SUnit Explained Revisited

Stéphane Ducasse

ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

SUnit is a minimal yet powerful framework that supports the creation of tests. SUnit is the mother of unit test frameworks. SUnit was developed originally by Kent Beck and get extended by Joseph Pelrine and others over several iterations to take into account the notion of resources that we will illustrate later. The interest for SUnit is not limited to Smalltalk or Squeak. Indeed legions of developers understood the power of unit testing and now versions of SUnit exist in nearly any language going from Java, Python, Perl, Oracle and lot others [SUnit]. The current version of SUnit is 3.1. The official web site of SUnit is http://sunit.sourceforge.net/.

Testing and building regression test suites is not new and everybody knows that regression tests are a good way to catch errors. Extreme Programming by putting testing in the core of its methodology is bringing a new light on testing which is a not so liked discipline. The Smalltalk community has a long tradition of test due to the incremental development supported by its programming environment. However, once you write tests in a workspace or as example methods there is no easy way to keep track of them and to automatically run them and tests that you cannot automatically run are of little interests. Moreover, having examples often does not tell to the reader what are the expect results, lot of the logic is left unspecified. That's why SUnit is interesting because it allows you to structure, describe the context of tests and to run them automatically. In less than two minutes you can write tests using SUnit instead of writing small code snippets and get all the advantage of stored and automatically executable tests.

In this article we start by discussing the interest of testing, then we present an exemple with SUnit and we go deep into the SUnit implementation.

1. Testing and Tests

Most of the developers believe that tests are a lost of time. Who has not heard: "I would write tests if I would have more time". If you write code that should never be changed indeed you should not write tests, but this also means that you application is not really used or useful. In fact tests are an investment for the future. In particular, having a suite of tests is extremely useful and allow one to gain a lot of time when your application changes.

Tests play several roles: first they are an active and *always synchronized* documentation of the functionality they cover. Second they represent the confidence that developers can have into a piece of functionality. They help you to find extremely fast the parts that break to due introduced changes. It is obvious but simply true. Finally, writing tests in the same time or even before writing code force you to think about the functionality you want to design. By writing tests first you have to clearly state the context in which your functionality will run, the way it will interact and more important the expected results. Moreover, when you are writing tests you are your first client and your code will naturally improves.

The culture of tests has always been present in the Smalltalk community because a method is compiled and we write a small expression to test it. This practice supports the extremely tight incremental development cycle promoted by Smalltalk. However, doing so does not bring the maximum benefit from testing. Because tests are not stored, reachable and run automatically. Moreover it often happens that the context of the tests is left unspecified so the reader has to interpret the obtained results and assess they are right or wrong.

It is clear that we cannot tests all the aspects of an application. Covering a complete application is simply impossible and should not be goal of testing. It may also happen that even with a good test suite some bugs can creep into the application and be left hidden waiting for an opportunity to damage your system. This is not a problem as soon as if you trap a bug you write a test that covers it.

Writing good tests is a technique that can be easily learnt by practising. Let us look at the properties that tests should have to get a maximum benefit

- Repeatable. We should be able to repeat a test as much as we want.
- Without human intervention. Tests should be repeated without any human intervention. You should be able to run them during the night.
- Telling a story. A test should cover one aspect of a piece of code. A test should act as a scenario that you would like to read to understand a functionality.
- Having a change frequency lower than the one of the covered functionality. Indeed you do not want to change all your tests every times you modify your application. One way to achieve this property is to write tests based on the interfaces of the tested functionality.

Besides the property of the test itself another important point while writing test suites is that the number of tests should be somehow proportional to the number of tested functionality. For example, changing one aspect of the system should not break all the tests you wrote but only a limited number. This is important because having 100 tests broken should be a much more important message for you than having 10 tests failing.

eXtreme Programming proposes to write tests even before writing code. This may seems against our deep developer habits. Here are the observations we made while practising up front tests writing. Up front testing help to know what you want to code, they help to know when you are done, they help to conceptualize the functionality of a class and to design the interface. Now it is time to write a first test and to convince you that this is a pity not using SUnit.

2. SUnit by Example

Before going into the detail of SUnit, we show an example step by step. We use the example testing the class `Set` that is included in the SUnit distribution, so that you can read the code directly in your favorite Smalltalk.

Step 1. First you should subclass the `TestCase` class as follow:

```
TestCase subclass: #ExampleSetTest
     instanceVariableNames: 'full empty'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'SUnit-Tests'
```

The class `ExampleSetTest` groups all tests related to the class test. It defines the context of all the tests that we will specify. Here the context is described by specifying two instance variables `full` and `empty` that represent a full and empty set.

Step 2. We define the method setUp as follow. The method setup acts as a context definer method or initiliaze method. It is invoked before the execution of any test method defined in this class. Here we initialize the `empty` variable to refer to an empty set and the `full` variable to refer to a set containing two elements.

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

This method defines the context of any tests defined in the class in testing jargon it is called the *fixture* of the test.

Step 3. We define some tests by defining some methods on the class `ExampleSetTest`. Basically one method represents one test. If your test methods start with the string ‘test’ the framework will collect them automatically for you into test suites ready to be executed.

The first test named `testIncludes`, tests the includes method of a `Set`. We say that sending the message `includes: 5` to a set containing 5 should return `true`.Here we see clearly that the test relies on the fact that the `setUp` method has been run before.

```
ExampleSetTest>>testIncludes
   self assert: (full includes: 5).
   self assert: (full includes: #abc)
```

The second test named `testOccurrences` verifies that the occurrences of 5 in the full set is equal to one even if we add another element 5 to the set.

```
ExampleSetTest>>testOccurrences
   self assert: (empty occurrencesOf: 0) = 0.
   self assert: (full occurrencesOf: 5) = 1.
   full add: 5.
   self assert: (full occurrencesOf: 5) = 1
```

Finally we test that if we remove the element 5 from a set the set does not contain it any more.

```
 ExampleSetTest>>testRemove
   full remove: 5.
   self assert: (full includes: #abc).
   self deny: (full includes: 5)
```

Step 4. Now we can execute the tests. This is possible using the user interface of SUnit. This interface depends on the dialect you use. In Squeak and VisualWorks, you should execute TestRunner open. You should obtain the Figure 1. You can also run you tests by executing the following code: `(ExampleSetTest selector: #testRemove) run`. This expression is equivalent to the shorter one `ExampleSetTest run: #testRemove`. We usually always include such kind of expression in the comment of our tests to be able to run them while browsing them as shown below.

```
ExampleSetTest>>testRemove
   "self run: #testRemove"

   full remove: 5.
   self assert: (full includes: #abc).
   self deny: (full includes: 5)
```

To debug a test use the following expressions: (`ExampleSetTest selector: #testRemove`) `debug` or `ExampleSetTest debug: #testRemove`.

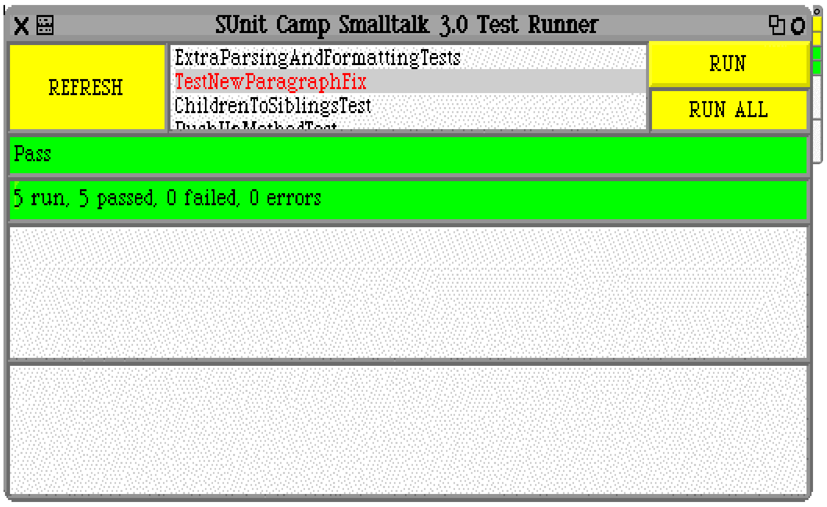
SUnit Camp Smalltalk 3.0 Test Runner
REFRESH
ExtraParsingAndFormattingTests
TestNewParagraphFix
ChildrenToSiblingsTest
RUN
RUN ALL
Pass
5 run, 5 passed, 0 failed, 0 errors

Figure 1 The user interface of SUnit in Squeak
Figure 2 . Here a test run and all the tests passed.

Some Explanations. The method assert: which is defined on the class `TestCase` requires a boolean as argument. This boolean represents the value of a tested expression. When the argument is true, the expression is considered to be correct, we say that the test is valid. When the argument is false, then the test failed. In fact SUnit consider two kinds of errors: the failures, i.e., when a test is not valid and the errors which are unexpected situations occurring while the test is running. An error is by its nature something that has not been tested but that happened like an out of bounds error. The method `deny:` is the negation of `assert:`. Hence `aTest deny: anExpression` is equal to `aTest assert: anExpression not`.

SUnit offers two methods `should:raise:` and `shouldnt:raise:` (`aTest should: aBlock raise: anException`) to test that exceptions have been raised during the execution of an expression. The following test illustrates the use of this method.

```
ExampleSetTest>>testIllegal
   self should: [empty at: 5] raise: Error.
   self should: [empty at: 5 put: #abc] raise: Error
```

Note that if you look in the example provided by SUnit you will found the following definition for the same test. Here the exception is provided via the `TestResult` class. This is because SUnit is running on all the Smalltalk dialects and the SUnit developers have factored out the variant part such as the name of the exception. So if you write tests that are intended to be cross dialects look at the class `TestResult`.

```
ExampleSetTest>>testIllegal
   self should: [empty at: 5] raise: TestResult error.
   self should: [empty at: 5 put: #abc] raise: TestResult error
```

3. Basic How To

If you are familiar with other testing frameworks such as JUnit, remember that JUnit has been widely inspired from SUnit so there are a lot of similarity. Normally SUnit has an associate UI that allows one to run tests. But you may have some questions that we will answer now.

How do I run a single test? Ask the testcase to build a suite for you by using the method run:

```
ExampleSetTest run: #testRemove
1 run, 1 passed, 0 failed, 0 errors
```

How do I run all the tests in a TestCase subclass? Just ask the class itself to build the test suite for you. Only the tests starting with the string ‘test’ will be added to the suite. Therefore we also reply to the question: How do I turn all the test* methods into a TestSuite?

```
ExampleSetTest suite run
9run, 9 passed, 0 failed, 0 errors
```

Must I subclass TestCase? In JUnit we can build a TestSuite from an arbitrary class containing test* methods. In Smalltalk you can do the same but you will have then to create it by hand and your class will have to implement all the essential TestCase methods so we suggest you not do it. The framework is there so use it.

How do I get my test cases/suites into the TestRunner tool? Depending on your dialect it may happen that the TestRunner tool does not updated when you created a new TestCase. So simply close it and reopen it.

4. The SUnit Framework

SUnit 3.1 introduces the notion of resources that are mandatory when one need to build tests that require long set up phases. A test resource specifies a set up that is only executed once for a set of tests contrary to the `TestCase` method which is executed before every test execution.

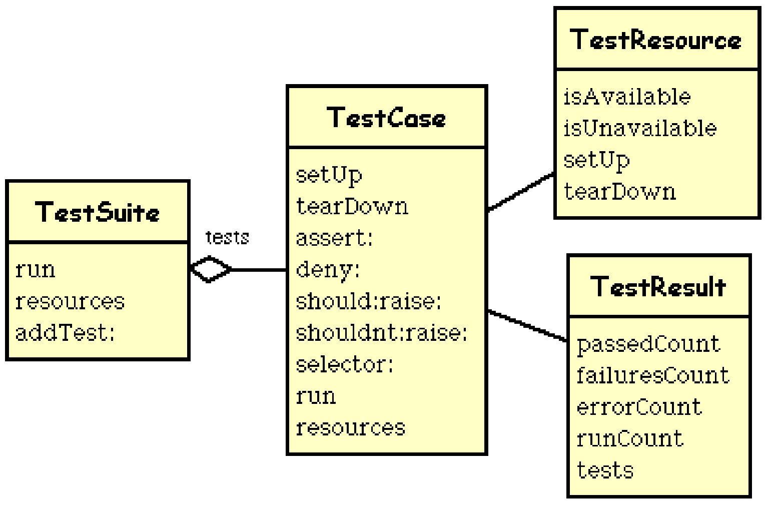

Figure 3 The four classes representing the core of SUnit.

SUnit is constituted by four main classes, namely `TestCase`, `TestSuite`, `TestResult` et `TestResource` as shown in the figure 2.

The class `TestCase` represents a test or more generally a family of tests that share a common context. The context is specified by the declaration of instance variables on a subclass of `Test-Case` and by the specialization of the method `setUp` which initializes the context in which the will be executed. The class `TestCase` defines also the method `tearDown` that is responsible for releasing if necessary the object allocated during the execution of the method `setUp`. The method `tearDown` is invoked after the execution of every tests.

The class `TestSuite` represents a collection of tests. An instance of `TestSuite` is composed by instance of `TestCase` subclasses (a instance of `TestCase` is characterized by the selector that should run) and `TestSuite`. The classes `TestSuite` and `TestCase` form a composite pattern in which `TestSuite` is the composite and `TestCase` the leaves.

The class `TestResult` represents the results of a `TestSuite` execution. This means the number of test passed, failed and the number of errors.

The class `TestResource` represents a resource that is used by a test or a set of tests. The point is that a resource is associated with subclass of `TestCase` and it is run automatically once before all the tests are executed contrary to the `TestCase` methods `setUp` and `tearDown` that are executed before and after any test.

A resource is run before a test suite is run. A resource is defined by specializing the class method resources as shown by the following example. By default, an instance of TestSuite consider that all its resources are the list of resources of the TestCase that compose it.

We define a subclass of `TestResource` called `MyTestResource` and we associate it with `MyTestCase` by specializing the class method `resources` to return an array of the test classes to which it is associated.

```
TestResource subclass: #MyTestResource
      instanceVariableNames: ''

TestResource>>setUp
  “here the resource is set up”

MyTestCase class>>resources
  “associate a resource with a testcase“

  ^ Array with: MyTestResource
```

As with a TestCase, we use the method `setUp` to define the actions that will be run during the set up of the resource.

5. The Cool Features of SUnit 3.1

In addition to TestResource SUnit 3.1 adds assertion description strings, logging support and resumable test failures.

5.1 Assertion description strings

The TestCase assertion protocol has been extended with a number of methods allowing the assertion to have a description.These methods take a String as second argument. If the test case fails, this string will be passed along to the exception handler, allowing more variety in messages than "Assertion failed" gives you. Of course, this string can be constructed dynamically.

```
| e |
e := 42.
self assert: e = 23 description: 'expected 23, got ' e printString
```

The added methods in TestCase are: #assert:description:, #deny:description:, #should:description:, and #shouldnt:description:.

5.2 Logging support

The description strings described above may also be logged to a Stream such as the Transcript, a file, stdout etc. You can choose whether to log by overriding TestCase>>#isLogging in your test case class, and choose where to log to by overriding TestCase>>#failureLog. Note that logging facilities will be really expanded in the release 3.2 of SUnit.

5.3 ResumableTestFailure

A resumable TestFailure has been added. This is a really powerful features that uses the powerful exception mechanisms offered by Smalltalk. What can this be used for? Take a look at this example:

```
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. In most cases, however, 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 to your logging device 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.

6. Key Implementation Aspects

We show now some key aspects of the implementation by following the execution of a test. This is not mandatory to use SUnit but can help you to customize it.

Running one Test. To execute one test, we evaluate the expression `(TestCase selector: aSymbol) run`. The method `TestCase>>run` defined on the class `TestCase` creates an instance of TestResult that will contains the result of the executed tests, then it invokes the method `TestCase>>run:`

```
TestCase>>run
   | result |
   result := TestResult new.
   self run: result.
   ^result
```

Note that in the future release, the class of the TestResult to be created will be returned by a method so that new TestResult can be introduced. The method `TestCase>>run:` invokes the method `TestResult>>runCase:`.

```
TestCase>>run: aResult
        aResult runCase: self
```

The method `TestResult>>runCase:` is the method that will invoke the method `TestCase>>runCase` that executes a test.

Without going into the details, `TestCase>>runCase` pays attention to the possible exception that may be raised during the execution of the test, invokes the execution of a testCase by calling the method `runCase` and counts the errors, failures and passed tests.

```
 TestResult>>rrunCase: aTestCase

   | testCasePassed |
   testCasePassed :=
   [
     [
       aTestCase runCase.
       true]
         sunitOn: self class failure
         do: [:signal |
           self failures add: aTestCase.
           signal sunitExitWith: false]]
             sunitOn: self class error
             do: [:signal |
               self errors add: aTestCase.
               signal sunitExitWith: false].

 testCasePassed
   ifTrue: [self passed add: aTestCase]The method TestCase>>runCase realizes
the calls to the methods setUp et tearDown as shown below:
 TestCase>>runCase
   self setUp.
   [self performTest] sunitEnsure: [self tearDown]
```

Running a TestSuite. To execute more than a test, we invoke the method `TestSuite>>run` on a `TestSuite`. The class `TestCase` provides some functionalities to get a test suite from its methods. The expression My`TestCase buildSuiteFromSelectors` returns a suite suite containing all the tests defined in the class `MyTestCase`.

The method `TestSuite>>run` creates an instance of `TestResult`, verifies that all the resource are available, then the method `TestSuite>>run:` is invoked which run all the tests that compose the test suite. All the resources are then reset.

```
run
  | result |
  result := TestResult new.
  self resources do: [ :res |
    res isAvailable ifFalse: [^res signalInitializationError]].
  [self run: result] sunitEnsure: [self resources do: [:each | each reset]].
  ^result

TestSuite>>run: aResult
  self tests do: [:each |
        self sunitChanged: each.
        each run: aResult]
```

The class `TestResource` and its subclasses keep track of the their currently created instances (one per class) that can be accessed and created using the class method `current`. This instance is cleared when the tests have finished to run and the resources are reset.

This is during the resource availability check that the resource is created if needed as shows the class method `TestResource class>>isAvailable`. During the TestResource instance creation, it is initialized and the method setUp is invoked. (Note it may happen that your version of SUnit 3.0 does not correctly initialize the resource. A version with this bug circulated a lot. Verify that `TestResource class>>new` calls the method `initialize`).

```
TestResource class>>isAvailable
  ^self current notNil and: [self current isAvailable]
TestResource class>>current
  current isNil ifTrue: [current := self new].
  ^current
TestResource>>initialize
  self setUp
```

7. Two Bits of Wisdom

Testing is difficult, here is a list of advices to build tests.

Test self described. Each time you change your code you do not want to change your tests, therefore try to write them in a way that they are self-contained. This is difficult but pay in the long term. Writing tests in terms of stable interfaces support self-contained tests.

Do not over test. Try to build your tests so that they do not overlap. It is annoying to have several tenth of tests covering all the same aspects and break all at the same time.

Unit vs. Acceptance Tests. Unit tests describe one functionality and as such make easier the identification of bugs. However for certain deeply recursive or complex setup situation it is easier to write tests that represent a scenario. So try as much as possible to have Unit tests and group them per class. For acceptance tests group then in terms of the functionality tested.

8. Extending SUnit

Here we briefly show how SUnit can be extended to provide setup that are shared by all the tests of a TestCase class. Note that the extension is not as robust as the core SUnit.

We start first with a simple example of the expected behavior. We define a new class which inherits from SharingSetUpTestCase as follow. We define two simple tests testOne and testTwo.

```
SharedOne
  superclass: SharingSetUpTestCase

SharedOne>>testOne
  Transcript show: 'Test one runs'; cr.

SharedOne>>testTwo
  Transcript show: 'Test Two runs'; cr.
```

Then we define the method setUp and tearDown that will be executed before and after the execution of the tests exactly in the same way as with non sharing tests. Note however the fact that with the solution we will present we have to explicitly invoke the setUp method and tearDown of the superclass.

```
SharedOne>>setUp
  "if you need to still have some setUp for a single tests, you have to invoke super
setUp"

  super setUp.
  Transcript show: 'SharedOne>>setUp'; cr

SharedOne>>tearDown

  Transcript show: 'SharedOne>>tearDown'; cr.
  super tearDown
```

Finally we define the methods sharedSetUp and sharedTearDown that will be only executed once for the two tests. Note that this solution implies that the tests are not destructively the shared fixture but just query it.

```
SharedOne class>>sharedSetUp

  Transcript show: 'SharedSetUp runs'  ; cr
  "my set up here"

SharedOne class>>sharedTearDown

  Transcript show: 'Shared TearDown runs' ;cr
  "my set up here"
```

Here is the trace you obtain

```
  SharedOne suite run
  SharedSetUp runs
  SharedOne>>setUp
  Test one runs
  SharedOne>> tearDown
  SharedOne>>setUp
  Test Two runs
  SharedOne>> tearDown
  Shared TearDown runs
2 run, 2 passed, 0 failed, 0 errors
```

The extension of the SUnit framework is based on the introduction of two classes: SharedSetupTestCase and SharedSetUpTestSuite. The basic idea is to use a flag that is flushed after a certain number of tests has been run. The class SharedSetupTestCase defines then one instance variable that indicate whether the test is run individually or in the context of a test suite and two class instance variables to indicate the number of time that the setup should hold and flag

```
SharedSetupTestCase
```

```
  superclass: testCase
  instanceVariables: ‘runIndividually’
  classinstanceVariables: ‘numberOfTestsToTearDown sharedSetUp '’

SharedSetupTestCase class>>suiteClass
  ^SharingSetUpTestSuite

SharedSetupTestCase class>>sharedSetUp
  "subclass should only override this hook to define a sharedSetUp"
SharedSetupTestCase class>>sharedTearDown
  "here we specify the teardown of the shared setup "

SharedSetupTestCase class>>flushSharedSetUp
  sharedSetUp := nil
```

The SharedSetupTestCase class is armed with the number of times it should hold.

```
SharedSetupTestCase class>>armTestsToTearDown: aNumber

  self flushSharedSetUp.
  numberOfTestsToTearDown := aNumber.
```

Everytimes a test is run, the method anothertestHasBeenRun is invoked. Once the number of tests is reached the sharedSetUp is flushed and the sharedTearDwon is executed.

```
SharedSetupTestCase class>>anotherTestHasBeenRun
  "Everytimes a test is run this method is called, once all the tests of the suite
are run the shared setup is reset"

  numberOfTestsToTearDown := numberOfTestsToTearDown - 1.
  numberOfTestsToTearDown isZero
    ifTrue:
      [self flushSharedSetUp.
      self sharedTearDown]
```

When a test is run its setUp is executed which somehow calls the privateSharedSetUp method which is only executed when the sharedSetUp test is not set. This execution invokes the sharedSetUp method.

```
SharedSetupTestCase class>>privateSharedSetUp

  sharedSetUp isNil
    ifTrue:
      [sharedSetUp := 1.
      self sharedSetUp]

SharedSetupTestCase>>setUp
  self class privateSharedSetUp

SharedSetupTestCase>>tearDown
  self class anotherTestHasBeenRun
```

When a testCase is created we assume that it will be run once. This may be invalidated laterinvoking the method executedFromASuite.

```
SharedSetupTestCase>>setTestSelector: aSymbol
  "I'm forced to do that because there is no initialize"

  runIndividually := true.
  super setTestSelector: aSymbol

SharedSetupTestCase>>executedFromASuite
  runIndividually := false
```

The two methods responsible for Test execution are then specialized as follow:

```
SharedSetupTestCase>>runCaseAsFailure
  self armTearDownCounter.
  super runCaseAsFailure.

SharedSetupTestCase>>runCase
  self armTearDownCounter.
  super runCase

SharedSetupTestCase>>armTearDownCounter
  self isIndividuallyExecuted
    ifTrue: [self class armTestsToTearDown: 1]
```

Now the SharedSetUpTestSuite define the instance variable testCaseClass and redefines the two methods to run test suite run: and run as follow. They check whether they contain tests and if this is the case they armed so that the shread setup is only executed a correct number of time.

```
SharedSetUpTestSuite>>run: aResult
  self checkAndArmSharedSetUp.
  ^super run: aResult

SharedSetUpTestSuite>>run
  self checkAndArmSharedSetUp.
  ^ super run

SharedSetUpTestSuite>>checkAndArmSharedSetUp
  self tests isEmpty
    ifFalse: [self tests first class armTestsToTearDown: self tests size]
```

Finally the method addTest: is specialize so that it marks all its tests with the fact that they are executed in a TestSuite and check whether all its tests are from the same class to avoid inconsistency.

```
SharedSetUpTestSuite>>addTest: aTest
  "Sharing a setup only works if the test case composing the test suite are from
the same class so we test it"

  aTest executedFromASuite.
  testCaseClass isNil
    ifTrue: [testCaseClass := aTest class.
          super addTest: aTest ]
    ifFalse: [aTest class == testCaseClass
         ifFalse: [self error: 'you cannot have test case of different classes in
a SharingSetUpTestSuite'.]
         ifTrue: [super addTest: aTest]]
```

SUnit is a powerful framework as we illustrated with the previous example. You can adapt it to your need.

9. Conclusion

We presented why writing tests are an important way of investing on the future. We recalled that tests should be repeatable, independent of any direct human interaction and cover a precise functionality to maximize their potential. We presented in a step by step fashion how to define a couple of tests for the class Set using SUnit. Then we gave an overview of the core of the framework by presenting the classes TestCase, TestResult, TestSuite and TestResources. Finally we dived into SUnit by following the execution of a tests and test suite. We hope that we convince you about the importance of repeatable unit tests and about the ease of writing them using SUnit.

10. Bibliography

[Beck] Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[FBBOR] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[RBJ1] D. Roberts, J. Brant and R. Johnson, “Why every Smalltalker should use the Refactoring Browser, Smalltalk Report, SIGS Press, http://st-www.cs.uiuc.edu/users/droberts/homePage.html#refactoring

[RBJ2]D. Roberts, J. Brant and R. Johnson, “A Refactoring Tool for Smalltalk”, TAPOS, vol. 3, no. 4, 1997, pp. 253-263, http://st-www.cs.uiuc.edu/~droberts/tapos/TAPOS.htm

[SUnit] http://www.xprogramming.com/software.htm