured to run whenever a developer checks *any changes whatsoever* into a code base.
- *Functional* — The tests not only test the code, they serve as examples of how the code is to be used.
- *Flexible* — You can now test that hard-to-reach communication layer by building unit tests right into that code.
- *Aggregateable* — You can pull together unit tests for all of your code into *test suites*.

How do we do it?

Generally, you will test individual units of functionality in your code. Thing classes, methods, functions. Unit tests are self contained, and not dependent on any single runtime environment. To do this you usually use mocks or stubs, which *emulate* dependencies. To do this, you need to design your code to be testable, usually using a pattern called *Dependency Injection*.

We can use unit testing frameworks to support a variety of other testing methods as well. You can use them for:

- *Functional testing* — no mocks, tied to runtime environment
- *Integration testing* — testing more than one unit; e.g. multiple layers or components

Unit testing allows more complete testing, less complexity, and better software through:

- documentation of programmatic use
- better design, forces use in more than one domain, design for test
- more agility, easier to change your software
- ease of later integration

Now keep in mind, unit tests aren't free.

They take time to write, and if you're not careful you can create *fragile tests*, which break easily when the code is changed, leading you to spend an inordinate amount of time altering your test suite. These are the two most sensible arguments people use against unit testing. Less sensible arguments include the **my code doesn't need them** or the **I already know how to do my job** arguments.

So what is Test Driven Development?

Well, it's another way to build software.

Remember, even if unit testing has been around for a while, it was primarily a niche practice. For the most part, industry was using large test teams, understood that they didn't work well, but they knew how to do it. So that's what people did. When unit testing started to gain more mainstream acceptance, the flexibility of the practice (and the xUnit frameworks) allowed developers to change how they did their jobs, to enable them to produce better code.

Some renegade developers started *writing tests for code that didn't exist*, and then they would create code so the tests would pass.

Ergo, **Test Driven Development** was born.

How does it work?

You use a simple algorithm:

1. *select a feature*
2. *write some tests*
3. *run tests (they should all fail)*
4. *implement feature a bit a a time, running tests after each bit of work until all tests pass*
5. *repeat*

That's it!

Metrics for tests?

Okay, so now, if we're doing TDD, we're writing tests before we're writing code. How can we tell we're writing good tests?

Writing software is *hard*, remember? It's challenging to write good tests that don't need to be reworked again and again. Some rules of thumb:

- *Think about design* — Most agile developers think that agile means you don't do design or architecture. This isn't true — rather, you do as *little* design and architecture as possible to build a quality product that meets needs (you write as little code as possible too). With TDD you can divide the task into smaller and smaller units to test (down to the method level) which makes it easier to think about issues like error handling and argument checking. Your tests should reflect this.
- *Black/white/grey box testing* — Try to stay away from examining class internals. They tend to change more than interfaces after the interfaces are established. Be wary of using white or grey box testing and avoid it if you can.

Metrics for tests?

- *Refactor* — One of the keys to using agile methods to develop good software is refactoring. Just like you refactor your code when you find it needs it, you need to aggressively to this with your tests as well.
- *Metrics* — Test coverage is a worthwhile metric, and there's plenty of tools that can generate that for you. Don't get lulled into a false sense of security though — if your tests are poor, or you're not testing the right things, this will be misleading. You'll also find that some areas need multiple tests under multiple different conditions (e.g. boundary conditions, error conditions, etc.)

What is Behavior Driven Development Then?

So, remember the definition from the first slide: **Behavior Driven Development (BDD) is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.**
— *Dan North, London, 2009*

This sure sounds nifty. But what does it really mean, and what kind of things do you use to do it?

- *Test Driven Development* — We've covered this; though BDD testing is slightly different.
- *Domain Driven Design* — A software design approach.
- *Scrum (or similar)* — An iterative software development methodology based on short cycles focused on clear user stories.

We'll touch on the last two now.

Domain Driven Design

Essentially, Domain Driven Design is a method of building software that separates system functions from domain functions, leading to cleaner, easier to understand code.

It does this by using groups of patterns to separate *data* from *application* functionality.

For more info:
http://www.infoq.com/minibooks/domain-driven-design-quickly

And Scrum, etc.?

These are various types of agile methodologies. They're characterized by:

- *Fluid Requirements* — They recognize that users don't usually know what they want until they see what they *don't* want.
- *Customer Involvement* — Customers are members of the team, go to meetings, sit with the developers, the whole bit.
- *Daily Standup Meetings* — The team gets together every morning to check the pulse of the project.
- *Iterative* — They use short iterations (2-6 weeks) to keep on target. At the end of each iteration, the client sees the current system and determines direction.

They all have tweaks beyond this, but this is really the core of most agile methodologies.

How BDD Testing differs

In the past, in Ruby, we'd use something like Test::Unit, an xUnit compatible unit testing suite delivered with Ruby installations. Your tests would look something like this:

```
class MyTests

def test_this
# Test something...
end

end
```

Pretty straightfoward. You can see what the tests are, code 'em up, run 'em, all that good stuff.

BDD Tests are Different!

BDD test differ, and that's show in the syntax you use. Generally, they're supposed to be clearly understandable and written in near english. You *describe* activities by illustrating what *it should* do under certain conditions.

In Ruby, we use RSpec to run these, and rspec tests look like this:

```
describe 'create activity' do

it 'should be creatable with a block' do
# test creating with a block
end

it 'should be creatable without a block' do
# test creating without a block
end

end
```

So what?

The nice thing about RSPec tests is that they are:

- *Clear* — The whole *describe it should...* makes the *intent* of the tests very clear, even to the author.
- *Limited* — The syntax also tends to enforce a more black-box testing style. Here, we're constantly reminded that we're testing what a component *should* do under specific condition, and that helps us from getting ourselves in trouble by testing things that perhaps we shouldn't.

The end result? **better tests**.

Questions?

Questions?