Unit Testing with *minitest*

Revision 2556

Andrew P. Black

April 18, 2016

Abstract

minitest is a testing dialect for Grace. It is intended to simplify the process of testing for students beginning to practice Test-Driven Development. This document illustrates how to use it.

1 An Example

The best way to explain how to use *minitest* is probably by example:

```
1  dialect "minitest"
2  import "setVector" as sV
3
4  testSuite {
5      def emptySet = sV.setVector.new
6      def set23 = sV.setVector.new
7      set23.add 2
8      set23.add 3
9
10     test "emptyness" by {
11         assert (emptySet.size == 0) description ("emptySet is not empty!")
12         deny (emptySet.contains 2) description ("emptySet contains 2!")
13     }
14     test "non-emptyness" by {
15         assert (set23.size) shouldBe 2
16         assert (set23.contains 2)
17         assert (set23.contains 3)
18     }
19     test "remove" by {
20         set23.remove 2
21         deny (set23.contains 3) description "{set23} contains 3 after it was removed"
22     }
23     test "duplication" by {
```

```
24          set23.add 2
25          assert (set23.size == 2) description "duplication of 2 not detected by set"
26      }
27  }
```

Line 1 says that this module is written in the dialect *minitest*. Line 2 imports the module under test, in this case, a (buggy!) implementation of sets using vectors, which is in the module *setVector*. Line 4 builds a *testSuite*: a collection of tests, here just four. A testSuite can also contain definitions, variable declarations, and executable code, as seen on lines 5–8.

As you can see, each test has a title (which is printed out if the test fails or errors) and a block of code, which should contain one or more assertions.

When the test module is executed, all of the test in the test suite are run, in some order. Any variables and definitions inside the testSuite are re-initialized before each test is run.

2 Naming TestSuites

If you have several test suites in one module, it's handy to give each test suite a name, so that it's easy to see how many tests pass in each suite. To do this, use the method testSuiteNamed()with():

```
dialect "minitest"

testSuiteNamed "for()and" with {
    test "first smaller" by {
        def result = []
        def as = [1, 2, 3]
        def bs = ["one", "two", "three", "four", "five"]
        for (as) and (bs) do { a, b ->
            result.add(a::b)
        }
        assert (result) shouldBe [1::"one", 2::"two", 3::"three"]
    }

    test "second smaller" by {
        def result = []
        def as = [1, 2, 3, 4, 5]
        def bs = ["one", "two", "three"]
        for (as) and (bs) do { a, b ->
            result.add(a::b)
        }
        assert (result) shouldBe [1::"one", 2::"two", 3::"three"]
    }
}

method trymatch(e) {
    match (e)
        case {n: Number -> n+1}
        case {s: String -> "Got " ++ s}
```

```
}

testSuiteNamed "match tests" with {
    test "number" by {
        assert(trymatch 4) shouldBe 5
    }
    test "string" by {
        assert(trymatch "beer") shouldBe "Got beer"
    }

    test "boolean" by {
        assert{trymatch (true)} shouldRaise (ProgrammingError)
    }
}
```

This will produce the output

```
for()and: 2 run, 0 failed, 0 errors
match tests: 3 run, 0 failed, 0 errors
```

3 What's in a test?

Typically, a test contains zero, one or two lines of code to set up a testable situation, and then one or more *assertions*. Here are the things that you can assert.

```
assert (bb: Boolean) description (message)
// asserts that bb is true.  If bb is not true, the test will fail with message
deny (bb: Boolean) description (message)
// asserts that bb is false.  If bb is not false, the test will fail with message
assert (bb: Boolean)
deny (bb: Boolean)
// short forms, with the default message "assertion failure"
assert (s1:Object) shouldBe (s2:Object)
// like assert (s1 == s2), but with a more appropriate default message.  Uses the == method of s1.
assert (s1:Object) shouldntBe (s2:Object)
// like assert (s1 != s2), but with a more appropriate default message. Uses the != method of s1.
assert (block0) shouldRaise (desiredException)
// asserts that the desiredException is raised during the execution of block0
assert(n1:Number) shouldEqual (n2:Number) within (epsilon:Number)
// asserts that n1 and n2 don't differ by more than epsilon.
// Use instead of assert(_)shouldBe(_) for floating point numbers.
assert (block0) shouldntRaise (undesiredException)
// asserts that the undesiredException is not raised during the execution of block0.
// The assertion holds if block0 raises some other exception, or if it completes
// execution without raising any exception.
failBecause (message)
// always fails; equivalent to assert (false) description (message)
assert(value:Object) hasType (Desired:Type)
// asserts that value has all of the methods of type Desired.
deny(value:Object) hasType (Undesired:Type)
// asserts that value is missing one of the methods in the type Undesired
```

The golden rule is to keep the tests simple. In addition to testing that the code in the module under test works, they define *what it means* for the code to work. So the readability of the tests is of utmost importance.

It's also important for the tests to depend only on the defined external behaviour of the code under test, and *not* on implementation decisions that might change. So, for example, if the result is a set, your test should not depend on the order in which the elements appear when you request `asString` on it.

4 What happens when you execute a *minitest* module?

Executing the above module will run all of the tests and print a summary of the results. 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`. (A test that does not check any assertions is also deemed to fail.)
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, *minitest* 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 from Section 1, we get the output

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

Note that even though the test with title *remove* errored, *minitest* went on to run the remaining tests.

If you want more information about testing, see the documentation for *gUnit*, Grace's more comprehensive testing framework, or any of many excellent books on test-driven development. I particularly recommend Kent Beck's *Test Driven Development: By Example*.

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