Unit Testing with *gUnit*

Revision 2127

Andrew P. Black

July 14, 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 gU
2  import "setVector" as sV
3
4  class setTest.forMethod(m) {
5      inherits gU.testCaseNamed(m)
6
7      var emptySet // :Set<Number>
8      var set23    // :Set<Number>
9
10     method setup {
11         super.setup
```

```
12          emptySet := sV.setVector.new
13          set23 := sV.setVector.new
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  gU.testSuite.fromTestMethodsIn(setTest).runAndPrintResults
```

Line 1 imports the *gUnit* package, so that we can use it to implement our tests. Line 2 imports the package under test, here an implementation of Sets called setVector.

Starting on line 4 we define a *class* whose instances will be the individual tests. The actual tests are methods in this class; the names of these methods are chosen to describe what is being tested. We call a class that contains test methods a test class.

After the definition of the class, 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 the test class; it builds a test suite containing one test for each of the test methods in the test 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 Classes and Test Methods

Why does *gUnit* insist that our tests are methods in a *class*, 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 its impotent 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 class*, here the class called aSetTest.

What makes a class a *test class*?

1. Its instances inherit from testCase.forMethod(m).
2. Its constructor method is called forMethod(m).
3. Its instances 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 class after the class or object that it's testing, and to include the word *Test* as a suffix. In our example, the class is called aSetTest because it's testing the class set.
5. The test class can have method 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 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 aTestCase.forMethod(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   method deny (bb: Boolean) description (message)
4   // asserts that bb is false.  If bb is not false, the test will fail with message
5   method assert (bb: Boolean)
6   method deny (bb: Boolean)
7   // short forms, with the default message "assertion failure"
8   method assert (s1:Object) shouldBe (s2:Object)
9   // like assert (s1 == s2), but with a more appropriate default message
10  method assert (block0) shouldRaise (desiredException)
11  // asserts that the desiredException is raised during the execution of block0
12  method assert (block0) shouldntRaise (undesiredException)
13  // asserts that the undesiredException is not raised during the execution of block0.
14  // The assertion holds if block0 raises some other exception, or if it completes
15  // execution without raising any exception.
16  method failBecause (message)
17  // equivalent to assert (false) description (message)
18  method assert(value:Object) hasType (Desired:Type)
19  // asserts that value has all of the methods of type Desired.
20  method deny(value:Object) hasType (Undesired:Type)
```

```
21      // asserts that value is missing one of the methods in the type Undesired
22      method assert(n1:Number) shouldEqual (n2:Number) within (epsilon:Number)
23      // asserts that n1 and n2 are equal, with a tolerance of epsilon
```

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 aSetTest:

```
method testRemove {
    set23.remove(2)
    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:

```
4 run, 0 failed, 1 error
Errors:
    testRemove: no method remove in setVector.
```

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

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. To get more information on the tests that don't pass, we *debug* them. To do this, we add the following line of code to the test module:

```
aSetTest.forMethod("testRemove").debugAndPrintResults
```

This creates a test suite that contains a single method (the method testRemove, named in the string argument to forMethod), and debugs it. Here is the output:

```
4 run, 0 failed, 1 error
Errors:
    testRemove: no method remove in setVector.

debugging method testRemove ...
No such method on line 35: no method remove in setVector.
  called from setVector.remove (defined nowhere in object at setVector:146) at setVectorTests:35
  called from setTest.testRemove (defined at setVectorTests:34 in object at setVectorTests:42) at gUnit:142
  called from MirrorMethod.request (defined at unknown:0) at gUnit:142
  called from Block[gUnit:139]._apply (defined at gUnit:130) at gUnit:143
  called from Block[gUnit:139].apply (defined at unknown:0) at gUnit:143
  called from Block[gUnit:136]._apply (defined at gUnit:130) at StandardPrelude:147
  called from Block[gUnit:136].apply (defined at unknown:0) at StandardPrelude:147
...
1 run, 0 failed, 1 error
Errors:
    testRemove: no method remove in setVector.
```

When we *debug* a test, we see the error message, but any code subsequent to the test that didn't pass is not run.

In this example, we see that the problem is that the object under test doesn't actually have a method called remove. This is hardly a surprise, because we haven't implemented it yet! If we implement this method, and run the same tests again, this is what happens:

```
4 run, 1 failed, 0 errors
Failures:
    testRemove: <SetVector: <Vector: 2 3>> contains 3 after it was removed

debugging method testRemove ...
Error around line 63: Assertion Failure: <SetVector: <Vector: 2 3>> contains 3 after it was removed
  Called aSetTest.debugAndPrintResults (defined at gUnit:149 in object at SetTests:44) on line 155
  Called aSetTest.debug (defined at gUnit:132 in object at SetTests:44) on line 151
  Called Block«gUnit:134».apply (defined at unknown:0 in object at unknown:0) on line 140
  Called Block«gUnit:134»._apply (defined at gUnit:128 in object at unknown:0) on line 140
  Called MirrorMethod.request (defined at unknown:0 in object at unknown:0) on line 139
  Called aSetTest.testRemove (defined at SetTests:36 in object at SetTests:44) on line 139
  Called aSetTest.deny()description (defined at gUnit:67 in object at SetTests:44) on line 287
  Called aSetTest.assert()description (defined at gUnit:60 in object at SetTests:44) on line 68
  Called Exception.raise (defined at unknown:0 in object at unknown:0) on line 63
    62:         then {
    63:             failure.raise(str)
    64:         }
```

```
minigrace: Program exited with error: SetTests
```

Whoops! We made a mistake when we implemented the remove method; it looks like it doesn't actually remove the argument.

The one line summary really tells us all that we need to know. The debugging information isn't all that useful: it tell us that the assertion failed, and then gives a stack trace of the path through *gUnit*, which isn't what we want. Perhaps one day Grace will have a debugger that will let us go back in time through the test that failed. Until then, if you make your tests small and simple, you will find that the test itself will give you most of the information that you need to fix the problem.

If I go back and correct my implementation of remove, this is the output on the next run:

```
4 run, 0 failed, 1 error
Errors:
    testRemove: undefined value used as argument to []:=

debugging method testRemove ...
RuntimeError on line 237: undefined value used as argument to []:=
  called from Block[vector:236]._apply (defined at vector:152) at vector:236
  called from Block[vector:236].apply (defined at unknown:0) at vector:236
  called from NativePrelude.while()do (defined at unknown:0 in StandardPrelude module) at vector:236
  called from vectorClass.removeFromIndex (defined at vector:228 in object at vector:343) at vector:127
  called from vectorClass.removeValue (defined at vector:122 in object at vector:343) at setVector:77
  called from setVector.remove (defined at setVector:76 in object at setVector:149) at setVectorTests:35
  called from setTest.testRemove (defined at setVectorTests:34 in object at setVectorTests:42) at gUnit:142
...
1 run, 0 failed, 1 error
Errors:
    testRemove: undefined value used as argument to []:=
```

This output is telling us about an error at line 231 of module vector, which is imported by setVector. On this line, the code accesses an undefined array element in method removeFromIndex. If the vector module had been developed using TDD, the developer would have found this bug, rather than having it annoy the clients of vector.

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 collect 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, and on math (for abs). 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.