Introducing JBehave

Improving Your Agile Process with Behavior Driven Development.

https://github.com/defano/jbehave-quickstart

Matt DeFano
Sr. Managing Architect
STA Group, LLC

matt.defano@cjug.org
matt.defano@stagrp.com

STA GROUP

What is it?

Behavior driven development: The work of author Dan North that represents an improvement to test driven development. Aims to define tests in terms of desired behavior using a “ubiquitous language” (i.e., plain English).

Fear not, even if you’re not practicing true TDD, there’s still plenty of benefit in his concepts.

JBehave: a stupid-simple tool that lets you bind plain-English descriptions of a software component’s behavior to Java. That’s it.

How did this come to be?

Dan North, the inventor of BDD, had a problem with Test Driven Development—several, really:

1. Classes are nouns, methods are verbs… but what should I name my tests?
   - ✓ North concluded that descriptive test method names should be sentences, like: `testFindsCityByZipCode()` or `testThrowsExceptionWhenDivisorIsZero()`.

2. How do I figure out why I broke [somebody else's] test?
   - ✓ If you follow the method-as-a-sentence convention, it should be very clear what *behavior* is expected by the test, and therefore what *behavior* was violated.

3. What should I put in a test? Do I want lots of little tests, or a few big tests?
   - ✓ Organize around behavior, not implementation. Each test should demonstrate correctness of a defined behavior, not necessarily a method or data member.
   - ✓ Fine grained tests are easier to maintain, and easier to understand. If you keep your "test sentences" you'll keep your tests fine-grained. Who wants a thousand character method name?

STA GROUP

So why are you so interested in it?

Matt DeFano, the inventor of nothing[1], had these problems with TDD:

1. The customer/client/boss wants a description of each test. I have 5,973 unit/integration tests. How am I supposed to provide that?
   - ✓ Behavior-driven tests are written in a "ubiquitous language" (called plain English) that's meaningful to both business people and engineers and are therefore self documenting.

2. Okay, fine. But they also want traceability from a requirement to a test. Which requirement does my [low-level-technical] test trace back to?
   - ✓ Test on the basis of desired system behavior, not on the basis of implementation. This does not preclude unit/class-level tests: You may describe unit tests in terms of that class's *behavior*.

3. We have to "pull teeth" to get detailed requirements from customer/client/product owner, and even then we make false assumptions or miss "inferred requirements".
   - ✓ Make your product owner specify requirements in terms of behavior-driven test scenarios. Use these scenarios as your acceptance criteria! This should leave no cracks for "inferred" functionality to fall through!

1. This is a lie. I'm pretty sure nothing existed before I "invented" it.

STA GROUP

Let's consider a simple example…

"As a smartphone user, I want a four-function calculator on my device so that I can replace my hand held calculator with my phone."

Most engineers should have little difficulty implementing this story correctly.

WRONG!

1	+	2	X	3	=	7 Expression Evaluation Mode	*or*	9 Immediate Execution Mode
5	+	1	+	+	+	=	6 Standard Input	*or*	8 Ten Key Input
4	*	-	5	=		-20 Contextual Operator Interpretation	*or*	-1 Fixed Operator Interpretation

Bottom line: Expressing intent is difficult, even for seemingly trivial applications!

For more examples, see http://en.wikipedia.org/wiki/Calculator_input_methods

STA GROUP

What is a behavioral test?

Every test has three parts:

1. **Initialization**: setup the unit under test and initialize the test context. Maps to the “given” step in JBehave; analogous to JUnit’s `@Before`.
2. **Stimulus**: provide input to the unit under test (i.e., perform the test). Maps to the “when” step; analogous to JUnit’s `@Test`.
3. **Verification**: assert that the unit under test responded correctly to the stimulus. Maps to the “then” step; analogous to JUnit’s assertions (i.e., `assertEquals()`).

```
Given a car
And the engine is running
And it’s in drive
When I press the gas pedal
Then the engine should rev
And the car should move forward
```

Test initialization

Test stimulus

Test verification

STA GROUP

How do I write a JBehave test?

1. Write a set of test scenarios in a `.story` text file:

```
Given a bank account with 100.00 in it
When I withdraw 20.00
Then the balance should be 80.00
```

2. Bind the steps of your scenario to Java methods in a POJO:

```
@Given("a bank account with $amount in it")
public void init (double amount) {
   Account acct = new Account(amount);
```

3. Configure JBehave (reporting style, JUnit integration, etc.)
4. Execute the test (use JUnit if you like—or run standalone)
5. Review the reports (console output, HTML—lots of options)

STA GROUP

You need three things to make Jbehave work

A stories file:

- A plain text file containing one or more scenarios (given/when/then stanzas). File should have same package prefix as its corresponding steps class. File name should mimic the steps class; replacing camel-case with underscores (i.e., `MyTest.java` → `my_test.story`). (Binding rules can be overridden.)

A steps POJO:

- A Java POJO containing a set of public, void methods annotated with `@Given`, `@When`, `@Then` (or `@Alias`) matching each statement in the story file.

A JBehave configuration:

- Configures reporting options (HTML, text, console)
- Defines binding of steps class to stories files.
- Can be specified directly in steps POJO, provided within a base class, or just about any other technique you might imagine.

STA GROUP

Static Input Example

calculator.story

```
Given a calculator with a cleared display
When I press 22
followed by %
Then I expect the calculator to display .22
```

Careful! Matchers are case sensitive. "`A Calculator…`" won't work!

CalculatorTest.java

```
@Given("a calculator with a cleared display")
public void initialize () {
   calculator = new Calculator();
}

@When("I press $input")
@Alias("followed by $input")
public void processInput(String inputToProcess) {
   calculator.enterInput(inputToProcess);
}

@Then("I expect the calculator to display $expectedValue")
public validate (String expectedValue) {
   assertEquals(calculator.getDisplayedValue(), expectedValue);
}
```

`@Alias` allows us to map different (but semantically equal) steps to the same Java method.

Variables are matched in the lexical order in which they appear in the step —Java parameter names don't matter.

Be careful not to duplicate"`then`" inside of the annotation!

STA GROUP

Variable Input Example

calculator.story

```
Given a calculator in its initial state
When I enter <sequence>
Then I expect <result> to be displayed

Examples:
|sequence|result|
|2 + 3 + 4 =|9|
|100 * 25 % =|25|
|1.25 - .10 =|1.15|
|10 / 5 =|2|
|1 + 2 * 3 =|9|
|99 =|99|
```

Write your scenario once!

Identify named variables with angled brackets.

Create an "`Examples:`" table (this is a keyword).

Pipe delimited columns; first row correlates with <variable> names.

Subsequent rows indicate test data.

CalculatorTest.java

```
@Then("I expect <result> to be displayed")
public validate (@Named("result") double expectedResult) {
    assertEquals(expectedResult, ...);
}
```

Example of named parameter; order and name of Java method parameters don't matter.

Invoked once for each row in the examples table (minus the header, of course).

STA GROUP

Configuring JBehave… (the ugly part)

Lots of different ways to execute a JBehave test (this flexibility is arguably a drawback—configuration is typically the most-heard complaint).

For example:

1. *My preferred approach:* Write a JBehave superclass that each steps class extends; use the superclass to provide the boilerplate described above. **Pros**: Steps classes look like JUnits; each steps class can be executed independently.
2. Wrap the tests in JUnit using the JBehave Embedder class and execute tests using JUnit. **Pros:** Works automatically with any toolset that supports JUnit. **Cons:** Requires cut-and-paste boilerplate for each test story.
3. Execute JBehave directly from your build system (using JBehave plugins for Maven and Ant). **Pros**: No wrapper code to write. **Cons**: Non-trivial cases hard to configure; awkward test execution within IDE.
4. Plus, many, many more!

STA GROUP

JBehave Base Class Example

```
@Ignore
public abstract class JBehaveJUnitTest extends JUnit

    @Override
    public Configuration configuration() {

        return new MostUsefulConfiguration()
            .useStoryLoader(new LoadFromClasspath(this.getCl
            .useStoryReporterBuilder(new StoryReporterBuilder()
            .withDefaultFormats()
            .withFormats(Format.CONSOLE, Format.TXT)
        );
    }

    @Override
    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(configuration(), this);
    }
}
```

`@Ingore` this class, otherwise JUnit will treat it as a test suite.

Makes this class (and its subclasses) look like JUnit suites.

Configure JBehave to your desire.

Binds your steps class (our subclass) to a story file based on naming convention.

STA GROUP

Using the JBehave base class

operator.story

```
Given the DIVIDE operator
Then isBinary() should return true

Given the NEGATE operator
Then isBinary() should return false
```

OperatorTest.java

```
public class OperatorTest extends JBehaveJUnitTest {

    private Operator operatorUnderTest;

    @Given("the $operatorName operator")
    public void given (String operatorName) {
        operatorUnderTest = Operator.valueOf(Operator.class, operatorName);
    }

    @Then("isBinary() should return $boolean")
    public void then (String bool) {
        assertEquals((Boolean) operatorUnderTest.isBinary(), Boolean.valueOf(bool));
    }
}
```

STA GROUP

Advanced Features

- Keyword synonyms, aliases, composite steps and pattern variants
  - ✓ Use other keywords in place of given/when/then; allow synonymous steps to bind to the same Java method or let multiple steps bind to a single method; define your syntax.
- Composite Steps
  - ✓ Subroutine steps—treat a “complicated” step as an alias/shortcut to several other steps.
- External Test Data Sources
  - ✓ Scenario data tables can be loaded as a classpath resource or URL.
- Tabular parameters with an “examples” table
  - ✓ Write a single set of steps, then a provide a table of inputs to test—no need to duplicate the English over and over again with slightly different values.

STA GROUP

Extra Credit

Don’t like the way Eclipse “rolls up” test results under a single test case entry in the UI? Use this third-party test runner.

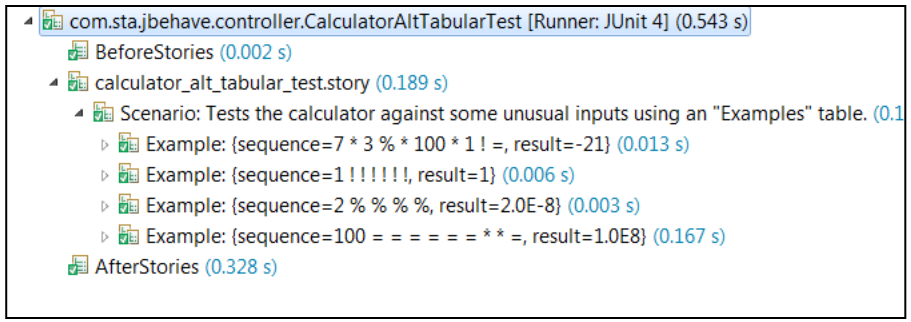
1. Include the test runner as a dependency in your POM:

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>jbehave-junit-runner</artifactId>
    <version>1.0.1</version>
</dependency>
```

2. Invoke the runner in your base class:

```
@Ignore
public class JBehaveJUnitTest extends JUnitStory {
```

```
@RunWith(JUnitReportingRunner.class)
public class JBehaveJUnitTest extends JUnitStory {
```



STA GROUP

How popular is it?

Still relatively small… but growing rapidly:

In absolute terms…

and relative terms…