Implementing and Using Resumable TestFailures in Smalltalk

Joseph Pelrine
MetaProg GmbH

Position paper for
Expanding the Boundaries of Unit Testing
OOPSLA 2002, Seattle, WA

The high performance aspect of extreme Programming derives in part from the rapid feedback cycles in unit testing. Collection testing and validation, however, can be very time-intensive, and can slow down the development process to the point where the advantages of test-driven programming are lost. Through the implementation of ”resumable” test failures, though, this deficit can be compensated for. The ResumableTestFailure (to be introduced in SUnit 3.1) offers a flexible implementation of this in Smalltalk.

The new SUnit release 3.1 adds more functionality at little cost to both Smalltalk's and extreme Programming's premier testing framework. In addition to the #assert:description: family of methods (well-known from JUnit), which allow you to attach arbitrary description strings to assertions, and the implementation of basic logging facilities, the major change is the introduction of a resumable TestFailure.

Why would you need a resumable TestFailure? Take a look at this example from a typical test case method:

```
      aCollection do: [ :each | self assert: each isFoo]
```

In this case, as soon as the first element of the collection isn't Foo, the test stops and returns a failure. Although this information is necessary for test-driven development, it normally isn't sufficient. In most cases, we would like to continue, and see both how many elements and which elements aren't Foo. It would also be nice to log this information. You can do this in this way:

```
aCollection do: [ :each |
      self
            assert: each isFoo
            description: each printString, 'is not Foo'
            resumable: true]
```

This will print out a message on the Transcript for each element that fails. It doesn't cumulate failures, i.e., if the assertion fail 10 times in your test method, you'll still only see one failure.

Implementation

As a result of SUnit being extremely lightweight, it required only minimal effort to implement the functionality required to support ResumableTestFailures.

1. The class ResumableTestFailure was created as a subclass of TestFailure, which itself is defined in the SUnitPreload package. (This package contains all dialect-specific Classes and Methods for SUnit, and makes it possible for the core SUnit package to be dialect-independent).
2. The method Exception>>#isResumable was overwritten to return **true**.
3. The method Exception>>#sunitExitWith: , which normally returns from the exception, was overwritten to resume execution.

While running the test cases, it was noticed that the SUnit framework had a conceptual inconsistency which was overlooked in the original implementation. The method TestResult>>#failures, which returns the collection of failures for a test run, was implemented to be an OrderedCollection. This led to each triggering of a ResumableTestFailure adding yet another failure to the collection. The implementation was changed to be a Set, based on the fact that a test case method is a failure regardless of how many assertions in the method are false. Also, implementing the failure collection as a Set reflects the fact that test cases should be non-deterministic, i.e., the order in which the test cases are executed is irrelevant.

The change in TestResult>>#failures led to a slight change in TestResult>>#defects, which was dependent on the failures being contained in an OrderedCollection. This change was minor, and will not be discussed further.

The implementation also required a method for triggering both regular and resumable TestFailures. The basic method, TestCase>>#assert:description:resumable: is illustrated below:

```
assert: aBoolean description: aString resumable:
resumableBoolean

      | exception |
      aBoolean ifFalse: [
            self logFailure: aString.
            exception := resumableBoolean
                  ifTrue: [ResumableTestFailure]
                  ifFalse: [TestResult failure].
            exception sunitSignalWith: aString]
```

Once again, the implementation of SUnit has proven to be very efficient and flexible when it comes to adding or extending behavior without changing the base packages. Of course, being in Smalltalk helps too – YMMV.

Joseph Pelrine wrote the reference implementation of SUnit 3.0. He is (together with Jeff Odell) currently maintainer of the Camp Smalltalk SUnit distribution on Sourceforge, and has just released SUnit 3.1 to the world.

He can be reached at:

Joseph Pelrine
MetaProg GmbH
Bachlettenstrasse 41
CH-4054 Basel
Switzerland
Email: jpelrine@metaprog.com