Simple Smalltalk Testing: With Patterns

Kent Beck,
First Class Software, Inc.
KentBeck@compuserve.com

(Transcribed to HTML by Ron Jeffries. The software is available for many Smalltalks, and for C++, on my FTP site.)

Introduction

Smalltalk has suffered because it lacked a testing culture. This column describes a simple testing strategy and a framework to support it. The testing strategy and framework are not intended to be complete solutions, but rather a starting point from which industrial strength tools and procedures can be constructed.

The paper is divided into three sections:

- Philosophy - Describes the philosophy of writing and running tests embodied by the framework. Read this section for general background.
- Cookbook - A simple pattern system for writing your own tests.
- Framework - A literate program version of the testing framework. Read this for in-depth knowledge of how the framework operates.
- Example - An example of using the testing framework to test part of the methods in Set.

Philosophy

I don’t like user interface-based tests. In my experience, tests based on user interface scripts are too brittle to be useful. When I was on a project where we used user interface testing, it was common to arrive in the morning to a test report with twenty or thirty failed tests. A quick examination would show that most or all of the failures were actually the program running as expected. Some cosmetic change in the interface had caused the actual output to no longer match the expected output. Our testers spent more time keeping the tests up to date and tracking down false failures and false successes than they did writing new tests.

My solution is to write the tests and check results in Smalltalk. While this approach has the disadvantage that your testers need to be able to write simple Smalltalk programs, the resulting tests are much more stable.

Failures and Errors

The framework distinguishes between failures and errors. A failure is an anticipated problem. When you write tests, you check for expected results. If you get a different answer, that is a failure. An error is more catastrophic, a error condition you didn't check for.

Unit testing

I recommend that developers write their own unit tests, one per class. The framework supports the writing of suites of tests, which can be attached to a class. I recommend that all classes respond to the message "testSuite", returning a suite containing the unit tests. I recommend that developers spend 25-50% of their time developing tests.

Integration testing

I recommend that an independent tester write integration tests. Where should the integration tests go? The recent movement of user interface frameworks to better programmatic access provides one answer- drive the user interface, but do it with the tests. In VisualWorks (the dialect used in the implementation below), you can open an ApplicationModel and begin stuffing values into its ValueHolders, causing all sorts of havoc, with very little trouble.

Running tests

One final bit of philosophy. It is tempting to set up a bunch of test data, then run a bunch of tests, then clean up. In my experience, this always causes more problems that it is worth. Tests end up interacting with one another, and a failure in one test can prevent subsequent tests from running. The testing framework makes it easy to set up a common set of test data, but the data will be created and thrown away for each test. The potential performance problems with this approach shouldn't be a big deal because suites of tests can run unobserved.

Cookbook

Here is a simple pattern system for writing tests. The patterns are:

Pattern	**Purpose**
Fixture	Create a common test fixture.
Test Case	Create the stimulus for a test case.
Check	Check the response for a test case.
Test Suite	Aggregate TestCases.

Fixture

How do you start writing tests?

Testing is one of those impossible tasks. You'd like to be absolutely complete, so you can be sure the software will work. On the other hand, the number of possible states of your program is so large that you can't possibly test all combinations.

If you start with a vague idea of what you'll be testing, you'll never get started. Far better to start with a single configuration whose behavior is predictable. As you get more experience with your software, you will be able to add to the list of configurations.

Such a configuration is called a "fixture". Examples of fixtures are:

Fixture	**Predictions**
1.0 and 2.0	Easy to predict answers to arithmetic problems
Network connection to a known machine	Responses to network packets
#() and #(1 2 3)	Results of sending testing messages

By choosing a fixture you are saying what you will and won't test for. A complete set of tests for a community of objects will have many fixtures, each of which will be tested many ways.

Design a test fixture.

- Subclass TestCase
- Add an instance variable for each known object in the fixture
- Override setUp to initialize the variables

In the example, the test fixture is two Sets, one empty and one with elements. First we subclass TestCase and add instance variables for the objects we will need to reference later:

```
Class: SetTestCase
   superclass: TestCase
   instance variables: empty full
```

Then we override setUp to create the objects for the fixture:

```
SetTestCase>>setUp
   empty := Set new.
   full := Set
   with: #abc
   with: 5
```

Test Case

You have a Fixture, what do you do next?

How do you represent a single unit of testing?

You can predict the results of sending a message to a fixture. You need to represent such a predictable situation somehow.

The simplest way to represent this is interactively. You open an Inspector on your fixture and you start sending it messages. There are two drawbacks to this method. First, you keep sending messages to the same fixture. If a test happens to mess that object up, all subsequent tests will fail, even though the code may be correct. More importantly, though, you can't easily communicate interactive tests to others. If you give someone else your objects, the only way they have of testing them is to have you come and inspect them.

By representing each predictable situation as an object, each with its own fixture, no two tests will ever interfere. Also, you can easily give tests to others to run.

Represent a predictable reaction of a fixture as a method.

- Add a method to TestCase subclass
- Stimulate the fixture in the method

The example code shows this. We can predict that adding "5" to an empty Set will result in "5" being in the set. We add a method to our TestCase subclass. In it we stimulate the fixture:

```
SetTestCase>>testAdd
   empty add: 5.
   ...
```

Once you have stimulated the fixture, you need to add a Check to make sure your prediction came true.

Check

A Test Case stimulates a Fixture.

How do you test for expected results?

If you're testing interactively, you check for expected results directly. If you are looking for a particular return value, you use "print it", and make sure that you got the right object back. If you are looking for side effects, you use the Inspector.

Since tests are in their own objects, you need a way to programmatically look for problems. One way to accomplish this is to use the standard error handling mechanism (Object>>error:) with testing logic to signal errors:

```
2 + 3 = 5 ifFalse: [self error: ‘Wrong answer’]
```

When you’re testing, you’d like to distinguish between errors you are checking for, like getting six as the sum of two and three, and errors you didn’t anticipate, like subscripts being out of bounds or messages not being understood.

There’s not a lot you can do about unanticipated errors (if you did something about them, they wouldn’t be unanticipated any more, would they?) When a catastrophic error occurs, the framework stops running the test case, records the error, and runs the next test case. Since each test case has its own fixture, the error in the previous case will not affect the next.

The testing framework makes checking for expected values simple by providing a method, "should:", that takes a Block as an argument. If the Block evaluates to true, everything is fine. Otherwise, the test case stops running, the failure is recorded, and the next test case runs.

Turn checks into a Block evaluating to a Boolean. Send the Block as the parameter to "should:".

In the example, after stimulating the fixture by adding "5" to an empty Set, we want to check and make sure it’s in there:

```
SetTestCase>>testAdd
   empty add: 5.
   self should: [empty includes: 5]
```

There is a variant on TestCase>>should:. TestCase>>shouldnt: causes the test case to fail if the Block argument evaluates to true. It is there so you don’t have to use "(...) not".

Once you have a test case this far, you can run it. Create an instance of your TestCase subclass, giving it the selector of the testing method. Send "run" to the resulting object:

```
(SetTestCase selector: #testAdd) run
```

If it runs to completion, the test worked. If you get a walkback, something went wrong.

Test Suite

You have several Test Cases.

How do you run lots of tests?

As soon as you have two test cases running, you’ll want to run them both one after the other without having to execute two do it’s. You could just string together a bunch of expressions to create and run test cases. However, when you then wanted to run "this bunch of cases and that bunch of cases" you’d be stuck.

The testing framework provides an object to represent "a bunch of tests", TestSuite. A TestSuite runs a collection of test cases and reports their results all at once. Taking advantage of polymorphism, TestSuites can also contain other TestSuites, so you can put

Joe's tests and Tammy's tests together by creating a higher level suite.

Combine test cases into a test suite.

```
(TestSuite named: 'Money')
   add: (MoneyTestCase selector: #testAdd);
   add: (MoneyTestCase selector: #testSubtract);
   run
```

The result of sending "run" to a TestSuite is a TestResult object. It records all the test cases that caused failures or errors, and the time at which the suite was run.

All of these objects are suitable for storing with the ObjectFiler or BOSS. You can easily store a suite, then bring it in and run it, comparing results with previous runs.

Framework

This section presents the code of the testing framework in literate program style. It is here in case you are curious about the implementation of the framework, or you need to modify it in any way.

When you talk to a tester, the smallest unit of testing they talk about is a test case. TestCase is a User's Object, representing a single test case.

```
Class: TestCase
   superclass: Object
```

Testers talk about setting up a "test fixture", which is an object structure with predictable responses, one that is easy to create and to reason about. Many different test cases can be run against the same fixture.

This distinction is represented in the framework by giving each TestCase a Pluggable Selector. The variable behavior invoked by the selector is the test code. All instances of the same class share the same fixture.

```
Class: TestCase
   superclass: Object
   instance variables: selector
   class variable: FailedCheckSignal
```

TestCase class>>selector: is a Complete Creation Method.

```
TestCase class>>selector: aSymbol
   ^self new setSelector: aSymbol
```

TestCase>>setSelector: is a Creation Parameter Method.

```
TestCase>>setSelector: aSymbol
   selector := aSymbol
```

Subclasses of TestCase are expected to create and destroy test fixtures by overriding the Hook Methods setUp and tearDown, respectively. TestCase itself provides Stub Methods for these methods which do nothing.

```
TestCase>>setUp
   "Run whatever code you need to get ready for the test to run."

TestCase>>tearDown
   "Release whatever resources you used for the test."
```

The simplest way to run a TestCase is just to send it the message "run". Run invokes the set up code, performs the selector, the runs the tear down code. Notice that the tear down code is run regardless of whether there is an error in performing the test. Invoking setUp and tearDown could be encapsulated in an Execute Around Method, but since they aren't part of the public interface they are just open coded here.

```
TestCase>>run
   self setUp.
   [self performTest] valueNowOrOnUnwindDo: [self tearDown]
```

PerformTest just performs the selector.

```
TestCase>>performTest
   self perform: selector
```

A single TestCase is hardly ever interesting, once you have gotten it running. In production, you will want to run many TestCases at a time. Testers talk of running test "suites". TestSuite is a User's Object. It is a Composite of Test Cases.

```
Class: TestSuite
   superclass: Object
   instance variables: name testCases
```

TestSuites are Named Objects. This makes them easy to identify so they can be simply stored on and retrieved from secondary storage. Here is the Complete Creation Method and Creation Parameter Method.

```
TestSuite class>>named: aString
   ^self new setName: aString

TestSuite>>setName: aString
   name := aString.
   testCases := OrderedCollection new
```

The testCases instance variable is initialized right in TestSuite>>setName: because I don't anticipate needing it to be any different kind of collection.

TestSuites have an Accessing Method for their name, in anticipation of user interfaces which will have to display them.

```
TestSuite>>name
   ^name
```

TestSuites have Collection Accessor Methods for adding one or more TestCases.

```
TestSuite>>addTestCase: aTestCase
```

```
    testCases add: aTestCase

TestSuite>>addTestCases: aCollection
    aCollection do: [:each | self addTestCase: each]
```

When you run a TestSuite, you'd like all of its TestCases to run. It's not quite that simple, though. If you have a suite that represents the acceptance test for your application, after it runs you'd like to know how long the suite ran and which of the cases had problems. This is information you would like to be able to store away for future reference.

TestResult is a Result Object for a TestSuite. Running a TestSuite returns a TestResult which records the information described above- the start and stop times of the run, the name of the suite, and any failures or errors.

```
Class: TestResult
    superclass: Object
    instance variables: startTime stopTime testName failures errors
```

When you run a TestSuite, it creates a TestResult which is timestamped before and after the TestCases are run.

```
TestSuite>>run
    | result |
    result := self defaultTestResult.
    result start.
    self run: result.
    result stop.
    ^result
```

TestCase>>run and TestSuite>>run are not polymorphically equivalent. This is a problem that needs to be addressed in future versions of the framework. One option is to have a TestCaseResult which measures time in milliseconds to enable performance regression testing.

The default TestResult is constructed by the TestSuite, using a Default Class.

```
TestSuite>>defaultTestResult
    ^self defaultTestResultClass test: self

TestSuite>>defaultTestResultClass
    ^TestResult
```

A TestResult Complete Creation Method takes a TestSuite.

```
TestResult class>>test: aTest
    ^self new setTest: aTest

TestResult>>setTest: aTest
    testName := aTest name.
    failures := OrderedCollection new.
    errors := OrderedCollection new
```

TestResults are timestamped by sending them the messages start and stop. Since start and stop need to be executed in pairs, they could be hidden behind an Execute Around

Method. This is something else to do later.

```
TestResult>>start
   startTime := Date dateAndTimeNow

TestResult>>stop
   stopTime := Date dateAndTimeNow
```

When a TestSuite runs for a given TestResult, it simply runs each of its TestCases with that TestResult.

```
TestSuite>>run: aTestResult
   testCases do: [:each | each run: aTestResult]
```

#run: is the Composite selector in TestSuite and TestCase, so you can construct TestSuites which contain other TestSuites, instead of or in addition to containing TestCases.

When a TestCase runs for a given TestResult, it should either silently run correctly, add an error to the TestResult, or add a failure to the TestResult. Catching errors is simple-use the system supplied errorSignal. Catching failures must be supported by the TestCase itself. First, we need a Class Initialization Method to create a Signal.

```
TestCase class>>initialize
   FailedCheckSignal := self errorSignal newSignal
   notifierString: 'Check failed - ';
   nameClass: self message: #checkSignal
```

Now we need an Accessing Method.

```
TestCase>>failedCheckSignal
   ^FailedCheckSignal
```

Now, when the TestCase runs with a TestResult, it must catch errors and failures and inform the TestResult, and it must run the tearDown code regardless of whether the test executed correctly. This results in the ugliest method in the framework, because there are two nested error handlers and valueNowOrOnUnwindDo: in one method. There is a missing pattern expressed here and in TestCase>>run about using ensure: to safely run the second halt of an Execute Around Method.

```
TestCase>>run: aTestResult
   self setUp.
   [self errorSignal
      handle: [:ex | aTestResult error: ex errorString in: self]
      do:
         [self failedCheckSignal
            handle: [:ex | aTestResult failure: ex errorString in: self]
            do: [self performTest]]] valueNowOrOnUnwindDo: [self tearDown]
```

When a TestResult is told that an error or failure happened, it records that fact in one of its two collections. For simplicity, the record is just a two element array, but it probably should be a first class object with a timestamp and more details of the blowup.

```
TestResult>>error: aString in: aTestCase
   errors add: (Array with: aTestCase with: aString)

TestResult>>failure: aString in: aTestCase
   failures add: (Array with: aTestCase with: aString)
```

The error case gets invoked if there is ever an uncaught error (for example, message not understood) in the testing method. How do the failures get invoked? TestCase provides two methods that simplify checking for failure. The first, should: aBlock, signals a failure if the evaluation of aBlock returns false. The second, shouldnt: aBlock, does just the opposite.

```
should: aBlock
   aBlock value ifFalse: [self failedCheckSignal raise]

shouldnt: aBlock
   aBlock value ifTrue: [self failedCheckSignal raise]
```

Testing methods will run code to stimulate the test fixture, then check the results inside should: and shouldnt: blocks.

Example

Okay, that's how it works, how do you use it? Here's a short example that tests a few of the messages supported by Sets. First we subclass TestCase, because we'll always want a couple of interesting Sets around to play with.

```
Class: SetTestCase
   superclass: TestCase
   instance variables: empty full
```

Now we need to initialize these variables, so we subclass setUp.

```
SetTestCase>>setUp
   empty := Set new.
   full := Set
      with: #abc
      with: 5
```

Now we need a testing method. Let's test to see if adding an element to a Set really works.

```
SetTestCase>>testAdd
   empty add: 5.
   self should: [empty includes: 5]
```

Now we can run a test case by evaluating "(SetTestCase selector: #testAdd) run".

Here's a case that uses shouldnt:. It reads "after removing 5 from full, full should include #abc and it shouldn't include 5."

```
SetTestCase>>testRemove
   full remove: 5.
   self should: [full includes: #abc].
```

```
    self shouldnt: [full includes: 5]
```

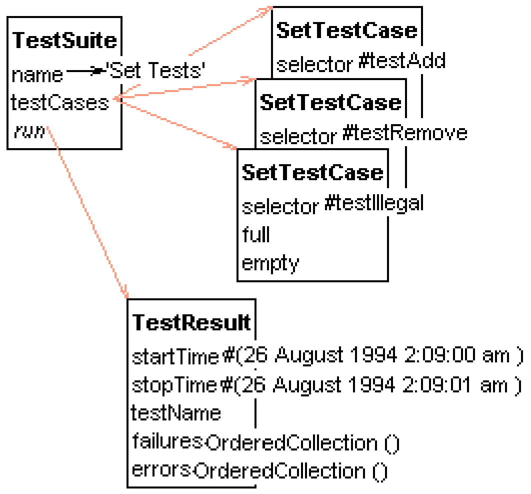
Here's one that makes sure an error is signalled if you try to do keyed access.

```
SetTestCase>>testIllegal
    self should: [self errorSignal handle: [:ex | true] do: [empty at: 5. false]]
```

Now we can put together a TestSuite.

```
| suite |
suite := TestSuite named: 'Set Tests'.
suite addTestCase: (SetTestCase selector: #testAdd).
suite addTestCase: (SetTestCase selector: #testRemove).
suite addTestCase: (SetTestCase selector: #testIllegal).
^suite
```

Here is an Object Explorer picture of the suite and the TestResult we get back when we run it.



The test methods shown above only cover a fraction of the functionality in Set. Writing tests for all the public methods in Set is a daunting task. However, as Hal Hildebrand told me after using an earlier version of this framework, "If the underlying objects don't work, nothing else matters. You have to write the tests to make sure everything is working."