Spock Framework Documentation

Release 1.0-SNAPSHOT

Peter Niederwieser

April 10, 2013

CONTENTS

Authors: Peter Niederwieser

Version: 1.0-SNAPSHOT

Note: This documentation effort is a work in progress. For the time being, we also recommend to check out the old documentation at http://wiki.spockframework.org.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Introduction

Spock is a testing and specification framework for Java and Groovy applications. What makes it stand out from the crowd is its beautiful and highly expressive specification language. Thanks to its JUnit runner, Spock is compatible with most IDEs, build tools, and continuous integration servers. Spock is inspired from JUnit, jMock, RSpec, Groovy, Scala, Vulcans, and other fascinating life forms.

1.2 Getting Started

It's really easy to get started with Spock. This section shows you how.

1.2.1 Spock Web Console

Spock Web Console is a website that allows you to instantly view, edit, run, and even publish Spock specifications. It is the perfect place to toy around with Spock without making any commitments. So why not run Hello, Spock! right now?

1.2.2 Spock Example Project

To try Spock on your own computer, download and unzip the Example Project (download link). It comes with fully working Ant, Gradle, and Maven builds that require no further setup. The Gradle build even bootstraps Gradle itself and gets you up and running in Eclipse or IDEA with a single command. See the README for detailed instructions.

1.2.3 Next Steps

The following sections provide further information on how to use Spock in a number of different environments:

- Ant
- Gradle
- Maven
- Groovy Console
- Eclipse
- IDEA

1.3 Data Driven Testing

Oftentimes, it is useful to exercise the same test code multiple times, with varying inputs and expected results. Spock's data driven testing support makes this a first class feature.

1.3.1 Introduction

Suppose we want to specify the behavior of the `Math.max` method:

```
class MathSpec extends Specification {
    def "maximum of two numbers"() {
        expect:
        // exercise math method for a few different inputs
        Math.max(1, 3) == 3
        Math.max(7, 4) == 7
        Math.max(0, 0) == 0
    }
}
```

Although this approach is fine in simple cases like this one, it has some potential drawbacks:

- Code and data are mixed and cannot easily be changed independently
- Data cannot easily be auto-generated or fetched from external sources
- In order to exercise the same code multiple times, it either has to be duplicated or extracted into a separate method
- In case of a failure, it may not be immediately clear which inputs caused the failure
- Exercising the same code multiple times does not benefit from the same isolation as executing separate methods does

Spock's data-driven testing support tries to address these concerns. To get started, let's refactor above code into a data-driven feature method. First, we introduce three method parameters (called *data variables*) that replace the hard-coded integer values:

```
class MathSpec extends Specification {
    def "maximum of two numbers"(int a, int b, int c) {
        expect:
        Math.max(a, b) == c

        ...
    }
}
```

We have finished the test logic, but still need to supply the data values to be used. This is done in a `where:` block, which always comes at the end of the method. In the simplest (and most common) case, the `where:` block holds a *data table*.

1.3.2 Data Tables

Data tables are a convenient way to exercise a feature method with a fixed set of data values:

```
class Math extends Specification {
    def "maximum of two numbers"(int a, int b, int c) {
        expect:
        Math.max(a, b) == c
```

```
        where:
        a | b | c
        1 | 3 | 3
        7 | 4 | 4
        0 | 0 | 0
    }
}
```

The first line of the table, called the *table header*, declares the data variables. The subsequent lines, called *table rows*, hold the corresponding values. For each row, the feature method will get executed once; we call this an *iteration* of the method. If an iteration fails, the remaining iterations will nevertheless be executed. All failures will be reported.

Data tables must have at least two columns. A single-column table can be written as:

```
where:
a | _
1 | _
7 | _
0 | _
```

1.3.3 Isolated Execution of Iterations

Iterations are isolated from each other in the same way as separate feature methods. Each iteration gets its own instance of the specification class, and the `setup` and `cleanup` methods will be called before and after each iteration, respectively.

1.3.4 Sharing of Objects between Iterations

In order to share an object between iterations, it has to be kept in a `@Shared` or static field.

Note: Only `@Shared` and static variables can be accessed from within a `where:` block.

Note that such objects will also be shared with other methods. There is currently no good way to share an object just between iterations of the same method. If you consider this a problem, consider putting each method into a separate spec, all of which can be kept in the same file. This achieves better isolation at the cost of some boilerplate code.

1.3.5 Syntactic Variations

The previous code can be tweaked in a few ways. First, since the `where:` block already declares all data variables, the method parameters can be omitted. [1]. Second, inputs and expected outputs can be separated with a double pipe symbol (`||`) to visually set them apart. With this, the code becomes:

```
class DataDriven extends Specification {
    def "maximum of two numbers"() {
        expect:
        Math.max(a, b) == c

        where:
        a | b || c
        3 | 5 || 5
```

[1] The idea behind allowing method parameters is to enable better IDE support. However, recent versions of IntelliJ IDEA recognize data variables automatically, and even infer their types from the values contained in the data table.

```
        7 | 0 || 7
        0 | 0 || 0
    }
}
```

1.3.6 Reporting of Failures

Let's assume that our implementation of the `max` method has a flaw, and one of the iterations fails:

```
maximum of two numbers   FAILED

Condition not satisfied:

Math.max(a, b) == c
    |    |  |  |  |
    |    7  0  |  7
    42          false
```

The obvious question is: Which iteration failed, and what are its data values? In our example, it isn't hard to figure out that it's the second iteration that failed. At other times this can be more difficult or even impossible [2]. In any case, it would be nice if Spock made it loud and clear which iteration failed, rather than just reporting the failure. This is the purpose of the `@Unroll` annotation.

1.3.7 Method Unrolling

A method annotated with `@Unroll` will have its iterations reported independently:

```
@Unroll
def "maximum of two numbers"() { ... }
```

Why isn't `@Unroll` the default?

One reason why `@Unroll` isn't the default is that some execution environments (in particular IDEs) expect to be told the number of test methods in advance, and have certain problems if the actual number varies. Another reason is that `@Unroll` can drastically change the number of reported tests, which may not always be desirable.

Note that unrolling has no effect on how the method gets executed; it is only an alternation in reporting. Depending on the execution environment, the output will look something like:

```
maximum of two numbers[0]   PASSED
maximum of two numbers[1]   FAILED

Math.max(a, b) == c
    |    |  |  |  |
    |    7  0  |  7
    42          false

maximum of two numbers[2]   PASSED
```

This tells us that the second iteration (with index 1) failed. With a bit of effort, we can do even better:

```
@Unroll
def "maximum of #a and #b is #c"() { ... }
```

[2] For example, a feature method could use data variables in its `setup:` block, but not in any conditions.

This method name uses placeholders, denoted by a leading hash sign (#), to refer to data variables a, b, and c. In the output, the placeholders will be replaced with concrete values:

```
maximum of 3 and 5 is 5   PASSED
maximum of 7 and 0 is 7   FAILED

Math.max(a, b) == c
    |    |  |  |  |
    |    7  0  |  7
    42          false

maximum of 0 and 0 is 0   PASSED
```

Now we can tell at a glance that the max method failed for inputs 7 and 0. See More on Unrolled Method Names for further details on this topic.

The @Unroll annotation can also be placed on a spec. This has the same effect as placing it on each data-driven feature method of the spec.

1.3.8 Data Pipes

Data tables aren't the only way to supply values to data variables. In fact, a data table is just syntactic sugar for one or more *data pipes*:

```
...
where:
a << [3, 7, 0]
b << [5, 0, 0]
c << [5, 7, 0]
```

A data pipe, indicated by the left-shift (<<) operator, connects a data variable to a *data provider*. The data provider holds all values for the variable, one per iteration. Any object that Groovy knows how to iterate over can be used as a data provider. This includes objects of type Collection, String, Iterable, and objects implementing the Iterable contract. Data providers don't necessarily have to *be* the data (as in the case of a Collection); they can fetch data from external sources like text files, databases and spreadsheets, or generate data randomly. Data providers are queried for their next value only when needed (before the next iteration).

1.3.9 Multi-Variable Data Pipes

If a data provider returns multiple values per iteration (as an object that Groovy knows how to iterate over), it can be connected to multiple data variables simultaneously. The syntax is somewhat similar to Groovy multi-assignment but uses brackets instead of parentheses on the left-hand side:

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

def "maximum of two numbers"() {
    ...
    where:
    [a, b, c] << sql.rows("select a, b, c from maxdata")
}
```

Data values that aren't of interest can be ignored with an underscore (_):

```
...
where:
[a, b, _, c] << sql.rows("select * from maxdata")
```

1.3.10 Data Variable Assignment

A data variable can be directly assigned a value:

```
...
where:
a = 3
b = Math.random() * 100
c = a > b ? a : b
```

Assignments are re-evaluated for every iteration. As already shown above, the right-hand side of an assignment may refer to other data variables:

```
...
where:
row << sql.rows("select * from maxdata")
// pick apart columns
a = row.a
b = row.b
c = row.c
```

1.3.11 Combining Data Tables, Data Pipes, and Variable Assignments

Data tables, data pipes, and variable assignments can be combined as needed:

```
...
where:
a | _
3 | _
7 | _
0 | _

b << [5, 0, 0]

c = a > b ? a : b
```

1.3.12 Number of Iterations

The number of iterations depends on how much data is available. Successive executions of the same method can yield different numbers of iterations. If a data provider runs out of values sooner than its peers, an exception will occur. Variable assignments don't affect the number of iterations. A `where:` block that only contains assignments yields exactly one iteration.

1.3.13 Closing of Data Providers

After all iterations have completed, the zero-argument `close` method is called on all data providers that have such a method.

1.3.14 More on Unrolled Method Names

An unrolled method name is similar to a Groovy `GString`, except for the following differences:

- Expressions are denoted with # instead of `$` [3], and there is no equivalent for the `${...}` syntax

[3] Groovy syntax does not allow dollar signs in method names.

- Expressions only support property access and zero-arg method calls

Given a class `Person` with properties `name` and `age`, and a data variable `person` of type `Person`, the following are valid method names:

```
def "#person is #person.age years old"() { ... } // property access
def "#person.name.toUpperCase()"() { ... } // zero-arg method call
```

Non-string values (like `#person` above) are converted to Strings according to Groovy semantics.

The following are invalid method names:

```
def "#person.name.split(' ')[1]" { ... } // cannot have method arguments
def "#person.age / 2" { ... } // cannot use operators
```

If necessary, additional data variables can be introduced to hold more complex expression:

```
def "#lastName"() {
    ...
    where:
    person << ...
    lastName = person.name.split(' ')[1]
}
```

1.4 Interaction Based Testing

Interaction-based testing is a design and testing technique that emerged in the Extreme Programming (XP) community in the early 2000's. Focusing on the behavior of objects rather than their state, it explores how the object(s) under specification interact, by way of method calls, with their collaborators.

For example, suppose we have a `Publisher` that sends messages to its `Subscriber`'s:

```
class Publisher {
    List<Subscriber> subscribers
    void send(String message)
}

interface Subscriber {
    void receive(String message)
}

class PublisherSpec extends Specification {
    Publisher publisher = new Publisher()
}
```

How are we going to test `Publisher`? With state-based testing, we can verify that the publisher keeps track of its subscribers. The more interesting question, though, is whether a message sent by the publisher is received by the subscribers. To answer this question, we need a special implementation of `Subscriber` that listens in on the conversation between the publisher and its subscribers. Such an implementation is often called a *mock object*.

While we could certainly create a mock implementation of `Subscriber` by hand, writing and maintaining this code can get unpleasant as the number of methods and complexity of interactions increases. This is where mocking frameworks come in: They provide a way to describe the expected interactions between an object under specification and its collaborators, and can generate mock implementations of collaborators that verify these expectations.

How Are Mock Implementations Generated?

Like most Java mocking frameworks, Spock uses JDK dynamic proxies (when mocking interfaces) and CGLIB proxies (when mocking classes) to generate mock implementations at runtime. Compared to implementations based on Groovy meta-programming, this has the advantage that it also works for testing Java code.

The Java world has no shortage of popular and mature mocking frameworks: JMock, EasyMock, Mockito, to name just a few. Although each of these tools can be used together with Spock, we decided to roll our own mocking framework, tightly integrated with Spock's specification language. This decision was driven by the desire to leverage all of Groovy's capabilities to make interaction-based tests easier to write, more readable, and ultimately more fun. We hope that by the end of this chapter, you will agree that we have achieved these goals.

Except where indicated, all features of Spock's mocking framework work both for testing Java and Groovy code.

1.4.1 Creating Mock Objects

Mock objects are created with the `MockingApi.Mock()` method [4]. Let's create two mock subscribers:

```
def subscriber = Mock(Subscriber)
def subscriber2 = Mock(Subscriber)
```

Alternatively, the following Java-like syntax is supported, which may give better IDE support:

```
Subscriber subscriber = Mock()
Subscriber subscriber2 = Mock()
```

Here, the mock's type is inferred from the variable type on the left-hand side of the assignment.

Note: If the mock's type is given on the left-hand side of the assignment, it's permissible (though not required) to omit it on the right-hand side.

Mock objects literally implement (or, in the case of a class, extend) the type they stand in for. In other words, in our example `subscriber` *is-a* `Subscriber`. Hence it can be passed to statically typed (Java) code that expects this type.

1.4.2 Default Behavior of Mock Objects

Lenient vs. Strict Mocking Frameworks

Like Mockito, we firmly believe that a mocking framework should be lenient by default. This means that unexpected method calls on mock objects (or, in other words, interactions that aren't relevant for the test at hand) are allowed and answered with a default response. Conversely, mocking frameworks like EasyMock and JMock are strict by default, and throw an exception for every unexpected method call. While strictness enforces rigor, it can also lead to over-specification, resulting in brittle tests that fail with every other internal code change. Spock's mocking framework makes it easy to describe only what's relevant about an interaction, avoiding the over-specification trap.

Initially, mock objects have no behavior. Calling methods on them is allowed but has no effect other than returning the default value for the method's return type (`false`, `0`, or `null`). An exception are the `Object.equals`, `Object.hashCode`, and `Object.toString` methods, which have the following default behavior: A mock object is only equal to itself, has a unique hash code, and a string representation that includes the name of the type it

[4] For additional ways to create mock objects, see *Other Kinds of Mock Objects (New in 0.7)* and *A la Carte Mocks*.

represents. This default behavior is overridable by stubbing the methods, which we will learn about in the Stubbing section.

1.4.3 Injecting Mock Objects into Code Under Specification

After creating the publisher and its subscribers, we need to make the latter known to the former:

```
class PublisherSpec extends Specification {
    Publisher publisher = new Publisher()
    Subscriber subscriber = Mock()
    Subscriber subscriber2 = Mock()

    def setup() {
        publisher.subscribers << subscriber // << is a Groovy shorthand for List.add()
        publisher.subscribers << subscriber2
    }
}
```

We are now ready to describe the expected interactions between the two parties.

1.4.4 Mocking

Mocking is the act of describing (mandatory) interactions between the object under specification and its collaborators. Here is an example:

```
def "should send messages to all subscribers"() {
    when:
    publisher.send("hello")

    then:
    1 * subscriber.receive("hello")
    1 * subscriber2.receive("hello")
}
```

Read out aloud: "When the publisher sends a 'hello' message, then both subscribers should receive that message exactly once."

When this feature method gets run, all invocations on mock objects that occur while executing the `when` block will be matched against the interactions described in the `then:` block. If one of the interactions isn't satisfied, a (subclass of) `InteractionNotSatisfiedError` will be thrown. This verification happens automatically and does not require any additional code.

Interactions

Is an Interaction Just a Regular Method Invocation?

Not quite. While an interaction looks similar to a regular method invocation, it is simply a way to express which method invocations are expected to occur. A good way to think of an interaction is as a regular expression that all incoming invocations on mock objects are matched against. Depending on the circumstances, the interaction may match zero, one, or multiple invocations.

Let's take a closer look at the `then:` block. It contains two *interactions*, each of which has four distinct parts: a *cardinality*, a *target constraint*, a *method constraint*, and an *argument constraint*:

```
1 * subscriber.receive("hello")
|   |          |       |
|   |          |       argument constraint
|   |          method constraint
|   target constraint
cardinality
```

Cardinality

The cardinality of an interaction describes how often a method call is expected. It can either be a fixed number or a range:

```
1 * subscriber.receive("hello")      // exactly one call
0 * subscriber.receive("hello")      // zero calls
(1..3) * subscriber.receive("hello") // between one and three calls (inclusive)
(1.._) * subscriber.receive("hello") // at least one call
(_..3) * subscriber.receive("hello") // at most three calls
_ * subscriber.receive("hello")      // any number of calls, including zero
                                     // (rarely needed; see 'Strict Mocking')
```

Target Constraint

The target constraint of an interaction describes which mock object is expected to receive the method call:

```
1 * subscriber.receive("hello") // a call to 'subscriber'
1 * _.receive("hello")          // a call to any mock object
```

Method Constraint

The method constraint of an interaction describes which method is expected to be called:

```
1 * subscriber.receive("hello") // a method named 'receive'
1 * subscriber./r.*e/("hello")  // a method whose name matches the given regular expression
                                // (here: method name starts with 'r' and ends in 'e')
```

When expecting a call to a getter method, Groovy property syntax *can* be used instead of method syntax:

```
1 * subscriber.status // same as: 1 * subscriber.getStatus()
```

When expecting a call to a setter method, only method syntax can be used:

```
1 * subscriber.setStatus("ok") // NOT: 1 * subscriber.status = "ok"
```

Argument Constraints

The argument constraints of an interaction describe which method arguments are expected:

```
1 * subscriber.receive("hello")     // an argument that is equal to the String "hello"
1 * subscriber.receive(!"hello")    // an argument that is unequal to the String "hello"
1 * subscriber.receive()            // the empty argument list (would never match in our example)
1 * subscriber.receive(_)           // any single argument (including null)
1 * subscriber.receive(*_)          // any argument list (including the empty argument list)
1 * subscriber.receive(!null)       // any non-null argument
1 * subscriber.receive(_ as String) // any non-null argument that is-a String
```

```
1 * subscriber.receive({ it.size() > 3 }) // an argument that satisfies the given predicate
                                          // (here: message length is greater than 3)
```

Argument constraints work as expected for methods with multiple arguments:

```
1 * process.invoke("ls", "-a", _, !null, { ["abcdefghiklmnopqrstuwx1"].contains(it) })
```

When dealing with vararg methods, vararg syntax can also be used in the corresponding interactions:

```
interface VarArgSubscriber {
    void receive(String... messages)
}

...

subscriber.receive("hello", "goodbye")
```

Spock Deep Dive: Groovy Varargs

Groovy allows any method whose last parameter has an array type to be called in vararg style. Consequently, vararg syntax can also be used in interactions matching such methods.

Matching Any Method Call

Sometimes it can be useful to match "anything", in some sense of the word:

```
1 * subscriber._(*_)      // any method on subscriber, with any argument list
1 * subscriber._          // shortcut for and preferred over the above

1 * _._                   // any method call on any mock object
1 * _                     // shortcut for and preferred over the above
```

Note: Although `(_..._) * _._(*_) >> _` is a valid interaction declaration, it is neither good style nor particularly useful.

Strict Mocking

Now, when would matching any method call be useful? A good example is *strict mocking*, a style of mocking where no interactions other than those explicitly declared are allowed:

```
when:
publisher.publish("hello")

then:
1 * subscriber.receive("hello") // demand one 'receive' call on 'subscriber'
_ * auditing._                  // allow any interaction with 'auditing'
0 * _                           // don't allow any other interaction
```

`0 *` only makes sense as the last interaction of a `then:` block or method. Note the use of `_ *` (any number of calls), which allows any interaction with the auditing component.

Note: `_ * ` is only meaningful in the context of strict mocking. In particular, it is never necessary when *stubbing* an invocation. For example, `_ * auditing.record(_) >> "ok"` can (and should!) be simplified to `auditing.record(_) >> "ok"`.

Where to Declare Interactions

So far, we declared all our interactions in a `then:` block. This often results in a spec that reads naturally. However, it is also permissible to put interactions anywhere *before* the `when:` block that is supposed to satisfy them. In particular, this means that interactions can be declared in a `setup` method. Interactions can also be declared in any "helper" instance method of the same specification class.

When an invocation on a mock object occurs, it is matched against interactions in the interactions' declared order. If an invocation matches multiple interactions, the earliest declared interaction that hasn't reached its upper invocation limit will win. There is one exception to this rule: Interactions declared in a `then:` block are matched against before any other interactions. This allows to override interactions declared in, say, a `setup` method with interactions declared in a `then:` block.

Spock Deep Dive: How Are Interactions Recognized?

In other words, what makes an expression an interaction declaration, rather than, say, a regular method call? Spock uses a simple syntactic rule to recognize interactions: If an expression is in statement position and is either a multiplication (`*`) or a left-shift (`>>`, `>>>`) operation, then it is considered an interaction and will be parsed accordingly. Such an expression would have little to no value in statement position, so changing its meaning works out fine. Note how the operations correspond to the syntax for declaring a cardinality (when mocking) or a response generator (when stubbing). Either of them must always be present; `foo.bar()` alone will never be considered an interaction.

Declaring Interactions at Mock Creation Time (New in 0.7)

If a mock has a set of "base" interactions that don't vary, they can be declared right at mock creation time:

```
def subscriber = Mock(Subscriber) {
    1 * receive("hello")
    1 * receive("goodbye")
}
```

This feature is particularly attractive for *stubbing* and with dedicated Stubs. Note that the interactions don't (and cannot [5]) have a target constraint; it's clear from the context which mock object they belong to.

Interactions can also be declared when initializing an instance field with a mock:

```
class MySpec extends Specification {
    Subscriber subscriber = Mock {
        1 * receive("hello")
        1 * receive("goodbye")
    }
}
```

Grouping Interactions with Same Target (New in 0.7)

Interactions sharing the same target can be grouped in a `Specification.with` block. Similar to *declaring interactions at mock creation time*, this makes it unnecessary to repeat the target constraint:

[5] The `subscriber` variable cannot be referenced from the closure because it is being declared as part of the same statement.

```
with(subscriber) {
    1 * receive("hello")
    1 * receive("goodbye")
}
```

A `with` block can also be used for grouping conditions with the same target.

Mixing Interactions and Conditions

A `then:` block may contain both interactions and conditions. Although not strictly required, it is customary to declare interactions before conditions:

```
when:
publisher.send("hello")

then:
1 * subscriber.receive("hello")
publisher.messageCount == 1
```

Read out aloud: "When the publisher sends a 'hello' message, then the subscriber should receive the message exactly once, and the publisher's message count should be one."

Explicit Interaction Blocks

Internally, Spock must have full information about expected interactions *before* they take place. So how is it possible for interactions to be declared in a `then:` block? The answer is that under the hood, Spock moves interactions declared in a `then:` block to immediately before the preceding `when:` block. In most cases this works out just fine, but sometimes it can lead to problems:

```
when:
publisher.send("hello")

then:
def message = "hello"
1 * subscriber.receive(message)
```

Here we have introduced a variable for the expected argument. (Likewise, we could have introduced a variable for the cardinality.) However, Spock isn't smart enough (huh?) to tell that the interaction is intrinsically linked to the variable declaration. Hence it will just move the interaction, which will cause a `MissingPropertyException` at runtime.

One way to solve this problem is to move (at least) the variable declaration to before the `when:` block. (Fans of *data-driven testing* might move the variable into a `where:` block.) In our example, this would have the added benefit that we could use the same variable for sending the message.

Another solution is to be explicit about the fact that variable declaration and interaction belong together:

```
when:
publisher.send("hello")

then:
interaction {
    def message = "hello"
    1 * subscriber.receive(message)
}
```

Since an `MockingApi.interaction` block is always moved in its entirety, the code now works as intended.

Scope of Interactions

Interactions declared in a `then:` block are scoped to the preceding `when:` block:

```
when:
publisher.send("message1")

then:
subscriber.receive("message1")

when:
publisher.send("message2")

then:
subscriber.receive("message2")
```

This makes sure that `subscriber` receives `"message1"` during execution of the first `when:` block, and `"message2"` during execution of the second `when:` block.

Interactions declared outside a `then:` block are active from their declaration until the end of the containing feature method.

Interactions are always scoped to a particular feature method. Hence they cannot be declared in a static method, `setupSpec` method, or `cleanupSpec` method. Likewise, mock objects should not be stored in static or `@Shared` fields.

Verification of Interactions

There a two main ways in which a mock-based test can fail: An interaction can match more invocations than allowed, or it can match fewer invocations than required. The former case is detected right when the invocation happens, and causes a `TooManyInvocationsError`:

```
Too many invocations for:

2 * subscriber.receive(_) (3 invocations)
```

To make it easier to diagnose why too many invocations matched, Spock will show all invocations matching the interaction in question (new in Spock 0.7):

```
Matching invocations (ordered by last occurrence):

2 * subscriber.receive("hello")   <-- this triggered the error
1 * subscriber.receive("goodbye")
```

According to this output, one of the `receive("hello")` calls triggered the `TooManyInvocationsError`. Note that because indistinguishable calls like the two invocations of `subscriber.receive("hello")` are aggregated into a single line of output, the first `receive("hello")` may well have occurred before the `receive("goodbye")`.

The second case (fewer invocations than required) can only be detected once execution of the `when` block has completed. (Until then, further invocations may still occur.) It causes a `TooFewInvocationsError`:

```
Too few invocations for:

1 * subscriber.receive("hello") (0 invocations)
```

Note that it doesn't matter whether the method was not called at all, the same method was called with different arguments, the same method was called on a different mock object, or a different method was called "instead" of this one; in either case, a `TooFewInvocationsError` error will occur. To make it easier to diagnose what happened

"instead" of a missing invocation, Spock will show all invocations that didn't match any interaction, ordered by their similarity with the interaction in question (new in Spock 0.7). In particular, invocations that match everything but the interaction's arguments will be shown first:

```
Unmatched invocations (ordered by similarity):

1 * subscriber.receive("goodbye")
1 * subscriber2.receive("hello")
```

Invocation Order

Often, the exact method invocation order isn't relevant and may change over time. To avoid over-specification, Spock defaults to allowing any invocation order, provided that the specified interactions are eventually satisfied:

```
then:
2 * subscriber.receive("hello")
1 * subscriber.receive("goodbye")
```

Here, any of the invocation sequences `"hello" "hello" "goodbye"`, `"hello" "goodbye" "hello"`, and `"goodbye" "hello" "hello"` will satisfy the specified interactions.

In those cases where invocation order matters, you can impose an order by splitting up interactions into multiple `then:` blocks:

```
then:
2 * subscriber.receive("hello")

then:
1 * subscriber.receive("goodbye")
```

Now Spock will verify that both `"hello"`'s are received before the `"goodbye"`. In other words, invocation order is enforced *between* but not *within* `then:` blocks.

Note: Splitting up a `then:` block with `and:` does not impose any ordering, as `and:` is only meant for documentation purposes and doesn't carry any semantics.

Mocking Classes

Besides interfaces, Spock also supports mocking of classes. Mocking classes works just like mocking interfaces; the only additional requirement is to put `cglib-nodep-2.2` or higher and `objenesis-1.2` or higher on the class path. If either of these libraries is missing from the class path, Spock will gently let you know.

1.4.5 Stubbing

Stubbing is the act of making collaborators respond to method calls in a certain way. When stubbing a method, you don't care if and how many times the method is going to be called; you just want it to return some value, or perform some side effect, *whenever* it gets called.

For the sake of the following examples, let's modify the `Subscriber`'s `receive` method to return a status code that tells if the subscriber was able to process a message:

```
interface Subscriber {
    String receive(String message)
}
```

Now, let's make the `receive` method return `"ok"` on every invocation:

```
subscriber.receive(_) >> "ok"
```

Read out aloud: "*Whenever* the subscriber receives a message, *make* it respond with 'ok'."

Compared to a mocked interaction, a stubbed interaction has no cardinality on the left end, but adds a *response generator* on the right end:

```
subscriber.receive(_) >> "ok"
|          |       |     |
|          |       |     response generator
|          |       argument constraint
|          method constraint
target constraint
```

A stubbed interaction can be declared in the usual places: either inside a `then:` block, or anywhere before a `when:` block. (See *Where to Declare Interactions* for the details.) If a mock object is only used for stubbing, it's common to declare interactions *at mock creation time* or in a `setup:` block.

Returning Fixed Values

We have already seen the use of the right-shift (>>) operator to return a fixed value:

```
subscriber.receive(_) >> "ok"
```

To return different values for different invocations, use multiple interactions:

```
subscriber.receive("message1") >> "ok"
subscriber.receive("message2") >> "fail"
```

This will return `"ok"` whenever `"message1"` is received, and `"fail"` whenever `"message2"` is received. There is no limit as to which values can be returned, provided they are compatible with the method's declared return type.

Returning Sequences of Values

To return different values on successive invocations, use the triple-right-shift (>>>) operator:

```
subscriber.receive(_) >>> ["ok", "error", "error", "ok"]
```

This will return `"ok"` for the first invocation, `"error"` for the second and third invocation, and `"ok"` for all remaining invocations. The right-hand side must be a value that Groovy knows how to iterate over; in this example, we've used a plain list.

Computing Return Values

To compute a return value based on the method's argument, use the the right-shift (>>) operator together with a closure. If the closure declares a single untyped parameter, it gets passed the method's argument list:

```
subscriber.receive(_) >> { args -> args[0].size() > 3 ? "ok" : "fail" }
```

Here `"ok"` gets returned if the message is more than three characters long, and `"fail"` otherwise.

In most cases it would be more convenient to have direct access to the method's arguments. If the closure declares more than one parameter or a single *typed* parameter, method arguments will be mapped one-by-one to closure parameters [6]:

[6] The destructuring semantics for closure arguments come straight from Groovy.

```
subscriber.receive(_) >> { String message -> message.size() > 3 ? "ok" : "fail" }
```

This response generator behaves the same as the previous one, but is arguably more readable.

If you find yourself in need of more information about a method invocation than its arguments, have a look at `org.spockframework.mock.IMockInvocation`. All methods declared in this interface are available inside the closure, without a need to prefix them. (In Groovy terminology, the closure *delegates* to an instance of `IMockInvocation`.)

Performing Side Effects

Sometimes you may want to do more than just computing a return value. A typical example is throwing an exception. Again, closures come to the rescue:

```
subscriber.receive(_) >> { throw new InternalError("ouch") }
```

Of course, the closure can contain more code, for example a `println` statement. It will get executed every time an incoming invocation matches the interaction.

Chaining Method Responses

Method responses can be chained:

```
subscriber.receive(_) >>> ["ok", "fail", "ok"] >> { throw new InternalError() } >> "ok"
```

This will return `"ok", "fail", "ok"` for the first three invocations, throw `InternalError` for the fourth invocations, and return `ok` for any further invocation.

1.4.6 Combining Mocking and Stubbing

Mocking and stubbing go hand-in-hand:

```
1 * subscriber.receive("message1") >> "ok"
1 * subscriber.receive("message2") >> "fail"
```

When mocking and stubbing the same method call, they have to happen in the same interaction. In particular, the following Mockito-style splitting of stubbing and mocking into two separate statements will *not* work:

```
setup:
subscriber.receive("message1") >> "ok"

when:
publisher.send("message1")

then:
1 * subscriber.receive("message1")
```

As explained in *Where to Declare Interactions*, the `receive` call will first get matched against the interaction in the `then:` block. Since that interaction doesn't specify a response, the default value for the method's return type (`null` in this case) will be returned. (This is just another facet of Spock's lenient approach to mocking.). Hence, the interaction in the `setup:` block will never get a chance to match.

Note: Mocking and stubbing of the same method call has to happen in the same interaction.

1.4.7 Other Kinds of Mock Objects (New in 0.7)

So far, we have created mock objects with the `MockingApi.Mock` method. Aside from this method, the `MockingApi` class provides a couple of other factory methods for creating more specialized kinds of mock objects.

Stubs

A *stub* is created with the `MockingApi.Stub` factory method:

```
def subscriber = Stub(Subscriber)
```

Whereas a mock can be used both for stubbing and mocking, a stub can only be used for stubbing. Limiting a collaborator to a stub communicates its role to the readers of the specification.

Note: If a stub invocation matches a *mandatory* interaction (like `1 * foo.bar()`), an `InvalidSpecException` is thrown.

Like a mock, a stub allows unexpected invocations. However, the values returned by a stub in such cases are more ambitious:

- For primitive types, the primitive type's default value is returned.
- For non-primitive numerical values (like `BigDecimal`), zero is returned.
- For non-numerical values, an "empty" or "dummy" object is returned. This could mean an empty String, an empty collection, an object constructed from its default constructor, or another stub returning default values. See class `org.spockframework.mock.EmptyOrDummyResponse` for the details.

A stub often has a fixed set of interactions, which makes *declaring interactions at mock creation time* particularly attractive:

```
def subscriber = Stub(Subscriber) {
    receive("message1") >> "ok"
    receive("message2") >> "fail"
}
```

Spies

(Think twice before using this feature. It might be better to change the design of the code under specification.)

A *spy* is created with the `MockingApi.Spy` factory method:

```
def subscriber = Spy(SubscriberImpl, constructorArgs: ["Fred"])
```

A spy is always based on a real object. Hence you must provide a class type rather than an interface type, along with any constructor arguments for the type. If no constructor arguments are provided, the type's default constructor will be used.

Method calls on a spy are automatically delegated to the real object. Likewise, values returned from the real object's methods are passed back to the caller via the spy.

After creating a spy, you can listen in on the conversation between the caller and the real object underlying the spy:

```
1 * subscriber.receive(_)
```

Apart from making sure that `receive` gets called exactly once, the conversation between the publisher and the `SubscriberImpl` instance underlying the spy remains unaltered.

When stubbing a method on a spy, the real method no longer gets called:

```
subscriber.receive(_) >> "ok"
```

Instead of calling `SubscriberImpl.receive`, the `receive` method will now simply return `"ok"`.

Sometimes, it is desirable to both execute some code *and* delegate to the real method:

```
subscriber.receive(_) >> { String message -> callRealMethod(); message.size() > 3 ? "ok" : "fail" }
```

Here we use `callRealMethod()` to delegate the method invocation to the real object. Note that we don't have to pass the `message` argument along; this is taken care of automatically. *callRealMethod()*' returns the real invocation's result, but in this example we opted to return our own result instead. If we had wanted to pass a different message to the real method, we could have used `callRealMethodWithArgs("changed message")`.

Partial Mocks

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Spies can also be used as partial mocks:

```
// this is now the object under specification, not a collaborator
def persister = Spy(MessagePersister) {
  // stub a call on the same object
  isPersistable(_) >> true
}

when:
persister.receive("msg")

then:
// demand a call on the same object
1 * persister.persist("msg")
```

1.4.8 Groovy Mocks (New in 0.7)

So far, all the mocking features we have seen work the same no matter if the calling code is written in Java or Groovy. By leveraging Groovy's dynamic capabilities, Groovy mocks offer some additional features specifically for testing Groovy code. They are created with the `MockingApi.GroovyMock()`, `MockingApi.GroovyStub()`, and `MockingApi.GroovySpy()` factory methods.

When Should Groovy Mocks be Favored over Regular Mocks?

Groovy mocks should be used when the code under specification is written in Groovy *and* some of the unique Groovy mock features are needed. When called from Java code, Groovy mocks will behave like regular mocks. Note that it isn't necessary to use a Groovy mock merely because the code under specification and/or mocked type is written in Groovy. Unless you have a concrete reason to use a Groovy mock, prefer a regular mock.

Mocking Dynamic Methods

All Groovy mocks implement the `GroovyObject` interface. They support the mocking and stubbing of dynamic methods as if they were physically declared methods:

```
def subscriber = GroovyMock(Subscriber)

1 * subscriber.someDynamicMethod("hello")
```

Mocking All Instances of a Type

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Usually, Groovy mocks need to be injected into the code under specification just like regular mocks. However, when a Groovy mock is created as *global*, it automagically replaces all real instances of the mocked type for the duration of the feature method [7]:

```
def publisher = new Publisher()
publisher << new RealSubscriber() << new RealSubscriber()

def anySubscriber = GroovyMock(RealSubscriber, global: true)

when:
publisher.publish("message")

then:
2 * anySubscriber.receive("message")
```

Here, we set up the publisher with two instances of a real subscriber implementation. Then we create a global mock of the *same* type. This reroutes all method calls on the real subscribers to the mock object. The mock object's instance isn't ever passed to the publisher; it is only used to describe the interaction.

Note: A global mock can only be created for a class type. It effectively replaces all instances of that type for the duration of the feature method.

Since global mocks have a somewhat, well, global effect, it's often convenient to use them together with `GroovySpy`. This leads to the real code getting executed *unless* an interaction matches, allowing you to selectively listen in on objects and change their behavior just where needed.

How Are Global Groovy Mocks Implemented?

Global Groovy mocks get their super powers from Groovy meta-programming. To be more precise, every globally mocked type is assigned a custom meta class for the duration of the feature method. Since a global Groovy mock is still based on a CGLIB proxy, it will retain its general mocking capabilities (but not its super powers) when called from Java code.

Mocking Constructors

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Global mocks support mocking of constructors:

```
def anySubscriber = GroovySpy(RealSubscriber, global: true)

1 * new RealSubscriber("Fred")
```

[7] You may know this behavior from Groovy's MockFor and StubFor facilities.

Since we are using a spy, the object returned from the constructor call remains unchanged. To change which object gets constructed, we can stub the constructor:

```
new RealSubscriber("Fred") >> new RealSubscriber("Barney")
```

Now, whenever some code tries to construct a subscriber named Fred, we'll construct a subscriber named Barney instead.

Mocking Static Methods

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Global mocks support mocking and stubbing of static methods:

```
def anySubscriber = GroovySpy(RealSubscriber, global: true)

1 * RealSubscriber.someStaticMethod("hello") >> 42
```

The same works for dynamic static methods.

When a global mock is used solely for mocking constructors and static methods, the mock's instance isn't really needed. In such a case one can just write:

```
GroovySpy(RealSubscriber, global: true)
```

1.4.9 Advanced Features (New in 0.7)

Most of the time you shouldn't need these features. But if you do, you'll be glad to have them.

A la Carte Mocks

At the end of the day, the `Mock()`, `Stub()`, and `Spy()` factory methods are just precanned ways to create mock objects with a certain configuration. If you want more fine-grained control over a mock's configuration, have a look at the `org.spockframework.mock.IMockConfiguration` interface. All properties of this interface [8] can be passed as named arguments to the `Mock()` method. For example:

```
def person = Mock(name: "Fred", type: Person, defaultResponse: ZeroOrNullResponse, verified: false)
```

Here, we create a mock whose default return values match those of a `Mock()`, but whose invocations aren't verified (as for a `Stub()`). Instead of passing `ZeroOrNullResponse`, we could have supplied our own custom `org.spockframework.mock.IDefaultResponse` for responding to unexpected method invocations.

Detecting Mock Objects

To find out whether a particular object is a Spock mock object, use a `org.spockframework.mock.MockDetector`:

```
def detector = new MockDetector()
def list1 = []
def list2 = Mock(List)

expect:
!detector.isMock(list1)
detector.isMock(list2)
```

[8] Because mock configurations are immutable, the interface contains just the properties' getters.

A detector can also be used to get more information about a mock object:

```
def mock = detector.asMock(list2)

expect:
mock.name == "list2"
mock.type == List
mock.nature == MockNature.MOCK
```

1.4.10 Further Reading

To learn more about interaction-based testing, we recommend the following resources:

- Endo-Testing: Unit Testing with Mock Objects

  Paper from the XP2000 conference that introduces the concept of mock objects.

- Mock Roles, not Objects

  Paper from the OOPSLA2004 conference that explains how to do mocking *right*.

- Mocks Aren't Stubs

  Martin Fowler's take on mocking.

- Growing Object-Oriented Software Guided by Tests

  TDD pioneers Steve Freeman and Nat Pryce explain in detail how test-driven development and mocking work in the real world.

1.5 Extensions

Spock comes with a powerful extension mechanism, which allows to hook into a spec's lifecycle to enrich or alter its behavior. In this chapter, we will first learn about Spock's built-in extensions, and then dive into writing custom extensions.

1.5.1 Built-In Extensions

Most of Spock's built-in extensions are *annotation-driven*. In other words, they are triggered by annotating a spec class or method with a certain annotation. You can tell such an annotation by its `@ExtensionAnnotation` meta-annotation.

Ignore

To temporarily prevent a feature method from getting executed, annotate it with `spock.lang.Ignore`:

```
@Ignore
def "my feature"() { ... }
```

For documentation purposes, a reason can be provided:

```
@Ignore(reason = "TODO")
def "my feature"() { ... }
```

To ignore a whole specification, annotate its class:

```
@Ignore
class MySpec extends Specification { ... }
```

In most execution environments, ignored feature methods and specs will be reported as "skipped".

IgnoreRest

To ignore all but a (typically) small subset of methods, annotate the latter with `spock.lang.IgnoreRest`:

```
def "I'll be ignored"() { ... }

@IgnoreRest
def "I'll run"() { ... }

def "I'll also be ignored"() { ... }
```

`@IgnoreRest` is especially handy in execution environments that don't provide an (easy) way to run a subset of methods.

IgnoreIf

To ignore a feature method under certain conditions, annotate it with `spock.lang.IgnoreIf`, followed by a predicate:

```
@IgnoreIf({ System.getProperty("os.name").contains("windows") })
def "I'll run everywhere but on Windows"() { ... }
```

To make predicates easier to read and write, the following properties are available inside the closure:

- `sys` A map of all system properties
- `env` A map of all environment variables
- `os` Information about the operating system (see `spock.util.environment.OperatingSystem`)
- `jvm` Information about the JVM (see `spock.util.environment.Jvm`)

Using the `os` property, the previous example can be rewritten as:

```
@IgnoreIf({ os.windows })
def "I'll run everywhere but on Windows"() { ... }
```

Requires

To execute a feature method under certain conditions, annotate it with `spock.lang.Requires`, followed by a predicate:

```
@Requires({ os.windows })
def "I'll only run on Windows"() { ... }
```

`Requires` works exactly like `IgnoreIf`, except that the predicate is inverted. In general, it is preferable to state the conditions under which a method gets executed, rather than the conditions under which it gets ignored.

TODO More to follow.

1.5.2 Writing Custom Extensions

TODO

1.6 New and Noteworthy

1.6.1 0.7

Snapshot Repository Moved

Spock snapshots are now available from http://oss.sonatype.org/content/repositories/snapshots/.

New Reference Documentation

The new Spock reference documentation is available at http://docs.spockframework.org. It will gradually replace the documentation at http://wiki.spockframework.org. Each Spock version is documented separately (e.g. http://docs.spockframework.org/en/spock-0.7-groovy-1.8). Documentation for the latest Spock snapshot is at http://docs.spockframework.org/en/latest. As of Spock 0.7, the chapters on *Data Driven Testing* and *Interaction Based Testing* are complete.

Improved Mocking Failure Message for `TooManyInvocationsError`

The diagnostic message accompanying a `TooManyInvocationsError` has been greatly improved. Here is an example:

```
Too many invocations for:

3 * person.sing(_)   (4 invocations)

Matching invocations (ordered by last occurrence):

2 * person.sing("do")   <-- this triggered the error
1 * person.sing("re")
1 * person.sing("mi")
```

Reference Documentation.

Improved Mocking Failure Message for `TooFewInvocationsError`

The diagnostic message accompanying a `TooFewInvocationsError` has been greatly improved. Here is an example:

```
Too few invocations for:

1 * person.sing("fa")   (0 invocations)

Unmatched invocations (ordered by similarity):

1 * person.sing("re")
1 * person.say("fa")
1 * person2.shout("mi")
```

Reference Documentation.

Stubs

Besides mocks, Spock now has explicit support for stubs:

```
def person = Stub(Person)
```

A stub is a restricted form of mock object that responds to invocations without ever demanding them. Other than not having a cardinality, a stub's interactions look just like a mock's interactions. Using a stub over a mock is an effective way to communicate its role to readers of the specification.

Reference Documentation.

Spies

Besides mocks, Spock now has support for spies:

```
def person = Spy(Person, constructorArgs: ["Fred"])
```

A spy sits atop a real object, in this example an instance of class `Person`. All invocations on the spy that don't match an interaction are delegated to that object. This allows to listen in on and selectively change the behavior of the real object. Furthermore, spies can be used as partial mocks.

Reference documentation.

Declaring Interactions at Mock Creation Time

Interactions can now be declared at mock creation time:

```
def person = Mock(Person) {
    sing() >> "tra-la-la
    3 * eat()
}
```

This feature is particularly attractive for Stubs.

Reference Documentation.

Groovy Mocks

Spock now offers specialized mock objects for spec'ing Groovy code:

```
def mock = GroovyMock(Person)
def stub = GroovyStub(Person)
def spy = GroovySpy(Person)
```

A Groovy mock automatically implements `groovy.lang.GroovyObject`. It allows stubbing and mocking of dynamic methods just like for statically declared methods. When a Groovy mock is called from Java rather than Groovy code, it behaves like a regular mock.

Reference Documentation.

Global Mocks

A Groovy mock can be made *global*:

```
GroovySpy(Person, global: true)
```

A global mock can only be created for a class type. It effectively replaces all instances of that type and makes them amenable to stubbing and mocking. (You may know this behavior from Groovy's `MockFor` and `StubFor` facilities.) Furthermore, a global mock allows mocking of the type's constructors and static methods.

Reference Documentation.

Grouping Conditions with Same Target Object

Inspired from Groovy's `Object.with` method, the `Specification.with` method allows to group conditions involving the same target object:

```
def person = new Person(name: "Fred", age: 33, sex: "male")

expect:
with(person) {
    name == "Fred"
    age == 33
    sex == "male"
}
```

Grouping Interactions with Same Target Object

The `with` method can also be used for grouping interactions:

```
def service = Mock(Service)
app.service = service

when:
app.run()

then:
with(service) {
    1 * start()
    1 * act()
    1 * stop()
}
```

Reference Documentation.

Polling Conditions

`spock.util.concurrent.PollingConditions` joins `AsyncConditions` and `BlockingVariable(s)` as another utility for testing asynchronous code:

```
def person = new Person(name: "Fred", age: 22)
def conditions = new PollingConditions(timeout: 10)

when:
Thread.start {
    sleep(1000)
    person.age = 42
    sleep(5000)
    person.name = "Barney"
```

```
}

then:
conditions.within(2) {
    assert person.age == 42
}

conditions.eventually {
    assert person.name == "Barney"
}
```

Experimental DSL Support for Eclipse

Spock now ships with a DSL descriptor that lets Groovy Eclipse better understand certain parts of Spock's DSL. The descriptor is automatically detected and activated by the IDE. Here is an example:

```
// currently need to type variable for the following to work
Person person = new Person(name: "Fred", age: 42)

expect:
with(person) {
    name == "Fred" // editor understands and auto-completes 'name'
    age == 42      // editor understands and auto-completes 'age'
}
```

Another example:

```
def person = Stub(Person) {
    getName() >> "Fred" // editor understands and auto-completes 'getName()'
    getAge() >> 42      // editor understands and auto-completes 'getAge()'
}
```

DSL support is activated for Groovy Eclipse 2.7.1 and higher. If necessary, it can be deactivated in the Groovy Eclipse preferences.

Experimental DSL Support for IntelliJ IDEA

Spock now ships with a DSL descriptor that lets Intellij IDEA better understand certain parts of Spock's DSL. The descriptor is automatically detected and activated by the IDE. Here is an example:

```
def person = new Person(name: "Fred", age: 42)

expect:
with(person) {
    name == "Fred" // editor understands and auto-completes 'name'
    age == 42      // editor understands and auto-completes 'age'
}
```

Another example:

```
def person = Stub(Person) {
    getName() >> "Fred" // editor understands and auto-completes 'getName()'
    getAge() >> 42      // editor understands and auto-completes 'getAge()'
}
```

DSL support is activated for IntelliJ IDEA 11.1 and higher.

Splitting up Class Specification

Parts of class `spock.lang.Specification` were pulled up into two new super classes: `spock.lang.MockingApi` now contains all mocking-related methods, and `org.spockframework.lang.SpecInternals` contains internal methods which aren't meant to be used directly.

Improved Failure Messages for `notThrown` and `noExceptionThrown`

Instead of just passing through exceptions, `Specification.notThrown` and `Specification.noExceptionThrown` now fail with messages like:

```
Expected no exception to be thrown, but got 'java.io.FileNotFoundException'

Caused by: java.io.FileNotFoundException: ...
```

`HamcrestSupport.expect`

Class `spock.util.matcher.HamcrestSupport` has a new `expect` method that makes [Hamcrest](http://code.google.com/p/hamcrest/) assertions read better in then-blocks:

```
when:
def x = computeValue()

then:
expect x, closeTo(42, 0.01)
```

@Beta

Recently introduced classes and methods may be annotated with @Beta, as a sign that they may still undergo incompatible changes. This gives us a chance to incorporate valuable feedback from our users. (Yes, we need your feedback!) Typically, a @Beta annotation is removed within one or two releases.

Fixed Issues

See the issue tracker for a list of fixed issues.

1.6.2 0.6

Mocking Improvements

The mocking framework now provides better diagnostic messages in some cases.

Multiple result declarations can be chained. The following causes method bar to throw an `IOException` when first called, return the numbers one, two, and three on the next calls, and throw a `RuntimeException` for all subsequent calls:

```
foo.bar() >> { throw new IOException() } >>> [1, 2, 3] >> { throw new RuntimeException() }
```

It's now possible to match any argument list (including the empty list) with `foo.bar(*_)`.

Method arguments can now be constrained with Hamcrest matchers:

```
import static spock.util.matcher.HamcrestMatchers.closeTo

...

1 * foo.bar(closeTo(42, 0.001))
```

Extended JUnit Rules Support

In addition to rules implementing `org.junit.rules.MethodRule` (which has been deprecated in JUnit 4.9), Spock now also supports rules implementing the new `org.junit.rules.TestRule` interface. Also supported is the new `@ClassRule` annotation. Rule declarations are now verified and can leave off the initialization part. I that case Spock will automatically initialize the rule by calling the default constructor. The `@TestName` rule, and rules in general, now honor the `@Unroll` annotation and any defined naming pattern.

See Issue 240 for a known limitation with Spock's TestRule support.

Condition Rendering Improvements

When two objects are compared with the == operator, they are unequal, but their string representations are the same, Spock will now print the objects' types:

```
enteredNumber == 42
|             |
|             false
42 (java.lang.String)
```

JUnit Fixture Annotations

Fixture methods can now be declared with JUnit's `@Before`, `@After`, `@BeforeClass`, and `@AfterClass` annotations, as an addition or alternative to Spock's own fixture methods. This was particularly needed for Grails 2.0 support.

Tapestry 5.3 Support

Thanks to a contribution from Howard Lewis Ship, the Tapestry module is now compatible with Tapestry 5.3. Older 5.x versions are still supported.

IBM JDK Support

Spock now runs fine on IBM JDKs, working around a bug in the IBM JDK's verifier.

Improved JUnit Compatibility

`org.junit.internal.AssumptionViolatedException` is now recognized and handled as known from JUnit. `@Unrolled` methods no longer cause "yellow" nodes in IDEs.

Improved `@Unroll`

The `@Unroll` naming pattern can now be provided in the method name, instead of as an argument to the annotation:

```
@Unroll
def "maximum of #a and #b is #c"() {
    expect:
    Math.max(a, b) == c

    where:
    a | b | c
    1 | 2 | 2
}
```

The naming pattern now supports property access and zero-arg method calls:

```
@Unroll
def "#person.name.toUpperCase() is #person.age years old"() { ... }
```

The `@Unroll` annotation can now be applied to a spec class. In this case, all data-driven feature methods in the class will be unrolled.

Improved `@Timeout`

The `@Timeout` annotation can now be applied to a spec class. In this case, the timeout applies to all feature methods (individually) that aren't already annotated with `@Timeout`. Timed methods are now executed on the regular test framework thread. This can be important for tests that rely on thread-local state (like Grails integration tests). Also the interruption behavior has been improved, to increase the chance that a timeout can be enforced.

The failure exception that is thrown when a timeout occurs now contains the stacktrace of test execution, allowing you to see where the test was "stuck" or how far it got in the allocated time.

Improved Data Table Syntax

Table cells can now be separated with double pipes. This can be used to visually set apart expected outputs from provided inputs:

```
...
where:
a | b || sum
1 | 2 || 3
3 | 1 || 4
```

Groovy 1.8/2.0 Support

Spock 0.6 ships in three variants for Groovy 1.7, 1.8, and 2.0. Make sure to pick the right version - for example, for Groovy 1.8 you need to use spock-core-0.6-groovy-1.8 (likewise for all other modules). The Groovy 2.0 variant is based on Groovy 2.0-beta-3-SNAPSHOT and only available from http://m2repo.spockframework.org. The Groovy 1.7 and 1.8 variants are also available from Maven Central. The next version of Spock will no longer support Groovy 1.7.

Grails 2.0 Support

Spock's Grails plugin was split off into a separate project and now lives at http://github.spockframework.org/spock-grails. The plugin supports both Grails 1.3 and 2.0.

The Spock Grails plugin supports all of the new Grails 2.0 test mixins, effectively deprecating the existing unit testing classes (e.g. UnitSpec). For integration testing, IntegrationSpec must still be used.

IntelliJ IDEA Integration

The folks from JetBrains have added a few handy features around data tables. Data tables will now be layed out automatically when reformatting code. Data variables are no longer shown as "unknown" and have their types inferred from the values in the table (!).

GitHub Repository

All source code has moved to http://github.spockframework.org/. The Grails Spock plugin, Spock Example project, and Spock Web Console now have their own GitHub projects. Also available are slides and code for various Spock presentations (like this one).

Gradle Build

Spock is now exclusively built with Gradle. Building Spock yourself is as easy as cloning the GitHub repo and executing `gradlew build`. No build tool installation is required; the only prerequisite for building Spock is a JDK installation (1.5 or higher).

Fixed Issues

See the issue tracker for a list of fixed issues.

1.7 Migration Guide

This page explains incompatible changes between successive versions and provides suggestions on how to deal with them.

1.7.1 0.7

Client code must be recompiled in order to work with Spock 0.7. This includes third-party Spock extensions and base classes.

No known source incompatible changes.

1.7.2 0.6

Class initialization order

> **Note:** This only affects cases where one specification class inherits from another one.

Given these specifications:

```
class Base extends Specification {
    def base1 = "base1"
    def base2

    def setup() { base2 = "base2" }
}

class Derived extends Base {
    def derived1 = "derived1"
    def derived2

    def setup() { derived2 = "derived2" }
}
```

In 0.5, above assignments happened in the order `base1`, `base2`, `derived1`, `derived2`. In other words, field initializers were executed right before the setup method in the same class. In 0.6, assignments happen in the order `base1`, `derived1`, `base2`, `derived2`. This is a more conventional order that solves a few problems that users faced with the previous behavior, and also allows us to support JUnit's new `TestRule`. As a result of this change, the following will no longer work:

```
class Base extends Specification {
    def base

    def setup() { base = "base" }
}

class Derived extends Base {
    def derived = base + "derived" // base is not yet set
}
```

To overcome this problem, you can either use a field initializer for `base`, or move the assignment of `derived` into a setup method.

`@Unroll` naming pattern syntax

Note: This is not a change from 0.5, but a change compared to 0.6-SNAPSHOT.

Note: This only affects the Groovy 1.8 and 2.0 variants.

In 0.5, the naming pattern was string based:

```
@Unroll("maximum of #a and #b is #c")
def "maximum of two numbers"() {
    expect:
    Math.max(a, b) == c

    where:
    a | b | c
    1 | 2 | 2
}
```

In 0.6-SNAPSHOT, this was changed to a closure returning a `GString`:

```
@Unroll({"maximum of $a and $b is $c"})
def "maximum of two numbers"() { ... }
```

For various reasons, the new syntax didn't work out as we had hoped, and eventually we decided to go back to the string based syntax. See *Improved @Unroll* for recent improvements to that syntax.

Hamcrest matcher syntax

Note: This only affects users moving from the Groovy 1.7 to the 1.8 or 2.0 variant.

Spock offers a very neat syntax for using Hamcrest matchers:

```
import static spock.util.matcher.HamcrestMatchers.closeTo

...

expect:
answer closeTo(42, 0.001)
```

Due to changes made between Groovy 1.7 and 1.8, this syntax no longer works in as many cases as it did before. For example, the following will no longer work:

```
expect:
object.getAnswer() closeTo(42, 0.001)
```

To avoid such problems, use `HamcrestSupport.that`:

```
import static spock.util.matcher.HamcrestSupport.that

...

expect:
that answer, closeTo(42, 0.001)
```

A future version of Spock will likely remove the former syntax and strengthen the latter one.