Chapter 37
Scala Testing

37.1 Introduction

This chapter examines the various facilities available within Scala to perform a range of tests.

37.2 Scala Runtime Test Facilities

37.2.1 Validation Checks

Scala provides three language facilities that can be used for validation. These are:

- *Assert*—this operation is used to validate the sate of some property and throws a java.lang.AssertionError if the condition it defines is not met.
- *Require*—this is intended as a pre-condition for a class, method or function. It can be used to validate parameters and throws an IllegalArgumentException if the condition specified is not met.
- *Assume*—this is essentially an alias for assert—it can therefore be used to validate the state of an instance of an object and also throws the AssertionError if the condition specified is not met. However, static analysis tools could decide to treat assume as being different. For example, it could be that assume is used to represent a concept which has already been verified and thus it can be assumed to be correct.

None of the above operations use the Java framework assertion and are always evaluated unless turned off at compile time. In fact, if you examine the `Predef.scala` source file you will find that the three validation type assertions are all very similar and differ only in the exceptions that they throw. An example illustrating the definitions of the assert, assume and require methods is shown below:

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_37

```
    def assert(assertion: Boolean) {
   if (!assertion)
          throw new java.lang.AssertionError(
                                          "assertion failed")
}
def assume(assumption: Boolean) {
   if (!assumption)
     throw new java.lang.AssertionError(
                                          "assumption failed")
}
def require(requirement: Boolean) {
   if (!requirement)
            throw new IllegalArgumentException(
                                   "requirement failed")
}
```

37.2.2 Using Require and Assert

The following simple listing illustrates how the *require* and the *assert* operations can be used:

```
package com.jjh.scala.validate

object AssertionTests {

  def main(args: Array[String]): Unit = {
    var count = 0
    // Try require
    require(count == 0)
    require(count == 0,
      {
        println("Require called should be 0")
      })
    // Assert
    assert(count == 0)
    assert(count == 0,
      {
        println("Assert called is not 0")
      })
  }

}
```

Note that the operations, require, assert and assume can all take a condition that returns true or false and an operation to perform if the condition is not met. Thus:

```
require(count == 0)
```

validates that count is currently set to Zero, while

```
require(count == 0,
    {
      println("Require called should be 0")
    })
```

also validates that count is Zero but evaluated the behaviour passed in as the second parameter if the Boolean test fails. In this case we merely print a message out to the console. The same is true for the assert example above.

37.3 Test Libraries in Scala

There are several libraries in Scala which extend the basic facilities available within the language itself to provide a richer way to represent and express tests. At the time of writing the three leading libraries are:

- ScalaTest
- Spec
- ScalaCheck

Each of these is briefly described below, while ScalaTest will be discussed in more detail in the next section.

37.3.1 ScalaTest

ScalaTest (see http://www.scalatest.org) provides a number of different testing options from Unit Tests, through functional testing to behaviour-based testing.

37.3.2 Spec

Spec is a Behaviour-Driven Development testing framework for Scala (see http://code.google.com/p/specs).

37.3.3 ScalaCheck

ScalaCheck (see http://www.scalacheck.org) is a testing framework developed from the Haskell library QuickCheck but has now extended beyond this. It is a property-based testing framework which means that each test is associated with a property that you wish to ensure for the elements being tested and how that property is verified. For example,

```
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll

object StringSpecification extends Properties("String")
{
    property("startsWith") = forAll { (a: String, b:
String) =>     (a+b).startsWith(a)
}
    property("concatenate") = forAll { (a: String, b:
String) =>     (a+b).length > a.length && (a+b).length >
b.length
   }
    property("substring") = forAll { (a: String, b:
String, c: String) =>     (a+b+c).substring(a.length,
a.length+b.length) == b
   }
}
```

37.4 Scalatest Testing Framework

The most established Scala test framework in use is ScalaTest. This framework actually allows several different testing styles to be adopted. The simplest of which is the unit testing approach based on direct integration between JUnit (originally a Java Unit Test framework) and Scala. Essentially, using ScalaTest around JUnit, a developer writes a simple Unit Test using Scala idioms that is then run through the JUnit infrastructure. This means that if a development project is using a mixture of Java and Scala code, then can write JUnit tests in either Java or Scala and run them through the same tool sets.

37.4.1 Setting up Your Scala Project

The first thing you will need to do if you are going to use ScalaTest and JUnit is to make them available to your applications. That is, to write test that will utilise the

ScalaTest to JUnit bridge you will need to have both the ScalaTest library and the JUnit library on your *classpath*. Note that you must ensure that you are using compatible versions of JUnit and ScalaTest. If the versions are not compatible you may find that you get unexpected results.

To set up your environment in Eclipse you will need to add the JUnit jar and the ScalaTest jar to your project.

First let us obtain the ScalaTest jar. The current release at the time of writing is the ScalaTest 3.0.4 release. This can be obtained from the *downloads* page of the main ScalaTest website (see http://www.scalatest.org/download). You can either download the libraries, add the appropriate Maven dependencies or if you are using SBT add the SBT dependencies to your build file. For example, for Maven you can add:

```
<dependency>
  <groupId>org.scalactic</groupId>
  <artifactId>scalactic_2.12</artifactId>
  <version>3.0.4</version>
</dependency>

<dependency>
  <groupId>org.scalatest</groupId>
  <artifactId>scalatest_2.12</artifactId>
  <version>3.0.4</version>
  <scope>test</scope>
</dependency>
```

This adds both the Scalatest library and its sister library Scalactic to your project. Strictly speaking, Scalactic is not required; however it is recommended as it focusses on quality through types.

The jUnit framework will be used to run the Scalatest test, and so this is also required. Again this can be downloaded, or if you are using Maven then you can just add the appropriate dependency to your POM file:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

37.4.2 ScalaTest and JUnit

In this section, we will look at how you can write simple Unit Tests in ScalaTest. A ScalaTest test mixes in the trait `org.scalatest.Suite` and can mix in one or more additional traits that allow you to control the behaviour of the tests. In general, we will call our test classes <something>Test or Test<something> where *something* is an appropriately descriptive name. Therefore we might write a class called PersonTest (which indicates that it provides one or more test methods associated with the class Person. To illustrate the idea let us assume that we have a simple class Person as shown in the following listing:

```
package com.jjh.scala

class Person(val name: String, var age: Int) {
  override def toString() = s"$name is $age"
}
```

A simple test class for this Person class might thus be:

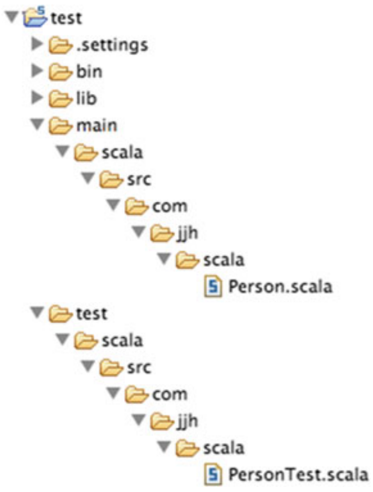
```
package com.jjh.scala

import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner
import org.junit.Assert._
import org.scalatest.FlatSpec

@RunWith(classOf[JUnitRunner])
class PersonTest extends FlatSpec {

  "A new Person instance" should "have an age 0" in {
    val p = new Person("John")
    assertEquals("Age shoud be Zero",0, p.age)
  }

  "A new Person with an age of 54" should "return set their age
appropriately" in {
    val p = new Person("John", 54)
    assertEquals("Age shoud be 54",54, p.age)
  }

}
```

There are a number of points to note about this simple test class.

37.4.2.1 The package

The first is that both the Person class and the PersonTest class are defined within the package com.jjh.scala. However, this does not mean that they are both defined in the same source location. I have created a second location that will contain all my tests. I thus have a route folder main/scala/src and a root folder test/scla/src that can both contain Scala source code. The main path indicates the main body of the system and the test path the location of test code.

This means that we can ensure that no test code is accidently included in the deployed system as the deployed system should not contain anything under the test root. However, for testing purposes we would like to allow the test class to have at least package visibility of all data and methods as this can make setting up test scenarios easier. In Scala it does not matter that Person and PersonTest are in different source locations, and the important point is that they are both part of the package com.jjh.scala. This is illustrated below.



37.4.2.2 RunWith Annotation

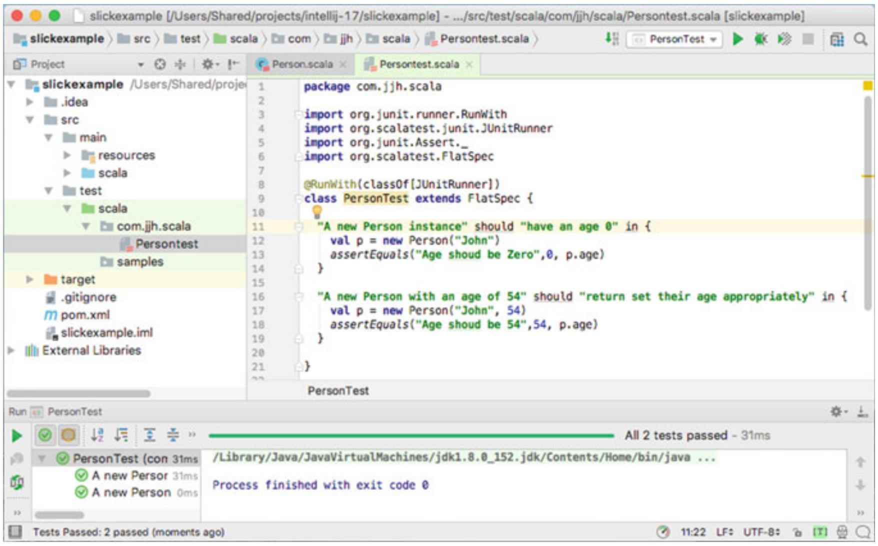
Secondly the class is *annotated* with a

```
@RunWith(classOf[JUnitRunner])
```

This indicates that when this class is run it is not run as a standard Scala application. Instead it should be run through the JUnit framework via the JUnitRunner.

37.4.2.3 Naming Conventions

The third point to note is that the test specifications within the PersonTest class are written in a very English style. This test specification style is of the format "X should Y" or "A must B".



```
package com.jjh.scala

import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner
import org.junit.Assert._
import org.scalatest.FlatSpec

@RunWith(classOf[JUnitRunner])
class PersonTest extends FlatSpec {

  "A new Person instance" should "have an age 0" in {
    val p = new Person("John")
    assertEquals("Age shoud be Zero",0, p.age)
  }

  "A new Person with an age of 54" should "return set their age appropriately" in {
    val p = new Person("John", 54)
    assertEquals("Age shoud be 54",54, p.age)
  }

}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

Fig. 37.1 JUnit runtime view in Eclipse

Finally, within the test specification methods, having created an instance of the Person class we validate the test using an Assert.assertEquals method. In fact the Assert class of the org.junit package provides many different assertion methods. We could have used the Assert.assert.Equals, but this is quiet long winded, and Assert. assert seems a bit repetitive. To avoid this we imported the methods on the Assert class into this file using:

```
import org.junit.Assert._
```

We can now run this simple test. In IntelliJ you can just select to the run the PersonTest class and IntelliJ will work that this is a test class with multiple test specifications and run each of them in turn. It will then open the test results view at the bottom of your IDE (Fig. 37.1).

There is a range of options that can be applied to these tests. For example, if you wish to define some behaviour that will be run before each test, referred to as test fixtures, you can mix in the *BeforeAndAfter* trait. This provides the ability to define a *before* and *after* expression. The following listing illustrates a test suite for a class Stack that has two tests and a before fixture and an after fixture:

```
package com.jjh.scala

import org.scalatest.junit.JUnitRunner
import org.junit.runner.RunWith
import org.scalatest.FlatSpec
import org.scalatest.BeforeAndAfter
import org.junit.Assert._

@RunWith(classOf[JUnitRunner])
class SuiteTest extends FlatSpec with BeforeAndAfter {

  before {
    println("Before behaviour")
  }

  after {
    println("After behaviour")
  }

  "A list treated as a pre 2.12 Stack" must "pop off value added" in {
    var stack = List.empty[Int]
    stack = stack.+:(52)
    assertEquals(1, stack.size)
  }

  "A list treated as a pre 2.12 Stack" should "return values in reverse
order" in {
    var stack = List.empty[Int]
    stack = stack.+:(52)
    stack = stack.+:(21)
    val x = stack.head
    assertEquals(21, x)
  }

}
```

The result of executing this test is that both tests pass and the console contain the output shown below:

```
Before behaviour
After behaviour
Before behaviour
After behaviour
```

As you can see from this example, the *before* and *after* behaviours have been run before (and after) each test.

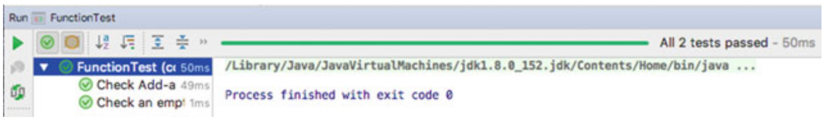
37.4.3 Scala Test and Functional Test Suites

The functional test suite facilities available in ScalaTest are aimed at supporting TDD style development. In a function style test each test is a *named* function within the test suite. A developer's class extends the FunSuite (which stands for Function Suite) and can use BeforeAndAfter fixtures for set-up and tear down style behaviour. In the function suite tests, names can be more description as the name of tests are strings and can have spaces within them. However, at runtime they are presented as normal JUnit tests.

The following listing illustrates a Function Test Suite:

```
package com.jjh.scala

import org.junit.Assert.assertEquals
import org.scalatest.junit.JUnitRunner
import org.junit.runner.RunWith
import org.scalatest.FunSuite

import scala.collection.mutable.Stack

@RunWith(classOf[JUnitRunner])
class FunctionTest extends FunSuite {

  test("Check Add-and-Pop a stack") {
    var stack = List.empty[Int]
    stack = stack.+:(52)
    assertEquals(1, stack.size)
  }

  test("Check an empty stack has no members") {
    var stack = List.empty[Int]
    stack = stack.+:(52)
    stack = stack.+:(21)
    val x = stack.head
    assertEquals(21, x)
  }

}
```

The results of running this FunctionTest within Eclipse are presented via the JUnit view as shown below.



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

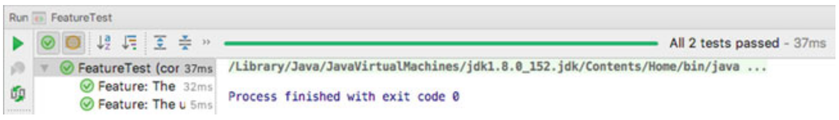
37.5 Scalatest and Feature Tests

Another test style supported by ScalaTest is that of the feature test. A feature is typically a very high-level concept that would be meaningful to a user. As such features are primarily those things that might be tested at the user acceptance testing level. ScalaTest provides the FeatureSpec trait that can be used to represent these very high-level concepts. The aim is to provide a structure that allows developers to work along side non-programmers to define the acceptance criteria. The following listing illustrates a ScalaTest Feature Specification test:

```
package com.jjh.scala

import org.junit.Assert.assertEquals
import org.scalatest.FeatureSpec
import org.scalatest.GivenWhenThen

import scala.collection.mutable.Stack
import org.scalatest.junit.JUnitRunner
import org.junit.runner.RunWith

@RunWith(classOf[JUnitRunner])
class FeatureTest extends FeatureSpec with GivenWhenThen {

  feature("The user can pop an element off the top of the stack") {

    info("As a programmer")
    info("I want to be able to pop items off the stack")
    info("So that I can get them in last-in-first-out order")

    scenario("pop is invoked on a non-empty stack") {
```

```
Given("a non-empty stack")
var stack = List.empty[Int]
stack = stack.+:(52)
stack = stack.+:(21)
val oldSize = stack.size

When("when pop style ops are invoked on the stack")
val result = stack.head
stack = stack.tail

Then("the most recently pushed element should be returned")
assert(result === 21)

And("the stack should have one less item than before")
assert(stack.size === oldSize - 1)
          }
scenario("pop is invoked on an empty stack") {

 Given("an empty stack")
 var emptyStack = List.empty[Int]

 When("when pop is invoked on the stack")
 Then("NoSuchElementException should be thrown")
 intercept[NoSuchElementException] {
   emptyStack.head
 }

 And("the stack should still be empty")
 assert(emptyStack.isEmpty)
   }
  }

 }
```

The result of running this as a Scala JUnit test in Eclipse is shown below



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

37.6 Test-Driven Development

Test-Driven Development (of TDD) is a development technique whereby developers write test cases *before* they write any implementation code. The tests thus drive or dictate the code that is developed. The implementation only provides as much functionality as is required to pass the test, and thus the tests act as a specification of *what* the code does (and some argue that the test are thus part of the that specification and provide documentation of what the system is capable of).

TDD has the benefit that as tests must be written first, there are always a set of tests available to perform unit, integration, regression testing, etc. This is good as developers can find that writing tests and maintaining tests are boring and of less interest than the actual code itself, and this put less emphasis into the testing regime than might be desirable. TDD encourages, and indeed requires, that developers maintain an exhaustive set of repeatable tests and that those tests are developed to the same quality and standards as the main body of code.

There are three rules of TDD as defined by Robert Martin for TDD; these are:

1. You are not allowed to write any production code unless it is to make a failing Unit Test pass.
2. You are not allowed to write any more of a Unit Test than is sufficient to fail, and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing Unit Test.

This leads to the TDD cycle described in the next section.

37.6.1 The TDD Cycle

There is a cycle to development when working in a TDD manner. The shortest form of this cycle is the TDD mantra:

Red / Green / Refactor

which relates to the JUnit suite of tools where it is possible to write a JUnit-based test. Within tools such as Eclipse, when you run a JUnit test a JUnit view is shown with Red indicating that a test failed, Green indicating that the test passed. Hence Red/Green, in other words write the test and let it fail, then implement the code to ensure it passes. The last part of this mantra is *Refactor* which indicates once you have it working make the code cleaner, better, fitter by Refactoring it. Refactoring is the process by which the behaviour of the system is not changed, but the implementation is altered to improve it.

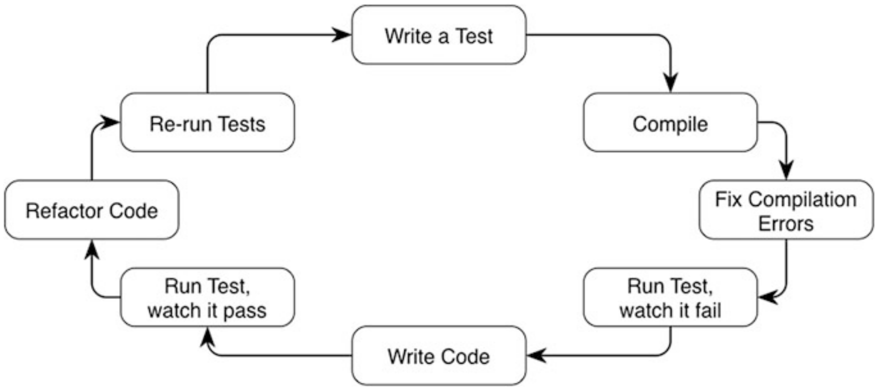


Fig. 37.2 TDD cycle

The TDD mantra can be seen in the TDD cycle that is shown in Fig. 37.2 and described in more detail below:

1. Write a single test.
2. Compile it. It should not compile because you have not written the implementation code.
3. Implement just enough code to get the test to compile.
4. Run the test and see it **fail**
5. Implement *just enough* code to get the test to pass.
6. Run the test and see it **pass**
7. **Refactor** for clarity and deal with any issue of reuse, etc.
8. Repeat for next test.

37.6.2 Test Complexity

The aim is to strive for simplicity in all that you do within TDD. Thus you write a test that fails and then do just enough to make that test pass (but no more). Then you Refactor the implementation code (that is change the internals of the unit under test) to improve the code base. You continue to do this until all the functionality for a unit has been completed. In terms of each test, you should again strive for simplicity with each test only testing one thing with only a single assertion per test (although this is the subject of a lot of debate within the TDD world).

37.6.3 *Refactoring*

The emphasis on Refactoring with TDD makes it more than just testing or Test First Development. This focus on Refactoring is really a focus on (re)design and incremental improvement. The tests provide the specification of what is needed as well as the verification that existing behaviour is maintained, but Refactoring leads to better design software. Thus with Refactoring TDD is not TDD!

References

- Astels D (2003) Test-driven development: a Practical Guide. Prentice-Hall/Pearson Education. ISBN 0-13-101649-0
- Beck K (2003) Test-driven development: by example. Addison-Wesley. ISBN 0-321-14653-0
- Dijkstra EW (1970) Notes on structured programming. Technical University of Eindhoven, The Netherlands, Department of Mathematics, Technical Report 70-WSK-03, April 1970 (see http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF).
- Myers G, Badgett T, Sandler C (2011) The art of software testing, 3rd edn. Wiley, 1118031962
- Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley, 0201485672, published June 1999

Online References

Cucumber behaviour driven development, see http://cukes.info/
ScalaTest see http://www.scalatest.org
ScalaCheck see http://www.scalacheck.org
Robert C. Martin (aka Uncle Bob) http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
http://www.testdriven.com
testdrivendevelopment group (Yahoo):
http://groups.yahoo.com/group/testdrivendevelopment
xUnit implementations:
http://www.xprogramming.com/software.htm
http://www.junit.org
Refactoring.com
http://www.refactoring.com/