Testing in Pharo

Stéphane Ducasse, Guillermo Polito and Juan-Pablo Sandoval

August 2, 2022

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations v

1 Introduction 1
1.1 Outline . . . 1
1.2 About SUnit . . . 2
1.3 About typographic conventions . . . 2
1.4 About this book . . . 3
1.5 Getting started . . . 3

2 Two minutes of theory 5
2.1 Automated tests . . . 5
2.2 Why testing is important . . . 6
2.3 What makes a good test? . . . 7
2.4 A piece of advice on testing . . . 8
2.5 Pharo testing rules as a conclusion . . . 9

3 SUnit by example 11
3.1 Step 1: Create the test class . . . 11
3.2 Step 2: A first test . . . 12
3.3 Step 3: Run the tests . . . 12
3.4 Step 4: Another test . . . 13
3.5 Step 5: Factoring out context . . . 13
3.6 Step 6: Initialize the test context . . . 14
3.7 Step 7: Debugging a test . . . 15
3.8 Step 8: Interpret the results . . . 15
3.9 Conclusion . . . 16

4 SUnit: The framework 17
4.1 Understanding the framework . . . 17
4.2 During test execution . . . 17
4.3 The framework in a nutshell . . . 18
4.4 Test states . . . 20
4.5 Glossary . . . 21
4.6 Chapter summary . . . 22

5 The SUnit cookbook **23**
5.1 Parameterized tests . . . 23
5.2 Matrix: a more advanced case . . . 24
5.3 Classes vs. objects as parameters . . . 25
5.4 Other assertions . . . 26
5.5 Running tests . . . 27
5.6 Advanced features of SUnit . . . 28
5.7 Test resources . . . 29
5.8 Customising tests . . . 32
5.9 Inheriting TestCase . . . 32
5.10 Conclusion . . . 33

6 Getting more with continuous integration **35**
6.1 Github Actions . . . 35
6.2 Travis . . . 37
6.3 With Travis . . . 38
6.4 With Appveyor . . . 38
6.5 Assessing test coverage . . . 39
6.6 Conclusion . . . 39

7 UI testing **41**
7.1 Testing Spec . . . 41
7.2 Spec user perspective . . . 42

8 Testing web applications with Parasol **47**
8.1 Getting started . . . 47
8.2 First steps with Selenium . . . 48
8.3 Locating elements with Parasol: The basics . . . 50
8.4 Finding elements using XPath . . . 53
8.5 Finding multiple elements . . . 53
8.6 Interacting with the elements . . . 54
8.7 Parasol in action . . . 55
8.8 Testing the page title . . . 56
8.9 Testing displayed information . . . 57
8.10 Testing interactions . . . 59
8.11 Conclusion . . . 64

9 MockObject and Teachable: Two super simple mocking approaches **65**
9.1 About MockObject design . . . 65
9.2 MockObject . . . 66
9.3 Stubs, Fakes, and Mocks . . . 66
9.4 Example . . . 67
9.5 About matching arguments . . . 68
9.6 Teachable . . . 69
9.7 Conclusion . . . 70

10 Mocketry **71**
10.1 How to install Mocketry? . . . 71
10.2 About Test-Driven Development and Mock Objects . . . 71
10.3 TDD is not just about testing . . . 72
10.4 How does outside-in TDD work? . . . 72
10.5 Getting started . . . 73
10.6 A simple shopping cart . . . 73
10.7 Writing our first test . . . 73
10.8 One product, no discount . . . 74
10.9 Many products, no discount . . . 76
10.10 Removing a product . . . 77
10.11 Products with discount price . . . 78
10.12 Reusing the cart for different sales . . . 79
10.13 Testing errors during checkout . . . 79
10.14 What's next . . . 80
10.15 Features overview . . . 81
10.16 Stubbing classes . . . 82
10.17 Expectations for set of objects . . . 83
10.18 Stub message sends with arguments . . . 84
10.19 Unexpected messages: Automocks . . . 86
10.20 Stub group of message sends . . . 87
10.21 Verify message sends . . . 87
10.22 Verify message sends with arguments . . . 89
10.23 Capture message arguments . . . 89
10.24 Verify message sends count . . . 90
10.25 Verify message send result . . . 91
10.26 Verify group of message sends . . . 91
10.27 Verify all expectations . . . 92
10.28 Conclusion . . . 92

11 Object validation with StateSpecs **93**
11.1 How to load StateSpecs . . . 93
11.2 Basic . . . 94
11.3 Two little DSLs . . . 94
11.4 Should expressions . . . 95
11.5 About equality for specification . . . 96
11.6 Specific case of String and ByteArray . . . 97
11.7 Floats . . . 97
11.8 Specification of class relationship . . . 98
11.9 Collection Specifications . . . 98
11.10 String Specifications . . . 99
11.11 Raising exception . . . 100
11.12 The fail message . . . 100
11.13 Predicate syntax . . . 101
11.14 Conclusion . . . 102

12 Performance testing with SMark **103**
12.1 Installing SMark 103
12.2 Measuring execution time 104
12.3 A first benchmark in SMark 105
12.4 Setup and teardown 105
12.5 SMark benchmark runners 106
12.6 Benchmark suites 107
12.7 Result reports 107
12.8 Conclusion 108

13 Miscellaneous **109**
13.1 Executable comments 109
13.2 Executable comment example 109
13.3 Supporting examples with assertions 110

Illustrations

3-1 An Example Set Test class 11
3-2 Testing includes 12
3-3 Running SUnit tests from the System Browser. 13
3-4 Testing occurrences 13
3-5 An Example Set Test class 14
3-6 Setting up a fixture 14
3-7 Testing includes 14
3-8 Testing occurrences 15
3-9 Testing removal 15
3-10 Introducing a bug in a test 15

4-1 During `run` execution. 18
4-2 setUp and teardDown in action. 18
4-3 The four classes representing the core of SUnit. 19
4-4 Testing removal of nonexisting element. 20
4-5 Testing removal of an nonexistent element. 20
4-6 An example of skipped test. 21
4-7 A not so nice example of skipped test. 21

5-1 Testing error raising 26
5-2 `setUp` and `tearDown` in action. 30
5-3 An example of a TestResource subclass 31

7-1 Spec Architecture. 42
7-2 A Spec application. 43

8-1 The Mercury Tours WebSite. 57
8-2 Registration From 60
8-3 Filling fields using Parasol. 62
8-4 Registration successful view. 63

CHAPTER 1

Introduction

Pharo is the unique enviroment in which you can start to write a test, execute the test and from the debugger grow your program. We coined this powerful technique Xtreme Test Driven Design. This is powerful because it gives you a unique situation where you are in close contact with the specific state of your program. There you can write your code interacting with a live organism (the set of objects that are currently executing your program). It gives you a unique opportunity to query and interact deeply with your objects.

Now describing Xtreme Test Driven Design feels like describing swimming among the fishes in a scuba diving session. It is difficult to transmit the sensation.

Xtreme TDD takes its root in testing and this is why in this book we will describe unit testing in Pharo. We will not explain Xtreme TDD more than that and if you want to try we suggest you to follow the video available at http://rmod-pharo-mooc.lille.inria.fr/MOOC/PharoMOOC-Videos/EN/Week2/W2-Redo-EN-final.mp4

1.1 Outline

In this book we will present how you to test and develop testing strategies in Pharo. We will present the SUnit framework, but also DrTests a plugin architecture to propose extensions and analyses to improve your tests. We show that contrary to what is commonly believed, testing UI is possible and that you can take advantage of it. We present how to connect your github repository to take advantage of integration services. We also present how to test web applications. We also show some mocking approaches and show

that benchmarks can be also supported even if they are not tests per se. We also describe the framework and its implementation.

1.2 About SUnit

Testing is getting more and more exposure. What is interesting to see is that Pharo inherits SUnit from its ancestors (Smalltalk) and it is worth knowing that most of the Unit frameworks are inheriting from the Smalltalk original version developed by K. Beck.

SUnit is a minimal yet powerful framework that supports the creation and deployment of tests. As might be guessed from its name, the design of SUnit focuses on *Unit Tests*, but in fact it can be used for integration tests and functional tests as well. SUnit was originally developed by Kent Beck and subsequently extended by Joseph Pelrine and others to incorporate the notion of a resource. Note that the version documented in this chapter and used in Pharo is a modified version of SUnit3.3.

1.3 About typographic conventions

In this book we use the following conventions. We use the new fluid class syntax introduced in Pharo 9. Fluid means that we use a cascade to define the class elements and omit the empty ones.

When you are used to define a class as follows:

```
TestCase subclass: #MyExampleSetTest
  instanceVariableNames: 'x'
  classVariableNames: ''
  package: 'MySetTest'
```

We use the following fluid definition

```
TestCase << #MyExampleSetTest
  slots: { #x };
  package: 'MySetTest'
```

Another point is that we always prefix method source code with the class of the method. The book shows it as:

```
MyExampleSetTest >> testIncludes
    | full empty |
    full := Set with: 5 with: 6.
    empty := Set new.
    self assert: (full includes: 5).
    self assert: (full includes: 6).
    self assert: (empty includes: 5) not
```

And it you want to type it into Pharo you should type the following in the corresponding class.

```
testIncludes
    | full empty |
    full := Set with: 5 with: 6.
    empty := Set new.
    self assert: (full includes: 5).
    self assert: (full includes: 6).
    self assert: (empty includes: 5) not
```

1.4 About this book

In Pharo by Example current revision (9), we decided to go to the essential of SUnit and removed parts that were too detailed and long. This gave us the idea that a ”Testing in Pharo” book was missing. Therefore instead of losing the parts that we removed, they grew in a new book. Therefore a part of the text of this book was written originally in Pharo by Example by Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker, it is mainly the motivation, description of SUnit and SUnit implementation. We thank them for this material that we revised. Our ultimate goal is to revisit the implementation of SUnit and to keep this book up to date.

Acknowledgments. We want to thank Jimmie Houchin for improving the english of this book.

1.5 Getting started

We areally encourage you to experience test driven design and in particular Xtreme test driven design. Yes writing tests looks like an extra effort but it is really worth. Tests force you to design API and they give you the insurance that you will be able to change your code without fear to break your code and do not get notified about it.

CHAPTER 2

Two minutes of theory

The interest in testing and Test Driven Development is not limited to Pharo. Automated testing has become a hallmark of the *Agile software development* movement, and any software developer concerned with improving software quality would do well to adopt it. Indeed, developers in many languages have come to appreciate the power of unit testing, and versions of *xUnit* now exist for every programming language.

What you will discover while programming in Pharo is that Pharo supports **Xtreme TDD**: you define your tests first as in TDD but you execute your program which breaks and you code in the debugger. This is simply such a productivity boost that we urge you to try. We are just addicted to it. In Pharo this is super cool to define a test, run the code and code in the debugger: this is such a speed up and positive energy.

Now lets us go for two minutes theory around Unit tests, TDD and other.

2.1 Automated tests

Neither testing, nor the building of test suites, is new. By now, everybody knows that tests are a good way to catch errors. eXtreme Programming, by making testing a core practice and by emphasizing *automated* tests, has helped to make testing productive and fun, rather than a chore that programmers dislike. The Pharo community has a long tradition of testing because of the incremental style of development supported by its programming environment.

Favor executable tests

During incremental development sessions in Pharo, the old fashioned programmer would write code snippets in a playground as soon as a method was finished. Sometimes a test would be incorporated as a comment at the head of the method that it exercised, or tests that needed some set up would be included as example methods in the class. The problem with these practices is that tests in a playground are not available to other programmers who modify the code. Comments and example methods are better in this respect, but there is still no easy way to keep track of them and to run them automatically. Tests that are not **systematically** run do not help you to find bugs! Moreover, an example method does not inform the reader of the expected result: you can run the example and see the (perhaps surprising) result, but you will not know if the observed behaviour is correct.

Favor TDD and XtremeTDD

With Test driven development, you write a little automated tests, execute it to make sure that it fails. Then in Pharo, you use the test execution and failure to create methods inside the debugger. Once your test passes, you rerun all your tests and commit if they are all passing.

SUnit is valuable to suport this scenario. It allows us to write tests that are automated and self-checking: the test itself defines what the correct result should be. It also helps us to organize tests into groups, to describe the context in which the tests must run, and to run a group of tests automatically. In less than two minutes you can write tests using SUnit, so instead of writing small code snippets in a playgound, we encourage you to use SUnit and get all the advantages of stored and automatically executable tests.

2.2 Why testing is important

Unfortunately, many developers believe that tests are a waste of their time. After all, *they* do not write bugs, only *other* programmers do that. Most of us have said, at some time or other: *I would write tests if I had more time.* If you never write a bug, and if your code will never be changed in the future, then indeed tests are a waste of your time. However, this most likely also means that your application is trivial, or that it is not used by you or anyone else. Think of tests as an investment for the future: having a suite of tests is quite useful now, but it will be *extremely* useful when your application, or the environment in which it runs, changes in the future.

Tests play several roles. First, they provide documentation of the functionality that they cover. This documentation is active: watching the tests pass tells you that the documentation is up to date. Second, tests help developers to confirm that some changes that they have just made to a package

have not broken anything else in the system, and to find the parts that break when that confidence turns out to be misplaced. Finally, writing tests during, or even before, programming forces you to think about the functionality that you want to design, *and how it should appear to the client code*, rather than about how to implement it.

By writing the tests first, i.e., before the code, you are compelled to state the context in which your functionality will run, the way it will interact with the client code, and the expected results. Your code will improve. Try it.

We cannot test all aspects of any realistic application. Covering a complete application is simply impossible and should not be the goal of testing. Even with a good test suite some bugs will still creep into the application, where they can lay dormant waiting for an opportunity to damage your system. If you find that this has happened, take advantage of it! As soon as you uncover the bug, write a test that exposes it, run the test, and watch it fail. Now you can start to fix the bug: the test will tell you when you are done.

2.3 What makes a good test?

Writing good tests is a skill that can be learned by practicing. Let us look at the properties that tests should have to get the maximum benefit.

Tests should be repeatable. You should be able to run a test as often as you want, and always get the same answer.

Tests should run without human intervention. You should be able to run them unattended.

Tests should tell a story. Each test should cover one aspect of a piece of code. A test should act as a scenario that you or someone else can read to understand a piece of functionality.

Tests should have a change frequency lower than that of the functionality they cover. You do not want to have to change all your tests every time you modify your application. One way to achieve this is to write tests based on the public interfaces of the class that you are testing. It is OK to write a test for a private *helper* method if you feel that the method is complicated enough to need the test, but you should be aware that such a test may have to be changed, or thrown away entirely, when you think of a better implementation.

One consequence of such properties is that the number of tests should be somewhat proportional to the number of functions to be tested: changing one aspect of the system should not break all the tests but only a limited number. This is important because having 100 tests fail should send a much stronger message than having 10 tests fail. However, it is not always possible to achieve this ideal: in particular, if a change breaks the initialization of an object, or the set-up of a test, it is likely to cause all of the tests to fail.

Several software development methodologies such as *eXtreme Programming* and Test-Driven Development (TDD) advocate writing tests before writing code. This may seem to go against our deep instincts as software developers. All we can say is: go ahead and try it. We have found that writing the tests before the code helps us to know what we want to code, helps us know when we are done, and helps us conceptualize the functionality of a class and to design its interface. Moreover, test-first development gives us the courage to go fast, because we are not afraid that we will forget something important.

Writing tests is not difficult in itself. Choosing *what* to test is much more difficult. The pragmatic programmers offer the "right-BICEP" principle. It stands for:

- Right: Are the results right?
- B: Are all the boundary conditions correct?
- I: Can you check inverse relationships?
- C: Can you cross-check results using other means?
- E: Can you force error conditions to happen?
- P: Are performance characteristics within bounds?

2.4 A piece of advice on testing

While the mechanics of testing are easy, writing good tests is not. Here is some advice on how to design tests.

Self-contained tests

You do not want to have to change your tests each time you change your code, so try to write the tests so that they are self-contained. This can be difficult, but pays off in the long term. Writing tests against stable interfaces supports this effort.

Do not over-test

Try to build your tests so that they do not overlap. It is annoying to have many tests covering the same functionality, because one bug in the code will then break many tests at the same time. This is covered by Black's rule, below.

Feathers' rules for unit tests

Michael Feathers, an agile process consultant and author, writes:

A test is not a unit test if: it talks to the database, it communicates across the network, it touches the file system, it can't run at the same time as any of your other

unit tests, or you have to do special things to your environment (such as editing config files) to run it. Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes. Never get yourself into a situation where you don't want to run your unit test suite because it takes too long.

Unit tests vs. Acceptance tests

Unit tests capture one piece of functionality, and as such make it easier to identify bugs in that functionality. As far as possible try to have unit tests for each method that could possibly fail, and group them per class. However, for certain deeply recursive or complex setup situations, it is easier to write tests that represent a scenario in the larger application. These are called acceptance tests (or integration tests, or functional tests).

Tests that break Feathers' rules may make good acceptance tests. Group acceptance tests according to the functionality that they test. For example, if you are writing a compiler, you might write acceptance tests that make assertions about the code generated for each possible source language statement. Such tests might exercise many classes, and might take a long time to run because they touch the file system. You can write them using SUnit, but you won't want to run them each time you make a small change, so they should be separated from the true unit tests.

Black's rule of testing

For every test in the system, you should be able to identify some property for which the test increases your confidence. It's obvious that there should be no important property that you are not testing. This rule states the less obvious fact that there should be no test that does not add value to the system by increasing your confidence that a useful property holds. For example, several tests of the same property do no good. In fact, they do harm in two ways. First, they make it harder to infer the behaviour of the class by reading the tests. Second, because one bug in the code might then break many tests, they make it harder to estimate how many bugs remain in the code. So, have a property in mind when you write a test.

2.5 Pharo testing rules as a conclusion

We can argue over and over why tests are important. The only real way to understand for real their values is by experience.

The Pharo core development has three basic rules about testing. Here they are:

Important A test that is not automated is not a test.

Important Everything that is not tested does not exist.

Important Everything that is not tested will break.

With this in mind, we urge ourselves to write tests. Sometimes we are sloppy and lazy but most of the time we push ourselves to stay bold. We encourage you to do the same.

CHAPTER 3

SUnit by example

In this chapter we present a small example showing how simple it is to use SUnit. Before going into the details of SUnit (see next Chapter), we will show a step by step example. We use an example that tests the class `Set`. Try entering the code as we go along. We will create a test i.e., create a context (also called a fixture), execute a stimulus and verify that some assertions are working.

If you already read the SUnit chapter on Pharo by Example book you can skip this chapter since the contents are the same.

3.1 Step 1: Create the test class

First you should create a new subclass of `TestCase` called `MyExampleSetTest` that looks like this:

The class `MyExampleSetTest` groups all the tests related to the class `Set`. As we will show latter, we will use it to define the context in which the tests will run.

The name of the class is not critical, but by convention it should end in `Test`. If you define a class called `Pattern` and call the corresponding test class `PatternTest`, the two classes will be alphabetized together in the browser

Listing 3-1 An Example Set Test class

```
TestCase subclass: #MyExampleSetTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'MySetTest'
```

Listing 3-2 Testing includes

```
MyExampleSetTest >> testIncludes
  | full empty |
  full := Set with: 5 with: 6.
  empty := Set new.
  self assert: (full includes: 5).
  self assert: (full includes: 6).
  self assert: (empty includes: 5) not
```

(assuming that they are in the same package). It *is critical* that your class is a subclass of `TestCase`.

Must I subclass TestCase?

In JUnit you can build a TestSuite from an arbitrary class containing `test*` methods. In SUnit you can do the same but you will then have to create a suite by hand and your class will have to implement all the essential `TestCase` methods like `assert:`. We recommend, however, that you not try to do this. The framework is there: use it.

3.2 Step 2: A first test

We start by defining a method named `testIncludes`. Pay attention the `'test'` part is important. Each method represents one test. The names of the methods should start with the string `'test'` so that SUnit will collect them into test suites. Test methods take no arguments.

This method creates two sets one empty and one full. This is the context or fixture of the test. Second we perform some action on the test: here we execute the method `includes:`, and third we validate the output using assertion via the `assert:` message. The method `assert:` checks that the argument is a boolean true.

Define the following test methods. The first test, named `testIncludes`, tests the `includes:` method of `Set`. For example, the test says that sending the message `includes: 5` to a set containing 5 should return `true`.

3.3 Step 3: Run the tests

The easiest way to run the tests is directly from the browser. Press on the icon on the side of the class name, or on an individual test method. The test methods will be flagged depending on whether they pass or not (as shown in 3-3).

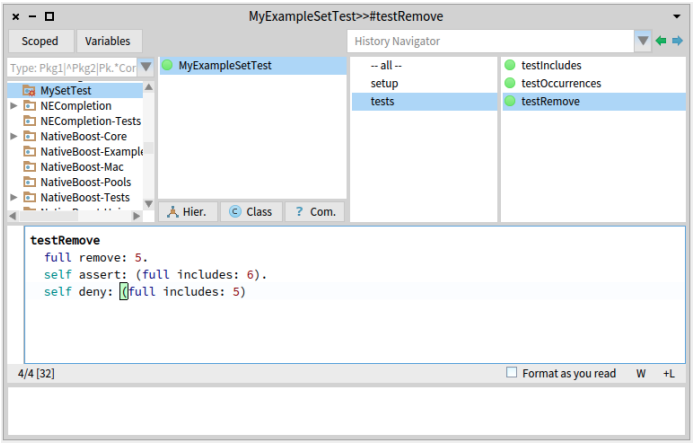

Figure 3-3 Running SUnit tests from the System Browser.

Listing 3-4 Testing occurrences

```
MyExampleSetTest >> testOccurrences
  | full empty |
  full := Set with: 5 with: 6.
  empty := Set new.
  self assert: (empty occurrencesOf: 0) equals: 0.
  self assert: (full occurrencesOf: 5) equals: 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) equals: 1
```

3.4 Step 4: Another test

Following the same pattern, here is a test verifying that the method `occurrencesOf:` works as expected. The second test, named `testOccurrences`, verifies that the number of occurrences of 5 in `full` set is equal to one, even if we add another element 5 to the set.

Note that we use the message `assert:equals:` and not `assert:` as in the first test. We could have used `assert:`. But the message `assert:equals:` is better since it reports the error in a much better way. When an assertion is failing, `assert:equals:` shows the expected value and the value received. While `assert:` just mentions that something is not true.

Now make sure that your test is passing too.

3.5 Step 5: Factoring out context

As you see in the two previous steps, we started to repeat the same context. This is not really nice, so we will factor the fixture out of the tests by defin-

Listing 3-5 An Example Set Test class

```
TestCase subclass: #MyExampleSetTest
  instanceVariableNames: 'full empty'
  classVariableNames: ''
  package: 'MySetTest'
```

Listing 3-6 Setting up a fixture

```
MyExampleSetTest >> setUp
  super setUp.
  empty := Set new.
  full := Set with: 5 with: 6
```

Listing 3-7 Testing includes

```
MyExampleSetTest >> testIncludes

  self assert: (full includes: 5).
  self assert: (full includes: 6).
  self assert: (empty includes: 5) not
```

ing instance variables in the class and a method `setUp` to initialize them.

A `TestCase` class defines the context in which the tests will run. We will add the two instance variables `full` and `empty` that we will use to represent a full and an empty set.

The next step is to initialize such instance variables.

3.6 Step 6: Initialize the test context

The message `TestCase >> setUp` defines the context in which the tests will run, a bit like an initialize method. `setUp` is invoked before the execution of each test method defined in the test class.

Define the `setUp` method as follows, to initialize the `empty` variable to refer to an empty set and the `full` variable to refer to a set containing two elements.

In testing jargon, the context is called the *fixture* for the test and the `setUp` method is responsible to initialize such fixture.

Updating existing tests

We change the two test methods to take advantage of the shared initialization. We remove the fixture code and obtain the following methods:

Clearly, these tests rely on the fact that the `setUp` method has already run.

Listing 3-8 Testing occurrences

```
MyExampleSetTest >> testOccurrences

  self assert: (empty occurrencesOf: 0) equals: 0.
  self assert: (full occurrencesOf: 5) equals: 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) equals: 1
```

Listing 3-9 Testing removal

```
MyExampleSetTest >> testRemove
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Listing 3-10 Introducing a bug in a test

```
MyExampleSetTest >> testRemove
  full remove: 5.
  self assert: (full includes: 7).
  self deny: (full includes: 5)
```

Taking advantage of setUp

We test that the set no longer contains the element 5 after we have removed it.

Note the use of the method `TestCase >> deny:` to assert something that should not be true. `aTest deny: anExpression` is equivalent to `aTest assert: anExpression not`, but is much more readable and expresses the intent more clearly.

3.7 Step 7: Debugging a test

Introduce a bug in `MyExampleSetTest >> testRemove` and run the tests again. For example, change `6` to `7`, as in:

The tests that did not pass (if any) are listed in the right-hand panes of the *Test Runner*. If you want to debug one, to see why it failed, just click on the name. Alternatively, you can execute one of the following expressions:

```
(MyExampleSetTest selector: #testRemove) debug

MyExampleSetTest debug: #testRemove
```

3.8 Step 8: Interpret the results

The method `assert:` is defined in the class `TestAsserter`. This is a superclass of `TestCase` and therefore all other `TestCase` subclasses and is respon-

sible for all kind of test result assertions. The `assert:` method expects a boolean argument, usually the value of a tested expression. When the argument is true, the test passes; when the argument is false, the test fails.

There are actually three possible outcomes of a test: *passing*, *failing*, and *raising an error*.

- **Passing**. The outcome that we hope for is that all of the assertions in the test are true, in which case the test passes. In the test runner, when all of the tests pass, the bar at the top turns green. However, there are two other ways that running a test can go wrong.
- **Failing**. The obvious way is that one of the assertions can be false, causing the test to *fail*.
- **Error**. The other possibility is that some kind of error occurs during the execution of the test, such as a *message not understood* error or an *index out of bounds* error. If an error occurs, the assertions in the test method may not have been executed at all, so we can't say that the test has failed; nevertheless, something is clearly wrong!

In the *test runner*, failing tests cause the bar at the top to turn yellow, and are listed in the middle pane on the right, whereas tests with errors cause the bar to turn red, and are listed in the bottom pane on the right.

Modify your tests to provoke both errors and failures.

3.9 Conclusion

- To maximize their potential, unit tests should be fast, repeatable, independent of any direct human interaction and cover a single unit of functionality.
- Tests for a class called `MyClass` belong in a class named `MyClassTest`, which should be introduced as a subclass of `TestCase`.
- Initialize your test data in a `setUp` method.
- Each test method should start with the word *test*.
- Use the `TestCase` methods `assert:`, `deny:` and others to make assertions.
- Run tests!

CHAPTER 4

SUnit: The framework

Now that you see that writing a test is easy, we will take the time to put in perspective the different aspects of SUnit. SUnit is a framework in the sense that it proposes an architecture that can be adapted. We will not cover the techniques to extend the framework but we will present the key classes and discuss some important points.

4.1 Understanding the framework

What we saw in the previous chapter is that a test method defines a test. Now there is a catch:

- A method (called a test method) of a subclass of the class `TestCase` represents a test.
- A `TestCase` subclass groups together all the tests sharing a similar context, called a *fixture*.
- A `setUp` method is run systematically before a test method is run and a `tearDown` after.
- The framework builds a `TestSuite` (a composite of tests) to execute the tests.

4.2 During test execution

Figure 4-1 shows the steps during the execution `run` of a test:

- An instance of `TestCase` is created.
- It creates an instance of `TestResult`.

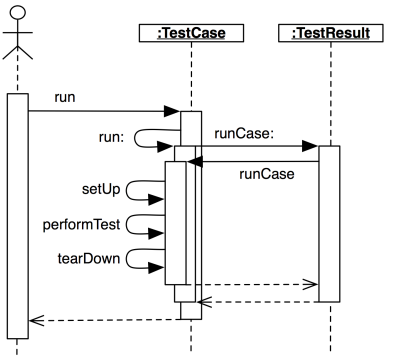

Figure 4-1 During `run` execution.

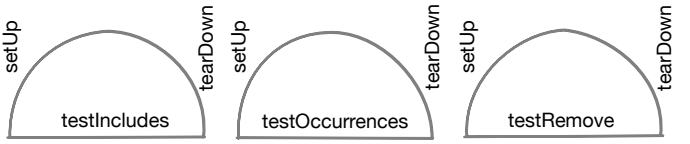

Figure 4-2 setUp and teardDown in action.

- This instance calls back the `TestCase` instance which executes
  - the test `setUp`
  - the test method
  - and finally the test `tearDown` method.

In fact the framework ensures that any test method is always surrounded by the execution of the `setUp` and `tearDown` methods. Figure 4-2 illustrates this point. It ensures that the fixture is always in a correct state and avoid dependencies between test execution.

4.3 The framework in a nutshell

SUnit consists of four main classes: `TestCase`, `TestSuite`, `TestResult`, and `TestResource`, as shown in Figure 4-3.

TestCase

`TestCase` is an abstract class that is designed to be subclassed. Each of its subclasses represents a **group** of tests that share a common context: such a group is called a **test suite**.

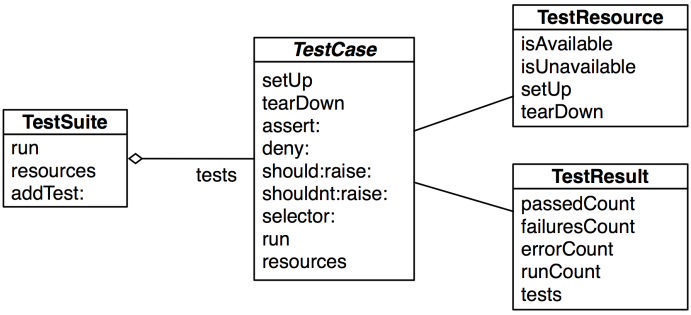

Figure 4-3 The four classes representing the core of SUnit.

Each test is run by creating a new instance of a subclass of `TestCase`, running `setUp` to initialize the test fixture, running the test method itself, and then sending the `tearDown` to clean up the test fixture.

The fixture is specified by instance variables of the subclass and by the specialization of the method `setUp`, which initializes those instance variables. Subclasses of `TestCase` can also override method `tearDown`, which is invoked after the execution of each test, and can be used to release any objects allocated during `setUp`.

TestSuite

Instances of the class `TestSuite` contain a collection of test cases. An instance of `TestSuite` contains tests, and other test suites. That is, a test suite contains sub-instances of `TestCase` and `TestSuite`. Both individual test cases and test suites understand the same protocol, so they can be treated in the same way (for example, both can be `run`). This is in fact an application of the Composite pattern in which `TestSuite` is the composite and the test cases are the leaves.

TestResult

The class `TestResult` represents the results of a `TestSuite` execution. It records the number of tests passed, the number of tests failed, and the number of errors signalled.

TestResources

The notion of a *test resource* represents a resource that is expensive to set-up but which can be used by a whole series of tests. A `TestResource` specifies a `setUp` method that is executed just once before a full suite of tests; this is in distinction to the `TestCase >> setUp` method, which is executed before

Listing 4-4 Testing removal of nonexisting element.

```
MyExampleSetTest >> testRemoveNonexistentElement
  empty remove: 5.
```

Listing 4-5 Testing removal of an nonexistent element.

```
MyExampleSetTest >> testRemoveNonexistentElement

  self should: [ empty remove: 5 ] raise: NotFound
```

each test. The class `TestResource` represents resources that can be shared among a test suite: Subclasses of `TestResource` are associated to test case classes and this means that all the suite of tests represented by the test case class (See Chapter 5).

4.4 Test states

As we mentioned earlier, the results of a test execution is mainly: passed (meaning that it succeeded), fails (that an assertion is not valid) or error (that an unexpected problem occurred).

Failures vs. Error

In fact this is really important to understand the difference between a failure and an error. A failure is something that you plan and you check whether is happened or not. When it happens your tests pass, else it fails.

Remove an element that is not in a set from that set raises an error. If we write the following test, the test will fail.

Now using the message `should:raise:` we make sure that we check that the expression effectively raises an error. It means that our test can either pass (i.e., it means that the test fulfil our requirement here that it should raise an error when removing an unknown element) or fail. We planned such behavior. It can fail or pass.

For an error this is different, an error occurs in an unplanned manner. The execution of the first version of `testRemoveNonexistentElement` would lead to an error.

Expected Failures and Skip

In addition a test can be in two other states: skipped and expectedFailures. Such states are handy during development.

The idea is that if you write a complex test and suddenly you realize that you will have more work than expected to make your test pass, you can mark it as skipped.

Listing 4-6 An example of skipped test.

```
MyExampleSetTest >> testToBeRevisitedLater

  self skip.
  ...
```

Listing 4-7 A not so nice example of skipped test.

```
MyExampleSetTest >> testToBeRevisitedLater

  true ifTrue: [^ self ].
  ...
```

In fact it is much better to use the message `skip` than to use a guard. Why because the framework can report clearly that your test is skipped. Using a guard as below is the best way to confuse yourself, because the framework will return that the test passes while it does NOT.

You can also tag your test with the `<expectedFailure>` pragma to indicate to the framework that you expect your test to fail. This is be taken into account by the framework when running and reporting all your tests. Hence you will be aware that some of your tests did not run correctly. Here is an example from Pharo.

```
testFromClassWhichTraitIsExtendedButNotItself
  "I'm tagging this expected failure because I'm not sure now if it
    is appropriate to keep
   or not with the removal of the 'traits-as-multiple-inheritance'
    stuff."
  <expectedFailure>

  self queryFromScope: ClyClassScope of: ClyClassWithTraits.
  self assert: resultItems size equals: 0
```

4.5 Glossary

- **Test case.** A test case is a single test. It defines a context, a stimulus and at the minimum one assertion. It is described as a method of a TestCase subclass starting with **test**.
- **Test suite.** A test suite (Composite design pattern) is a group of tests. The tests belonging to a test suite do not have to be from the same class. In practice the test runner builds a test suite for all the test cases defined in a class and executes them one by one. Instances of the class `TestSuite` contain a collection of test cases. An instance of `TestSuite` contains tests, and other test suites. That is, a test suite contains sub-instances of `TestCase` and `TestSuite`.

- **Fixture.** A fixture is a context in which a stimulus will be executed and assertions verified. To share a fixture amongst multiple tests you can define the `setUp` method. It is automatically invoked before any test method execution.
- **Failure.** A failure is the situation where you planned something and checked it and it is not correct.
- **Error.** An error is the situation where your test is not working but it is not covered by an assertion. For example, a message not understood is sent and it was unexpected.

4.6 Chapter summary

This chapter we gave an overview of the core of the SUnit framework by presenting the classes `TestCase`, `TestResult`, `TestSuite` and `TestResources`. We described the setUp/tearDown logic and the different states a test can be in.

CHAPTER 5

The SUnit cookbook

This chapter will give you more details on how to use SUnit. If you have used another testing framework such as JUnit, much of this will be familiar, since all these frameworks have their roots in SUnit. Normally you will use SUnit's GUI to run tests, but there are situations where you may not want to use it. But let us start with a powerful feature of SUnit: parametrized tests.

5.1 Parameterized tests

Since Pharo 7.0 you can express parameterized tests. Parametrized tests are tests that can be executed on multiple configurations: your tests will run on different contexts that you can specify basically as test arguments. Parametrized tests are really powerful when you want to check whether two implementations pass the same set of tests.

To declare a parameterized test you have to:

- define your test case class as a subclass of `ParametrizedTestCase` instead of `TestCase`. This class should define accessors that will be used to configure the tests.
- define a *class* method named `testParameters` which specifies the actual parameters.

A simple example first

Here is an example taken from the Enlumineur project which is a pretty printer for Pharo code. Using paramterized tests let us know whether two different pretty printers produce the same outputs.

We define the class `BIEnlumineurTest`. It has different parameters expressed as instance variables such as `formatterClass` and `contextClass`.

```
ParametrizedTestCase subclass: #BIEnlumineurTest
  instanceVariableNames: 'configurationSelector formatterClass
    contextClass'
  classVariableNames: ''
  package: 'Enlumineur-Tests'
```

This class should define accessors for its parameters, here for `formatterClass` and `contextClass`. The tests should use the tests instance variables and should not refer directly to the classes held by the instance variables. Else this would shortcut the idea of a parametrized test itself.

Then we define the class method `testParameters` as follows.

```
BIEnlumineurTest class >> testParameters
  ^ ParametrizedTestMatrix new
    addCase: { (#formatterClass -> BIEnlumineurPrettyPrinter) .
    (#contextClass -> BIEnlumineurContext) };
    yourself
```

Now the framework will run the test using the parameters we mentioned. To add a new variation we just have to add a case using the `addCase:` message.

Controlling configuration

The following example generates 2 cases. Exactly the 2 cases listed in `testParameters` method. The values for `number1` and `number2` will be set and the test will be executed.

```
PaSelectedCasesExampleTest class >> testParameters

  ^ ParametrizedTestMatrix new
    addCase: { #number1 -> 2. #number2 -> 1.0. #result -> 3 };
    addCase: { #number1 -> (2/3). #number2 -> (1/3). #result -> 1 };
    yourself
```

5.2 Matrix: a more advanced case

Sometimes you do not want to enumerate all the combinations by hand. In that case you can use a matrix and specify all the possible values of a parameter. The class `PaSimpleMatrixExampleTest` contains some examples.

The following test executes 27 different cases. All the combinations in the matrix are executed, i.e. item1 values will be enumerated, and for each ones, all the values of the other parameters will be also enumerated. This way all possible combinations are generated and tests run for each of them.

```
PaSimpleMatrixExampleTest class >> testParameters

  ^ ParametrizedTestMatrix new
    forSelector: #item1 addOptions: { 1. 'a'. $c };
    forSelector: #item2 addOptions: { 2. 'b'. $d };
    forSelector: #collectionClass addOptions: { Set. Bag.
    OrderedCollection }
```

The test matrix generates using a cartesian product the configurations or a set of well known cases. Each option is constituted from a set of possible values and a selector that is the name of the parameter to set in the test case instance. Another example of `testParameters` is:

```
testParameters

  ^ ParametrizedTestMatrix new
    forSelector: #option1 addOptions: #(a b c);
    forSelector: #option2 addOptions: {[1].[2].[3]};
    yourself.
```

This example will generate 9 different configuration. One per each combination of `option1` and `option2`. Do not forget that the test case should have a setter for each option.

In addition each option can be a literal or a block to generate that value. The block has an optional parameter, the parameter is the test case to configure.

5.3 Classes vs. objects as parameters

There is a subtle but important point about the kind of the parameters. Indeed, we may wonder whether it is better to pass a class or an instance as parameter of a test. Theoritically there is not much difference between passing a class or an object. However in practice there is a difference because when we pass an object, as in the following configuration, the framework does not recreate the object during each test execution. Therefore if your object accumulates information, then such information will be shared amongs your tests and this is a bad idea.

```
CbkDlittleImporterTest class >> testParameters

  ^ ParametrizedTestMatrix new
      addCase: { #importer -> CBkCollectorDLittleImporter new };
    yourself.
```

The solution is to favor passing classes as follows and to explicitly create objects in the `setUp`. This way you are sure that your object does not hold state from previous execution.

Listing 5-1 Testing error raising

```
MyExampleSetTest >> testIllegal
  self should: [ empty at: 5 ] raise: Error.
  self should: [ empty at: 5 put: #zork ] raise: Error
```

```
CbkDlittleImporterTest class >> testParameters

  ^ ParametrizedTestMatrix new
      addCase: { #importerClass -> CBkCollectorDLittleImporter };
    yourself.

CbkDlittleImporterTest >> setUp
  super setUp.
  importer := importerClass new.

CbkDlittleImporterTest >> importerClass: anImporterClass
  importerClass := anImporterClass
```

In conclusion, we suggest to pass instances as parameters when the objects are not complex and to favor classes otherwise.

5.4 Other assertions

In addition to `assert:` and `deny:`, there are several other methods that can be used to make assertions.

First, `TestAsserter >> assert:description:` and `TestAsserter >> deny:description:` take a second argument which is a message string that describes the reason for the failure, if it is not obvious from the test itself. These methods are described in Section 5.4.

Next, SUnit provides two additional methods, `TestAsserter >> should:raise:` and `TestAsserter >> shouldnt:raise:` for testing exception propagation.

For example, you would use `self should: aBlock raise: anException` to test that a particular exception is raised during the execution of `aBlock`. The method below illustrates the use of `should:raise:`.

Try running this test. Note that the first argument of the `should:` and `shouldnt:` methods is a block that contains the expression to be executed.

Note that this is usually not really good to catch exception using the `Error` class, since it is catching basically everything. In that current case, the `at:` primitive signals an instance of `Error` so we have deal with it.

Using `assert:equals:`

In addition to `assert:`, there is also `assert:equals:` that offers a better report in case of error (incidentally, `assert:equals:` uses `assert:description:`).

For example, the two following tests are equivalent. However, the second one will report the value that the test is expecting: this makes easier to understand the failure. In this example, we suppose that `aDateAndTime` is an instance variable of the test class.

```
testAsDate
  self assert: aDateAndTime asDate = ('February 29, 2004' asDate
    translateTo: 2 hours).

testAsDate
  self
    assert: aDateAndTime asDate
    equals: ('February 29, 2004' asDate translateTo: 2 hours).
```

Assertion description strings

The `TestAsserter` assertion protocol includes a number of methods that allow the programmer to supply a description of the assertion. The description is a `String`; if the test case fails, this string will be displayed by the test runner. Of course, this string can be constructed dynamically.

```
...
e := 42.
self assert: e = 23 description: 'expected 23, got ', e printString
...
```

The relevant methods in `TestAsserter` are:

```
assert:description:
deny:description:
should:description:
shouldnt:description:
```

5.5 Running tests

Running a single test

Normally, you will run your tests using the Test Runner or using your code browser. You can also run a single test as follows:

```
MyExampleSetTest run: #testRemove
>>> 1 run, 1 passed, 0 failed, 0 errors
```

Running all the tests in a test class

Any subclass of `TestCase` responds to the message `suite`, which builds a test suite that contains all the methods in the class whose names start with the string *test*.

To run the tests in the suite, send it the message `run`. For example:

```
MyExampleSetTest suite run
>>> 5 run, 5 passed, 0 failed, 0 errors
```

5.6 Advanced features of SUnit

In addition to `TestResource` that we present just in subsequent section, SUnit contains assertion description strings, logging support, the ability to skip tests, and resumable test failures.

Logging support

The description strings mentioned above may also be logged to a `Stream`, such as the `Transcript` or a file stream. You can choose whether to log by overriding `isLogging` in your test class; you can also choose where to log by overriding `failureLog` to answer an appropriate stream. By default, the `Transcript` is used to log.

To enable logging, you should redefine the method `isLogging` to say so.

```
MyExampleSetTest class >> isLogging
  ^ true
```

Skipping tests

Sometimes in the middle of a development, you may want to skip a test instead of removing it or renaming it to prevent it from running. You can simply invoke the `TestAsserter` message `skip` on your test case instance. For example, the following test uses it to define a conditional test.

```
OCCompiledMethodIntegrityTest >> testPragmas

  | newCompiledMethod originalCompiledMethod |
  (self class environment hasClassNamed: #Compiler) ifFalse: [ ^
    self skip ].
  ...
```

It is better to use `skip` than to use a simple `^ self` because in the latter case you may think that you test is executed while it is not!

Continuing after a failure

SUnit also allows us to specify whether or not a test should continue after a failure. This is a really powerful feature that uses Pharo's exception mechanisms. To see what this can be used for, let's look at an example.

Consider the following test expression:

```
aCollection do: [ :each | self assert: each even ]
```

In this case, as soon as the test finds the first element of the collection that isn't `even`, the test stops. However, we would usually like to continue, and see both how many elements, and which elements, aren't `even` (and maybe also log this information).

You can do this as follows:

```
aCollection do: [ :each |
  self
    assert: each even
    description: each printString, ' is not even'
    resumable: true ]
```

This will print out a message on your logging stream for each element that fails. It doesn't accumulate failures, i.e, if the assertion fails 10 times in your test method, you'll still only see one failure. All the other assertion methods that we have seen are not resumable by default; `assert: p description: s` is equivalent to `assert: p description: s resumable: false`.

5.7 Test resources

One of the important features of a suite of tests is that they should be independent of each other. The failure of one test should not cause an avalanche of failures of other tests that depend upon it, nor should the order in which the tests are run matter. Performing `setUp` before each test and `tearDown` afterwards helps to reinforce this independence.

However, there are occasions where setting up the necessary context is just too time-consuming for it to be done before the execution of each test. Moreover, if it is known that the test cases do not disrupt the resources used by the tests, then it is wasteful to set them up afresh for each test. It is sufficient to set them up once for each suite of tests. Suppose, for example, that a suite of tests needs to query a database, or do analysis on some compiled code. In such cases, it may make sense to set up the database and open a connection to it, or to compile some source code, before any of the tests start to run.

Where should we cache these resources, so that they can be shared by a suite of tests? The instance variables of a particular `TestCase` subclass won't do, because a TestCase instance persists only for the duration of a single test (as mentioned before, the instance is created anew *for each test method*). A global

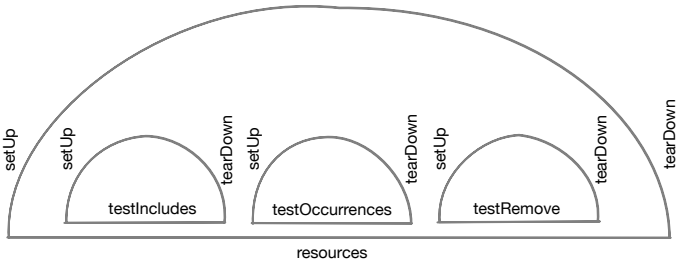

Figure 5-2 `setUp` and `teardDown` in action.

variable would work, but using too many global variables pollutes the name space, and the binding between the global and the tests that depend on it will not be explicit.

A better solution is to define a TestResource and use it. The class `TestResource` implements a singleton to manage the execution of setUp and tearDown around a complete test suite as shown in Figure 5-2. Each subclass of `TestResource` understands the message `current`, which will answer a singleton instance of that subclass. Methods `setUp` and `tearDown` should be overridden in the subclass to ensure that the resource is initialized and finalized.

One thing remains: somehow, SUnit has to be told which resources are associated with which test suite. A resource is associated with a particular subclass of `TestCase` by overriding the *class* method `resources`.

By default, the resources of a `TestSuite` are the union of the resources of the `TestCases` that it contains.

Here is an example. We define a subclass of `TestResource` called `MyTestResource`. Then we associate it with `MyTestCase` by overriding the class method `MyTestCase class >> resources` to return an array of the test resource classes that `MyTestCase` will use.

You can also defines the instance side method `isAvailable` to indicate whether the resource is available. But if you need this, better read the code of the `TestResource` class.

Checking the described behavior

The following trace (written to the `Transcript`) illustrates that a global set up is run before and after each test in a sequence. Let's see if you can obtain this trace yourself.

Listing 5-3 An example of a TestResource subclass

```
TestResource subclass: #MyTestResource
  instanceVariableNames: ''
  ...


MyTestResource >> setUp
  ...

MyTestCase class >> resources
  "Associate the resource with this class of test cases"

  ^ { MyTestResource }
```

```
MyTestResource >> setUp has run.
MyTestCase >> setUp has run.
MyTestCase >> testOne has run.
MyTestCase >> tearDown has run.
MyTestCase >> setUp has run.
MyTestCase >> testTwo has run.
MyTestCase >> tearDown has run.
MyTestResource >> tearDown has run.
```

Create new classes `MyTestResource` and `MyTestCase` which are subclasses of `TestResource` and `TestCase` respectively. Add the appropriate methods so that the following messages are written to the `Transcript` when you run your tests.

Solution

You will need to write the following six methods.

```
MyTestCase >> setUp
  Transcript show: 'MyTestCase>>setUp has run.'; cr

MyTestCase >> tearDown
  Transcript show: 'MyTestCase>>tearDown has run.'; cr

MyTestCase >> testOne
  Transcript show: 'MyTestCase>>testOne has run.'; cr

MyTestCase >> testTwo
  Transcript show: 'MyTestCase>>testTwo has run.'; cr

MyTestCase class >> resources
  ^ Array with: MyTestResource

MyTestResource >> setUp
  Transcript show: 'MyTestResource>>setUp has run'; cr
```

```
MyTestResource >> tearDown
  Transcript show: 'MyTestResource>>tearDown has run.'; cr
```

5.8 Customising tests

In this section, we show that SUnit offers two hooks to define what a test selector is and how to perform the test. Imagine that we want to support a class method on a given class returned by `classWithExamplesToTest` whose selector follow the pattern `example*` methods are considered as tests. We can define method `testSelectors` as follows:

```
HiExamplesTest class >> testSelectors [
  ^ self classWithExamplesToTest class methods
    select: [ :each | (each selector beginsWith: 'example') and: [
    each numArgs = 0 ] ]
    thenCollect: [ :each | each selector ]
```

Then we can redefine the method `performTest` to example the method on the class itself.

```
HiExamplesTest >> performTest
  example := self class classWithExamplesToTest perform:
    testSelector asSymbol
```

5.9 Inheriting TestCase

A new `TestCase` can inherit tests from a superclass. The logic is a bit cumbersome. By default a new test case class inherits from a subclass of `TestCase` that is abstract. If your new subclass as no test methods it will inherit from its superclass.

Otherwise if your new class has selectors and inherits from a concrete superclass, you should redefine `shouldInheritSelectors` to return `true`.

What developers use in practice is the last part: to redefine the method `shouldInheritSelectors`. For example, this is what the `CoCompletionEngineTest` class is doing to inherit the tests of `CompletionEngineTest`

```
CoCompletionEngineTest >> shouldInheritSelectors

  ^ true
```

Here is the definition of the method `shouldInheritSelectors`.

```
TestCase class >> shouldInheritSelectors
  "I should inherit from an Abstract superclass but not from a
    concrete one by default,
  unless I have no testSelectors in which case I must be expecting
    to inherit them from my superclass.
```

```
  If a test case with selectors wants to inherit selectors from a
    concrete superclass, override this to true in that subclass."

  ^self ~~ self lookupHierarchyRoot
    and: [self superclass isAbstract or: [self testSelectors
    isEmpty]]
```

5.10 Conclusion

SUnit is a simple framework but it already provides a powerful set of mechanisms to take real advantage of writing tests. In particular parametrized tests are a powerful method when you have several objects that expose the same API, then you can reuse your tests.

CHAPTER 6

Getting more with continuous integration

Once you have tests you want to run them as many times as possible. In this chapter we show the basic setup to execute your tests automatically on each commit. We present Github actions, and Travis integration both using SmalltalkCI to execute your tests on the server.

6.1 Github Actions

If you use github to store your Pharo code, then you can use a nice continuous integration: GitHub actions. You can edit your script by pressing the Actions button of your github project. You can follow the template. It will create a `.github/workflows` folder on your project, and store your scripts. You can have multiple scripts.

In addition you should add a file to configure SmalltalkCI. This file is in STON format (a.k.a JSON for Smalltalk).

SmalltalkCI configuration lets you specify:

- how to load your project (`loading:`)
- which entities (classes, packages, projects...) to include in the tests (`testing:` and `#include:`)
- which entities (classes, packages, projects...) to exclude from the tests (`testing:` and `#exclude:`).
- several options such as time out,
- but also test coverage (`testing:` and `#coverage:`)

There is a good documentation of SmalltalkCI at https://github.com/hpi-swa/smalltalkCI.

Example

But let us look at an example. Let us take for example the project Containers-Array2d from http://github.com/pharo-containers.

You should first define a file named `.smalltalk.ston` to indicate SmallttalkCI (the application that will run your tests) how to identify them.

Here we indicate how SmalltalkCI should find the source code and which packages should be executed by the tests in the context of also computing the test coverage of that package.

```
SmalltalkCISpec {
  #loading : [
    SCIMetacelloLoadSpec {
      #baseline : 'ContainersArray2D',
      #directory : 'src',
      #platforms : [ #pharo ]
    }
  ],
  #testing : {
    #coverage : {
      #packages : [ 'Containers-Array2D' ]
    }
  }
}
```

Here is the currentStablePharo file.

```
name: currentStablePharo

env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

on:
  push:
    branches:
      - master
  workflow_dispatch:

jobs:
  build:
    strategy:
      matrix:
        platform: [ubuntu-latest, macos-latest, windows-latest ]
    runs-on: ${{ matrix.platform }}
    steps:
      - uses: actions/checkout@v2
```

```
      - uses: hpi-swa/setup-smalltalkCI@v1
        id: smalltalkci
        with:
          smalltalk-version: Pharo64-8.0
      - run: smalltalkci -s ${{
    steps.smalltalkci.outputs.smalltalk-version }}
        shell: bash
        timeout-minutes: 15
```

You can also use matrix setup to run multiple configuration of platforms: here we run on Pharo 8 and Pharo 7 for mac and windows.

```
name: matrix

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

# A workflow run is made up of one or more jobs that can run
    sequentially or in parallel
jobs:
  build:
    strategy:
      matrix:
        os: [ macos-latest, windows-latest ]
        smalltalk: [ Pharo64-8.0, Pharo64-7.0 ]
    runs-on: ${{ matrix.os }}
    name: ${{ matrix.smalltalk }} on ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v2
      - name: Setup smalltalkCI
        uses: hpi-swa/setup-smalltalkCI@v1
        with:
          smalltalk-version: ${{ matrix.smalltalk }}
      - name: Load Image and Run Tests
        run: smalltalkci -s ${{ matrix.smalltalk }}
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        timeout-minutes: 15
```

6.2 Travis

First you should enable the travis services on your github account. We sketch here the process. Better refer to the documentation since the process was

changed in the past. Nowadays travis does not look to work as it was used to.

It usually means that:

- you should have an account on travis-ci.com (travis-ci.org has been deprecated).
- you should get a token from Github `GH_Token` that you add to your travis account.
- you have to encrypt (using `travis encrypt key`) your repository with the one time shown github key.

Then you should add configuration files, for linux and mac a `.travis.yml` file and windows a `appveyor.yml` one.

6.3 With Travis

Here is a configuration file for travis. It should be named `.travis.yml`.

```
language: smalltalk
sudo: false

# Select operating system(s)
os:
  - linux
  - osx

# Select compatible Smalltalk image(s)
smalltalk:

  - Pharo32-8.0
  - Pharo-7.0
  - Pharo-6.1

  - Pharo64-8.0
  - Pharo64-7.0
```

6.4 With Appveyor

If you want to validate your tests on Windows, you should create the file `appveyor.yml` file. Here is an example:

```
environment:
  CYG_ROOT: C:\cygwin
  CYG_BASH: C:\cygwin\bin\bash
  CYG_CACHE: C:\cygwin\var\cache\setup
  CYG_EXE: C:\cygwin\setup-x86.exe
  CYG_MIRROR: http://cygwin.mirror.constant.com
  SCI_RUN: /cygdrive/c/smalltalkCI-master/run.sh
```

```
  matrix:
    - SMALLTALK: Pharo-6.1
    - SMALLTALK: Pharo-7.0

platform:
  - x86

install:
  - '%CYG_EXE% -dgnqNO -R "%CYG_ROOT%" -s "%CYG_MIRROR%" -l
    "%CYG_CACHE%" -P unzip'
  - ps: Start-FileDownload
    "https://github.com/hpi-swa/smalltalkCI/archive/master.zip"
    "C:\smalltalkCI.zip"
  - 7z x C:\smalltalkCI.zip -oC:\ -y > NULL

build: false

test_script:
  - '%CYG_BASH% -lc "cd $APPVEYOR_BUILD_FOLDER; exec 0</dev/null;
    $SCI_RUN"'
```

6.5 Assessing test coverage

Finally once you have automated servers up and running, you can take advantage of other services such as computing the code coverage of your tests. SmalltalkCI is ready for this and you can simply use http://coveralls.io. First create an account to http://www.coveralls.io and enable the communication between coverall and the project you want to cover.

The previous smalltalkCI configuration you showed at the beginning of this chapter is already ready to be used with Coverall.

6.6 Conclusion

Running automatically your tests is a super great feeling. Do not miss this opportunity. This chapter shows you how easy it is to do it with github or travis. Have a look at the documentation of SmalltalkCI https://github.com/hpi-swa/smalltalkCI to see what you do automatically.

CHAPTER 7

UI testing

Developers often think that testing UI is difficult. This is true that fully testing the placement and layout of widgets can be tedious. However, testing the logic of an application and in particular the interaction logic is possible and this is what we will show in this chapter. We show that testing Spec application is simple and effective.

7.1 Testing Spec

Tests are key to ensure that everything works correctly. In addition, they free us from the fear to break something without being warned about it. Tests support refactorings. While such facts are general and applicable to many domains, they also true for user interfaces.

Spec architecture

Spec is based on an architecture with three different layers as shown in Figure~7-1:

- **Presenters:** Presenters defined the interaction logic and manipulate domain objects. They access back-end widgets but via an API that is specified by Adapters.
- **Adapters:** Adapters are objects exposing low-level back-end widgets. They are a bridge between presenters and low-level widgets.
- **Back-end widgets**. Back-end widgets are plain widgets that can be used without Spec.

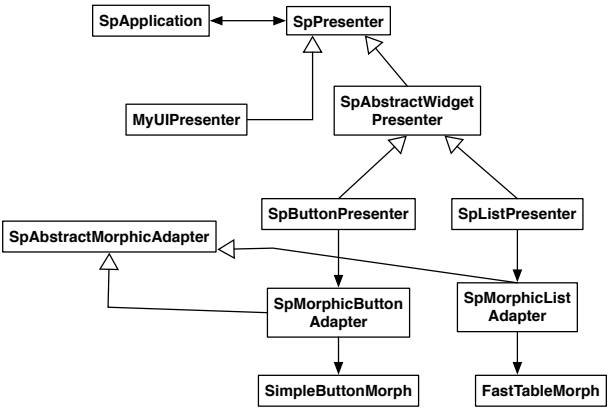

Figure 7-1 Spec Architecture.

Three roles and concerns

To help you understand the different possibilities of testing that you can engage, we identify the following roles and their related concerns.

- **Spec Users.** Spec users are developers that build a new application. They define the logic of the application by assembling together presenters and domain objects.
- **Spec Developers.** Spec developers are more concerned with the development of new Spec presenter and their link with the adapter.
- **Widget Developers.** Widget developers are concerned about the logic and working of a given widget is a given back-end.

We will focus on the first role. For the reader interested in the second role, the class `SpAbstractBackendForTest` is a good starting place.

7.2 Spec user perspective

As a Spec user, you should consider that the back-ends are working and your responsibilities is to test the logic of the user interface components. We should make sure that when the model changes, the user interface components reflect the changes. Inversely when the user interface components change, we should ensure that the model is updated.

Example

We will test a simple spec application. The model for this application can be any class. It shows in a tree presenter all the subclasses of the model. Also, has a text presenter that shows the definition string for the selected class.

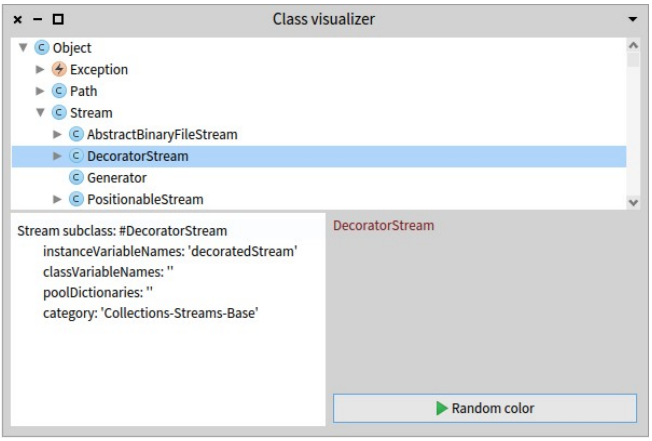

Figure 7-2 A Spec application.

Finally, has a string morph and a button. When the button is pressed, the colour of the morph changes randomly.

Spec user test 1: the tool should be initialized correctly

The tool will be instantiated with a model. In this case, we will use `Object` because it is the root of almost all classes. So, when we instantiate the spec application of the figure above, all the sub presenters of the application must show the data of the model.

```
testInitialization

  | model |
  model := String.
  spApplication := ClassVisualizerPresenter on: model.
  self assert: spApplication model equals: model.
  self
    assert: spApplication textPresenter text
    equals: model classDefinitions first definitionString.
  self
    assert: spApplication morphPresenter morph contents
    equals: model name
```

Spec user test 2: when selecting a new item in the tree presenter the text presenter and the morph should change

The tree presenter shows a tree of classes. When a class of the tree presenter is selected, the text presenter should change according to the definition of the new selected class. The morph must change as well.

```
testSelectItemOnTreePresenter

  "As we have initialized the tree with Object as its roots. The
    class OrderedCollection is a subclass of Object. We would
    simulate that a user selectes OrderedCollection from the tree
    presenter."

  spApplication := ClassVisualizerPresenter on: Object.
  spApplication hierarchyTreePresenter selectItem: OrderedCollection.
  self
    assert: spApplication hierarchyTreePresenter selectedItem
    equals: OrderedCollection.
  self
    assert: spApplication textPresenter text
    equals: OrderedCollection classDefinitions first
    definitionString.
  self
    assert: spApplication morphPresenter morph contents
    equals: OrderedCollection name
```

Spec user test 3: triggering the button action

The action of the colour button changes the colour of the morph randomly. When the button is clicked the morph must change its colour.

```
testButtonChangesMorph

  | previousColor |
  spApplication := ClassVisualizerPresenter on: Object.
  previousColor := spApplication morphPresenter morph color.
  spApplication colorButton click.
  self
    deny: spApplication morphPresenter morph color
    equals: previousColor
```

Spec user test 4: the text presenter should not be editable

For this application, we only want that the text presenter shows the class definition. We do not want the user to be able to edit it.

```
testTextPresenterIsNotEditable

  spApplication := ClassVisualizerPresenter on: Object.
  self deny: spApplication textPresenter isEditable
```

Spec user test 5: the window is built correctly

Here, we will test that the title and the initial extent of the window are correct. Also, we will test if the window was built correctly.

```
testInitializeWindow

  | window |
  spApplication := ClassVisualizerPresenter on: Object.
  window := spApplication openWithSpec.
  self assert: window isBuilt.
  self assert: window title equals: ClassVisualizerPresenter title.
  self assert: window initialExtent equals: 600 @ 400.
  window close
```

Known limitations and conclusion

We show in this chapter that you can take advantage of Spec to define tests that will help you to evolve the visual part of your application.

Currently Spec does not offer a way to script and control popup window. It is not possible to script a button that opens a dialog for a value. Future versions of Spec20 should cover this missing feature.

CHAPTER 8

Testing web applications with Parasol

Chapter Contributors: Evelyn Cusi and Daniel Aparicio

During the construction of a web application, it is exhausting to have to test the entire flow of the application each time you modify it, and much more as the application grows. This chapter introduce Parasol to allow developers to automate tests with a collection of language specific bindings, giving us the facility to test complex interactions in matter of seconds.

Parasol, or also called *Beach Parasol*, is a Smalltalk framework to write functional/acceptance tests using Selenium WebDriver. Through Parasol, you can access to all functionalities of Selenium WebDriver in an intuitive way. Parasol gives a convenient API to access to the WebDrivers of Selenium such as: Chrome, Firefox and others. Actually, Parasol supports Pharo 6.0 and 7.0, also supports GemStone version 3.1, 3.2 and 3.3. Parasol was developed by Johan Brichau and Andy Kellens from Two Rivers.

8.1 Getting started

To load Parasol into your Pharo image, you can execute the following script in Playground:

```
Metacello new
    baseline: 'Parasol';
    repository: 'github://SeasideSt/Parasol/repository';
    load: 'tests'.
```

Once this is done you can start

```
ZnZincServerAdaptor startOn: 8080.
```

Since, Parasol uses Selenium WebDriver, you must download Selenium WebDriver and a browser driver, this last one depends on which browser you want to use to test your application.

Downloading Selenium web driver

Selenium server is a project that contains a set of tools and libraries that enable and support the automation of web browsers. Selenium is a Java program, therefore it requires a Java Runtime Environment (JRE) 1.6 or a new version to run it. If Java Runtime Environment (JRE) is not installed on your system, you can download the JRE from the Oracle website.

In this chapter, we use Selenium server 3.141.x to run our examples. You can download it from the official page of Selenium. The name of file should contain the following prefix:

```
selenium-server-standalone-3.141.x.jar
```

You may use the following commant to run Selenium server in your computer.

```
java -Dwebdriver.chrome.driver=chromedriver -jar
    selenium-server-standalone-3.141.x.jar
```

Note that for running the previous command you need to configure java in the PATH (environmente variable).

Browser driver

Selenium requires a driver to interact with the web browser. For instance, Chrome requires Chromedriver, which must be installed before the following examples can be run. The table below shows browsers compatible with Parasol and their respective links to download.

Browser Driver	URL
Chrome	https://sites.google.com/a/chromium.org/chromedriver/downloads
Firefox	https://github.com/mozilla/geckodriver/releases
Safari	https://webkit.org/blog/6900/webdriver-support-in-safari-10/

We use Chromedriver version 80.0.3987.106 to run the chapter examples, but you can use other browser drivers or another version that you want.

8.2 First steps with Selenium

This section describes a simple test wrote using parasol. This section assumes that you have already installed Parasol and Selenium server is already running.

A first test

Let's start easy and assume we want to test if the title of the http://pharo.org website is correct. For this, first we need to create a class that inherits from `TestCase`.

```
TestCase subclass: #PharoOrgTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FT-Parasol'
```

Then, we create a method called `testTitleOfPharoPage` as follows:

```
PharoOrgTest >> testTitleOfPharoPage

  | driver |
  driver := BPRemoteWebDriver withCapabilities: BPChromeOptions new.
  driver get: 'https://pharo.org/'.
  self assert: 'Pharo - Welcome to Pharo!' equals: driver getTitle.
  driver close.
```

Step by step explanation

First, we create a subclass of `TestCase`, we do not need instance or class variables for now. We placed this subclass is in the FT-Parasol package.

```
TestCase subclass: #PharoOrgTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FT-Parasol'
```

The `testTitleOfPharoPage` method contains a temporary variable that we use to save our instance of the Chrome WebDriver.

```
| driver |
driver := BPRemoteWebDriver withCapabilities: BPChromeOptions new.
```

Note that if you want to use another driver, you must change the class `BPChromeOptions` to another browser driver compatible with Parasol.

The `get` method loads a given URL and allow you to navigate through the website. The WebDriver will wait that the page is fully loaded before returning the control to the test. If your page loads a large amount of AJAX, then WebDriver may not know when the page has been fully loaded.

```
driver get: 'https://pharo.org/'.
```

The next line of our method is an assertion to confirm that the title of the page is equal to: `'Pharo - Welcome to Pharo!'`.

```
self assert: 'Pharo - Welcome to Pharo!' equals: driver getTitle.
```

Finally, we close browser window.

```
driver close.
```

You may also use the method `quit` instead `close`. The difference is that `quit` will come out of the browser, and `close` will close a tab. Note that the `close` method will close a tab only if there is an open tab, by default most browsers will close completely.

Improve the structure of your test

In our previous example, we wrote the entire test in one method, however, is a best practice add all the prerequisites of the tests on the `setUp` method and all the steps of cleaning on the `tearDown` method.

Therefore, to improve the structure of these tests, we first will convert the temporary variable driver into an instance variable.

```
TestCase subclass: #PharoOrgTest
  instanceVariableNames: 'driver'
  classVariableNames: ''
  package: 'FT-Parasol'
```

Next, we will place the statements that load the drive the `setUp` method:

```
PharoOrgTest >> setUp
  super setUp.
  driver := BPRemoteWebDriver withCapabilities: BPChromeOptions new.
  driver get: 'https://pharo.org/'
```

In the same way, we move the cleaning statements in the `tearDown` method:

```
PharoOrgTest >> tearDown
  super tearDown.
  driver quit
```

Finally, our test method will be rewrited as follows:

```
PharoOrgTest >> testTitleOfPharoPage
  self assert: 'Pharo - Welcome to Pharo!' equals: driver getTitle.
```

Now, if you run this test again, you should behave in the same way as the first version of our test we saw.

8.3 Locating elements with Parasol: The basics

In our tests, we would like to verify also if some particular HTML elements display the information we want to. But before to perform this verification, we first need to find these elements. Parasol uses what are called *Locators* to find and match the elements of the web page. Parasol has eight locators as shown in the following table.

Locator	Example
ID	findElementByID: 'user'
Name	findElementByName: 'username'
Link Text	findElementByLinkText: 'Login'
Partial Link Text	findElementByPartialLinkText: 'Next'
XPath	findElementsByXPath: '//div[@id="login"]/input'
Tag Name	findElementsByTagName: 'body'
Class Name	findElementsByClassName: 'table'
CSS	findElementByCSSSelector: '#login > input[type="text"]'

You can use any of them to find the element that you are looking for in your application. The following paragraphs briefly describe how to use a number of these locators.

Find element by ID

The use of ID is the easiest and probably the safest way to locate an element on HTML. Test scripts that use IDs are less prone to changes in the application. For example, consider the following HTML form:

```
<form method="get">
    <input id="fullName" name="textInfo" type="text"/>
    <input id="submitButton" type="submit"/>
</form>
```

We may locate the input field by its ID, as follows:

```
textField := driver findElementByID: 'fullName'.
```

If no element has the ID attribute that match with the provided one, a `BPNoSuchElementException` exception will be raised.

Find element by name

An HTML element may have an attribute called `name`, they are normally used in the forms such as text fields and selection buttons. The values of the attributes of name are passed to the server when a form is sent. In terms of lower probability of change, the attribute name is probably the second in importance after ID.

Considering the previous HTML code of example, you can locate the elements using the `findElementByName` method:

```
textField := driver findElementByName: 'textInfo'.
```

Find element by link and partial link

It is possible to find an element based on its link. You can use this way of location of elements when you know the link text used inside of an anchor tag.

With this strategy, the first element with the value of text link that matches with the location will be returned. For instance, we may find the following anchor element:

```
<a href="https://pharo.org/">Go to Pharo!</a>
```

The HTML element can be located in the following ways:

```
linkPharo := driver findElementByLinkText: 'Go to Pharo!'.
linkPharo := driver findElementByPartialLinkText: 'Go to'.
```

Notice that with `findElementByPartialLinkText` we don't need to give the full text link, just part of it.

Find element by tag name

Finding elements by a tag name is used when you want to locate an element by the name of its label. However, because there is a limited set of tag names, it is very possible that more than one element with the same name of tag exists, so this locator is not normally used to identify an element, instead, it is more common to use it in chained locations.

Consider our previous example form:

```
<form method="get">
  <input id="inputField" name="textInfo" type="text"/>
  <input id="submitButton" type="submit"/>
</form>
```

We could locate the input element (input) as follows:

```
input := driver findElementByTagName: 'input'
```

Find element by class name

The class attribute of a HTML element is used to add style to our pages. And it can be used to identify elements too.

In the next example:

```
<p id="testParagraph" class="testclass1">Bla bla</p>
<a class="testclass2">foobar</a>
```

We can use any of the following line to locate the *testclass* elements.

```
testClassOne := driver findElementByClassName: 'testclass1'.
testClassTwo := driver findElementByClassName: 'testclass2'.
```

It is common in web applications to use the same class attribute in several elements, so be careful if you try to get the elements by class name.

Find element by CSS selector

You can locate an element through the syntax of the CSS selector.

By example, the element p of the next HTML code:

```
<p class="content">Hello, how are you?</p>
```

It could be located like this:

```
paragraphHello := driver findElementByCSSSelector: 'p.content'
```

8.4 Finding elements using XPath

XPath, the XML Path Language, is a query language for selecting nodes from an XML document. When a browser renders a web page, the HTML contents may be parsed into a DOM tree or similar. Therefore, it is possible to find a specific node in the tree using XPath. To illustrate how to find a element with XPath we will use the next HTML code as example:

```
    <div id="testDiv1">
  <p id="testDiv1p" class="c1"></p>
    </div>
    <div id="testDiv2">
  <p id="testDiv2p1" class="c2"></p>
  <p id="testDiv2p2" class="c1"></p>
  <p id="testDiv2p3" class="c1"></p>
    </div>
```

Imagine that we need to get the div with ID equal to testDiv2, so we use the next code snippet to get it:

```
testDiv := driver findElementByXPath: '//div[@id=''testDiv2'']'
```

For locating elements with XPath, we first need to understand their syntax. Therefore, to use this option, you will first need to learn XPath syntax.

Note that XPath is a very powerful way to locate web elements when by id, by name or by link text are not applicable. However, XPath expression are vulnerable to structure changes around the web element, because the path from the root to the target element may change.

8.5 Finding multiple elements

There are some cases in which you can have more than one element with the same attributes.

Chaining findElement to find a child element

The following example

```
<div id = "div1">
  <input type="checkbox" name = "same" value="on">Same checkbox in
    Div1</input>
</div>
<div id = "div2">
  <input type="checkbox" name = "same" value="on">Same checkbox in
    Div2</input>
</div>
```

In these cases, XPATH can be used, however, there is a simpler way, using nested selectors which is nothing else than locating elements in the result of a previous location. For example the following locates the div entry with id ”div2” :

```
inputOfDiv2 := (driver findElementByID: 'div2') findElementByName:
    'same'
```

Multiple elements

As the name suggests, `findElementsByTagName` returns a list of matching elements. Its syntax is exactly the same as `findElement`, but in plural `findElements`.

For example, to get all div of our previous example:

```
divElements := driver findElementsByTagName: 'div'.
```

Sometimes, findElement fails due to multiple matching elements on a page, of which one was not aware. findElements will be useful to find them.

8.6 Interacting with the elements

Until now we saw how to navigate through a URL and how to select elements. Now, our next step is to interact with these elements. You can do different things with these elements depending on their type. Using the following form as an example we will show how to interact with it and its elements.

```
<html>
<body>
<h1>Sign in</h1>
<form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <button name="login" type="submit" class= "btn
    btn-primary">Login</button>
    <a href="forgotPasswordsd.html">Do you forgot your password?</a>
```

```
    <p class="content">
        "Are you new here?"
        <a href="register.html">Create an account</a>
    </p>
</form>
</body>
<html>
```

Filling text in a text field

To fill the username and password fields in this form, we first have to select them, for this, we will use the following code:

```
name := driver findElementByName: 'username'.
password := driver findElementByName: 'password'
```

Now it is possible to fill text in these fields as follows:

```
name sendKeys: 'John'.
password sendKeys: 'xxxxxxx'.
```

It is possible to send the message `sendKeys:` in any element: This makes it possible to validate keyboard shortcuts. However, writing something in a text field does not automatically delete it. Instead, what you write will be attached to what is already there. You can easily delete the content of a text field or text area with the message `clear`.

Activating links and buttons

Another very useful action in the navigation of web pages is to click on links and buttons, such action is activated using the message `click` on the selected element. Below is an example of how to use it with our form.

```
loginButton := driver findElementByName: 'login'.
loginButton click.
```

8.7 Parasol in action

Previous sections introduced Parasol features though basic examples. This section applies eveything learned during the chapter to create a number of tests for a small but real website called *Mercury Tours*. *Mercury Tours* is an agency that offers trips. Maybe if you worked with automated web test before, you are familiar with it.

Setting up tests

First, we will create a subclass of test called `EPTest`:

```
TestCase subclass: #EPTest
  instanceVariableNames: 'driver'
  classVariableNames: ''
  package: 'Example-Parasol-Tests'
```

As we see in previous sections, the instance variable driver represents our browser driver necessary to work with Parasol and its methods. Second, we need to initiallize the driver and load the page, since we need to do this step for all the tests we will place this in the `setUp` method.

```
EPTest >> setUp
  super setUp.
  driver := BPRemoteWebDriver withCapabilities: BPChromeOptions new.
  driver get: self baseURL

EPTest >> baseURL
  ^ 'http://newtours.demoaut.com/'
```

As you notice, we use `self baseURL` to get the URL. It is not necessary to separate the URL in a method.

Finally, we define the `tearDown` method:

```
EPTest >> tearDown
  driver quit.
  super tearDown
```

It is important to use the `quit` message to close the browser when a test ends, as we mentioned in second section of this chapter. With the `setUp` and `tearDown` method, we are able to do any test on the page.

8.8 Testing the page title

Our first test will verify if our URL is the principal URL of web page, which is defined as follows:

```
EPTest >> testPageEntry
  | title |
  title := driver getTitle.
  self assert: title equals: 'Welcome: Mercury Tours'
```

Remember that if we run Parasol tests we must first run the Selenium server. With the Selenium server and the test running, we can see that our browser was opened and can see the home of our tested page:

After the web browser is closed, we can see that the test passed because our page have the title *Welcome: Mercury Tours* in our tab. So it's time to do something more complex.

Figure 8-1 The Mercury Tours WebSite.

8.9 Testing displayed information

The Mercury Tours page has a lot of information on its home page. So, why not try to test this information? As an example, we will write a test to verify the information in the table called *specials*. Assume that you want to test that the first row is the flight from Atlanta to Las Vegas and the price is $398. How do we do that? First step is to know how our page represents this information.

Use the developer tools of Chrome browser and select the desired row. We get the following HTML code:

```
<tr bgcolor="#CCCCC">
    <td width="80%">
    <font face="Arial, Helvetica, sans-serif, Verdana" size="2">
        Atlanta to Las Vegas
    </font>
```

```
    </td>
    <td width="20%">
    <div align="right">
    <font face="Arial, Helvetica, sans-serif, Verdana" size="2">
        <b>$398</b>
    </font>
    </div>
    </td>
</tr>
```

We can see the following details:

- The row is inside a table.
- The table or the row does not contain an ID or a Class that we can use to find this element.
- The CSS or the Tag Name can't help us to find this row.

In this special cases, XPath is a useful locator in this particular case. First, we need to create a subclass of `EPTest`. We are using `EPTest` as a base class for future test classes. This decision give us a number of benefits, for instance, we may find easily all test classes by inspecting the subclasses of `EPTest`, and we can reuse some methods in the subclasses, in particular, the `setUp` and `tearDown` methods.

Therefore, we define the following class:

```
EPTest subclass: #EPHomePageTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Example-Parasol-Tests'
```

`EPHomePageTest` is a class only dedicated to testing the home page and all elements that are part of this. Taking our HTML code as a base, create a method to get the flight and and its price, then test if the value of these elements is the same as the expected result. The following method give us the expected result, but it contains some things that we can improve.

```
EPHomePageTest >> testPriceAndInfoForFlightFromAtlantaToLasVegas

self assert: ((driver findElementByXPath:
    '/html/body/div/table/tbody/tr/td[2]/table/tbody/tr[4]/td/table/tbody/tr/td[2]/ta
    getText)
     equals: 'Atlanta to Las Vegas'.
self assert: ((driver findElementByXPath:
    '/html/body/div/table/tbody/tr/td[2]/table/tbody/tr[4]/td/table/tbody/tr/td[2]/ta
    getText)
    equals: '$398'
```

We will refactor this method to make it more understandable. We will add a method to get the controller element.

```
EPHomePageTest >> descriptionFlightFromAtlantaToLasVegas
  ^ driver findElementByXPath: '/html/body/div/table/tbody/tr/td[2]/
  table/tbody/tr[4]/td/table/tbody/tr/td[2]/table/tbody/tr[2]/td[1]/
  table[1]/tbody/tr[3]/td/table/tbody/tr[1]/td[1]/font'

EPHomePageTest >> priceFlightFromAtlantaToLasVegas
  ^ driver findElementByXPath: '/html/body/div/table/tbody/tr/td[2]/
  table/tbody/tr[4]/td/table/tbody/tr/td[2]/table/tbody/tr[2]/
  td[1]/table[1]/tbody/tr[3]/td/table/tbody/tr[1]/td[2]/div/font/b'
```

Then, we may now refactor the test method as follows:

```
EPHomePageTest >> testFlightFromAtlantaToLasVegas
  self
    assert: self descriptionFlightFromAtlantaToLasVegas getText
    equals: 'Atlanta to Las Vegas'.
  self
    assert: self priceFlightFromAtlantaToLasVegas getText
    equals: '$398'
```

We test a part of the home page, if you want to test other parts, follow the previous steps and try to construct multiple tests. The tests above are a good example of how we get elements from a HTML page and test if the displayed page contains the correct information we want to display.

8.10 Testing interactions

We will now show how to test interactions. It's an important step to test, so we will explain how you could you do it. In this example, we will create a test that registers a user in the website. First, we will create another subclass of `EPTest` called `EPRegisterUserTest`.

```
EPTest subclass: #EPRegisterUserTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Example-Parasol-Tests'
```

In this website to register a user we need first click the *REGISTER* button in the bar menu. Therefore, we will create a method that emulates a click in the *REGISTER* button.

```
EPRegisterUserTest >> clickInRegisterButton
  (driver findElementByLinkText: 'REGISTER') click
```

We use this method in our `setUp` method as follows:

```
EPRegisterUserTest >> setUp
  super setUp.
  self clickInRegisterButton
```

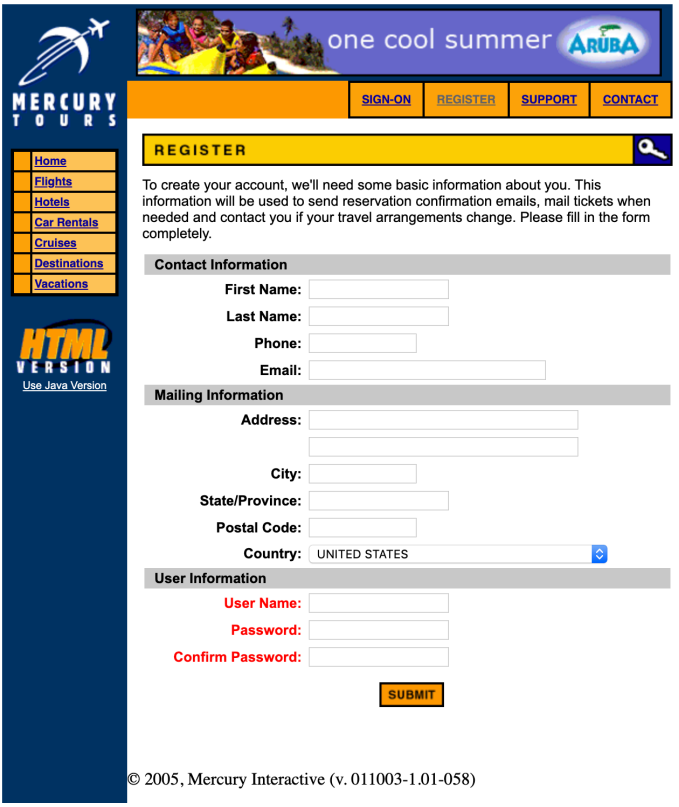

Figure 8-2 Registration From

With this little change, the first step that each test will perform is to move to the *REGISTER* form. You may see the *REGISTER* form in Figure 8-2.

Now, we are prepared to create tests in `EPRegisterUserTest`. Our first test is to verify if we are in the correct page when we click on the register button. So, create the test.

```
EPRegisterUserTest >> testEntryToRegisterPage
  | title |
  title := driver getTitle.
  self assert: title equals: 'Register: Mercury Tours'
```

In this page we have a registration form with multiple fields. If we want to register on the page we must complete three sections in the form. Contact, mailing and user information, so we define three tests to complete these sections.

As you remember, the first step is to see how the page render the elements in the form. For this, we inspect the HTML code using the browser tools. The

following code shows the contact information HTML code:

```
<tr>
<td align=right>
<font face="Arial, Helvetica, sans-serif" size="2">
<b>First Name: </b>
</font>
</td>
<td>
<input maxlength=60 name="firstName" size="20">
</td>
</tr>
 <tr>
<td align=right>
<font face="Arial, Helvetica, sans-serif" size="2">
<b>Last Name: </b>
</font>
</td>
<td>
<input maxlength=60 name="lastName" size="20">
</td>
</tr>
<tr>
<td align=right>
<font face="Arial, Helvetica, sans-serif" size="2">
<b>Phone:</b>
</font>
</td>
<td>
<input maxlength="20" name="phone" size="15">
</td>
</tr>
<tr>
<td align=right>
<font face="Arial, Helvetica, sans-serif" size="2">
<b>Email:</b>
</font>
</td>
<td>
<input name="userName" id="userName" size="35" maxlength="64">
</td>
</tr>
```

We can draw the following conclusions from the HTML code:

- All inputs have a name or an ID.
- All forms are in a table.

With these conclusions we can start with the tests. The test `firstNameField` tests whether the correct information is entered into the contact information section. Like our previous methods, we need to create multiple methods

Figure 8-3 Filling fields using Parasol.

to get the fields elements to put some information into them. At following we show one method that returns the field with the name `firstName`.

```
EPRegisterUserTest >> firstNameField
  ^ driver findElementByName: 'firstName'
```

You need to create as many of these you need to get all the form fields you need for your test. The next step is create a method to set information to fields. We use the message `sendKeys` to set the information that we want.

```
EPRegisterUserTest >> fillInformationToContactSection
  self firstNameField sendKeys: 'Jhon'.
  self lastNameField sendKeys: 'Stuart'.
  self phoneField sendKeys: '10276123'.
  self emailField sendKeys: 'example@mail.com'
```

Finally, we create the following test to verify if the introduced information is the same as the fields.

```
EPRegisterUserTest >> testIntroduceInformationInContactSection
  self fillInformationToContactSection.
  self assert: (self firstNameField getAttribute: 'value') equals:
    'Jhon'.
  self assert: (self lastNameField getAttribute: 'value') equals:
    'Stuart'.
  self assert: (self phoneField getAttribute: 'value') equals:
    '10276123'.
  self assert: (self emailField getAttribute: 'value') equals:
    'example@mail.com'
```

If we run the test we can see how Parasol introduces the information to the fields (see Figure 8-3).

Now, we can create tests to fill information to other sections. An important observation is the select box in mailing information section. If we want to test it, we need to use another method to assert it. In this case, the selected option contains the attribute called `selected`. So, we define the next method to get this element:

Figure 8-4 Registration successful view.

```
EPRegisterUserTest >> getSelectedOfCountry
  (self countryField findElementsByTagName: 'option') do: [ :each |
    (each getAttribute: 'selected')
      ifNotNil: [ ^ each ] ]
```

When you create the test, with this method you only get the element that contain the information that we need. So we use the `getText` message and assert if the fields contain the correct information.

```
EPRegisterUserTest >> testIntroduceInformationInMailingSection
  self assert: (self getSelectedOfCountry getText) equals: 'UNITED
    STATES '.
```

It is also posible change the value of the combo box using the `sendKeys` message as we saw in previous examples.

Now, with all fields tested, we define a test to the user registration on the page. If you created the `fillInformationToMailingSection` and `fillInformationToUserSection` methods, we will use these methods. If you didn't, you need to create these methods.

We need another method to find the submit button and click it, so we create it.

```
EPRegisterUserTest >> clickInSubmitButton
  (driver findElementByName: 'register') click
```

In the page, when you register a user you obtain the following view:

We have two options to assert if the user is registered:

- Get the text 'sign-off' of sign-off option
- Get the description text below the register title.

You can use any of these options, as an example we will get the description text. So define the method to find the description text.

```
EPRegisterUserTest >> descriptionText
  ^ driver findElementByXPath: '/html/body/div/table/tbody/tr/td[2]/
table/tbody/tr[4]/td/table/tbody/tr/td[2]/table/tbody/tr[3]/td/p[2]/font'
```

And finally create the test to verify if the user is registered successfully.

```
EPRegisterUserTest >> testRegistrationOfUser
  <timeout: 10>
  self sendInformationToContactSection.
  self sendInformationToMailingSection.
  self sendInformationToUserSection.
  self clickInSubmitButton.
  self
    assert: (self descriptionText getText)
    equals: 'Thank you for registering.
You may now sign-in using the user name
and password you''ve just entered.'
```

Sometimes this test can fail because the page needs to save the new user and load the successfully message. Another problem that can cause the test failure is the time it takes for our Selenium server to use the browser driver. So we use the timeout pragma to try to avoid this, but you can also use the `Delay` class. If you don't have this problem, delete the line.

Finally run the test and observe its result. If the test passed, you are now able to create multiple tests using Parasol.

8.11 Conclusion

The purpose of this chapter was to introduce the basics of Parasol to create a test suite. You should now be able to find elements in a website, fill information in fields, and interact with the website throught links and buttons.

CHAPTER 9

MockObject and Teachable: Two super simple mocking approaches

Imagine that you want to test a network protocol, you do now want to rely on real network but you would like to fix certain parameters and make that your protocol is reacting correctly. You can use Mock objects to represent a part of the world during your tests. A mock object is an object that supports testing in isolation and represents a behavior that should be fixed from a test point of view.

There are several frameworks for defining mock objects: BabyMock and Mocketry are two of the most sophisticated. In this chapter we present two super simple and minimalist mocking approach. We start with an extension that has been introduced as an extension of SUnit in Pharo 9. It has been designed by Giovanni Corriga.

9.1 About MockObject design

While simpler and less sophisticated than other libraries, this implementation is still quite powerful. This implementation takes a different approach compared to BabyMock/BabyMock2 and Mocketry:

- there are no methods such as `should`, `can`, or `be:`,
- the mocks are stricter in their behaviour – users need to send all and only the defined messages, in the same order as they were defined.

- the mocks need to be manually verified using the new message `TestCase>>verify:` defined on `TestCase`.

These limitations are on purpose, mainly for two reasons:

- to discourage the use of these objects unless they are really needed.
- to keep the implementation simple so that it can be integrated in Pharo's SUnit library, as opposed to being its own framework.

This extension is similar to Pharo Teachable that you can find at: https://github.com/astares/Pharo-Teachable that we present below as a complement.

9.2 MockObject

I am a test double object that can be used as a stub, a fake, or a mock. I provide a simple protocol so that the user can teach me what messages to expect, and how to behave or respond to these messages.

Usage

A new object can be created with`MockObject new`, or using the utility methods on the class side 'instance creation' protocol. The main message to teach a MockObject is`on:withArguments:verify:`; the other methods in the 'teaching' protocol all send this message. This message takes a selector symbol, a collection of arguments, and a block. By sending this message, the mock object is trained to evaluate the block when receiving a message matching the selector and the arguments.

Other variations of this message have more specialized behavior:`on:withArguments:respond` will simply return its third argument when the mock receives a matching message; likewise `on:withArguments:` will return the mock itself. The other methods in the 'teaching' protocol provide an ergonomic API for this behaviour.

A mock object will expect to receive only the messages it has been trained on, in the same order and number as it was trained. If it receives an unknown message, or a known message but in the wrong order, it will simply return itself.

9.3 Stubs, Fakes, and Mocks

A MockObject can be used as a stub by not using the `verify:` variants of the 'teaching' protocol.

To use the MockObject as a real mock, the user needs to verify its use. This is done by means of the `TestCase>>verify:` message. Verification needs to be triggered by the user - it's not automatic.

The`verify:` message will assert

- that the mock has received all the messages it has been trained on
- that it has not received only those messages
- that it has received the messages it has been trained on.

9.4 Example

```
TestCase << #MyClassUsingMock
  slots: { #mock };
  package: 'SUnit-MockObjects-Tests'
```

We create a mock object and teach it two messages `meaningOfLife` and `secondMeaning` that should be send one after the other.

```
MyClassUsingMock >> setUp

  super setUp.
  mock := MockObject new.
  mock
    on: #meaningOfLife
    respond: 42.
  mock
    on: #secondMeaning
    respond: 84.
```

Then we can write a test that will send the message to the object and expect a given result. We can also verify that all the messages have been sent.

```
MyClassUsingMock >> testVerifyChecksThatAllTheMessageGotSent

  self assert: mock meaningOfLife equals: 42.
  self assert: mock secondMeaning equals: 84.
  self verify: mock
```

We can also specify which precise argument a message should be passed using the message `on:with:respond:` or `on:with:with:respond:`.

```
MyClassUsingMock2 >> setUp

  super setUp.
  mock := MockObject new.
  mock
    on: #meaningOfLife:
    with: 22
    respond: 42.
  mock
    on: #secondMeaning:and:
    with: 32
    with: 64
```

```
    respond: 84.
```

The following

```
MyClassUsingMock2 >> testMeaningOfLifeDoesNotPassCorrectValue

  self should: [ self assert: (mock meaningOfLife: 33) equals: 42]
    raise: TestFailure
```

```
MyClassUsingMock2 >> testMeaningOfLife

  self assert: (mock meaningOfLife: 22) equals: 42.
  self assert: (mock secondMeaning: 32 and: 64) equals: 84
```

9.5 About matching arguments

Regarding the arguments matching, the code always goes through the matching logic but it's still quite flexible. The following three possibilities are supported:

- ignoring the arguments: if the message send has no arguments of interest,
- full match: if all arguments of the actual message send match the expected arguments,
- partial match: the user of the mock should use `MockObject class>>any` for any argument that can be ignored. This is a special wildcard object to be used when we don't care about an argument.

For example, let's assume we have this code under test:

```
aMock messageToBeMocked: 1 withArguments: 'two'
```

This can be mocked the three following way: ignoring arguments, full match or partial match.

Ignoring arguments.

```
aMock on: #messageToBeMocked:withArguments:
```

Full match.

```
aMock
  on: #messageToBeMocked:withArguments:
  with: 1
  with: 'two'
```

Partial match.

```
aMock
  on: #messageToBeMocked:withArguments:
  with: MockObject any
  with: 'two'
```

The above training methods are to be used when we don't care about the behaviour when mocking, but only that the message is sent. If we want to control what is returned by the message send, we can use the `respond:` variants. If we want even finer control e.g. side effects, raising exceptions we can use the `verify:` variants.

9.6 Teachable

The other simple library for mocking object is Pharo Teachable. It is not integrated into SUnit but still worth checking it. It is developed by Torsten Bergman.

Installation

You can install teachable using the following expression:

```
Metacello new
  repository: 'github://astares/Pharo-Teachable/src';
  baseline: 'Teachable';
  load
```

Teachable is a class whose instances can be taught to respond to messages. It's usefull for creating mocks who should behave like other objects (for instance inside of a test case) without actually implementing a real mock class.

Here is an example how it can be used:

```
| teachable |
teachable := Teachable new.
teachable
    whenSend: #help return: 'ok';
     whenSend: #doit evaluate: [ 1 inspect ];
     acceptSend: #noDebugger;
     whenSend: #negate: evaluate: [ :num | num negated ].
```

After teaching the object we can use it as if it had a normal implementation in a class:

```
teachable help.
  "this will return the string 'ok'"
teachable doit.
  "this will open the inspector on the SmallInteger 1"
teachable noDebugger.
```

```
  "this will accept the send of #noDebugger and return the teachable"
teachable negate: 120
  "this will return -120"
```

9.7 Conclusion

We presented two little approaches to support simple mock objects by providing a way to teach values or action to be performed by the mock. More mature and powerful frameworks exist: BabyMock and Mocketry.

CHAPTER 10

Mocketry

Mocketry is an alternate mock framework developed by Denis Kudriashov. It provides a simple way to stub any message to any object and to verify any behavior.

10.1 How to install Mocketry?

You can install Mocketry using the following expression

```
Metacello new
  baseline: 'Mocketry';
  repository: 'github://dionisiydk/Mocketry';
  load.
```

10.2 About Test-Driven Development and Mock Objects

Test-Driven Development is a software development technique in which programmers write tests before they write the code itself. In the tests they make assertions about the expected behavior of the object under test. There are two kinds of assertions, the ones where you assert that an object answered correctly in response to a message. Here is an example expressed using Mocketry:

```
result := calculator add: 'V' to: 'VII'.
result should equal: 'XII'.
```

And the ones where you assert that an object sent the correct message to another object. A mock object library helps in the latter case by providing a framework for setting up those assertions:

```
publisher publish: testEvent.
subscriber should receive eventReceived: testEvent.
```

A mock library also provides tools to specify the expected behavior of interacting objects to simulate particular test case. Here is an example:

```
order := Order on: 50 of: #product.
warehouse := Mock new.
(warehouse stub has: 50 of: #product) willReturn: true.

order fillFrom: warehouse.

warehouse should receive remove: 50 of: #product.
```

10.3 TDD is not just about testing

Test-Driven Development combined with mock objects supports good object-oriented design by emphasizing how the objects collaborates with each other rather than their internal implementation. Mock objects can help you discover object roles as well as design the communication protocols between the objects.

We'll describe how to use mock objects and outside-in TDD in practice. We're going to use SUnit as our test framework (we assume you're familiar with it) and Mocketry as our mock object library. This will be more like a message (or interaction) oriented approach of TDD, so we choose the example codes accordingly. Depending on what problem you're trying to solve and what design strategy you're following, you might want to use simple assertions instead of mocks and expectations. But in an object oriented code there are plenty of places where objects *tell each other to do things* therefore mocks will come in handy.

10.4 How does outside-in TDD work?

We start with the action that triggers the behavior we want to implement and work our way from the outside towards the inside of the system. We work on one object at a time, and substitute its collaborator objects with mocks. Thus the interface of the collaborators are driven by the need of the object under test. We setup expectations on the mocks, which allows us to finish the implementation and testing of the object under test without the existence of concrete collaborators. After we finished an object, we can move on towards one of its collaborator and repeat the same process. Now In the following sections, we will develop a complete sample application written with TDD.

10.5 Getting started

With SUnit you create test case classes by subclasing TestCase. Then in any test you can create mock by simple `new` message:

```
yourMock := Mock new.
```

Mocketry does not require any special context variables for this. Also Mocketry implements auto naming logic to retrieve variable names from test context. Inside test yourMock will be shown as "a Mock(yourMock)" (in debugger).

But if you need special name you can specify it:

```
Mock named: 'yourMock'
```

Also there is way to get multiple mocks at once:

```
[ :mockA :mockB | "your code here" ] runWithMocks
```

Then to specify expected behaviour for mock just send `stub` message to it. Following expression will be recorded as expectation. And to verify what was going on with objects just send `should receive` messages and following expression will be verified. Look at 10.15 section for all detailes on Mocketry API.

Now In the next sections, we will develop a complete sample application written with TDD.

10.6 A simple shopping cart

Let's design a simple shopping cart that will calculate the price of the products and charge the customer's credit card. The total charge is calculated by subtracting the discount price from the total order amount. During this process we are going to discover several roles that mock objects will play.

10.7 Writing our first test

After an incoming checkout message, the shopping cart will send a `charge:` message to a credit card object if it is not empty. This outgoing message and its argument is what we need to test in order to verify the behaviour of our shopping cart.

Let's start with the simplest test when the shopping cart is empty. In this case no `charge:` message should be sent.

```
ShoppingCartTest >> testNoChargeWhenCartIsEmpty
  | creditCard cart |
  creditCard := Mock new.
  cart := ShoppingCart payment: creditCard.

  cart checkout.

  creditCard should not receive charge: Any.
```

The `creditCard` is a dependency for the shopping cart, so we pass it through a constructor method. Then we trigger the behavior we want to test by sending a `checkout` message to the shopping cart. And at the end we check our expected behaviour - credit card should not be charged by any amount because nothing exists in the cart.

And empty implementation of `checkout` method will pass the test:

```
ShoppingCart >> checkout
  "empty implementation"
```

Usually we want to start with a failing test because we want to be sure that the test is capable of failing, then we can write just enough production code pass the test. This is an exceptional case, so we'll keep an eye on this test and come back to this later.

10.8 One product, no discount

Now let's add one product id (we'll call it SKU which is an abbreviation of stock keeping unit) to the shopping cart and check the price being sent to the credit card. But to do so we need to know the price of the product. It would be helpful if there were an object that could find out the price of an item by its SKU. We can bring this object to existence with the help of the mock library.

Let's create a mocked `priceCatalog` and move the initialization of the shopping cart in the `setUp` method.

```
ShoppingCartTest >> setUp
  super setUp.
  priceCatalog := Mock new.
  creditCard := Mock new.
  cart := ShoppingCart catalog: priceCatalog payment: creditCard.
```

We converted the `cart` and `creditCard` temporary variables to instance variables because we are using them in both tests.

Now we can setup and use the `priceCatalog` without worrying about its internal structure of an actual catalog.

```
ShoppingCartTest >> testChargingCardAfterBuyingOneProduct
  (priceCatalog stub priceOf: #sku1) willReturn: 80.

  cart addSKU: #sku1.
  cart checkout.

  (creditCard should receive charge: 80) once.
```

This test will fail if:

- the message `charge:` is not sent to the `creditCard` during the test.
- the message `charge:` is sent more than once.
- the argument of the message `charge:` is different than 80.

This test gives us our first failure:

```
a Mock(creditCard) never received charge: 80
```

Notice that Mocketry complains only about the missing `charge:` message. It doesn't care about the `priceOf:`. Stubbed behaviour is only used to simulate reaction of system on particular messages. Stubs do not add any requirements to tested behaviour. If business rules will require to check catalog for every `checkout` operation then we will add extra assertion in our test. Or better we will add separate test to describe this aspect of system. But right now we don't care about it. Also imagine we're introducing a price caching mechanism in the shopping cart. So after we asked the price of a product we store it and next time we won't ask it again. This will not break our test. Charging the credit card twice would be a serious mistake and Mocketry will report this failure. But querying the product price twice is an implementation detail.

The following implementation passes the test.

```
ShoppingCart >> addSKU: aString
  sku := aString.
```

```
ShoppingCart >> checkout
  creditCard charge: (priceCatalog priceOf: sku).
```

This implementation isn't complete but it is just enough to pass the second test. We don't want to do more than it necessary to pass the test. This practice ensures high code coverage and helps keeping the implementation simple.

Now our first test is finally broken because we charge the credit card (with nil) even if the cart is empty. This can be easily fixed by a nil check.

```
ShoppingCart >> checkout
  sku ifNotNil:
    [creditCard charge: (priceCatalog priceOf: sku)]
```

10.9 Many products, no discount

The shopping cart that can hold only one product isn't particularly useful. The next test will force us to generalize our implementation.

```
ShoppingCartTest >> testChargingCardAfterBuyingManyProducts
  (priceCatalog stub priceOf: #sku1) willReturn: 10.
  (priceCatalog stub priceOf: #sku2) willReturn: 30.
  (priceCatalog stub priceOf: #sku3) willReturn: 50.
  cart addSKU: #sku1.
  cart addSKU: #sku2.
  cart addSKU: #sku3.

  cart checkout.

  creditCard should receive charge: 90.
```

This generates a failure message:

```
a Mock(creditCard) never received charge: 90
```

And in debugger you can see what messages was actually sent between objects. For this select top item on stack pane of debugger. And then select `subject` on variables pane. It will show single message:

```
a Mock(creditCard) charge: 50 returned default mock(73422)
```

The shopping cart keeps the last SKU only that's why it reports the wrong price. First we rename the `sku` instance variable to `skus` and intialize it to an empty `OrderedCollection`.

```
ShoppingCart >> initialize
  super initialize.
  skus := OrderedCollection new.
```

Then we replace the assignment to an `add:` message in the `addSKU:` method.

```
ShoppingCart >> addSKU: aString
  skus add: aString.
```

Finally we change the nil check to an empty check and add the `orderPrice` calculation.

```
ShoppingCart >> checkout
  skus ifNotEmpty:
    [ creditCard charge: (skus sum: [ :each | priceCatalog priceOf:
    each ])]
```

Refactoring is part of TDD. We should check after each test if there is anything we can do to make the test and the code better. The setup of the `priceCatalog` adds a lot of noise to the last test. Let's extract it to a private method.

```
ShoppingCartTest >> testChargingCardAfterBuyingManyProducts
  self prices: {#sku1 -> 10. #sku2 -> 30. #sku3 -> 50}.
  cart addSKU: #sku1.
  cart addSKU: #sku2.
  cart addSKU: #sku3.

  cart checkout.

  creditCard should receive charge: 90.
```

```
ShoppingCartTest >> prices: associations
  associations do: [:each |
      (priceCatalog stub priceOf: each key) willReturn each value].
```

Then extract the `orderPrice` calculation into a new method:

```
ShoppingCart >> checkout
  skus ifNotEmpty:
    [creditCard charge: self orderPrice]
```

```
ShoppingCart >> orderPrice
  ^ skus sum: [:each | priceCatalog priceOf: each]
```

10.10 Removing a product

The shopping cart should provide methods for removing a product. We don't want to test the internal state of the shopping cart, because it would make our tests brittle. We assume that the interface of the shopping cart is more stable than its implementation, so we want to couple our tests to the interface of the object under test. We can verify the behaviour by checking that a product removal is reflected in the price.

```
ShoppingCartTest >> testProductRemovalReflectedInPrice
  self prices: {#sku1 -> 30. #sku2 -> 70.}.
  cart addSKU: #sku1.
  cart addSKU: #sku2.
  cart removeSKU: #sku2.

  cart checkout.

  creditCard should receive charge: 30.
```

Making the test pass is fairly simple.

```
ShoppingCart >> removeSKU: aSymbol
  skus remove: aSymbol.
```

10.11 Products with discount price

We know that there are rules for calculating the discount price. Those rules take the total order amount into account. Luckily we don't need to worry about what exactly the rules are. Let's introduce a new mock that will play the role of the `DiscountCalculator`.

```
ShoppingCartTest > >setUp
  super setUp.
  priceCatalog := Mock new.
  discountCalculator := Mock new.
  creditCard := Mock new.
  cart := ShoppingCart catalog: priceCatalog discount:
    discountCalculator payment: creditCard.
```

```
ShoppingCartTest >> testChargingCardWithDiscount
  self prices: {#sku1 -> 20. #sku2 -> 30}.
  (discountCalculator stub calculateDiscount: 50) willReturn: 10.
  cart addSKU: #sku1.
  cart addSKU: #sku2.

  cart checkout.

  creditCard should receive charge: 40.
```

Mocketry complains that the `creditCard` was charged with an incorrect price (50 instead of 40). We need to subtract the `discountPrice` from the `orderPrice` to make the test pass.

```
ShoppingCart >> checkout
  | orderPrice discountPrice |
  skus ifNotEmpty:
    [orderPrice := self orderPrice.
    discountPrice := self discountPrice: orderPrice.
    creditCard charge: orderPrice - discountPrice]
```

```
ShoppingCart >> discountPrice: orderPrice
  ^ discountCalculator calculateDiscount: orderPrice
```

Now the last test passed.

Interesting that other tests will pass too while we are not mentioned `discountCalculator` there. In these tests `calculateDiscount:` message will be sent to mock object without explicit stub for it. In such cases (when stub behaviour is not specified) Mocketry returns new special mock as result of message. This mock plays role of zero in arithmetic operations and false in boolean logic. And in our case it will lead to zero result of `calculateDiscount:`. It is very usefull behaviour which makes tests less brittle and allows to specify only needed requirement for given aspect of functionality. In our first tests we don't care about discount and tests continue work without it.

10.12 Reusing the cart for different sales

We want the shopping cart to unload itself after the checkout so we'll able to reuse the same instance over and over. We'll simulate two separated purchase in the next test to verify this behaviour. This will trigger two `charge:` messages. We'll check if the correct price is sent when the shopping cart is in the correct state (`firstPurchase` or `secondPurchase`).

```
ShoppingCartTest>>testUnloadsItselfAfterCheckout
  |purchase|
  self prices: {#sku1 -> 10. #sku2 -> 30}.
  cart addSKU: #sku1.
  cart checkout.
  cart addSKU: #sku2.
  cart checkout.

  [creditCard receive charge: 10.
  creditCard receive charge: 30] should beDoneInOrder
```

`should beDoneInOrder` expression verifies that messages were sent and it was happen in same order as defined in given block.

The test generates a failure message:

```
a Mock(creditCard) never received charge: 30
```

And in debugger you can see actual messages:

```
a Mock(creditCard) charge: 10 returned default mock(73422).
a Mock(creditCard) charge: 40 returned default mock(34614).
```

But cleaning the `skus` collection in the `checkout` fixes the test.

```
ShoppingCart>>checkout
  | orderPrice discountPrice |
  skus ifNotEmpty:
    [orderPrice := self orderPrice.
    discountPrice := self discountPrice: orderPrice.
    skus removeAll.
    creditCard charge: orderPrice - discountPrice]
```

10.13 Testing errors during checkout

We expect the `creditCard` to report `PaymentError` when the payment was unsuccessful. When this happens, the shopping cart should not forget the content because we may want to retry the checkout later.

In the next test case we'll send two `checkout` message to the shopping cart, whereof the first will fail. The `creditCard` will signal `PaymentError` then answer `true` consecutively.

```
ShoppingCartTest>>testFirstPaymentFailedSecondSucceed
  self prices: {#sku -> 25}.
  cart addSKU: #sku.

  (creditCard stub charge: 25) willRaise: PaymentError new.
  [cart checkout] should raise: PaymentError.

  (creditCard stub charge: Any) willReturn: true.
  cart checkout.
  (creditCard should receive charge: 25) twice.
```

The failure report is:

```
a Mock(creditCard) should receive charge: 10 2 times but it was 1
    times
```

The test failed because we clean the `skus` collection before charging the credit card. So after the first checkout the shopping cart is empty, and the second time it won't charge the credit card.

Let's move the `skus removeAll` at the end, to make the test pass.

```
ShoppingCart>>checkout
  | orderPrice discountPrice |
  skus ifNotEmpty:
    [orderPrice := self orderPrice.
    discountPrice := self discountPrice: orderPrice.
    creditCard charge: orderPrice - discountPrice.
    skus removeAll]
```

10.14 What's next

We finished the shopping cart, so what's next? We can continue with either of its dependencies: `CreditCard`, `PriceCatalog`, `DiscountCalculator`.

For example we may want to use a simple in-memory object for the `price-Catalog` and implement its internals using a `Dictionary`. In this case a simple assertion based test would cover its behaviour.

```
PriceCatalogDictionaryTest >> testKnowsThePriceOfAproduct
  | catalog |
  catalog := PriceCatalogDictionary withPrices {#sku1 -> 10}.
  (catalog priceOf: #sku1) should equal: 10.
```

Another implementation might be an HTTP based remote price catalog which would use a REST web service to find out the price.

```
RemoteHttpPriceCatalog >> priceOf: sku
  | response |
  response := ZnClient new
    url: 'http://example.com/rest/sku/', sku;
```

```
    get;
    response.
  ^ self parseResponse: repsonse.
```

We could mock the http client to test the `RemoteHttpPriceCatalog` but it would result duplications and wouldn't improve our confidence in the code. Mocking a third-party code often causes brittle test and doesn't tell us too much about code itself (its design and correctness). So we recommended wrapping the third-party API with an adapter (See Alistair Cockburn Ports And Adapters Architecture) that will provide a high level of abstraction. Think about the adapter like a translator that translates messages between two different domains (`priceOf:` to http `get`). But here the `RemoteHttp-PriceCatalog` is already an adapter, introducing a new one would just delay the problem. We free to mock the adapter just as we did in the shopping cart tests, but we still need to test the `RemoteHttpPriceCatalog` somehow. We recommend testing the adapter itself with integration tests. In the contrast with unit tests (what we did before), integration tests don't test object in isolation. Writing a test like this would require firing up a local http server that could provide canned responses in an appropriate format.

We'll leave those as an exercise for the reader.

10.15 Features overview

Create mocks easily

To create mock just use the message `new`.

```
yourMock := Mock new.
```

Mocketry does not require any special context variables for this. Also Mocketry implements auto naming logic to retrieve variable names from test context. In the debugger, inside a test yourMock will be shown as "a Mock(yourMock)".

But if you need special name you can specify it:

```
Mock named: 'yourMock'
```

You can look at it live in MockTests.

Multiple Mocks at once

Also there is way to get multiple mocks at once:

```
[ :mockA :mockB | "your code here" ] runWithMocks
```

Stub any message sends

To stub a message send just send message **stub** to an object and the following message send will create an expectation.

> **To do** I not understand why following example is not enough to demonstrate previous sentence. I would stay only example

Once you have a stub object and send it the message `stub`, you can then send it messages such as `willReturn:` and `should be:` to verify certain condition as follows:

```
mock := Mock new.
mock stub someMessage willReturn: 100.
mock someMessage should be: 100.
```

You can stub any objects. It is not only about mocks:

```
rect := 0@0 corner: 2@3.
rect stub width willReturn: 1000.

rect area should be: 3000 "area = width * height"
```

Note that `stub` message activates, message interception for real objects. Without it following sentence will not work.

```
rect area should be: 300.
```

10.16 Stubbing classes

And you can do this with globals and classes too but this can be dangerous.

```
DateAndTime stub now willReturn: #constantValue.

DateAndTime now should be: #constantValue.
```

But you should be carefull with classes and globals. Don't try

```
Array stub new.
```

It will crash image because stub performs some black magic so that the object understands the mocking messages and `Array` is not a class that can be modified because it is central to the execution engine.

Now for the classes that are stubbed, you can turn them back to their default status (we call this behavior to recover). So if you stub a class from a workspace, it is your responsibility to *recover* it from stub behaviour. Do it using the message `recoverFromGHMutation` as follows:

```
DateAndTime recoverFromGHMutation.
```

In case when you stub a global inside test Mocketry will automatically recover all global stubs when test completes.

10.17 Expectations for set of objects

Also with Mocketry you can define expectations for set of objects. For example you can stub message to **ANY object**:

```
Any stub width willReturn: 100.
```

Or you can stub **ANY message** to particular object:

```
mock := Mock new.

mock stub anyMessage willreturn:: 100.

mock someMessage should be: 100.
mock someMessage2 should be: 100.
```

And both variants are supported:

```
Any stub anyMessage willReturn: 100.

mock := Mock new.
mock someMessage should be: 100.

rect := 0@0 corner: 2@3.
rect stub.

rect area should be: 100.
rect width should be: 100.
```

Any class is a specific object specification which means ”any” object. You can use any kind of specifications:

```
(Kind of: Rectangle) stub width willReturn: 100.

rect := 0@0 corner: 2@3.
rect stub.

rect area should be: 300.

rect2 := 0@0 corner: 4@5.
rect2 stub.

rect2 area should be: 500
```

10.18 Stub message sends with arguments

In place of message arguments you can use expected objects itself. Or you can put specifications for expected arguments:

```
mock stub messageWith: arg1 and: arg2
mock stub messageWith: Any and: arg2
mock stub messageWith: [:arg | true]
mock stub messageWith: (Kind of: String) and: arg2
mock stub messageWith: (Instance of: Float) & (Satisfying for: [:arg
    | arg > 10]).
```

Last defined expectation has more priority than previous one. It allows you to define default expectations in `setUp` methods and override it in particular tests. Following example shows it:

```
mock := Mock new.
(mock stub messageWith: (Instance of: SmallInteger)) willReturn:
    #anyInt.
(mock stub messageWith: (Kind of: String)) willReturn: #anyString.
(mock stub messageWith: 10) willReturn: #ten.

(mock messageWith: 10) should be: #ten.
(mock messageWith: 20) should be: #anyInt.
(mock messageWith: 'test' should be: #anyString
```

Expected actions for stubs

There are different kind of expected actions

To do look.

Return value from message

```
mock stub someMessage willReturn: #result.
mock someMessage should be: #result.
```

Raise error on message send

```
mock stub someMessage willRaise: ZeroDivide new.
[mock someMessage] should raise: ZeroDivide.
```

Evaluate block on message send and return it result

```
(mock stub someMessageWith: #arg) will: [ #result ].
(mock someMessageWith: #arg) should be: #result.
```

Given block can accept argument of message in original order:

```
(mock stub someMessageWith: #arg1 and: #arg2) will: [:arg1 :arg2 |
    arg1, arg2].
(mock someMessageWith: #arg1 and: #arg2) should equal: 'arg1arg2'.
```

Return values for subsequent series of messages

```
mock stub someMessage willReturnValueFrom: #(result1 result2).
mock someMessage should be: #result1.
mock someMessage should be: #result2
```

Return receiver of message

```
mock stub someMessage willReturnYourself.
mock someMessage should be: mock.
```

Extra conditions on message sends

It is possible to verify arbitrary condition when expected message is going to be executed. For example:

```
mock := Mock new.
mock stub someMessage
  when: [ flag ] is: (Kind of: Boolean);
  when: [ flag ] is: true;
  when: [ flag ] satisfy: [ :object | true or: [ false ] ].

flag := true.
mock someMessage. "does not fail"

flag := false.
mock someMessage "will fail immediately on call by last condition:
    flag should be true"

flag := #flag.
mock someMessage "will fail immediately on call by first condition:
    flag should be boolean"
```

To do I do not get the flag should be boolean. We should discuss it.

Process condition

Mocketry implements process related condition to check that a message was synchronously sent (relative to test process). Using the messages `shouldBeSentInThisProcess` and `shouldBeSentInAnotherProcess`, you can control the process where the execution of the test should happen.

```
mock stub someMessage shouldBeSentInThisProcess.
[ mock someMessage ] fork. "will fail immediately on call".

mock stub someMessage shouldBeSentInAnotherProcess.
[ mock someMessage ] fork. "will not fail".
mock someMessage. "will fail immediately on call"
```

Message sends usage rules

It is possible to specify how many times expectation can be used using the messages `use: useTwice` and `useOnce`.

```
mock := Mock new.

mock stub someMesage willReturn: #default.
mock stub someMessage willReturn: 300; use: 3.
mock stub someMessage willReturn: 200; useTwice.
mock stub someMesage willReturn: 100 useOnce.
```

Note that in case of conflict the last defined expectation takes precedence over previously defined ones.

```
mock someMessage should be: 100.

mock someMessage should be: 200.
mock someMessage should be: 200.

mock someMessage should be: 300.
mock someMessage should be: 300.
mock someMessage should be: 300.

mock someMessage should be: #default
```

10.19 Unexpected messages: Automocks

Mock returns another special mock for unexpected messages (when no expectation is defined for received message):

```
mock := Mock new.

automock := mock someMessage.

automock should beInstanceOf: MockForMessageReturn.
```

And any message to this mock will produce another automock. It means that your tests will not fail if you will not define any expectation for your mocks. It allows you put only required detailes inside your tests which really make sense for tested aspect of functionality. Anything else does not matters.

Also to improve this idea automock try to play role of false in boolean expressions.

```
mock := Mock new.
returnedMock := mock someMessage.

result := returnedMock ifFalse: [ #falseBranch ] ifTrue: [
    #trueBranch ].

result should be: #falseBranch.
returnedMock should be: false
```

And play zero in arithmetic

```
mock := Mock new.
returnedMock := mock someMessage.

result := 1 + returnedMock.
result should equal: 1.
returnedMock should equal: 0
```

10.20 Stub group of message sends

There is way to stub multiple message sends at once:

```
mock := Mock new.
rect := 0@0 corner: 2@3.
rect stub.

[ mock someMessage willReturn: 10.
rect width willReturn: 1000 ] should expect.

mock someMessage should be: 10.
rect area should be: 3000.
```

Inside ”should expect” block you don’t need to send **extra #stub** message to objects

10.21 Verify message sends

With Mocketry you can check that particular object received particular message. Use **”should receive**” expression for this:

```
mock := Mock new.

mock someMessage.

mock should receive someMessage.
mock should not receive anotherMessage
```

You can verify that message was send to real objects. It is not only about mocks:

```
rect := 0@0 corner: 2@3.

rect stub "it should be here to enable message interception"
rect area

rect should receive width. "area = width * height"
```

And you can do this with globals too:

```
DateAndTime stub.
DateAndTime midnight.

DateAndTime should receive now. "inside midnight #now is called"
```

But you should be carefull with globals. Look at section 10.15.

Also with Mocketry you can verify that message was sent to set of objects. For example you can verify that message was sent to **ANY object**:

```
mock := Mock new.
rect := 0@0 corner: 2@3.
rect stub.

mock width.
rect area.

Any should receive width. "it will check that mock and rect received
    message #width"
Any should receive area "it will fail because mock not received
    #area message".
```

Also you can verify that **ANY message** was sent to particular object:

```
mock := Mock new.

mock someMessage should be: 100.

mock should receive anyMessage.
```

And both variants are supported:

```
mock := Mock new.
rect := 0@0 corner: 2@3.
rect stub.

mock someMessage.

Any should receive anyMessage. "will fail because rect not received
    any message".
```

```
rect width.

Any should receive anyMessage. "will not fail because both objects
    received at least one message"
```

Any class is specific object spec which means ”any” object. You can uses any kind of specs to verify message send for set of objects:

```
rect := 0@0 corner: 2@3.
rect stub.

rect area.

rect2 := 0@0 corner: 4@5.
rect2 width.

(Kind of: Rectangle) should receive width. "will not fail because
    both rect's received message #width"
(Kind of: Rectangle) should receive area "will fail because rect2
    not received message #area"

mock := Mock new.
(Kind of: Rectangle) should receive width. "will not fail because
    mock is not kind of Rectangle"
```

10.22 Verify message sends with arguments

In place of message arguments you can use expected objects itself. Or you can put specifications for expected arguments:

```
mock := Mock new.

(mock messageWith: 10) should be: #ten.
(mock messageWith: 'test' should be: #anyString.

mock should receive messageWith: 10.
mock should receive messageWith: (Instance of: SmallInteger).
mock should receive messageWith: 'test'.
mock should receive messageWith: (Kind of: String).
mock should receive messageWith: [:arg | arg isNumber].
```

10.23 Capture message arguments

Mocketry provides suitable tool to capture arguments of messages for subsequent verification:

```
mock := Mock new.
mock someMessageWith: Arg argName.

mock someMessageWith: #argValue.

Arg argName should be: #argValue.
```

An argument specification can play the role of any object. So it does not restrict message send expectation. Capture will store all received argument values. To verify concrete arguments use message `fromCall:`.

```
Arg argName fromFirstCall should be: #value1.
Arg argName fromLastCall should be: #value3.
(Arg argName fromCall: 2) should be: #value2.
```

Short version:

```
Arg argName should be: #argValue.
```

will signal error if there are multiple different captured values.

Also ”should” expression on capture will verify that owner message send occurred required number of times.

When argument is captured it value is stubbed. It allows you to verify subsequent message sends to captured arguments:

```
mock stub someMessageWith: Arg rectangle.

rect := 0@0 corner: 2@3.
mock someMessageWith: rect.
rect area.

Arg rectangle should be: rect.
Arg rectangle should receive width.
```

10.24 Verify message sends count

Mocketry allows one to verify how many times object received particular message:

```
mock := Mock new.

mock someMessage.
mock should receive someMessage once.

mock someMessage.
mock should receive someMessage twice.

mock someMessage.
mock should receive someMessage exactly: 3.
mock should receive someMessage atLeast: 2.
```

```
mock should receive someMessage atMost: 3.
mock should receive someMessage atLeast: 1 atMost: 5.
```

Same works to verify that set of objects received particular message expected number of times:

```
mock := Mock new.
mock2 := Mock new.

mock someMessage; someMessage.
mock2 someMessage.

Any should receive someMessage twice. "will fail because mock2
    received #someMessage only once"

mock2 someMessage.
Any should receive someMessage twice. "will not fail because both
    mocks received #someMessage twice"
```

10.25 Verify message send result

There are two ways how to verify result of occurred message:

First you can continue `should receive` expression with `which should` clause to validate actual returned value:

```
rect := 0@0 corner: 2@3.
rect stub.

rect area.

rect should receive area which should equal: 6.
rect should receive width which should beKindOf: Number
```

And you can validate sender message of any object:

```
mock := Mock new.
result := mock someMessage.
result should beReturnedFrom: [ mock someMessage ].
```

10.26 Verify group of message sends

There is way to verify group of message sends at once:

```
mock := Mock new.
rect := 0@0 corner: 2@3.
rect stub.

mock someMessage.
rect area.
```

```
[ rect width.
mock someMessage ] should beDone.

[ mock someMessage.
rect width ] should beDoneInOrder.
```

Messages

- **beDone** measn that we don't care about order of message sends.
- **beDoneInOrder** verifies that messages were set in same order as they defined inside given block

10.27 Verify all expectations

There is way how to verify that all defined expectations were occurred:

```
mock1 := Mock new.
mock2 := Mock new.

[ mock1 someMessage. mock2 someMessage2 ]
   should lenient satisfy:
[ mock2 someMessage2.
mock1 someMessage willReturn: 'some' ].
```

- **lenient** means that we don't care about order in which expected messages were happened.

```
mock1 := Mock new.
mock2 := Mock new.

[ mock1 someMessage. mock2 someMessage2 ]
   should strictly satisfy:
[ mock1 someMessage willReturn: 'some'.
mock2 someMessage2 ].
```

- **strictly** means that we want expected messages were happened in same order in which they were defined.

10.28 Conclusion

Mocketry is a powerful system and we hope that it will help you to improve your tests and development.

CHAPTER 11

Object validation with StateSpecs

Chapter contribution by Denis Kudriashov.

StateSpecs is object state specification framework developed by Denis Kudriashov. it describes particular object states with first class specifications.

StateSpecs introduces two little DSLs: should expression and word classes. Should expressions were originally invented by Dave Astels in project SSpec as part of general rethinking the Testing Driven Development (TDD) methodology in favor of Behavior Driven Design (BDD). SSpec approach has been ported to many different languages (NSpec in C#, RSpec in Ruby for example).

StateSpecs provides fluid DSL to validate objects over these specification.

StateSpecs offers different kind of validations. For the classes `SpecOfCollectionItem`, `SpecOfObjectClass`, and `SpecOfObjectSuperclass` verify different properties of an object.

11.1 How to load StateSpecs

You can load StateSpecs using the following expression.

```
Metacello new
  baseline: 'StateSpecs';
  repository: 'github://dionisiydk/StateSpecs';
  load.
```

11.2 Basic

Specifications can match and validate objects. In case an object does not satisfy a specification you will get failure result with detailed descriptiong about the problem. For example the following snippet use `SpecOfObjectClass` and validate objects.

We create a specification stating that we expect a small integer and we validate whether different objects verify it.

```
spec := SpecOfObjectClass requiredClass: SmallInteger.
spec validate: 10.
>>> a SpecOfValidationSuccess

spec validate: 'some string'.
>>> a SpecOfValidationFailure(Got 'some string' but it should be an
    instance of SmallInteger)
```

Instead of validation you can simply match objects with a specification using the `matches:` predicate.

```
spec matches: 10.
>>> true
```

```
spec matches: 'string'.
>>> false
```

Specifications can be negated using the message `not`.

```
spec not validate: 10.  "==> a SpecOfValidationFailure(Got 10 but it
    should not be an instance of SmallInteger)"
spec not validate: 'some string'. "==> a SpecOfValidationSuccess"
```

The message `not` creates a new spec instance. You can also negate the current one with the message `invert`.

11.3 Two little DSLs

To easily create specifications and validate objects using them, StateSpecs provides two DSLs: `should` expressions and ”word” classes.

The first allows you to write ”assertions” of the following form:

```
x should be: 2
y should equal: 10
```

Such expressions verify that the receiver follows the constraints they represent.

The ”word” classes DSL allows you to instantiate specifications using natural readable words. They are builders that return expectation objects.

```
Kind of: Number.
>>> a SpecOfObjectSuperclass(should be a kind of Number)
Instance of: String.
>>> a SpecOfObjectClass(should be an instance of String)
Equal to: 'test'.
>>> a SpecOfEquality(should equal 'test')
```

Word classes were introduced to get fluid interface for mock expectations like for example stating that the results of a given message should be a string. But they are very handy shorcuts to access specifications in general. The same word can return different specifications in different expressions which allows very fluid instantiation interface:

```
Equal to: 'test'.
>>> a SpecOfEquality(should equal 'test')

Equal to: 10.0123 within: 0.01.
>>>  a SpecOfApproxEquality(should be within 0.01 of 10.0123)
```

11.4 Should expressions

Should expressions were created with the goal to replace SUnit assertions (`self assert: a equals: b`). The implementation of should expression creates specific specifications and it verifies that receiver satisfies it. When an object is not valid, a should expression signals `SpecOfFailed` error. Such validation error can be inspected in the debugger to analyze and understand the reason.

Sending a `not` message to a `should` expression negates the logic of following expression:

```
3 should not equal: 3.
>>> fail with message: Got '3' but it should not equal '3'
```

To explore complete set of expressions look at the class `SpecOfShould-Expression`. It is also place where to extend them. The test class `SpecOf-ShouldExpressionTests` describes them using tests.

Specification of object identity

The message `be:` is used to verify that receiver is identical to a given argument: The following two expressions will fail, obviously.

```
1 should be: 2.
1 should not be: 1.
```

Specification of object equality

The `equal:` message is used to verify that receiver is equivalent to given argument: The following two expressions will fail, obviously.

```
3 should equal: 2.
3 should not equal: 3.
```

11.5 About equality for specification

In many languages, the equality operation = is redefined by many classes according to their domain logic. This is a problem because such redefinition may not be adapted from the specification point of view. Imagine that we want compare two different types of collections:

```
#(1 2 3) asOrderedCollection = #(1 2 3).
>>> false
```

It returns false which is correct from the point of view of collection library. But what should we expect from a specification perspective? We could express like the following snippet:

```
#(1 2 3) asOrderedCollection should equal: #(1 2 3).
```

But it is not fully adapted because it forces us to always think about collection types when we would like assert their equality. In fact we are supposed to assert collection *items* with this expression and not necessarily instances of collections.

In StateSpecs we define the equality specification using a specific message named `checkStateSpecsEqualityTo:` instead of `#=`.

The idea is that general equality specification should be as much simple equality as possible. And if you want some extra details you should use a different explicit specification which describes them. In case of collections you should check for collection class explicitly if it is important for your business case where you use specification:

The following snippet illustrates the point: the messahe `beInstanceOf:` will fail.

```
actual := #(1 2 3) asOrderedCollection.
expected := #(1 2 3).

actual should beInstanceOf: expected class.
actual should equal: expected.
```

Following this logic, StateSpecs does not check order when comparing basic collection classes:

```
#(1 2 3) should equal: #(2 1 3).
>>> true
#(1 2 3) asSet should equal: #(2 1 3).
>>> true
```

When the order is important, use the message `equalInOrder:` as illustrated below:

```
#(1 2 3) asOrderedCollection should equalInOrder: #(2 1 3).
>>> Fails #(1 2 3)" but it should equal in order to "#(2 1 3)
```

11.6 Specific case of String and ByteArray

There are collection classes such as `String` or `ByteArray` whose order and type are important. For them theses properties are always taken into account for equality comparison. The two following expressions fail:

```
'123' should equal: #($1 $2 $3).
>>> Fails
```

It fails with message: Got '123' but it should equal "#($1 $2 $3)".

```
'123' should equal: '132'.
```

It fails with message: Got '123' but it should equal '132'.

11.7 Floats

Floats are another example where specification behaves differently then standard language comparison:

```
0.1 + 0.2 = 0.3
>>> false"
```

It is correct result because of rounding errors in float arithmetics. But it is completelly not suitable to be part of specification. So in StateSpecs following expression will succeed:

```
(0.1 + 0.2) should equal: 0.3
>>> true
```

Float implements the method `checkStateSpecsEqualityTo:` by comparing numbers with default accuracy.

In addition, the message `equals:within:` should be used for special specifications for floats when concrete accuracy is important:

```
10.123 should equal: 10.1 within: 0.1
>>> true
10.123 should equal: 10.1 within: 0.01 "
>>> Fails
```

Last expression fails with message: Got 10.123 but it should be within 0.01 of 10.1.

The same logic is used by equality specification of Point class.

11.8 Specification of class relationship

StateSpec offers also a way to express class validation with two different messages: `bekindOf:` and `beInstanceOf:`.

```
3 should beKindOf: String
>>> Fails
```

It will fail with the message: Got 3 but it should be a kind of String.

```
3 should beInstanceOf: String
>>> Fails
```

It fails with the message: Got 3 but it should be an instance of String.

11.9 Collection Specifications

StateSpec proposes specific messages to handle different facets of collection.

To specify the size of an expected collection, use the `haveSize:` message:

```
#(1 2) should haveSize: 10
>>> Fails
```

It fails with message: Got #(1 2) but it should have 10 elements.

There is a simple expression for checking empty collections. It uses the predicate syntax explained below in Section 11.13.

```
#(1 2) should be isEmpty
>>> Fails
```

It fails with message: #(1 2) should be isEmpty.

To require concrete item in collection use one of `include:` messages:

```
#(1 2) should include: 10.
>>> Fails
```

Index position

It fails with message: Got #(1 2) but it should include 10.

```
#(1 2) should include: 10 at: 1
>>> Fails
```

It fails with message: Got 1 at key 1 of #(1 2) but should equal 10.

Note that the argument of the `include` messages can be specification itself:

```
#(1 2) should include: (Instance of: String) at: 1.
>> Fails
```

It fails with message: Got 1 at key 1 of #(1 2) but should be an instance of String.

```
#(1 2) should include: (Instance of: String)
>>> Fails
```

It fails with message: Got #(1 2) but should include (be an instance of String).

The following expression will succeed without error:

```
#(1 2) should include: [:each | each > 1]
```

To specify expected key in dictionary, use the `includeKey:` message:

```
{ #key1 -> #value1 } asDictionary should includeKey: #key2
```

It fails with message: Got a Dictionary{#key1->#value1} but it should include key #key2

11.10 String Specifications

To specify the substring of expected string use `includeSubstring:` message: The following fails with message: Got 'some test string' but it should include 'test2'.

```
'some test string' should includeSubstring: 'test2'
>>> Fails
```

Prefix and suffix

To specify prefix of expected string use `beginWith:` message. The following fails with message: Got 'string for test' but it should begin with 'test':

```
'string for test' should beginWith: 'test'
>>> Fails
```

To specify suffix of expected string use `endWith:` message. The following fails with message: Got 'test string' but it should end with 'test':

```
'test string' should endWith: 'test'
>>> Fails
```

To specify regex expression which expected string should satisfy use `matchRegex:` message:

```
'string for test' should matchRegex: '^test'
>>> Fails
```

It fails with message: Got 'string for test' but it should match regex '^test'.

By default all these specifications validate strings ignoring case. If you want case sensitive specs just add `caseSensitive: true` keyword to all examples:

```
'some test string' should includeSubstring: 'Test' caseSensitive:
    true
'test string' should beginWith: 'Test' caseSensitive: true
'string for test' should endWith: 'Test' caseSensitive: true
'test string' should matchRegex: '^Test' caseSensitive: true
```

11.11 Raising exception

Specifying the signalling of exceptions is important. StateSpecs proposes the `raise:` message to specify expected failure of given block: The following fails with message: Got no failures but should be an instance of ZeroDivide.

```
[1 + 2] should raise: ZeroDivide.
>>> Fails
```

```
[1/0] should raise: ZeroDivide.
>>> true
```

The following snippet fails with message: Got ZeroDivide but it should be an instance of Error.

```
[1/0] should raise: Error.
```

The following does not fail because `ZeroDivide` is kind of `Error`.

```
[1/0] should raise: (Kind of: Error).
```

In addition, you can use instance of expected exception instead of class. For example, the following snippet does not fail.

```
| errorInstance |
errorInstance := Error new messageText: 'test error'.
[ error signal ] should raise: errorInstance
```

But the following fails with message: Got "Error: another error" but it should equal "Error: test error".

```
[self error: 'another error'] should raise: errorInstance
```

11.12 The fail message

StateSpec proposes a simple message `fail` to expect general failure:

```
[ 1 / 0 ] should fail
```

The previous snippet does not fail beause block is really failed as expected. The following one fails with message: Got `ZeroDivide` but it should not be a kind of `Error`.

```
[ 1 / 0 ] should not fail
>>> Fails
```

```
[ 1 + 2 ] should fail.
```

It fails with message: Got no failures but should be a kind of Error.

11.13 Predicate syntax

In many cases the only thing that we want to specify is some boolean state of objects using their own methods. For this purpose the special `SpecOfBooleanProperty` specification is available. It should be created with a given boolean message as follows:

```
| spec |
spec := SpecOfBooleanProperty fromMessage: (Message selector:
    #isEmpty)
spec validate: #()
>>> true
```

A message can include arguments as follows:

```
spec := SpecOfBooleanProperty fromMessage: (Message selector:
    #between:and: arguments: #(1 10))
spec validate: 5
```

Explaining object properties

One of the last feature of StateSpecs is ability to explain why a given object does not validate a specification. Imaging that we want to validate the x coordinate of a rectangle' origin. It could be done like this:

```
(1@3 corner: 20@1) origin x should equal: 100
>>> Fail
```

Such expression fails with message: "Got 1 but it should equal 100". However, it is unclear which exact property of the rectangle is wrong.

StateSpecs introduces the message `where` which should be sent to receiver and all following messages up to `should` will be recorded as object property. At the end a should expression will validate the retrieved property instead of receiver itself.

```
(1@3 corner: 20@30) where origin x should equal: 100.
```

It fails with message: Got 1 from (1@3) corner: (20@30) origin x but it should equal 100.

11.14 Conclusion

StateSpecs is flexible object validation framework. StateSpecs is a basic brick of other libraries such as Mocketry.

CHAPTER 12

Performance testing with SMark

Chapter Contributors: Juan P. Sandoval A.

Measuring performance is not an easy activity. It involves many considerations that you need to take into account such as the pertinence of what you want to measure, how you can isolate it from other computations, ... Indeed an execution may be affected by different factors, for instance, the hardware you have, the VM you are using, the run-time optimizations that the VM perform while your program is running, the unpredictable garbage collection, among many others. All these factors combined make it difficult to have a good estimation of the performance of an execution.

This chapter introduces SMark, a testing framework for benchmarking Pharo applications. SMark is developed and maintained by Stephan Marr. SMark encapsulates a number of benchmarking best practices which are useful to take into account these factors and help us to have a decent execution time estimation.

12.1 Installing SMark

You can install SMark with the following incantation:

```
Metacello new
    baseline: 'SMark';
    repository: 'github://smarr/SMark';
    load.
```

The examples in this chapter were performed in Pharo 8, but SMark is supported since Pharo 7.

12.2 Measuring execution time

Measuring execution time needs to be done carefully, otherwise we can get a very far estimation of it. To illustrate the extent of this situation, we will use the recursive fibonacci function as a subject under analysis. Define a new class named `Math` and the following method:

```
Math >> fib: n
  ^n <= 1
    ifTrue: [ n ]
  ifFalse: [ (self fib: (n - 1)) + (self fib: (n - 2)) ]
```

We now will measure how many seconds take to compute the 40th fibonacci, using the message `timeToRun` method of the BlockClosure class as follows:

```
[ Math new fib: 40 ] timeToRun
```

The method `timeToRun` basically records the system clock time before and after the execution of the code block. We execute this script 5 times, you may found results of each execution in the following lines:

```
1 execution: 1091 milliseconds
2 execution: 1046 milliseconds
3 execution: 1098 milliseconds
4 execution: 1069 milliseconds
5 execution: 1085 milliseconds
```

As you can see, even in this small example, there is a variation between the results in the five iterations. Even though the measurements were done on the same computer, in the same Pharo Image. What happened? As we mentioned in the introduction the VM and the Hardware perform plenty of activities at the moment to execute a piece of code, and some of these activities are not deterministic, for instance, without going to far, some collections like the class Set that save objects without a defined (or let say, deterministic) order.

For this reason, it is necessary to consider a number of good practices to measure execution time, and minimize the measurement bias and uncertainty. For instance, execute the benchmarks a number of times before starting our measurements to let the VM to perform the dynamic optimizations. Another recommendation is to execute multiple times the benchmark, use the media as a point of reference, and consider the error margin. These recommendations may vary depending on the VM you are using, and the characteristics of your benchmark. For instance, it is well known that time measurements have more variations in micro-benchmarks than macro-benchmarks.

But, don't worry, SMark takes many of these recommendations into account to help you in executing your benchmarks and get a decent execution time estimation.

12.3 A first benchmark in SMark

SMark has been designed to run benchmarks in a similar fashion than we run tests. Therefore, their creation is quite similar too. First, we need to create a subclass of the class `SMarkSuite`.

```
SMarkSuite subclass: #MyBenchSuite
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SMark-Demo'
```

Then, we need to write the piece of code on which we are interested to measure the execution time (the benchmark), in our case, we will use our fibonacci benchmark that we introduced in the previous section.

```
MyBenchSuite >> benchFib40
  self fib: 40.
```

To execute our benchmark, we only need to execute the following script in a playground.

```
MyBenchSuite runOnly: #benchFib40
```

If you print the script answer it shows a small report like this one:

```
Report for: MyBenchSuite
Benchmark Fib40
Fib40 total: iterations=1 runtime: 1099ms
```

As you can see, the execution time is similar to the ones we got in the previous section, but now we can use SMark to execute multiple times the benchmark and get a better time estimation.

```
MyBenchSuite runOnly:#benchFib40 iterations: 25
```

And we got the following report:

```
Report for: MyBenchSuite
Benchmark Fib40
Fib40 total: iterations=25 runtime: 1022.7ms +/-1.5
```

There you go, now we have the average execution time and error margin, the error margin measure with a 90% of confidence degree assuming that the variable distribution is normal.

12.4 Setup and teardown

Similarly to SUnit, SMark offers two hook methods to define the actions to be done before and after a benchmark is executed.

12.5 SMark benchmark runners

The way that SMark will run benchmarks depends on the Smark runner you use. There are five different runners:

- `SMarkRunner`
- `SMarkAutosizeRunner`,
- `SMarkCogRunner`,
- `SMarkProfileRunner`, and
- `SMarkWeakScalingRunner`.

Each runner provides a particular way to perform the benchmarks. You may specify which runner you want to use by overriding the following method.

```
SMarkSuite class >> defaultRunner
  ^self onCog: [ SMarkCogRunner ] else: [ SMarkRunner ]
```

The message `defaultRunner` returns a CogRunner or a normal Runner depending of the VM on which the benchmark is executed. Next subsection describes each one of these runners.

SMarkRunner. It is the standard way to run a benchmark provided by SMark. `SMarkRunner` only performs the benchmark N times, where N is the number of iterations defined by the user, as we saw in the previous section. It records the time measurements of each execution to build a report.

SMarkCogRunner. It adds a warning up instructions before executing the benchmark. `SMarkCogRunner` executes a benchmark twice before taking the measurements. The first execution is to execute the inline cache optimization done by the VM. The second execution is to trigger the JIT compiler to produce code. Once these two executions are performed the benchmark is executed N times in similarly as SMarkRunner does.

SMarkAutosizeRunner. It increases the execution time of the benchmark to reach a targetTime, this is used mostly with micro-benchmarks. The default targetTime is 300 milliseconds. Therefore, if a benchmark takes less than 300 milliseconds to run, this runner will measure how many times this benchmark needs to be executed to meet the targetTime. Once the benchmark execution time meets the targetTime, this runner executes this increased benchmark N iterations to take the time measurements similar to SMarkRunner.

SMarkProfileRunner. It automatically opens and uses Pharo standard time profiler to monitor the execution of all the benchmarks in the suite. Then, the time profilers show the call context tree of this execution reporting which methods were executed by the benchmarks, and how much time each method consumes.

SMarkWeakScallingRunner. It is specific to platforms with support for real parallelism. Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

12.6 Benchmark suites

Having good benchmark already packaged and identified is important to be able to compare changes for example in the compiler or other aspects of execution. SMark provides four benchmarks suites:

- *SMarkCompiler*, a benchmark that measures the time needed to compile a regular size method.
- *SMarkLoops*, a set of microbenchmarks measuring a number of basic aspects such as message send, instance field access, and array access cost.
- *SMarkSlopstone*, a Smalltalk Low level OPeration Stones Portable Low Level Benchmarks for ST80 and ST/V (using 16-bit SmallIntegers) - Placed in public domain January 1993 (c) Bruce Samuelson Permission is given to place this in public Smalltalk archives.
- *SMarkSmopstone*, a Smalltalk Medium level OPeration Stones Portable Medium level Benchmarks for ST80 and ST/V (using 16-bit SmallInts) Placed in public domain January 1993 (c) Bruce Samuelson Permission is given to place this in public Smalltalk archives.

You may run any of these benchmark suites by performing the class method run of any of the previous clases. For instance:

```
SMarkLoops run
```

12.7 Result reports

SMark provides an option to export the benchmarking results using the class `SMarkSimpleStatisticsReporter`. For instance, consider the following example:

```
| stream result textReport|
result := SMarkLoops run.
stream := TextStream on: String new.
SMarkSimpleStatisticsReporter reportFor: result on: stream.
testReport := stream contents.
```

It runs the benchmark suite SMarkLoops and uses the class `SMarkSimpleStatisticsReporter` to export the results in a stream, in this particular case on a TextStream. This report is particularly useful to integrate SMark in a continuous integration environment.

12.8 Conclusion

This chapter introduces SMark, a framework to help you define, execute and report your benchmarks. It provides different ways to run your benchmarks, collect the results and perform a simple statistical analysis to measure the error margin. In addition, SMark provides four benchmark suites that are useful to test the performance of a number of core functionalities in Pharo.

CHAPTER 13

Miscellaneous

In this little chapter we present some little points around tests.

13.1 Executable comments

An important problem that developers are facing is how to document their code. Often they would like to provide a little example to help their users. Unit Tests are already a good foundation for this.

Since Pharo 7.0 you can add to your method comment, executable expressions. This way a user can execute an expression and see its results. Since Pharo 8.0 and the introduction of DrTests executable comments are crawled by the DrTests plugin and report the executable comments that are not working.

This way you gain on multiple levels:

- you get more user friendly and self explainable comments,
- you get the insurance that your comments are correct,
- you get tests that your code is doing what you specified.

13.2 Executable comment example

To define an executable comment you should use the message >>> to separate an expression from its results. Here is an example `'a' asByteArray >>> #[97]`

```
String >> asByteArray
  "Convert to a ByteArray with the ascii values of the string."
  "'a' asByteArray >>> #[97]"
  "'A' asByteArray >>> #[65]"
  "'ABA' asByteArray >>> #[65 66 65]"

  | b |
  b := ByteArray new: self byteSize.
  1 to: self size * 4 do: [:i |
    b at: i put: (self byteAt: i)].
  ^ b
```

Note that the message >>> is a real message, a binary message, so you have to pay attention when documenting other binary messages about the execution order. You can simply surround the first expressions with parantheses. Here we show `contractTo:` and there is no need to put extra parenthesis as in the example.

```
String >> contractTo: smallSize
  "return myself or a copy shortened by ellipsis to smallSize"
  "('abcd' contractTo: 10) >>> 'abcd'"
  "('Pharo is really super cool' contractTo: 10) >>> 'Phar...ool'"
  "('A clear but rather long-winded summary' contractTo: 18) >>> 'A
    clear ...summary'"

  | leftSize |
  self size <= smallSize
    ifTrue: [^ self].  "short enough"
  smallSize < 5
    ifTrue: [^ self copyFrom: 1 to: smallSize].    "First N
    characters"
  leftSize := smallSize-2//2.
  ^ self copyReplaceFrom: leftSize+1    "First N/2 ... last N/2"
    to: self size - (smallSize - leftSize - 3)
    with: '...'
```

DrTests plugin

Once you defined an executable comment, you can check that it is valid by comparing the results with the expected one. You can also use the DrTests plugin for executable comments.

13.3 Supporting examples with assertions

Some people wants to be able to have test methods (with assertions) that can call each others and without been forced that the method starts with the 'test' prefix. SUnit can be simply extended to support such example testing methods.

This is as simple as overriding the method `testSelectors` in a subclass of `TestCase`.

```
TestCase << #ExampleTest
  package: 'SUnit-Extensions'
```

Here we decided to only consider test methods when they are tagged with the pragma `#example`. In addition we check that the method should not expect an argument.

```
ExampleTest class >> testSelectors

  ^ self methods
    select: [ :each | (each hasPragmaNamed: #example) and: [ each
    selector isUnary ] ]
    thenCollect: [ :each | each selector ]
```

Here is an example taken from Bloc

```
simulateMouseMoveOutside
  <gtExample>
  | element mouseLeave mouseMove mouseEnter |

  element := self blueElement.
  element size: 100@100.
  element relocate: 100@100.

  mouseLeave := mouseMove := mouseEnter := 0.

  element addEventHandlerOn: BlMouseMoveEvent do: [ mouseMove :=
    mouseMove + 1 ].
  element addEventHandlerOn: BlMouseEnterEvent do: [ mouseEnter :=
    mouseEnter + 1 ].
  element addEventHandlerOn: BlMouseLeaveEvent do: [ mouseLeave :=
    mouseLeave + 1 ].

  BlSpace simulateMouseMoveOutside: element.

  self assert: mouseMove equals: 0.
  self assert: mouseEnter equals: 0.
  self assert: mouseLeave equals: 0.

  ^ element
```