The Evol

Behavior Modification:

ution of Behavior-Driven Development

by Dan North

I had a problem. While using and teaching agile practices like test-driven development (TDD) on projects in different environments, I kept coming across the same confusion and misunderstandings. Programmers wanted to know where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails.

The deeper I got into TDD, the more I felt that my own journey had been less of a wax-on, wax-off process of gradual mastery than a series of blind alleys. I remember thinking "If only someone had told me that!" far more often than "Wow, a door has opened." I decided it must be possible to present TDD in a way that gets straight to the good stuff and avoids all the pitfalls.

My response is behavior-driven development (BDD). It has evolved out of established agile practices and is designed to make them more accessible and effective for teams new to agile software delivery.

Test method names should be sentences

My first "Aha!" moment occurred as I was being shown a deceptively simple utility called agiledox, written by my colleague, Chris Stevenson. It takes a JUnit test class and prints out the method names as plain sentences, so a test case that looks like this:

```
public class CustomerLookupTest extends TestCase {
 testFindsCustomerById() {
  ...
 }

 testFailsForDuplicateCustomers() {
  ...
 }
 ...
}
```

renders something like this:

```
CustomerLookup
- finds customer by id
- fails for duplicate customers
- ...
```

The word "test" is stripped from both the class name and the method names, and the camel-case method name is converted into regular text. That's all it does, but its effect is amazing. Developers discovered it could do at least some of their documentation for them, so they started to write test methods that were real sentences. What's more, they found that when they wrote the method name in the language of the *business domain*, the generated documents made sense to business users, analysts, and testers.

A simple sentence template keeps test methods focused

Then I came across the convention of starting test method names with the word "should." This sentence template—The class *should* do something—means you can only define a test for the current class. This keeps you focused. If you find yourself writing a test whose name doesn't fit this template, it suggests the behavior may belong elsewhere.

For instance, I was writing a class that validates input from a screen. Most of the fields are regular client details—forename, surname, etc.—but then there is a field for date of birth and one for age. I started writing a `ClientDetailsValidatorTest` with methods like `testShouldFailForMissingSurname` and `testShouldFailForMissingTitle`.

Then I got into calculating the age and entered a world of fiddly business rules: What if the age and date of birth are both provided but don't agree? What if the birthday is today? How do I calculate age if I only have a date of birth? I was writing increasingly cumbersome test method names to describe this behavior, so I considered handing it off to something else. This led me to introduce a new class I called `AgeCalculator`, with its own `AgeCalculatorTest`. All the age calculation behavior moved into the calculator, so the validator needed only one test around the age calculation to ensure it interacted properly with the calculator.

If a class is doing more than one thing, I usually take it as an indication that I should introduce other classes to do some of the work. I define the new service as an interface describing *what it does*, and I pass this service in through the class's constructor:

```
public class ClientDetailsValidator {

 private final AgeCalculator ageCalc;

 public ClientDetailsValidator(AgeCalculator ageCalc) {
  this.ageCalc = ageCalc;
 }
}
```

This style of wiring objects together, known as "dependency injection," is especially useful in conjunction with *mocks*, or *actors*, which I discuss in detail on StickyMinds.com (www.stickyminds.com/mocks).

An expressive test name is helpful when a test fails

After a while, I found that when changed code caused a test to fail, I could look at the test method name and identify the intended behavior of the code. Typically one of three things had happened:

- I had introduced a bug. Bad me. Solution: Fix the bug.
- The intended behavior was still relevant but had moved elsewhere. Solution: Move the test and maybe change it.
- The behavior was no longer correct—the premise of the system had changed. Solution: Delete the test.

The latter is likely to happen on agile projects as your understanding evolves. Unfortunately, novice TDDers have an innate fear of deleting tests, as though it somehow reduces the quality of their code.

A more subtle aspect of the word "should" becomes apparent when compared with the more formal alternatives of "will" or "shall." "Should" implicitly allows you to challenge the premise of the test: "Should it? Really?" This makes it easier to decide whether a test is failing due to a bug you have introduced or simply because your previous assumptions about the system's behavior are now incorrect.

"Behavior" is a more useful word than "test"

Now I had a tool—agiledox—to remove the word "test" and a template for each test method name. It suddenly occurred to me that people's misunderstandings about TDD almost always came back to the word "test." That's not to say that testing isn't intrinsic to TDD—the resulting set of methods is an effective way of ensuring your code works. However, if the methods do not comprehensively describe the behavior of your system, then they are lulling you into a false sense of security.

I started using the word "behavior" in place of "test" in my dealings with TDD and found that not only did it seem to fit but also that a whole category of coaching questions magically dissolved. I now had answers to some of those TDD questions. What to call your test is easy—it's a sentence describing the next behavior in which you are interested. How much to test becomes moot—you can only describe so much behavior in a single sentence. When a test fails, simply work through the process described above—either you introduced a bug, the behavior moved, or the test is no longer relevant.

I found the shift from thinking in tests to thinking in behavior so profound that I started to refer to TDD as BDD, or *behavior-driven development*.

JBehave emphasizes behavior over testing

At the end of 2003, I decided it was time to put my money—or at least my time—where my mouth was. I started writing a replacement for JUnit called JBehave, which removed any reference to testing and replaced it with a vocabulary built around verifying behavior. I did this to see how such a framework would evolve if I adhered strictly to my new behavior-driven mantras. I also thought it would be a valuable teaching tool for introducing TDD and BDD without the distractions of the test-based vocabulary.

To define the behavior for a hypothetical `CustomerLookup` class, I would write a behavior class called, for example, `CustomerLookupBehavior`. It would contain methods that started with the word "should." The behavior runner would instantiate the behavior class and invoke each of the behavior methods in turn, as JUnit does with its tests. It would report progress as it went and print a summary at the end.

My first milestone was to make JBehave self-verifying. I only added behavior that would enable it to run itself. I was able to migrate all the JUnit tests to JBehave behaviors and get the same immediate feedback as with JUnit.

Determine the next most important behavior

I then discovered the concept of business value. Of course, I had always been aware that I wrote software for a reason, but I had never really thought about the value of the code I was writing *right now*. Another colleague, business analyst Chris Matts, set me thinking about business value in the context of behavior-driven development.

Given that I had the target in mind of making JBehave self-hosting, I found that a really useful way to stay focused was to ask, "What's the next most important thing the system *doesn't* do?"

This question requires you to identify the value of the features you haven't yet implemented and to prioritize them. It also helps you formulate the behavior method name: The system doesn't do X (where X is some meaningful behavior), and X is important, which means it *should* do X; so your next behavior method is simply:

```
public void shouldDoX() {
}
```

Now I had an answer to another TDD question, namely *where to start*.

Requirements are behavior, too

At this point, I had a framework that helped me understand—and more importantly, explain—how TDD works and an approach that avoided all the pitfalls I had encountered.

Toward the end of 2004, while I was describing my new found, behavior-based vocabulary to Matts, he said, "But that's just like analysis." There was a long pause while we processed this, and then we decided to apply all of this behavior-driven thinking to defining requirements. If we could develop a consistent vocabulary for analysts, testers, developers, and the business, then we would be well on the way to eliminating some of the ambiguity and miscommunication that occur when technical people talk to business people.

BDD provides a "ubiquitous language" for analysis

Around this time, Eric Evans published his bestselling book *Domain-Driven Design*. In it, he describes the concept of modeling a system using a *ubiquitous language* based on the business domain, so that the business vocabulary permeates right into the codebase.

Chris and I realized we were trying to define a ubiquitous language for the analysis process itself! We had a good starting point. In common use within the company there was already a story template that looked like this:

As a [X], I want [Y], so that [Z]

where Y is some feature, Z is the benefit or value of the feature, and X is the person (or role) who will benefit. Its strength is that it forces you to identify the value of delivering a story at the point at which you define it. When there is no real business value for a story, it often comes down to something like " . . . I want [some feature] so that [I just do, ok?]." This can make it easier to descope some of the more esoteric requirements.

Matts and I set about discovering what every agile tester already knows: A story's behavior is simply its acceptance criteria—if the system fulfills all the acceptance criteria, it's behaving correctly; if it doesn't, it isn't. So we set about creating a template to capture a story and its acceptance criteria.

The template had to be loose enough that it wouldn't feel artificial or constraining to analysts but structured enough that we could break the story into its constituent fragments and automate them. We started describing the acceptance criteria in terms of *scenarios*, which took the following form: **given** some *initial state* (the *givens*), **when** an *event* occurs, **then** ensure some *outcomes*.

The context is expressed as one or more *givens*, and the result is expressed as one or more *outcomes*.

To illustrate, let's use the classic example of an ATM machine. One of the story cards might look like this:

Customer withdraws cash

As a customer, **I want** to withdraw cash from an ATM, **so that** I don't have to wait in line at the bank.

So how do we know when we have delivered this story? Here are some possible scenarios to consider:

- The account is in credit.
- The account is overdrawn past the overdraft limit.
- The account is overdrawn within the overdraft limit.

Of course, there will be other scenarios, such as the account is in credit but this withdrawal makes it overdrawn, or the dispenser has insufficient cash.

Using the given-when-then template, the first two scenarios might look like this:

Scenario 1: Account is in credit
Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned

Notice the use of "and" to connect multiple givens or multiple outcomes in a natural way.

Scenario 2: Account is overdrawn past the overdraft limit
Given the account is overdrawn
And the card is valid
When the customer requests cash
Then ensure a rejection message is displayed
And ensure cash is not dispensed
And ensure the card is returned

Both scenarios are based on the same event and even have some givens and outcomes in common. We want to capitalize on this by reusing givens, events, and outcomes.

Acceptance criteria should be executable

The fragments of the scenario—the givens, event, and outcomes—are fine-grained enough to be represented directly in code. JBehave defines an object model that enables us to directly map the scenario fragments to Java classes.

You write a class representing each given:

```
public class AccountIsInCredit implements Given {
 public void setup(World world) {
   ...
 }
}

public class CardIsValid implements Given {
 public void setup(World world) {
   ...
 }
}

and one for the event:
public class CustomerRequestsCash implements Event {
 public void occurIn(World world) {
   ...
 }
}
```

and so on for the outcomes. JBehave then wires these all together and executes them. It creates a "world," which is just somewhere to store your objects, and passes it to each of the givens in turn so they can populate the world with known state. JBehave then tells the event to "occur in" the world, which carries out the actual behavior of the scenario. Finally it passes control to any outcomes we have defined for the story. Having a class to represent each fragment enables us to reuse fragments in other scenarios or stories.

At first, the fragments are implemented using mocks to set an account to be in credit or a card to be valid. These form the starting points for implementing behavior. As you implement the application, the givens and outcomes are changed to use the actual classes you have implemented, so that by the time the scenario is completed, they have become proper end-to-end functional tests.

The present and future of BDD

After a brief hiatus, JBehave is back under active development. The core is fairly complete and robust. The next step will be integration with popular Java IDEs like IntelliJ IDEA and Eclipse. It now can run from a console within the IDE, but this isn't ideal.

Dave Astels has been actively promoting BDD and is in the process of writing a book about it. His Weblog and various published articles have provoked a flurry of activity, most notably the *rspec* project to produce a BDD framework in the Ruby language. I have started work on *rbehave*, which will be a port of JBehave to Ruby.

A number of my co-workers have been using BDD techniques on a variety of real-world projects and have found the techniques very successful. The JBehave story runner—the part that verifies acceptance criteria—is under active development. The vision is to have a round-trip editor so that BAs and testers can capture stories in a regular text editor that can generate stubs for the behavior classes, all in the language of the business domain. BDD evolved with the help of many people. Please see the StickyNotes for acknowledgements and a link to the BDD wiki. **{end}**

Dan North is a senior consultant with ThoughtWorks, where he coaches development teams in agile software delivery and project automation. A programmer with fifteen years of delivery experience, he has published a number of articles and spoken at conferences on topics ranging from agile enablement to NLP. You can contact Dan at dan.north@thoughtworks.com.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- BDD phrasebook
- Acknowledgements
- The BDD wiki
- References

Step into a new testing generation

GUIdancer is the unique new tool from Bredex GmbH for automated software testing, offering the ability to create GUI tests without programming.

Creating reusable, easily maintainable tests using straightforward high level specifications – welcome to the world of **GUI**dancer.

To find out more – www.guidancer.com

GUIdancer®

Additional features:
- *XML and HTML reports*
- *Intuitive user interface*
- *Runs as standalone application or Eclipse plugin*
- *User-defined hierarchical organization of test elements*
- *Platform independent*
- *Event Handling*
- *Context-sensitive help and sample projects*
- *Comprehensive documentation*
- *Observation Mode*
- *Multi-User Capacity*

Requirements:
- *Java 1.5 or later to run* **GUI**dancer
- *Java 1.3 or later for Applications under Test*
- *Swing corresponding to VM*
- *Microsoft Windows/Linux/Solaris*
- *Oracle 9i or later (optional)*

BREDEX
Software-Entwicklung und Beratung
www.bredexsw.com