CHAPTER 1

SUnit

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 focussed 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 (discussed below).

In this chapter we start by discussing why we test, and what makes a good test. We then present a series of small examples showing how to use SUnit. Finally, we look at the implementation of SUnit, so that you can understand how Pharo uses the power of reflection in supporting its tools. Note that the version documented in this chapter and used in Pharo is a modified version of SUnit3.3.

1.1 Introduction

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.

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 Smalltalk community has a long tradition of testing because of the

incremental style of development supported by its programming environment. In traditional Smalltalk development, the programmer would write tests in a workspace 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 workspace 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 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.

SUnit is valuable because it allows us to write tests that are 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 workspace, we encourage you to use SUnit and get all the advantages of stored and automatically executable tests.

1.2 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 at the same time as, 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.

1.3 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?

Now let's write our first test, and show you the benefits of using SUnit.

1.4 SUnit by example

Before going into the details of SUnit, 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.

Step 1: Create the test class

First you should create a new subclass of `TestCase` called `MyExampleSetTest`. Add two instance variables so that your new class looks like this:

Listing 1.1: An Example Set Test class

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

We will use the class `MyExampleSetTest` to group all the tests related to the class `Set`. It defines the context in which the tests will run. Here the context is described by the two instance variables `full` and `empty` that we will use to represent a full and an empty set.

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

Step 2: 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.

Listing 1.2: Setting up a fixture

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

In testing jargon the context is called the *fixture* for the test.

Step 3: write some test methods

Let's create some tests by defining some methods in the class `MyExampleSetTest`. 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.

Define the following test methods. The first test, named `testIncludes`, tests the `includes:` method of `Set`. The test says that sending the message `includes: 5` to a set containing 5 should return `true`. Clearly, this test relies on the fact that the `setUp` method has already run.

Listing 1.3: Testing set membership

```
MyExampleSetTest>>testIncludes
  self assert: (full includes: 5).
  self assert: (full includes: 6)
```

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.

Listing 1.4: Testing occurrences

```
MyExampleSetTest>>testOccurrences
  self assert: (empty occurrencesOf: 0) = 0.
  self assert: (full occurrencesOf: 5) = 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) = 1
```

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

Listing 1.5: Testing removal

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

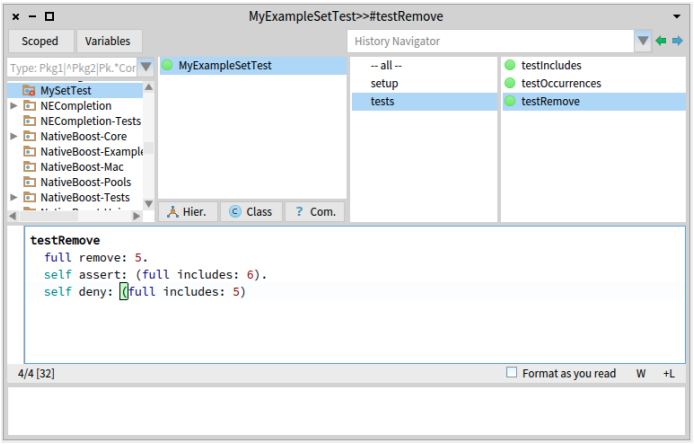

Figure 1.1: Running SUnit tests from the System Browser

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.

Step 4: Run the tests

The easiest way to run the tests is directly from the browser. Simply click on the icon of the class name, or on an individual test method, and select *Run tests (t)* or press the icon. The test methods will be flagged green or red, depending on whether they pass or not (as shown in 1.1).

You can also select sets of test suites to run, and obtain a more detailed log of the results using the SUnit Test Runner, which you can open by selecting `World > Test Runner`.

The *Test Runner*, shown in Figure 1.2, is designed to make it easy to execute groups of tests.

The left-most pane lists all of the packages that contain test classes (i.e., subclasses of `TestCase`). When some of these packages are selected, the test classes that they contain appear in the pane to the right. Abstract classes are italicized, and the test class hierarchy is shown by indentation, so subclasses of `ClassTestCase` are indented more than subclasses of `TestCase`. `ClassTestCase` is a class offering utilities methods to compute test coverage.

Open a Test Runner, select the package *MySetTest*, and click the `Run Selected` button.

You can also run a single test (and print the usual pass/fail result summary) by executing a *Print it* on the following code: `MyExampleSetTest run: #testRemove`.

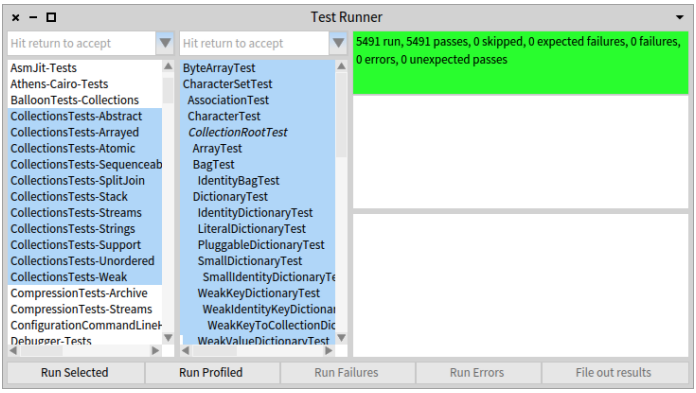

Figure 1.2: Running SUnit tests using the

Some people include an executable comment in their test methods that allows running a test method with a *Do it* from the browser, as shown below.

Listing 1.6: Executable comments in test methods

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

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

Listing 1.7: Introducing a bug in a test

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

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
```

Step 5: Interpret the results

The method `TestCase>>assert:`, which is defined in the class `TestCase`, 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.

1.5 The SUnit cook book

This section 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.

Other assertions

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

First, `TestCase>>assert:description:` and `TestCase>>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 1.7.

Next, SUnit provides two additional methods, `TestCase>>should:raise:` and `TestCase>>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:`.

Listing 1.8: Testing error raising

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

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.

Running a single test

Normally, you will run your tests using the Test Runner or using your code browser. If you don't want to launch the Test Runner from the World menu, you can execute `TestRunner open`. 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 will build 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
```

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.

1.6 The SUnit framework

SUnit consists of four main classes: `TestCase`, `TestSuite`, `TestResult`, and `TestResource`, as shown in Figure 1.3. 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 suite of tests; this is in distinction to the `TestCase>>setUp` method, which is executed before each test.

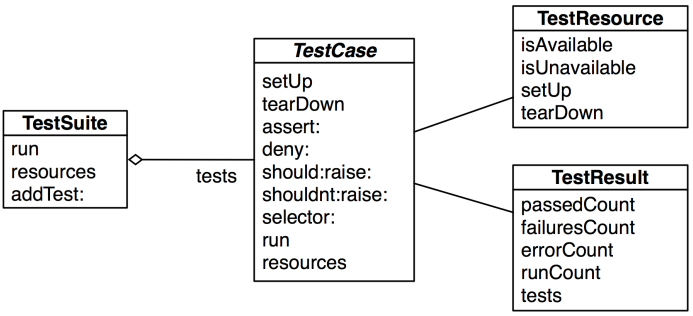

Figure 1.3: The four classes representing the core of

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 (that is, a test suite). Each test is run by creating a new instance of a subclass of `TestCase`, running `setUp`, running the test method itself, and then sending the `tearDown`.

The context 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.

TestResource

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 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 put the necessary resources in a singleton object of some class. The class `TestResource` exists to be subclassed by such resource classes. 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` and we associate it with `MyTestCase` by overriding the class method `resources` to return an array of the test classes that it will use (note the use of a dynamic array, for convenience).

Listing 1.9: An example of a TestResource subclass

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

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

```
  ^{ MyTestResource }
```

Exercise

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.

```
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
```

1.7 Advanced features of SUnit

In addition to TestResource, SUnit contains assertion description strings, logging support, the ability to skip tests, and resumable test failures.

Assertion description strings

The TestCase 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 TestCase are:

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

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 that the value that the test is expecting and this is 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).
```

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 must also choose where to log by overriding `failureLog` to answer an appropriate stream.

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 `TestCase` message `skip` on your test case instance. For example, the following test uses it to define a conditional test.

```
OCCompiledMethodIntegrityTest>>testPragmas

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

1.8 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`.

1.9 SUnit implementation

The implementation of SUnit makes an interesting case study of a Pharo framework. Let's look at some key aspects of the implementation by following the execution of a test.

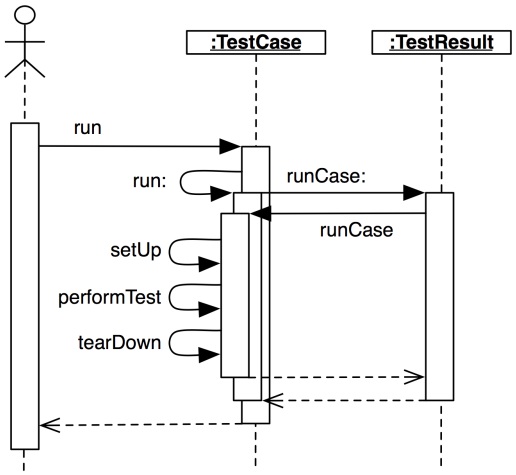

Figure 1.4: Running one test

Running one test

To execute one test, we execute the expression `(aTestClass selector: aSymbol) run`.

The method `TestCase>>run` creates an instance of `TestResult` that will accumulate the results of the test, then it sends itself the message `TestCase>>run:` (See Figure 1.4).

```
TestCase>>run
  | result |
  result := self classForTestResult new.
  [ self run: result ]
    ensure: [ self classForTestResource resetResources: self resources ].
  ^ result
```

The method `TestCase>>run:` sends the message `runCase:` to the test result:

Listing 1.10: Passing the test case to the test result

```
TestCase>>run: aResult
  aResult runCase: self
```

The method `TestResult>>runCase:` sends the message `TestCase>>runCase` to an individual test, to execute the test. `TestResult>>runCase` deals with any exceptions that may be raised during the execution of a test, runs a `TestCase` by sending it the `runCase`, and counts the errors, failures and passes.

Listing 1.11: Catching test case errors and failures

```
TestResult>>runCase: aTestCase
```

```
  [
  aTestCase announce: TestCaseStarted withResult: self.
  aTestCase runCase.
  aTestCase announce: TestCaseEnded withResult: self.
  self addPass: aTestCase ]
    on: self class failure, self class skip, self class warning, self
        class error
    do: [ :ex | ex sunitAnnounce: aTestCase toResult: self ]
```

The method TestCase>>runCase sends the messages TestCase>>setUp and TestCase>>tearDown as shown below.

```
TestCase>>runCase
    self resources do: [ :each | each availableFor: self ].
    [ self setUp.
    self performTest ] ensure: [
      self tearDown.
      self cleanUpInstanceVariables ]
```

Running a TestSuite

To run more than one test, we send the message run to a TestSuite that contains the relevant tests. TestCase class provides some functionality to build a test suite from its methods. The expression MyTestCase buildSuiteFromSelectors returns a suite containing all the tests defined in the MyTestCase class. The core of this process is:

Listing 1.12: Auto-building the test suite

```
TestCase class>>testSelectors
  ^(self selectors select: [ :each | (each beginsWith: 'test') and:
      [each numArgs isZero]])
```

The method TestSuite>>run creates an instance of TestResult, verifies that all the resources are available, and then sends itself the message TestSuite>>run:, which runs all the tests in the suite. All the resources are then released.

```
TestSuite>>run: aResult
  self setUp.
  [ self tests
    do: [ :each |
      each run: aResult.
      self announceTest: each.
      self changed: each ] ]
    ensure: [ self tearDown ]

TestSuite>>setUp
  self resources do: [ :each |
```

```
        each isAvailable ifFalse: [ each signalInitializationError ]
    ].

TestSuite>>tearDown
  self resourceClass resetResources: self resources
```

The class `TestResource` and its subclasses keep track of their currently created singleton instances that can be accessed and created using the class method `TestResource class>>current`. This instance is cleared when the tests have finished running and the resources are reset.

The resource availability check makes it possible for the resource to be recreated if needed, as shown in the class method `TestResource class>>isAvailable`. During the `TestResource` instance creation, it is initialized and the message `setUp` is sent to a test resource.

Listing 1.13: Test resource availability

```
TestResource class>>isAvailable
  "This is (and must be) a lazy method. If my current has a value, an
      attempt to make me available has already been made: trust its
      result. If not, try to make me available."

  current ifNil: [self makeAvailable].
  ^self isAlreadyAvailable
```

Listing 1.14: Test resource creation

```
TestResource class>>current
  "This is a lazy accessor: the assert of self isAvailable does no work
      unless current isNil. However this method should normally be sent
      only to a resource that should already have been made available,
      e.g. in a test whose test case class has the resource class in
      its #resources, so should never be able to fail the assert.
  If the intent is indeed to access a possibly-unprepared or
      reset-in-earlier-test resource lazily, then preface the call of
      'MyResource current' with 'MyResource availableFor: self'."

  self
    assert: self isAvailable
    description:
      'Sent #current to unavailable resource ', self name,
       '. Add it to test case'' class-side #resources (recommended) or
            send #availableFor: beforehand'.
  ^ current
```

1.10 Some 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.

1.11 Chapter summary

This chapter explained why tests are an important investment in the future of your code. We explained in a step-by-step fashion how to define a few tests for the class `Set`. Then we gave an overview of the core of the SUnit framework by presenting the classes `TestCase`, `TestResult`, `TestSuite` and `TestResources`. Finally we looked deep inside SUnit by following the execution of a test and a test suite.

- 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 classed `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!