

Better Testing Through Behaviour

Tom Adams

Workingmouse
303 Coronation Drive
Milton QLD 4064 Australia
+61 7 3369 1933

tom.adams@workingmouse.com

ABSTRACT

Test Driven Development (TDD) is an established development practice that provides significant benefits throughout the entire software development process. While TDD provides many advantages, it is often met with resistance and can be easily misused. Behaviour Driven Development (BDD) is a refinement to TDD that shifts the emphasis from testing to specification. BDD practitioners cite several advantages to this approach covering organisational, managerial and technical TDD concerns. This paper explores the benefits of BDD using Instinct, a purpose built open source Java BDD framework. Instinct provides flexible annotation of behaviour contexts, specifications and actors; automatic creation of test doubles and test subjects; a state and behaviour expectation API; JUnit test runner integration; Ant support and an IntelliJ IDEA plugin.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging - Testing tools

General Terms

Documentation, Design, Verification.

Keywords

Behaviour Driven Development, Test Driven Development, Testing Tools, Java, Agile Software Development.

1. INTRODUCTION

Originating as an eXtreme Programming (XP) practice, Test Driven Development (TDD) has emerged as an efficient way of creating high quality code. TDD often leads to higher code quality in terms of reduced defects, high cohesion and low coupling [7]. TDD also focuses on the design and usability of production code, by allowing a developer to become a client of their own code, using it from the outside. TDD has many other benefits including providing a ready-built regression suite, helping code maintenance by providing examples of code usage, providing a safety net for refactoring, reducing time spent debugging, providing tight feedback loops (an essential factor of agile methodologies) and allowing a developer to verify that their code is functioning correctly. However notice that only two of TDDs benefits (as defined above) are about testing!

Behaviour Driven Development (BDD) combines the “best practices” that have emerged from using TDD with features from other development practices to attempt to overcome the technical and organisation impediments to TDD adoption and effectiveness.

2. WHAT IS BEHAVIOUR DRIVEN DEVELOPMENT?

“The act of writing a unit test is more an act of design than of verification. It’s also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function.” Robert C. Martin

Behaviour Driven Development (BDD) is a development practice arising out of agile development methodologies, first developed by Dan North in response to the problems experienced using and teaching TDD [10]. At its core, BDD is a refinement to TDD that shifts the emphasis from testing to specification and is in effect the best practices of what developers practising TDD have been doing all along. While this doesn’t seem like a major change, the shift in emphasis from testing to specification brings a number of important flow on effects that positively impact the development of both test and production code. The following sections describe these effects.

Ubiquitous Language

For many, the language used to express a concept has an impact on the way they think about that thing. Goh states that the “abstractions and concepts you use to express yourself in any language ... shape the way you think about the problem you are solving” [5]. This is supported in linguistics by the Sapir-Whorf hypothesis which postulates 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” [18]. For software development, this implies that the language we use to describe the software constructs has an impact on how we create those constructs; by changing the language we affect the code we create. As a concrete example, consider that this line of reasoning is used to justify assigning meaningful names to functions and variables. Also, developers implicitly feel the impact of language when working with a well designed API; an API that provides well named classes and functions within a well designed architecture.

BDD supports using meaningful language in two ways. Firstly, it focuses on “getting the words right”, encouraging good naming of classes, methods and variables. Secondly, it borrows concepts from Domain Driven Development (DDD) to bridge the gap between technical and business artefacts [3]. BDD attempts to capture the behaviour of the domain using a clear and concise syntax that everyone can understand, forming consensus around not only the domain artefacts but also the run-time behaviour of the system [13, 16]. BDD calls the shared language used by all stakeholders the ubiquitous language of the domain.

Design Focus

“A waterfall ‘designer’ starts from an understanding of the problem and builds up some kind of model for a solution, which they then pass on to the implementers. An agile developer does exactly the same, but the language they use for the model happens to be executable source code rather than documents or UML.” Kerry Buckley

TDD advocates acknowledge that one of the most important outcomes of practising TDD is its influence on the design of the resulting code. Test driven code has less defects [7] than non-test driven code and is often more cohesive and less coupled than non-test driven code¹. However the emphasis on “testing” limits TDD’s uptake and effectiveness in driving design (from both a managerial and technical point of view). BDD acknowledges that design is one of the most important outcomes of testing by providing language and tools that support the creation of well-designed code.

Behaviour Focus

“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.” Dave Astels

The difference between testing and specifying may be subtle, but it leads to an increased focus on the behaviour of a piece of code. By focusing on behaviour, BDD frameworks break the traditional 1-to-1 mapping of a unit test class to a production class. The location of the behaviour is no longer important, resulting in test code that is less coupled to its production counterpart and less fragile when production code is refactored. Rather than a 1-to-1 mapping, BDD encourages an M-to-N mapping, allowing as many specification classes as needed to specify the required behaviour. Furthermore, behaviour contexts provide the ability to group related specifications into a single class. Contexts are usually created around different states of an object, such as an empty and non-empty stack. A single context focuses on one thing, thereby reducing the confusion of having multiple states (of a subject) and multiple assertions in a single test.

Figure 1 shows several problems encountered test-driving, and identifies the BDD practices and Instinct features that address the problem.

	Test Driven Development	Behaviour Driven Development	Instinct
Organisational	Too much time/money, optional extra	Design focus	
	Overlapping with traditional QA testers	Design focus	
	Developers require training		
	Developer resistance		

¹ Note that this does not mean that the non-tested code cannot exhibit the same quality characteristics, nor that tested code is always of high quality.

	Test Driven Development	Behaviour Driven Development	Instinct
Technical	Vocabulary difference	Ubiquitous language	
	Test vocabulary affects thinking	Ubiquitous language	
	1-1 mapping of test/production code	Behaviour focus, contexts	
	Tight tests coupled to code	Behaviour focus	
	Repetitive infrastructure setup		Automatic test actor creation
	State vs. interaction affects design	Behaviour focus, ubiquitous language	
	Test intent not clear	Behaviour focus, ubiquitous language	Properties, expectation API, multiple contexts
	Framework inflexibility		Flexible extension
	Hard to know where to start?	Behaviour focus	
Design focus obscured	Design focus		
Documentation	Behaviour focus, ubiquitous language	Expectation API, Properties	

Figure 1. Problems with traditional TDD that BDD and/or Instinct overcome.

BDD Frameworks

Early BDD frameworks focused on acceptance or story-level behaviour. Frameworks such as JBehave [6] allow an application's behaviour to be described in terms of stories and scenarios. Subsequent frameworks such as RSpec [15] (for Ruby) and NSpec [12] (for C#) focused more on the code level of testing, offering an xUnit equivalent feature set for low level specifications. Both approaches are legitimate, and use ubiquitous language to describe the desired behaviour of the system at the appropriate level.

Story-level frameworks provide APIs to allow business stakeholders (e.g. XP's Customer, Scrum's Product Owner, business/functional analyst or traditional QA tester) to understand the behaviour of the system. Typically these frameworks use the "as a/i want/so that" form of story definition and "given/when/then" form of test lifecycle, and also provide an API to plug in the implementation of the acceptance test (the code that supports the wording). Listing 1 shows a story-level specification in Ruby.

```

Story "transfer to cash account",
%(As a savings account holder
  I want to transfer money from my savings account
  So that I can get cash easily from an ATM) do

  Scenario "savings account is in credit" do
    Given "my savings account balance is", 100
    Given "my cash account balance is", 10
    When "I transfer", 20
    Then "my savings account balance should be", 80
    Then "my cash account balance should be", 30
  end

  Scenario "savings account is overdrawn" do
    Given "my savings account balance is", -20
    Given "my cash account balance is", 10
    When "I transfer", 20
    Then "my savings account balance should be", -20
    Then "my cash account balance should be", 10
  end
end
end

```

Listing 1. Ruby BDD story framework (RBehave) [11]

Code-level frameworks provide APIs that allow developers to design and specify the behaviour of low level components of the system. These frameworks provide xUnit equivalent feature sets and support ubiquitous language by encoding it into the APIs used to describe the expected behaviour of a system, as well as the interaction between the specification code and the BDD framework itself. The APIs used for describing expected behaviour typically take the form of Domain Specific Languages (DSLs) or fluent APIs. Consider the following abridged code samples from the Ruby framework RSpec and the C# framework NSpec.

```
describe "non-empty Stack" do
  it "should return the top item when sent #peek" do
    @stack.peek.should == @last_item_added
  end
end

[Specification]
public void NSpecSpecificationExample() {
  Specify.That(someObj.Foo()).ShouldEqual(someOtherObj);
  Specify.That(someObj.GetSomeFloat()).ShouldEqual(5.3)
    .WithAToleranceOf(0.1);
  Specify.That(someObj.SomeMethod()).ShouldBeNull();
  Specify.That(someObj.Bar()).ShouldBeOfType(typeof(Foo));
}
```

Listing 2. RSpec & NSpec behaviour specifications.

BDD frameworks also make use of meaningful naming and readability when defining the way a developer interacts with the framework. They provide flexible, explicit and meaningful ways of defining specification methods and classes. Owing to the flexibility of the language, Ruby frameworks set the standard on readability. Java and C# frameworks simulate the freedom provided by more flexible languages by exploiting generics, chained method calls and method naming techniques.

Listing 3 shows the same specification using RSpec syntax and Instinct syntax respectively, highlighting the readability provided by the framework syntax. In these examples the specification should be read as “a non-empty Stack should return the top item when sent peek”.

```
describe "non-empty Stack" do
  it "should return the top item when sent #peek" do
    @stack.peek.should == @last_item_added
  end
end

class ANonEmptyStack {
  void shouldReturnTheTopItemWhenSentPeek() {
    expect.that(stack.peek()).equalTo(lastItemAdded);
  }
}
```

Listing 3. Behaviour specifications using RSpec and Instinct.

BDD Terminology

As BDD focuses on “getting the words right”, it introduces subtle changes to the traditional TDD vocabulary. BDD frameworks such as Instinct also adopt the terminology introduced by Meszaros² [9]. The major concepts introduced in BDD are defined below.

- **Specification** - A method/function in which the behaviour of a piece of code is specified. Specifications are executable examples that guide the design process and provide both documentation and tests. Specifications are analogous to test methods in xUnit frameworks.
- **Context** - A context in which a certain behaviour is valid. Contexts can be used to group specifications together and also to set up the state in which expectations hold.

² See also <http://xunitpatterns.com/Test%20Double%20Patterns.html>.

- **Collaborator** - An object that interacts with another object, providing a service to it. Collaborators are usually mocked or stubbed out in specifications in order to verify interactions between them and the subject.
- **Actor** - An actor is an object that has a role in a specification. Subjects, dummies, stubs and mocks are all actors.
- **Subject** - The class whose behaviour is under scrutiny.
- **Double** - An implementation of an interface (or extension of a class) that is only used for testing. Doubles can be manually created (plain classes that implement an interface) or auto-created via a framework.
- **Dummy** - Dummy objects are passed around but never actually used, they are usually used just to fill parameters to simplify specifications (where their behaviour may not be important). Dummies will throw exceptions if methods are called on them. They are the simplest form of test double implementation.
- **Stub** - Stubs respond to method calls made during a test by providing canned answers. Stubs may record information about calls, such as an email gateway stub that remembers the messages it 'sent'. Stubs do not fail if methods are called or the order in which they are called (if at all).
- **Mock** - Mocks are more advanced stubs, that not only respond to calls made during a test but are also pre-programmed with expectations which form a specification of the calls they are expected to receive. Mocks will throw an exception if they receive a call they weren't expecting and are checked (called verification) to ensure they received all the calls they expected. Some mocks also verify the order of calls made.
- **Fixture** - A fixture is a known set of data (or commands to setup that data) that provide the environment for a set of tests. Fixtures work well when you have a bunch of tests that work on similar data reducing the complexity of your testing environment (fixtures can also have a downside when overused between tests that should have independent data).

3. INTRODUCING INSTINCT

Instinct is a Java BDD framework, which like all good frameworks, was created using ideas drawn out of several real XP projects. Instinct differs from other TDD and BDD frameworks by focusing on flexibility, explicitness and simplicity. Instinct aims to be flexible and non-intrusive in the way you interact with the framework, allowing marking of actors and specification methods in several different ways. The default specification lifecycle (actor auto-creation, pre-specification, post-specification and specification) can be overridden to provide custom behaviour. Instinct is written to allow flexible extension, and is not tied to an inheritance model.

Instinct encourages explicitness by clearly marking the roles actors play with a specification. This allows the framework to auto-create and insert the actor into the context (known as auto-wiring) and also allows a reader to differentiate the role the collaborator plays in the context. Auto-wiring of actors reduces unnecessary setup infrastructure, clarifying the intent of the specifications.

Instinct provides a fluent style of API (DSL) for expectation setting which unifies state and behaviour expectations into a single API. This API encourages readability, taking advantage of generics to provide type-safe state and interaction checkers. Instinct integrates the jMock 2 [8] mocking library for interaction-based testing.

Contexts in Instinct are created as classes, which can be either standalone (in their own class file) or nested within a parent class, for easy grouping of contexts that operate on different states of the same subject (e.g. an empty and non-empty stack) and navigation between specification and production code³. Instinct formalises test nomenclature (subjects, actors, etc.) by encoding actors as first class citizens of the framework rather than leaving them to be defined on an ad-hoc project-by-project basis.

To simplify specifications, Instinct also takes the motto that “the framework is everybody’s friend”, invoking marked private methods, inserting actors into fields and not failing when on missing return types on lifecycle methods are specified (in contrast to most xUnit frameworks). Instinct does not attempt to provide mechanisms for creating and validating proofs, but instead allows state-based falsification. In the future this feature will be expanded in the tradition of QuickCheck [14], by offering a property API, further simplifying specifications. Instinct also offers integration with Apache Ant [1] for integrating specification running into a build and JUnit 3 & 4 to enable specification running within IDEs.

For comparison between Instinct specifications and traditional xUnit tests, listings 4 and 5 show a (abridged) specification written using Instinct and a unit test written using JUnit 4 for a simple shopping cart ⁴.

³ Grouping all contexts for a subject into a single class reintroduces the 1-to-1 mapping between specification and production code.

⁴ Note. For brevity, code examples have visibility modifiers and required imports removed.

```

class ShoppingCartTest {
    Mockery mockery = new Mockery();
    ShoppingCart cart = new ShoppingCartImpl();
    Item item = mockery.mock(Item.class);

    @Test
    void addItemToEmptyCart() {
        assertTrue(cart.isEmpty());
        cart.addItem(item);
        assertFalse(cart.isEmpty());
        assertEquals(1, cart.size());
        assertTrue(cart.contains(item));
    }

    @Test
    void removeItemFromEmptyCartDoesNothing() {
        assertTrue(cart.isEmpty());
        cart.remove(item);
        assertTrue(cart.isEmpty());
    }
}

```

Listing 4. JUnit shopping cart unit test.

```

class AnEmptyShoppingCart {
    @Subject ShoppingCart cart;
    @Dummy Item item;

    @Specification
    void canHaveAnItemAdded() {
        expect.that(cart.isEmpty()).isTrue();
        cart.addItem(item);
        expect.that(cart.isEmpty()).isFalse();
        expect.that(cart.size()).equalTo(1);
        expect.that(cart.contains(item)).isTrue();
    }

    @Specification
    void doesNotFailWhenAnItemIsRemoved() {
        expect.that(cart.isEmpty()).isTrue();
        cart.remove(item);
        expect.that(cart.isEmpty()).isTrue();
    }
}

```

Listing 5. Instinct shopping cart specification.

While in this case the code is very similar, the Instinct code provides better readability through a fluent expectation API, automatic injection of actors and explicit marking of the roles of collaborators⁵.

4. CONCLUSION

*“This lesson is that programming is not about building software; programming is about designing software.”
 “The only way we validate a software design is by building it and testing it.” Jack W. Reeves*

BDD does not represent a fundamental shift away from TDD. In fact, it’s often said that BDD is simply TDD practised correctly. By practising BDD you get all the benefits of TDD as well as a shift in focus to the things that really matter. BDD shifts the emphasis from testing to specification and provides a ubiquitous language, a strong focus on design and emphasises system behaviour, independent of where the behaviour resides. Even if the “soft” benefits of BDD are not considered, BDD frameworks such as Instinct provide additional features that reduce the overhead and increase the effectiveness of designing and testing software.

⁵ See for example <http://www.jmock.org/oopsla2004.pdf>.

5. ACKNOWLEDGEMENTS

Thanks to the EasyDoc, GCS and KeyManager project teams, out of which the ideas that became Instinct were drawn. Thanks also to Instinct project members and Workingmouse who have provided support for the development of Instinct and of this paper.

6. REFERENCES

- [1] Apache Ant, <http://ant.apache.org/>, 2007.
- [2] Astels, D., A New Look at Test-Driven Development, http://blog.daveastels.com/files/BDD_Intro.pdf, 2005.
- [3] Bellware, S., Behavior Driven Development is..., <http://codebetter.com/blogs/scott.bellware/archive/2007/08/04/166415.aspx>, 2007.
- [4] Buckley, K., Concurrent design and development – a better spin?, <http://www.kerrybuckley.com/2007/09/26/concurrent-design-and-development-a-better-spin/>, 2007.
- [5] Goh, T., TDD vs BDD, <http://www.progprog.com/articles/2007/08/17/tdd-vs-bdd>, 2007.
- [6] JBehave, 2007, <http://jbehave.org/>, 2007.
- [7] Jeffries, R. and Melnik, G., TDD: The Art of Fearless Programming, IEEE Software, http://www.computer.org/portal/cms_docs_software/software/homepage/2007/s3024.pdf, 2007.
- [8] jMock - A Lightweight Mock Object Library for Java, <http://www.jmock.org/>, 2007.
- [9] Meszaros, G., xUnit Test Patterns: Refactoring Test Code, Addison-Wesley, 2007.
- [10] North, D., Introducing BDD, <http://dannorth.net/introducing-bdd/>, 2006.
- [11] North, D., Introducing rbehave, <http://dannorth.net/2007/06/introducing-rbehave>, 2007.
- [12] NSpec, <http://nspec.tigris.org/>, 2007.
- [13] Ocampo, J., Comment on Adverbs - A Partial Obstruction to BDD, <http://codebetter.com/blogs/scott.bellware/archive/2007/09/09/167744.aspx>, 2007.
- [14] QuickCheck: An Automatic Testing Tool for Haskell, <http://www.cs.chalmers.se/~rjmh/QuickCheck/>, 2007.
- [15] RSpec 1.0.8 Examples, <http://rspec.rubyforge.org/examples.html>, 2007.
- [16] Sahayam, S., Behaviour-Driven Development (BDD) - A first look, Workingmouse Internal Blog (not publically available), 2007.
- [17] TestDox, <http://agiledox.sourceforge.net/>, 2007.
- [18] Wikipedia, Sapir-Whorf hypothesis, http://en.wikipedia.org/wiki/Sapir-Whorf_hypothesis, 2007.