Unit Testing with *gUnit*

Revision 2052

Andrew P. Black

April 17, 2015

Abstract

Unit testing has become a required part of software development. Every method that might possibly go wrong should have one or more unit tests that describe its behaviour. These tests act as executable documentation: they describe what the method should do in readable terms, by giving examples. If the tests pass, we can be sure that the implementation conforms to this specification. Of course, specification by example cannot be complete, and testing cannot ensure correctness, but it helps enormously in finding bugs, and speeds up development.

Test Driven Development (TDD) takes testing to the next level: with TDD, you write your tests before you write your code. This helps you choose good interfaces for your methods — interfaces that make sense to the client, not to the implementor. You also know what to do next, and when you are done.

The unit testing framework for Grace is called *gUnit*. This document outlines how to use it.

1 An Example

The best way to explain how to use *gUnit* is probably by example.

```
1  import "gUnit" as gUnit
2  import "collectionsPrelude" as mot
3
4  def setTest = object {
5      class forMethod(m) {
6          inherit gUnit.testCaseNamed(m)
7
8          var emptySet:Set⟦Number⟧
9          var set23:Set⟦Number⟧
10
11         method setup {
```

```
12              emptySet := mot.set.empty
13              set23 := mot.set.empty
14              set23.add(2)
15              set23.add(3)
16          }
17
18          method testEmpty {
19              assert (emptySet.size == 0) description ("emptySet is not empty!")
20              deny (emptySet.contains 2) description ("emptySet contains 2!")
21          }
22
23          method testNonEmpty {
24              assert (set23.size) shouldBe 2
25              assert (set23.contains 2)
26              assert (set23.contains 3)
27          }
28
29          method testDuplication {
30              set23.add(2)
31              assert (set23.size == 2) description "duplication of 2 not detected by set"
32          }
33      }
34  }
35
36  gUnit.testSuite.fromTestMethodsIn(setTest).runAndPrintResults
```

Line 1 imports the *gUnit* package, so that we can use it to implement our tests. Line 2 imports the module under test (*mot)*. Normally this will be some code that you are developing, but for the sake of this example, we are importing the pre-defined module "collectionPrelude". We will test the implementation of sets in the module under test.

On line 4 we define a test factory — an object that contains a class. Starting on line 5 we define that *class*; its instances will be the individual tests. There will be one test for each method in this class whose name begins with test; the names of these methods are chosen to describe what is being tested. Naturally enough, we call a class that contains *test* methods a *test class*.

After the definition of the test factory, the last line of the example runs the tests. It's a bit complicated, so lets look at it in parts.

A testSuite (pronounced like "sweet") is a collection of tests. Like an individual test, you can run a testSuite, which runs all of the tests that it contains. There are several ways of making a testSuite; here we use the method fromTestMethodsIn on the object testSuite. This method takes as its argument a *test factory* object; it builds a test suite containing one test for each of the test methods in the objects created by the test factory's class.

What do we do with the test suite once we've made it? Run all of the tests, and print the results! There are other things that we could do too; we will look at test suites later in a bit more detail.

2 Test Factories, Test Classes, and Test Methods

Why does *gUnit* insist that our tests are methods in a *test factory*, rather than simply objects with a run method? The answer is that it's sometimes useful to run a test more than once (for example, when hunting for a bug), and it is important that each test should start with a "clean slate", and not be contaminated by previous failed tests. So *gUnit* needs to be able to generate a new instance of a test when required. This is the function of the *test factory*, here the object called setTest.

What makes an object a *test factory*?

1. Its contains a method called forMethod(m) that returns a fresh object (line 5). The most convenient way to write such a method is using the class syntax, so we call it a test class.
2. The fresh objects created by the test class inherit from gUnit.testCaseNamed(m) (line 6). These objects are *test objects*.
3. the test objects have *methods* corresponding to the tests that we want to run; these *test methods* have no parameters and *must* have a name starting with test. For example, starting on line 18 we see a test called testEmpty.
4. It's conventional to name the factory object after the class or object that it is testing, and to include the word *Test* as a suffix. In our example, the class is called setTest because it is testing the class set.
5. The test class can have methods setup and teardown; if they exist, these methods will be run before and after each test method (whether or not the test passes). If you do override these methods, be sure to request **super**.setup or **super**.teardown before you do anything else.
6. The test class and the test factory can have fields and other methods as necessary. For example, it is sometimes convenient to have helper methods for clarity, such as asset()isApproximatelyEqualTo(). Any **def**s and **var**s in the test class will be initialised afresh *before each test method* is run.

What's in a test method? The only thing that has to be in a test is one or more *assertions*, which are self-requests of various assertion methods inherited from gUnit.testCaseNamed(m).

```
1   method assert (bb: Boolean) description (message)
2   // asserts that bb is true.  If bb is not true, the test will fail with message.
3   // Hence, the message should describe the situation that applies when the assertion is false.
4   method deny (bb: Boolean) description (message)
5   // asserts that bb is false.  If bb is not false, the test will fail with message.
6   // Hence, the message should describe the situation that applies when the assertion is true.
7   method assert (bb: Boolean)
8   method deny (bb: Boolean)
9   // short forms, with the default message "assertion failure"
10  method assert (s1:Object) shouldBe (s2:Object)
11  // like assert (s1 == s2), but with a more appropriate default message
12  method assert (block0) shouldRaise (desiredException)
```

```
13      // asserts that the desiredException is raised during the execution of block0
14      method assert (block0) shouldntRaise (undesiredException)
15      // asserts that the undesiredException is not raised during the execution of block0.
16      // The assertion holds if block0 raises some other exception, or if it completes
17      // execution without raising any exception.
18      method failBecause (message)
19      // equivalent to assert (false) description (message)
20      method assert(n1:Number) shouldEqual (n2:Number) within (epsilon:Number)
21      // asserts that n1 and n2 are equal, with a tolerance of epsilon
22      method assert(value) hasType (Desired:Type)
23      // asserts that value has a type that conforms to the type Desired; if it fails,
24      // the message will tell you which methods are missing from value.
25      method assertType(T:Type) describes (value)
26      // asserts that type T describes the object value; the assertion will be false if value has
27      // methods that are missing from T, and the message will tell you which methods are missing.
28      method deny(value) hasType (Undesired:Type)
29      // assets that value does not have type Undesired.
```

In addition to the assertions, a test can contain arbitrary executable code. However, because part of the function of a test is to serve as documentation, its a good idea to keep tests as simple as possible.

What happens when a test runs? In general, one of three things might happen when a test runs.

1. The test *passes*, that is, all of the assertions that it makes are **true**.
2. The test *fails*, that is, one of the assertions is **false**.
3. The test *errors*[1], that is, a runtime error occurs that prevents the test from completing. For example, the test may request a method that does not exist in the receiver, or might index an array out of bounds.

In all cases, *gUnit* will record the outcome, *and then go on to run the next test.* This is important, because we generally want to be able to run a suite of tests, and see how many pass, rather than have testing stop on the first error or failure. For example, when we run the set test suite shown above, we get the output

```
3 run, 0 failed, 0 errors
```

What happens when your tests don't pass? The rule for Test Driven Development (TDD) is to write the test that documents new functionality *before* you implement that functionality. Let's illustrate this by adding the remove operation to sets.

First, we add another test to setTest:

[1]Yes, I'm using "to error" as a verb. How daring!

```
method testRemove {
    set23.delete 3
    deny (set23.contains 3) description "{set23} contains 3 after it was removed"
}
```

When we run the tests again, we get this output, which summarizes the test run:

```
setTest: 4 run, 0 failed, 1 error
Errors:
    testRemove: NoSuchMethod: no method 'delete(_)' on set.ofCapacity(_) set{2, 3}.
```

The summary output will contain a list of all of the tests that failed, and a list of all of the tests that errored, but not a lot of debugging information.

GUnit will then run some or all of the tests that errored again. (The number of tests that it re-runs can be controlled by assigning to the variable gUnit.numberOfErrorsToRerun.) Here is the output:

```
Re−running 1 error.

debugging method testRemove ...
NoSuchMethod on line 36: no method 'delete(_)' on set.ofCapacity(_) set{2, 3}.
  called from setTest.forMethod(_).testRemove at line 36 of gUnit doc
```

In this example, we see that the problem is that the object under test doesn't actually have a method called delete. This is hardly a surprise, because we haven't implemented it yet! The normal process of test-driven development would then have us go and implement this method, and then run the tests again. In this case, because this is only an example, the set class does in fact have a method that does the right thing — the problem is that it is called remove rather than delete. So the fix is to change our test:

```
method testRemove {
    set23.remove 2
    deny (set23.contains 3) description "{set23} contains 3 after it was removed"
}
```

```
setTest: 4 run, 1 failed, 0 errors
Failures:
    testRemove: set{3} contains 3 after it was removed
```

Whoops! The test failed. What's going on? Why did the test fail? Notice that the test is not re-run, because the one line summary really tells us all that we need to know. the stack trace would just tell us that the test was run, which won't help is to find the problem.

When a test fails, we have discovered an inconsistency between what the test says and what the module under test does. This means that *either* the module under test contains a bug, or the *test* contains a bug. We usually assume that we made a mistake when we implemented the method under test; here it looks like remove doesn't actually remove the argument. But it's worth checking that we wrote the test correctly.

In this case, if you look carefully, you will see that the *test* is wrong. We are removing 2, but then checking that the set doesn't contain 3. Let's fix the test!

```
method testRemove {
    set23.remove 3
    deny (set23.contains 3) description "{set23} contains 3 after it was removed"
}
```

Now we get:

```
setTest: 4 run, 0 failed, 0 errors
```

3 Test Suites

A lot of the power of automated testing comes from being able to run large numbers of tests unattended, and to have a human alerted only when there are errors or failures. To this end, it's useful to be able to group tests together into collections called test suites.

gUnit's testSuite implements the *Composite* pattern, and lets the testing framework treat individual tests and compositions of tests uniformly. A testSuite contains zero or more tests, and zero or more testSuites. A testSuite implements the enumerable interface of collections; you can ask a test suite its size, add a new test or test suite to it, and create an iterator to sequence through it. It also implements the runnable interface of a test: you can run a test suite, runAndPrintResults, or debugAndPrintResults.

As with any collection factory, testSuite responds to the with(), empty, and withAll() requests, answering a new test suite. You can also make a test suite using testSuite.fromTestMethodsIn(aTestClass), which populates the suite with all of the test methods in aTestClass.

4 Dependencies

The implementation of *gUnit* depends on collections and on mirrors. It also uses methods on exceptions to print diagnostics about failing tests. Specifically:

1. testSuites contain lists of tests.
2. testResults contain sets of failures and errors.
3. testSuite.fromTestMethodsIn(aTestClass) reflects on an instance of aTestClass using mirror.reflect , and looks at the names of the available methods.
4. testCaseNamed().run uses reflection to request execution of the test method.

5. debugging a test involves extracting the exceptionKind, message, lineNumber and backtrace from a dynamically-generated Exception object, so that these things can be printed.