sSpec

by

Dave Astels

July 2, 2006

for version 0.11

Copyright 2006 David R. Astels

Introduction 3

What Is sSpec? 3

Core API 4

General 4

Class/Type 5

Blocks 5

Collections 6

Mock API 7

Creating a mock 7

Expecting Messages 7

Arbitrary Message Handling 7

Expecting Arguments 7

Argument Constraints 8

Receive Counts 8

Return Values 9

Raising and Throwing 10

Ordering 10

Using a Mock 11

Executing Expectations 12

TextSpecRunner 12

RefactoringBrowser Integration 13

Copyright 2006 David R. Astels

Introduction

What Is sSpec?

sSpec is a framework for writing executable specifications for Smalltalk programs.

Expectations in sSpec are chains of message sends. For example:

```
stack top should not equal: 3
```

The target object here is the result of **`stack top`**, the message **`should`** is sent to the target, which returns a **`ShouldHelper`** instance. This is then sent the **`not`** message, returning a **`ShouldNegator`**, which is sent the **`equal:`** message with the **`3`** as an argument.

Copyright 2006 David R. Astels

Core API

sSpec defines a method **should** on every object in the system. This **should** method is your entry to the magic of sSpec.

Almost all expectation forms have a corresponding negated form. It is listed when it is supported and, unless otherwise stated, is met when ever the non-negated form would be violated.

General

Arbitrary Block

```
target should satisfy: [:arg | …]
target should not satisfy: [:arg | …]
```

The supplied block is evaluated, passing **target** as the sole argument. If the block evaluates to **false**, **ExpectationNotMetError** is raised.

```
target should satisfy: [:arg | arg > 0]
```

Equality

```
target should equal: <value>
target should not equal: <value>
```

The target object is compared to **value** using =. If the result is **false**, **ExpectationNotMetError** is raised.

Floating Point Comparison

```
target should be within: <tolerance> of: <value>
target should not be within: <tolerance> of: <value>
```

The target object is compared to **value**. If they differ by more that **tolerance**, **ExpectationNotMetError** is raised. In the negated case, **ExpectationNotMetError** is raised if they differ by less than **tolerance**.

```
target should be within: 0.05 of: 27.35
```

Identity

```
target should be: <value>
target should not be: <value>
```

The target object is compared to **value** using ==. If the result is **false**, **ExpectationNotMetError** is raised.

Arbitrary Predicate

```
target should predicate
target should be predicate
target should not predicate
target should not be predicate
```

The message predicate can be any selector understood by **target**. It will be sent to **target** and if the result is **false**, **ExpectationNotMetError** is raised. Note, that it can be any selector, even though only a unary selector is shown here.

If the given selector is not understood by **target**, sSpec tries prepending (and capitalizing) ‘is’ and ‘has’ to find a selector that **target** understands.

For example:

```
container should be empty => container isEmpty
```

Class/Type

Direct Instance

```
target should be an instance of: <class>
target should not be an instance of: <class>
```

An **ExpectationNotMetError** is raised if **target** is not or is, respectively, an direct instance of **class**. As expected this correlates to **target isMemberOf: class**.

Ancestor Class

```
target should be a kind of: <class>
target should not be a kind of: <class>
```

As above, but uses **target isKindOf: class**... checking whether **class** is the direct class of **target**, or an ancestor of **target**'s direct class.

Type

```
target should respond to: <selector>
target should not respond to: <selector>
```

Uses **target respondsTo: symbol** to check whether **symbol** is a selector that **target** understands.

Blocks

Raising

```
aBlock should raise: <exception>
aBlock should not raise: <exception>
```

Checks that **aBlock** causes the named exception to be raised or not The latter is actually one of two cases: some other exception is raised, or no exception is raised. For example:

```
[3 / 0] should raise: ZeroDivide
```

There is a more general form as well.

```
aBlock should raise
aBlock should not raise
```

Copyright 2006 David R. Astels

These forms don't worry about what exception is raised (or not). All they are concerned with is that some except was raised, or that no exception was.

Collections

Containment

```
target should include: <object>
target should not include: <object>
```

This is simply a specific case of the arbitrary predicate form. It uses **`target includes: object`** and raises an **`ExpectationNotMetError`** if that returns false.

The remaining collection forms are a little more involved. They rely on two things: 1) **`target`** responds to the message **`things`** by returning an object that 2) responds to either **`length`** or **`size`**, which return a number that is a measure of size. Currently **`length`** is used if is appropriate, otherwise **`size`** is attempted.

Exact Size

```
target should have: <number> on: #things
```

The **`things`** of **`target`** has a length/size of exactly **`number`**.

Lower Bound

```
target should have at least: <number> in: #things
```

The **`things`** of **`target`** has a length/size of no less than **`number`**.

Upper Bound

```
target should have at most: <number> in: #things
```

The **`things`** of **`target`** has a length/size of no more than **`number`**.

Copyright 2006 David R. Astels

Mock API

Sspec contains a full featured Mock Objects framework.

Creating a mock

```
mock := Mock named: <name>
```

This creates a new mock with the given **name** (a string) and registers it. When the specification finishes, all registered mocks are verified.

```
mock := mock named: <name> withOptions: <options>
```

As above, but allows you to specific options to tweak the mock's behaviour. The **options** argument is a hash. Currently the only supported option is**#nullObject**. Setting this to true instructs the mock to ignore (quietly consume) any messages it hasn't been told to expect. I.e.:

```
mock := Mock named: "blah" wuthOptions: #(nullObject)
```

Expecting Messages

```
mock shouldReceive: #selector
```

The **selector** argument is a symbol that is the name of a message that you want the mock to expect.

Arbitrary Message Handling

You can supply a block to a message expectation. When the message is received by the mock, the block is evaluated, and passed any arguments. The result is the return value of the message. For example:

```
(mock shouldReceive:#randomCall)
      andDo: [:a | a should be true]
```

This allows arbitrary argument validation and result computation. It's handy and kind of cool to be able to do this, but I advise against it. Mocks should not be functional. they should be completely declarative. That said, it's sometimes useful to give them some minimal behaviour.

Expecting Arguments

```
(mock shouldReceive: #msg) with: <arg>
```

for example:

```
(mock should receive: #msg) with: 1
```

There are forms supporting up to 4 arguments, as well as a form supporting an array of arguments:

```
(mock shouldReceive: #msg) with: <arg1> with: <arg2>
(mock shouldReceive: #msg) with: <arg1> with: <arg2> with: <arg3>
(mock shouldReceive: #msg) with: <arg1> with: <arg2> with: <arg3> with: <arg4>
(mock shouldReceive: #msg) withAll: <args>
```

There are two special forms as well:

```
(mock shouldReceive: #msg) withNoArgs
```

The message (**`msg`**) is expected to be passed no arguments.

```
(mock shouldReceive: #msg) withAnyArgs
```

Any arguments (and any number of arguments) are to be accepted. This includes cases where no arguments are provided. *This is the default when no **`with`** clause is specified.* Even so, sometimes you want to be explicit about it.

Argument Constraints

Constraints can be placed on individual arguments which are looser than value equivalence (as above).

#anything

accepts any value of any type for this argument, e.g.:

```
(mock shouldReceive: #msg) with: 1 with: #anything
```

#numeric

accepts any numeric value for this argument, e.g.:

```
(mock shouldReceive: #msg) with: 1 with: #numeric
```

#boolean

accepts any boolean value for this argument, e.g.:

```
(mock shouldReceive: #msg) with: 1 with: #boolean
```

#string

accepts any string value for this argument, e.g.:

```
(mock shouldReceive: #msg) with: 1 with: #string
```

duck typing

accepts any object that responds to all of the prescribed selectors (1 or more)

```
DuckTypeArgConstraint with: #size
```

There are forms with up to 4 **`with:`** arguments as well as **`withAll:`** form that takes a collection of selectors

```
(mock shouldReceive: #randomCall:)
       anyNumberOfTimes;
       with: (DuckTypeArgConstraint with: #size with: #includes:).

(mock shouldReceive: #randomCall:)
              with: (DuckTypeArgConstraint withAll: #(#size  #do:  #collect:
                                                      #select: #detect:)).
```

Receive Counts

never

An exception is raised if the message is ever received.

```
(mock shouldReceive: #msg) never
```

anyNumberOfTimes

The message can be received 0 or more times.

```
(mock shouldReceive: #msg) anyNumberOfTimes
```

once

An exception is raised if the message is never received, or it is received more than once.

```
(mock shouldReceive: #msg) once
```

twice

An exception is raised if the message is received anything but two times.

```
(mock shouldReceive: #msg) twice
```

exactly:

An exception is raised if the message is received anything but **n** times.

```
(mock shouldReceive: #msg) exactly: n times
```

atLeastOnce

An exception is raised if the message is never received.

```
(mock shouldReceive: #msg) atLeastOnce
```

atLeastTwice

An exception is raised if the message is never received or is received only once.

```
(mock shouldReceive: #msg) atLeastTwice
```

atLeast:

An exception is raised if the message is received fewer than **n** times.

```
(mock shouldReceive: #msg) atLeast: n times
```

atMost:

An exception is raised if the message is received more than **n** times.

```
(mock shouldReceive: #msg) atMost: n times
```

Return Values

Single return value

Each time the expected message is received, **value** will be returned as the result.

Copyright 2006 David R. Astels

```
(mock shouldReceive: #msg) andReturn: <value>
```

Consecutive return values

When the expected message is received, val_i will be returned as the result for the ith reception of the message. After the message has been received **n** times, val_n is returned for all subsequent receives.

```
(mock shouldReceive: #msg) andReturnConsecutively: <value_1 .. value_n>
```

Computed return value

When the expected message is received, the result of evaluating the supplied block will be returned as the result. The block is passed any arguments passed as arguments of the message.

```
(mock shouldReceive: #msg) andReturn: […]
```

This capability can be used to compute return values based on the arguments. For example:

```
(mock shouldReceive: #msg)
      once;
      with: #(numeric numeric);
      andReturn: [:a :b| a + b]
```

Raising and Throwing

These instruct the mock to raise an exception or throw a symbol, respectively, instead of returning a value.

```
(mock shouldReceive: #msg) andRaise: <exception>
```

Ordering

There are times when you want to specify the order of messages sent to a mock.

It shouldn't be the case very often, but it can be handy at times.

Labeling expectations as being ordered is done by the **`ordered`** call:

```
(mock shouldReceive: #flip) once; ordered.
(mock shouldReceive: #flop) once; ordered.
```

If the send of **`flop`** is seen before **`flip`** the specification will fail.

Of course, chains of ordered expectations can be set up:

```
(mock shouldReceive: #one) ordered.
(mock shouldReceive: #two) ordered.
(mock shouldReceive: #three) ordered.
```

The expected order is the order in which the expectations are declared.

Order-independant expectations can be set anywhere in the expectation sequence, in any order. Only the order of expectations tagged with the **`ordered`** call is significant. Likewise, calls to order-

Copyright 2006 David R. Astels

independant methods can be made in any order, even interspersed with calls to order-dependant methods. For example:

```
mock shouldReceive: #zero.
(mock shouldReceive: #one) ordered.
(mock shouldReceive: #two) ordered.
mock shouldReceive: #onePointFive.

mock proxy
       one;
       onePointFive;
       zero;
       two
```

Using a Mock

Sending proxy to a mock answers a proxy object that *fronts* for the mock in the system under test. It can be passed into the system and used. Interactions with it are trapped and checked against the expectation that have been set.

```
mock proxy
```

Copyright 2006 David R. Astels

Executing Expectations

There are two ways to execute sSpec specifications:

1. using the TextSpecRunner, and

2. using the Refactoring Browser integration.

Both will be examined in turn.

TextSpecRunner

This approach allows the most flexibility, control, and reporting.

There are two choices to make when creating a **`TextSpecRunner`** (actually the cross product of 2 ways):

1. the format of the output, terse or verbose, and

2. where the output goes, **`Transcript`** or a supplied stream.

Output from the spec run can be in one of two forms. Terse provides very basic output, for example:

```
X.
```

Where a '.' indicates a specification that is met, and an 'X' indicates a failing specification. Running in verbose mode provide a list of contexts and specification are being executed, noting which failed:

```
sample context with failing spec
        should fail (FAILED 1)
        should pass
```

In both cases, this output is followed by a summary of how many specs were tun, how many failed, and how long the run took:

```
2 specs, 1 failures
0.208 seconds
```

Typically, a terse run will go much faster. For example, the above run in terse mode reported:

```
2 specs, 1 failures
0.1 seconds
```

Finally, as well in both cases, any failure will result in a summary of the failure, numbered in order of appearance. These numbers are noted in the verbose output with the failure indication, e.,g.:

```
1) ExpectationNotMetError in 'sample context with failing spec should fail'
5 should be even
SSpec.SampleContextWithFailingSpec>>shouldFail
```

There are four pieces of useful information here:

1. The exception that is responsible for the failure, e.g. **`ExpectationNotMetError`**

Copyright 2006 David R. Astels

2. The context & spec, identified in a prosy way. This is a good reason for carefully choosing context & spec names… they shoudl read well together and make sense. E.g. `sample context with failing spec should fail`

3. A message specific to the exact failure, e.g. `5 should be even`

4. The class/method of the failing specification, e.g. `SSpec.SampleContextWithFailingSpec >> shouldFail`

RefactoringBrowser Integration

sSpec is integrated into the Refactoring Browser (which is VW's standard class browser) in exactly the same way sUnit is. When there are specification in the context of the rightmost selected item in the navigation lists, a panel is displayed below the code edit pane:

Not run: 6 specs | Run Failures | List Failures | Profile | Debug | Run

On the left is the current status (Not run) and the number of spec in scope (6 specs). On the right are the command buttons, in order right to left:

Run: Runs all specs in scope. If all pass, the status areas turns green:

Passed: 2 run, 0 failed | Run Failures | List Failures | Profile | Debug | Run

If there are failing specs, the status area turns red:

Failed: 6 run, 3 failed | Run Failures | List Failures | Profile | Debug | Run

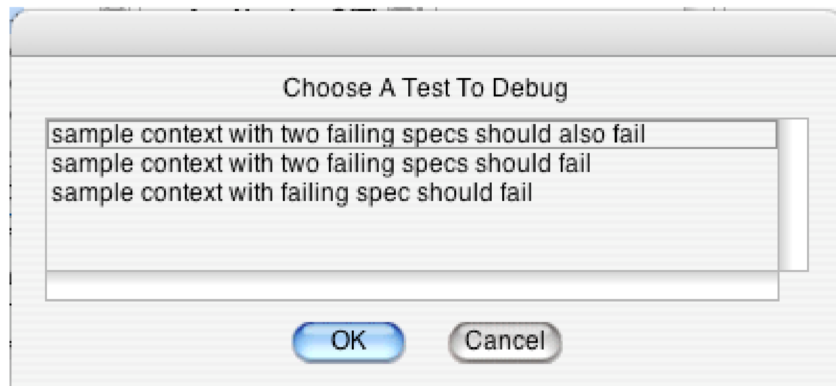
In both cases, the number of spec run, and the number that failed are displayed.

Debug: Runs all specs in scope, but failures pop up a notifier allowing you to debug the failures. Getting a spec to pass will continue the run, stopping at the next failure, if any.

Profile: Enabled if AT Profiling is installed. Runs all specs in scope with the profiler active.

List Failures: Enabled only after a run has produced failures. It pops up a list of failing specs:

Copyright 2006 David R. Astels



Selecting one of these failing tests will run it, opening a notifier on the offending exception, allowing you to debug that spec.

Run Failures: Enabled only after a run has produced failures. It runs only those specs that just previously failed, rather than rerunning all those in scope.