```
   [self run: result] ensure: [self resources do: [:each | each reset]].
   ↑result
```

Method 1.15: *Passing the test result to the test suite*

```
TestSuite»run: aResult
   self tests do: [:each |
      self changed: each.
      each run: aResult].
```

The class TestResource and its subclasses keep track of their currently created instances (one per class) that can be accessed and created using the class method 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 re-created if needed, as shown in the class method TestResource class»isAvailable. During the TestResource instance creation, it is initialized and the method setUp is invoked.

Method 1.16: *Test resource availability*

```
TestResource class»isAvailable
   ↑self current notNil and: [self current isAvailable]
```

Method 1.17: *Test resource creation*

```
TestResource class»current
   current isNil ifTrue: [current := self new].
   ↑current
```

Method 1.18: *Test resource initialization*

```
TestResource»initialize
   super initialize.
   self setUp
```

1.9 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.

Feathers' Rules for Unit tests. Michael Feathers, an agile process consultant and author, writes:[3]

[3]See http://www.artima.com/weblogs/viewpost.jsp?thread=126923. 9 September 2005

> *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 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.

1.10 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 called 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 using the SUnit test runner tool (in the tool bar).