Chapter

30

Testing

It seems easier to write error free code in Smalltalk than in other languages. I suspect that this is partly because it's difficult to have off-by-one errors, the garbage collector takes care of memory management, there is a very rich class library, and there are no pointers. However, we still do create errors, both coding errors and functionality errors — ie, we didn't code the correct functionality. So we need to test the software. Smalltalk code is easy to test and debug because it is so interactive. However, as with any other language, once we've done our initial development and testing, it is psychologically difficult to thoroughly retest after making changes. What we need is way to develop test cases that can be retained and replayed after changes.

In this chapter we will look at a way to develop tests, and a user interface to run the tests. I want to acknowledge Kent Beck and his article, *Simple Smalltalk Testing* (The Smalltalk Report, October 1994), for presenting some interesting ideas that I used in developing the testing scheme that follows. This chapter contains a lot of code. If you work through it then by the end of the chapter you should have a working test facility. Hopefully you will also have learned some useful ideas for other problems you are working on. As in other chapters, to save on space I won't generally use accessors for instance variables and the formatting is sometimes tighter than usual. The code can also be found in the file `testing.st`.

The testing scheme here is not designed to test user interfaces. There are specialist products available to capture and replay keystrokes and mouse actions. A good application will separate out the user interface from the domain model, and it should be possible to test the domain model in isolation from the user interface. This is where the classes described in this chapter can help. The scheme described uses a test manager to runs tests. Rather than embed the tests within the classes that we want to test, we have separate test classes that exist in a well-defined hierarchy. The test manager makes use of this hierarchy when figuring out which tests to run.

The Test class

The basic concept is that we have a *TstTest* class (Tst is the prefix I am using for my testing classes) that provides the behavior for running tests cases. We organize our tests by subclassing off TstTest. Each test class can have both individual tests to run, and subclasses that have their own individual tests and their own subclasses.

This hierarchy mechanism gives us a way to organize our tests in some logical fashion. For example, you might have the immediate subclasses of TstTest be ProductTests, SupportClassTests, and MyTests. Subclassed off ProductTests you might have SystemTests and ComponentTests. Under MyTests could be tests that you have written in conjunction with new code you are developing. When you are satisfied with the code and have officially blessed it, you might move the tests you wrote into the ProductTests or SupportClassTests hierarchy. Figure 30-1 illustrates the hierarchy.

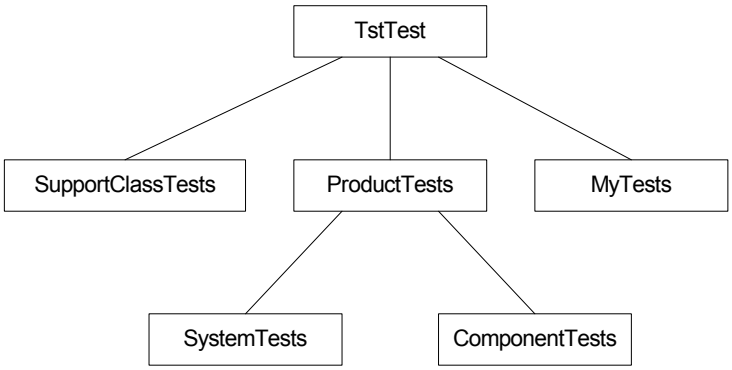

Figure 30-1.
The TstTest class hierarchy.

We run the tests using a test manager that organizes the test classes by their inheritance hierarchy. There are two ways to run tests. The first way runs a hierarchy of tests and reports on the results in a *TstResult*. The second way runs individually selected tests and raises an exception when a failure occurs, allowing you to go into the debugger and look at the values.

Building the TstTest class

```
Object subclass: #TstTest
   instanceVariableNames: 'selector '
   classVariableNames: 'TestFailedSignal '
   poolDictionaries: ''
   category: 'TST-Test'

TstTest class
   instanceVariableNames: 'DisplayName '
```

If a test fails, our code raises a TestFailedSignal. Rather than create a signal for each test, we create it for the class, which means we need a class initialization method. In this method we also set the name to be displayed by the Test Manager.

```
TstTest class>>initialize
   "self initialize"
   DisplayName := 'All Tests'.
   TestFailedSignal := (self errorSignal newSignal)
         notifierString: 'Test failed: ';
         nameClass: self message: #testFailedSignal;
         yourself

TstTest class>>testFailedSignal
   ^TestFailedSignal
```

We don't really need to send the `nameClass:message:` message when creating the signal, but it's good practice to include it when creating a signal because it allows other methods to create their own instance of a TestFailedSignal.

The general idea is that a subclass of TstTest can have its own subclass hierarchy and some individual tests to run. The test manager will find all the individual tests for a given test class, and create an instance of the class for each test to run. The individual tests each have their own method name in the `tests` protocol. So when an instance is created, the method selector to run is specified.

```
TstTest class>>newTest: aSelector
   ^self new initialize: aSelector

TstTest>>initialize: aSelector
   selector := aSelector

TstTest>>printOn: aStream
   aStream
      print: self class;
      nextPutAll: '>>';
      print: selector
```

What does an individual test look like? Below is an example of one. Notice that it does some work, then sends the `expect:string:` message. This tells the test code that it expects the code in the block to evaluate to *true*. If the code evaluates to *false*, the test fails. The string that is sent as a parameter is simply a way of making it easier to find the part of the test that failed. If you only have one result, you can do `self expect: [some code].`

```
SomeTest>>driveBusy
   |  request response |
   request := self myMountRequest: 'PAYROLL' drive: 'DRIVE2'.
   response := self mySendMessage: request.
   self expect: [response isSuccessResponse] string: 'Mount first
volume'.
   request := self myMountRequest: 'BKUP950524' drive: 'DRIVE2'.
   response := self mySendMessage: request.
   self expect: [response isFailureResponse] string: 'Mount second
volume'
```

Let's take a look at what `expect:` and `expect:string:` do.

```
TstTest>>expect: aBlock
   self expect: aBlock string: nil

TstTest>>expect: aBlock string: aStringOrNil
   aBlock value ifFalse:
      [self class testFailedSignal raiseErrorString: aStringOrNil]
```

If the block evaluates to false, we ask our TestFailedSignal to raise an exception. If we specified a string, then it raises the exception using our string as the error string. I'm not going to show them, but you can write the corresponding `reject:` and `reject:string:` methods that reject the result — ie, raise the exception if the block evaluates to *true*.

Now we need to see how the exception is handled. To do this we look at how the tests are run. Remember that we specified the selector (or method) to run when we created the instance of the test and stored the selector in an instance variable. To run the method, we have to perform the selector.

```
TstTest>>performTest
   self perform: selector
```

We have two ways of performing the test. We can send the `run` message or the `run:` message to the test instance. If we send `run`, there is no exception handling and if a test fails, the exception will raise a notifier window, allowing us to immediately debug the code that failed. If we send `run:`, we specify a test result object in which we will record information about the test.

```
TstTest>>run
   self performTest

TstTest>>run: aTestResult
   self errorSignal
      handle: [:ex | aTestResult error: ex localErrorString in: self]
      do: [self class testFailedSignal
         handle: [:ex | aTestResult failure: ex localErrorString in:
self]
         do:
            [aTestResult incrementCount.
            self performTest]]
```

There are two things that can go wrong with a test: a *failure*, where the result is different from what we expected, and an *error*, where there is an error in the test (for example, the test might send a message that is not understood). We trap both conditions in the `run:` method.

The innermost exception handler looks specifically for TestFailedSignal exception that we raise if we get an unexpected result. If it comes across one, in the `handle:` block it logs the *failure* in the test result. The outermost exception handler traps all exceptions other than those raised by TestFailedSignal. So if we send a not understood message, or do a division by zero, or try to access a non-existent array element, they will all be handled in the `handle:` block, which logs the *error* in the test result.

Finally, on the class side, we write several methods that will be used when interacting with the test manager. The first method specifies the protocol that contains all the individual test case selectors. The second method uses meta-programming to return the names of all the individual test cases in sorted order, and the third method returns the actual test cases.

```
TstTest class>>testProtocol
   ^#tests

TstTest class>>individualTestNames
   ^(self organization listAtCategoryNamed: self testProtocol)
asSortedCollection

TstTest class>>individualTests
   ^self individualTestNames
      collect: [:each | self newTest: each]
```

The next two methods are used to return a collection of the test superclasses of the test that has been selected to run. This collection will be used for doing setup and cleanup work prior to and after running the test.

```
TstTest class >>rootTestClass
   ^TstTest

TstTest class>>testSuperclasses
   "Returns a collection of the superclasses up to the root test
class"
   | collection class |
   collection := OrderedCollection new.
   class := self.
   [class == self rootTestClass]
      whileFalse:
         [class := class superclass.
         collection add: class].
   ^collection
```

The Test Result

The test result is the object that stores information about the errors and failures.

```
Object subclass: #TstResult
   instanceVariableNames: 'test startTime stopTime count failures
errors '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'TST-Test'

TstResult class>>newFor: aTest
   ^super new initialize: aTest

TstResult>>initialize: aTest
   test := aTest.
   count := 0.
   failures := OrderedCollection new.
   errors := OrderedCollection new

TstResult>>incrementCount
   count := count + 1
```

The `failure:in:` method below stores the test case and string in the failure collection. There is an equivalent (but not shown) method for `error:in:`. Since there are only two parameters, the test case and the string, we store them in an Association. If we ever wanted to record more than two pieces of information, we would create a new class.

```
TstResult>>failure: aString in: aTestCase
   failures add: aTestCase -> aString

TstResult>>start
   startTime := Timestamp now
```

There is a corresponding `stop` method. The final thing is to provide a `printOn:` method for TstResult. Figure 30-2 shows how a test result might appear when displayed in an inspector window. Following the figure is the code that generates the output.

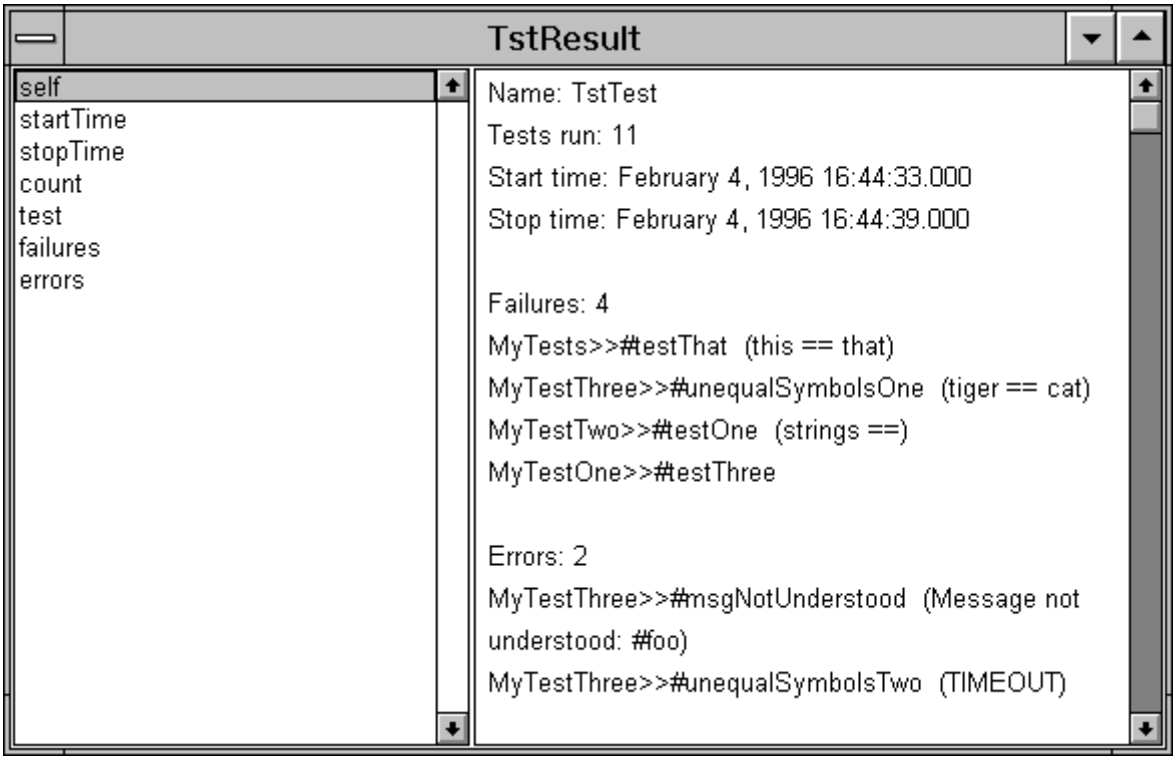

Figure 30-2. The output of a test result.

```
TstResult>>printOn: aStream
   aStream nextPutAll: 'Name: '.
   test printOn: aStream.
   aStream
      cr; nextPutAll: 'Tests run: '; print: count;
      cr; nextPutAll: 'Start time: '; print: startTime;
      cr; nextPutAll: 'Stop time: '; print: stopTime.
   self myPrintFailuresOn: aStream.
   self myPrintErrorsOn: aStream

TstResult>>myPrintFailuresOn: aStream
   failures isEmpty ifFalse:
      [self
         myPrintCollection: failures
         label: 'Failures'
         on: aStream]

TstResult>>myPrintCollection: aCollection label: aString on: aStream
   aStream
      cr; cr;
      nextPutAll: aString;
      nextPutAll: ': ';
      print: aCollection size.
   aCollection do:
      [:each |
         aStream cr.
         each key printOn: aStream.
         each value notNil
            ifTrue:
               [aStream nextPutAll: '  (('.
               aStream nextPutAll: each value.
               aStream nextPut: $)]]
```

The Test Manager

Now that we have the test class, let's look at how we might create a user interface to run the tests. First, create a window with a read-only input field, three action buttons, a check box, and two list widgets. Figure 30-3 shows the window.

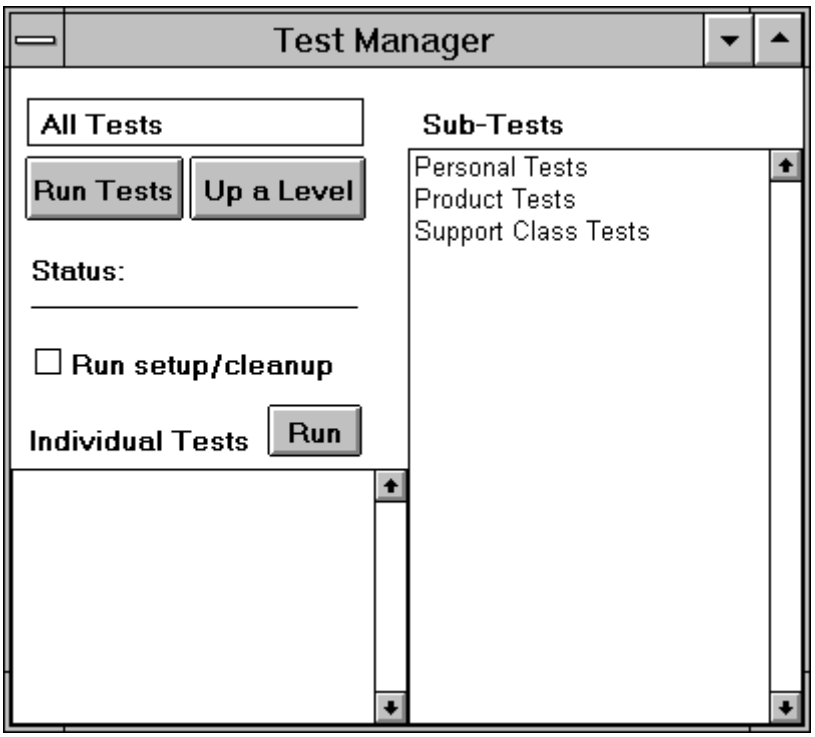

Figure 30-3.
The Test Manager.

The read-only input field at the top left displays the currently selected test. To the right is another window showing all the immediate subtests of the current test. Below is a pane showing all the individual tests associated with the current test. There are three action buttons: Run Tests, which runs the hierarchy of tests from the displayed test down; Up a Level, which makes the next higher test in the hierarchy the current test; and Run, which runs the tests that have been selected in the Individual Test pane. Below are the class definition and `initialize` method. In the `aspects` protocol we also need to write methods to get the instance variables (we won't show the getters here).

```
ApplicationModel subclass: #TstTestManager
   instanceVariableNames: 'mainTest subTestList individualTestList
doSetupCleanup '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'STBE-Testing'

TstTestManager>>rootTestClass
   ^TstTest

TstTestManager>>initialize
   mainTest := self rootTestClass asValue.
   doSetupCleanup := false asValue.
   subTestList := SelectionInList new.
   subTestList selectionIndexHolder onChangeSend: #changedSubTest to:
self.
   individualTestList := MultiSelectionInList new.
```

```
    self myBuildLists.
```

Let's take a look at what we are doing in the `initialize` method. First, we set our main test from the root test class. Next we set our *subTestList* variable to be a SelectionInList and we register the Test Manager as a dependent of the SelectionInList, asking to be sent the `changedSubTest` message if the user selects or deselects a test in the right hand pane. Finally, we initialize the individual test list to be a MultiSelectionInList, which allows the user to select multiple individual test cases to run. Finally, after setting up the variables, we build the subtest and individual test lists for the current main test.

In the `initialize` method we told the SelectionInList object to send us a message if the selection index changes. Here's what we do in the `changeSubTest` method. If the user selected a test in the right hand pane, we want to make it the main test then display its sub-tests and its individual tests, if any.

```
TstTestManager>>changedSubTest
   self subTestList selection notNil
      ifTrue:
         [self mainTest value: self subTestList selection.
         self myBuildLists]
```

If the user wants to go back up a level in the test hierarchy and presses the Up a Level button, we execute the method `showSuperclass`, which sets the main test to be the superclass of the current test (unless we are already at the root test class), and rebuilds the subtest and individual test lists.

```
TstTestManager>>showSuperclass
   | mainTestClass |
   mainTestClass := self mainTest value.
   mainTestClass == self rootTestClass
      ifFalse:
         [self mainTest value: mainTestClass superclass.
         self myBuildLists]
```

We've now seen several references to `myBuildLists`, so let's take a look at the method.

```
TstTestManager>>myBuildLists
   | test |
   test := self mainTest value.
   self subTestList list: test subclasses asSortedCollection.
   self individualTestList list: test individualTestNames
```

There are several points to note when looking at this method. When we create the list of subtests, we want to display the subtests in alphabetic order, so we send the `asSortedCollection` message to the collection. For this to work, we need to make sure that our test case classes respond to the <= message which is used to sort a collection.

```
TstTest class>> <= anObject
   ^self displayString <= anObject displayString

TstTest class>>displayString
   ^DisplayName notNil
      ifTrue: [DisplayName]
      ifFalse: [self name]

ProductTests class>>initialize
   "self initialize"
```

```
super initialize.
DisplayName := 'Product Tests'
```

Note that these methods are all written on the class side since we are displaying classes in the input field and in the subtest list. The `displayString` message is sent to an object by a SelectionInList and by an Input field (for the input field to work, in the Properties Tool you must define it as holding an Object). We override the default `displayString` to return the display name, assuming that the DisplayName class instance variable has been set in the class. This allows us to display something more reasonable than the class name. If we haven't set the DisplayName, the class name will be displayed instead. The third method show here gives an example of how a class would set its DisplayName.

Running Tests

All Tests

When you press the Run Tests button, you run all the individual tests associated with the displayed test, then recursively go through the sub-tests, running all the individual tests associated with each sub-test and all their sub-tests. The action associated with the Run Tests button is the `runTests` method.

```
TstTestManager>>runTests
   | testClass  testResult |
   testClass := self mainTest value.
   testResult := self myTestResultFor: testClass.
   self myDoHierarchySetupFor: testClass.
   testResult start.
   self myRunTestsForClass: testClass result: testResult.
   testResult stop.
   self myDoHierarchyCleanupFor: testClass.
   testResult inspect.
```

We start by creating a test result which we use to record the results of the tests. The code to return a test result is shown here.

```
TstResult>>myTestResultFor: aTest
   ^TstResult newFor: aTest
```

We then do any setup for higher level classes, run the test hierarchy, and do any cleanup for higher level classes. Each test class in the hierarchy can do setup before and cleanup after running the tests. This allows the classes to set up any conditions that are required for the test. Typically this involves such things as populating classes with needed data or making sure that databases have the correct records. We give each class in the hierarchy the opportunity to do any setup and cleanup. For setup we start at the root test class and work down to our immediate superclass. For cleanup we do this in reverse, starting at our superclass and working up to the root test class. We'll run setup and cleanup for the current test and its subtests later.

(One thing to note is that we assume the setup and cleanup will work. This test manager doesn't have the ability to handle problems in setup and cleanup, or handle exceptions raised there.)

We have a method that determines whether any setup should be done, and another method that does the setup for all the superclasses in the hierarchy. We won't show them, but there are similar methods for doing cleanup.

```
TstTestManager>>myDoSetupFor: aClass
   (aClass class includesSelector: #setUp)
      ifTrue: [aClass setUp]

TstTestManager>>myDoHierarchySetupFor: aClass
   aClass testSuperclasses
      reverseDo: [:each | self myDoSetupFor: each]
```

A feature of our setup is that we don't want to inherit setup methods from superclasses. If we allowed inheritance, we would do the same setup more than once if a particular class in the hierarchy didn't need to do any setup. We handle this with the code:

```
each class includesSelector: #setUp
```

This code checks to see if the named selector has been defined by the class. We can't use `respondsTo:` because this will also return *true* if the method is inherited from a superclass, which we specifically don't want. The cleanup methods looks very similar except that they use `cleanUp` rather than `setUp`, and because cleanup should occur in the opposite order to setup, `myDoHierarchyCleanupFor:` sends `do:` rather than `reverseDo:`.

A point to note is that we define the `setUp` and `cleanUp` methods on the class of the test classes side rather than the instance side. This is because we want to potentially invoke these methods for several classes in the test case hierarchy. The only way we could do this on the instance side would be to create an instance of each class in the hierarchy, then send it the message. It's a lot cleaner to put the behavior on the class side.

Here is another method for which we need to look at the code.

```
TstTestManager>>myRunTestsForClass: aClass result: aTestResult
   self myDoSetupFor: aClass.
   aClass individualTests
      do: [:each | each run: aTestResult].
   aClass subclasses do:
      [:each | self myRunTestsForClass: each result: aTestResult].
   self myDoCleanupFor: aClass
```

When we run tests for a class, we do any set up for the test class we are running, we run any individual tests defined on the test class, we go through all our sub tests, asking them to do exactly what we are doing, then we do any clean up for the test class. So in this method, we recursively ask all our subclasses to execute this same method. This gives us a depth first recursion through the whole sub test hierarchy for this test. Well, that's it for running a hierarchy of tests. The other way to run tests is by selecting individual tests to run.

Selected Tests

The Test Manager has a different philosophy when running individual tests. Rather than going through all the setup and cleanup of classes higher in the test hierarchy, it assumes that any setup has already been done (you can change this by clicking the Run setup/cleanup check box). However, it *does* do setup and cleanup for the current test class. It also runs the tests without a test result wrapper, which means that if an individual test fails, it will raise a Notifier window and let you start debugging. You can select multiple tests to run in the individual test window, then when you press the Run button, the `runSelected` method is executed.

```
TstTestManager>>runSelected
```

```
   | selectedTests testClass testCollection |
   selectedTests := self individualTestList selections.
   selectedTests isEmpty ifTrue: [^self].
   testClass := self mainTest value.
   testCollection:= selectedTests collect: [:each | testClass
newTest: each].
   self doSetupCleanup value ifTrue: [self myDoHierarchySetupFor:
testClass].
   self myRunSelectedCases: testCollection forClass: testClass.
   self doSetupCleanup value ifTrue: [self myDoHierarchyCleanupFor:
testClass]].
```

If any individual tests have been selected, we create a collection of test objects, one for each test selected. We then run the selection in another method.

```
TstTestManager>>myRunSelectedCases: aCollection forClass: aClass
   self myDoSetupFor: aClass.
   aCollection do: [:each | each run].
   self myDoCleanupFor: aClass
```

At this point we do setup for the class we are in, we run each test without a test result wrapper, then we do any cleanup. Because we are not using a test result, any failure will cause a Notifier window to be displayed.

Running tests that never complete

Some tests may never complete because they are waiting for something that doesn't happen. This presents a problem in the current scheme because the rest of the tests will never run, and you will not see the test result. If you have such a situation, you can replace the `performTest` method with the following code. However, a more likely scenario is that you have only a few tests with the potential to wait forever. For example, your System Tests might have the potential to hang, but other tests will always complete. In this case, you would override `performTest` in the SystemTest class. Or you could set up a two branch hierarchy under TstTest, one branch for tests that have the potential to hang and one branch for those that don't.

```
TstTestManager>>performTest
   | testProcess timeoutProcess sharedQueue exceptionOrNil |
   super class showContext: thisContext.
   sharedQueue := SharedQueue new.
   testProcess :=
      [self errorSignal
         handle: [:ex | sharedQueue nextPut: ex]
         do:
            [self perform: selector.
            sharedQueue nextPut: nil]] fork.
   timeoutProcess :=
      [(Delay forSeconds: self timeoutValue) wait.
      sharedQueue nextPut: (self errorSignal newException
errorString: 'TIMEOUT')] fork.
   exceptionOrNil := sharedQueue next.
   testProcess terminate.
   timeoutProcess terminate.
   exceptionOrNil notNil ifTrue:
      [exceptionOrNil propagateFrom: thisContext]
```

We run the test in a forked process and create another forked process that will wait for some timeout period. Both forked processes put something on a SharedQueue when they have done their work, and the main flow

waits until it can read something from the queue. It then takes action based on what it gets off the queue. Let's look at the details.

In *testProcess* we run the test, wrapped in an all-encompassing signal handler. If the test succeeds, we put *nil* on the shared queue. Seeing *nil* tells the main flow that the test succeeded. If the test does not succeed, an exception is raised as we saw earlier in the chapter. We trap the exception and put it on the shared queue. In *timeoutProcess* we simply wait for some timeout period then we create an exception and put it on the shared queue. Since we are using a message send to get the timeout value (you'll have to write the `timeoutValue` method and have it return a number), subclasses can override the value if they need a different timeout period.

The main flow of control waits for one of the processes to put something on the shared queue. This will either be *nil*, which means the test succeeded, or an exception, which we raise again so that the code in the `run` or `run:` method can handle it appropriately. It also terminates both forked processes since they are no longer needed (sending `terminate` to an already terminated process is benign).

We do things this way because when a process is forked it gets its own context. A signal handler in one context can't trap an exception raised in another context, so we trap the signal in the forked process, give it to the main process, and let the main process start it up again as though it came from the main process.

Loading in the tests

Besides having a mechanism for filing in all the tests or a subset of the tests, you might find it useful to have each class responsible for filing in tests that test it. A mechanism for doing this is to have fileIn code in a class-side method such as:

```
MyClass class>>fileInTests
   "self fileInTests"
   #( 'MyTest1.st'
      'MyTest2.st'
   ) do: [:each | (self myTestDir construct: each) fileIn]
```

In this chapter we have looked at a test scheme that makes use of a test manager to run the tests. We could extend this scheme and give each class the ability to test itself using the TstClass functionality. For example, we might write a method such as the following on the class side of classes we wish to test.

```
MyClass class>>runTests
   "self runTests"
   #( MyTestOne
      MyTestTwo
   ) do: [:className | (Smalltalk at: className) individualTests do:
[:test | test run]]
```

However, the technique shown still relies on there being separate classes to do the testing. An alternative scheme would be to keep the test cases completely within the classes to be tested, but such a scheme goes beyond the scope of this chapter.