THE INTRODUCTION TO THE SPOCK FRAMEWORK

Vladimír Oraný
AppSatori s.r.o, vladimir.orany@appsatori.eu

ABSTRACT:

This article introduces the essentials of testing applications using the Spock framework. It uses comparison to the JUnit testing framework so the readers can much simpler understand its usage. It also shows the benefits of using the Groovy language instead of the Java language for writing tests.

KEYWORDS:

Spock, Groovy, test driven development, Java, JUnit, specification

INTRODUCTION

The Spock framework [1] is specification framework with mock capabilities build on the top of Groovy language [6]. It was created by Peter Niederwieser as a perfect example of using the language to create new powerful domain specific language (DSL) which uses operator overloading and abstract syntax tree manipulation on the daily basis. The goal of the framework is allow developer to write more readable and writable specification. The Spock framework is inspired by many well known tools such as JUnit [12], jMock [14] or RSpec [13].

WHY TO USE SPOCK FRAMEWORK?

Before we start to learn about the Spock framework one question sure arises in your heads. Why should you use yet another testing framework? Next two lists summarize the main befits. More ideas could be found at Why Spock wiki page [2].

The Benefits of Using Groovy Instead of Java

The Spock framework is written using the Groovy language so developers can benefit from all the features it provides.

Dynamic language

Dynamic language helps to implement true test driven development. You can call methods and use properties which do not exist yet without having to deal with the compiler errors. This can help you to implement features step by step.

No access check for private methods and fields

Groovy does not check privacy access to the methods and fields. According to the Groovy bug #3010 [7], there is no run time penalty if you can call private methods and access private fields.

This could be particularly useful for white box testing. The calculator example uses this feature to access the registry field.

Less boilerplate code using the Groovy syntax and language enhancements

Groovy adds a lot of syntax sugar to the Java language syntax. Closures, ranges and list and map literals are just a few of them. Groovy also adds many useful methods to standard Java classes e.g. `every` and `any` methods for collections.

Power asserts

The power assertions were first introduced in the Spock framework but now they are part of the Groovy language itself. Instead of pointless assertion error shows Groovy values for each object involved in the assertion as shown in following example.

Example 1 - Power assert

```
Condition not satisfied:

calc.display == "E"
|    |       |
|    =       false
|            1 difference (0% similarity)
|            (=)
|            (E)
eu.appsatori.ts2011.Calc@46dd75a4

	at eu.appsatori.ts2011.CalcSpec.Zero division shows 'E' on 
display(CalcSpec.groovy:151)
```

The Benefits of Using the Spock framework Instead of JUnit

Although many helpful features of the Groovy language help testing software the Spock framework adds much more.

Implicit assertion in expect and then clauses

There is no need to use the assert keyword in expect and then clauses. Each line is evaluated against the Groovy truth.

Easy parameterization and data driven tests

It is extremely easy to parameterize any feature method using the where block. The parameters generator can be anything which cans Groovy iterates e.g. the SQL query result.

Easy extensible

The Spock framework is easy to extend. There are already extensions for the Spring or the Grails framework. The Geb functional testing tool also flawlessly integrates with it.

Easy mocking

Mocking is essential part of the Spock framework. It has a powerful DSL to specify object interactions.

Compatible with JUnit

Every Spock specification extends the spock.lang.Specification class which is just plain JUnit 4 test case which uses special runner called Sputnik. For example, if your IDE supports Groovy and JUnit you can start using Spock anytime you want.

Examples

Example code showing full Spock and JUnit comparison can be found at http://link.appsatori.eu/ts2011. There you can find full specification used in examples here and its JUnit counterparts.

THE SPOCK FRAMEWORK ESSENTIALS

The first thing to do if you want to write the Spock specification is to extend spock.lang.Specification class. According to Spock Basics [3] page the typical layout of the specification class is following:

- fields
- fixture methods
- feature methods
- helper methods

Because the most interesting things happen in the feature methods let discover the other ones first.

Fields

Fields used by more feature methods should be declared as instance fields of the specification class. Those fields are not shared between features method. The initialization is called before each feature method instead. If there is a need to share field between feature methods it must be annotated by spock.lang.Shared annotation. This is particularly useful for expensive resources such as SQL connections.

Example 2 - Fields definition

```
@Shared sql = Sql.newInstance("jdbc:h2:mem:", "org.h2.Driver")
def calc = new Calc()
```

Fixture methods

Fixture methods are equivalents of @Before, @BeforeClass, @After and @AfterClass annotated methods and their purpose are to set up and clean up fixtures for feature methods. Following feature methods are available:

- setup() - called before each feature method

- cleanup() - called after each feature method
- setupSpec() - called before the first feature method
- cleanupSpec() - called after the last feature method

Example 3 - Fixture methods examples

```
def setup(){
	println "This is called before each feature method."
}

def cleanup(){
	println "This is called after each feature method."
}

def setupSpec(){
	sql.execute '''
		create table basics (
			id int primary key,
			first int,
			operator char(1),
			second int,
			result int);
	'''
	sql.execute '''
		insert into basics
			(id, first, operator, second, result)
		values
			(1, 4, '+', 2, 6),
			(2, 4, '-', 2, 2),
			(3, 4, '/', 2, 2),
			(4, 4, '*', 2, 8)
	'''
}

def cleanupSpec(){
	sql.execute 'drop table basics'
}
```

Helper methods

Helper methods are just regular methods which help to modularize feature method by extracting repetitive code into one place.

Example 4 - Helper method example

```
def pushButtons(numbers){
	numbers.each {
		calc.push it as String
	}
}
```

Feature methods

The most important part of specification lies in the feature methods. They are just instance methods of the class which are usually named by meaningful string name such as “The stack ‘pop’ method must throw exception if empty”. What makes method a feature method is presence of one of the blocks defined by Spock: setup (given), when, then, expect, cleanup, where. Each block begins with label of the same name optionally followed by describing string. What are the blocks good for you can read in following parts.

Setup Blocks

Setup is optional block where the fixtures for the feature methods are created. It has an alias “given” to allow writing specification in usual form given-when-then. It is the only one block which label can be omitted. All statements between the beginning of the method and the first block is considered as setup block. Setup block is called before each assumption. Setup block must always be the first block of the method.

When and Then Blocks

When and then blocks comes always together. When block gives the stimulus and the then block asserts the response is adequate. When blocks can contain any code but then blocks must contain only conditions, exception conditions and interactions. Conditions are any statements which can be evaluated to the Groovy truth. Exception conditions are defined by calling one of thrown or a notThrown method which assumes that exception of given type was or wasn’t thrown.

Example 5 - Feature method with exception condition and condition

```
def "Zero division shows 'E' on display"(){
	when:
	calc.push '5'
	calc.push '/'
	calc.push '0'
	calc.push '='
	then:
	notThrown ArithmeticException
	calc.display == "E"
}
```

Exception conditions are defined by calling one of thrown or a notThrown method which assumes that exception of given type was or wasn’t thrown.

Example 6 - Feature method with interactions

```
def "Calc uses given operators at the right time"(){
	setup:
	Operator plus = Mock()
	Operators operators = Mock()
	Operators old = calc.operators
	calc.operators = operators
```

```
        when:
        pushButtons([1,2])
        then:
        0 * operators.get(_)

        when:
        calc.push '+'
        then:
        1 * operators.get('+') >> plus
        0 * plus.operate(_,_)

        when:
        pushButtons([1,2])
        then:
        0 * operators.get(_)
        0 * plus.operate(_,_)

        when:
        calc.push '='
        then:
        1 * plus.operate(12, 12) >> 24

        cleanup:
        calc.operators = old
}
```

Interactions verify that some method was called on the mock object and how many times it happened. It uses powerful syntax which can easy specify how many times (number or range) should be some method (identified by name or regular expression) on particular (or even any) mock object called and what the result (or results) of the call should be. For more information, visit Interactions page of Spock framework wiki [4].
Feature method can contain as many when-then block pairs as needed to simulate scenario approach.

Expect

Expect block is just like then blocks. It is useful when there is no need for when block. It can contain any parts which the when block can contain.

Example 7 - Feature method with expect block

```
def "Zero must be shown on new calculator’s display"(){
        expect:
        calc
        calc.display == 0 as String
}
```

Cleanup

The cleanup block is counterpart to the setup block. Its purpose is to undo all undesirable changes which were introduced during the feature method. Only the where block or the end of method can follow the cleanup block.

Where

The where block serves for feature method parameterization. You can use any variable and use it as feature method parameter. Parameters are defined using the table like block with variables in headers and its values in columns or you can use the left shift operator in single or multiple assignment. The right hand side must be anything the Groovy language can iterate e.g. SQL results.

Example 8 - Feature method using table like where block

```
def "Pushing numbers multiple times must append number to the display"(){
	when:
	pushButtons numbers
	then:
	calc.display == result as String
	where:
	numbers | result
	[1,2,3] | 123
	[2,2]   | 22
	[0,1,2] | 12
}
```

Example 9 - Feature method using SQL as parameters generator

```
def "Basic operators must work as expected"(){
	when:
	calc.push first as String
	calc.push operator as String
	calc.push second as String
	calc.push '='
	then:
	calc.display == result as String
	where:
	[first, operator, second, result] << sql.rows(
		'select first, operator, second, result from basics'
	)
}
```

SUMMARY

This article introduced the Spock specification framework. When compared to JUnit the Spock specifications are much more like test suites than individual tests. Each feature method could contain its setup and cleanup block as well as they could be parameterized easily by where blocks. This could be in JUnit achieved only at test class level (see source codes for further more details). In general, Spock specifications are more flexible and readable. This also applies even when things go wrong thanks to detailed power asserts which dumps shows all involved objects.

LITERATURE

1. NIEDERWIESER, Peter. *Spock : the enterprise ready specification framework* [online]. 2011 [cit. 2011-04-08]. Available on the World Wide Web: <http://code.google.com/p/spock/>.
2. NIEDERWIESER, Peter. *Why Spock* [online]. 2011 [cit. 2011-04-08]. Available on the World Wide Web: <http://code.google.com/p/spock/wiki/WhySpock>.
3. NIEDERWIESER, Peter. *Spock Basics*[online]. 2011 [cit. 2011-04-08]. Available on the World Wide Web: <http://code.google.com/p/spock/wiki/SpockBasis>.
4. NIEDERWIESER, Peter. *Interactions* [online]. 2011 [cit. 2011-04-08]. Available on the World Wide Web: <http://code.google.com/p/spock/wiki/Interactions>.
5. NIEDERWIESER, Peter. *Spock Web Console* [online]. 2011 [cit. 2011-04-08]. Available on the World Wide Web: <http://meetspock.appspot.com/>.
6. *Groovy : Home* [online]. 2011 [cit. 2011-04-08]. Available on the World Wide Web: <http://groovy.codehaus.org/>.
7. *[#GROOVY 3010] : fix private field visibility* [online]. 2009 [cit. 2011-04-08]. Codehaus Jira. Available on the World Wide Web: <http://jira.codehaus.org/browse/GROOVY-3010>.
8. *Groovy JDK : Overview* [online]. 2011 [cit. 2011-04-08]. Available on the World Wide Web: <http://groovy.codehaus.org/groovy-jdk/>.
9. *Groovy Truth* [online]. 2010 [cit. 2011-04-08]. Available on the World Wide Web: <http://docs.codehaus.org/display/GROOVY/Groovy+Truth>.
10. LAFORGE, Guillaume. *Groovy 1.7 release notes* [online]. 2010 [cit. 2011-04-08]. Available on the World Wide Web: <http://docs.codehaus.org/display/GROOVY/Groovy+1.7+release+notes>.
11. *JUnit 4 Tutorial 6 : Parameterized Tests* [online]. 2009 [cit. 2011-04-08]. Available on the World Wide Web: <http://www.mkyong.com/unittest/junit-4-tutorial-6-parameterized-test/>.
12. *Welcome to JUnit.org* [online]. 2009 [cit. 2011-04-08]. Available on the World Wide Web: <http://junit.org>.
13. *RSpec.info: Home* [online]. 2009 [cit. 2011-04-08]. Available on the World Wide Web: <http://rspec.info>.
14. *JMock: An Expressive Mock Object Library for Java* [online]. 2009 [cit. 2011-04-08]. Available on the World Wide Web: <http://jmock.org>.
15. *Duckapter: Duck Typing Support for Java* [online]. 2010 [cit. 2011-04-08]. Available on the World Wide Web: <http://duckapter.googlecode.com>