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*

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.

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.

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.

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.

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?

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*.

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.


THE UNIVERSITY of
NEW MEXICO