University of Berne
Institute of Computer Science
http://www.iam.unibe.ch/~ducasse/

SUnit Explained

Stéphane Ducasse
(revised by Rick Zaccone)

Directory

- **Table of Contents**
- **Begin Article**

Last Revision Date: March 6, 2006

1. Introduction

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 extended by Joseph Pelrine and others over several iterations to take into account the notion of resources that we will illustrate later. The interest in 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 including Java, Python, Perl, Oracle and many others [4]. The current version of SUnit is 3.1. The official web site of SUnit is `<http://sunit.sourceforge.net/>`.

Testing and building test suites is not new and everybody knows that tests are a good way to catch errors. eXtreme Programming, by putting testing in the core of its methodology, is shedding a new light on testing, an often disliked discipline. The Smalltalk community has a long tradition of testing 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 inform the reader of expected results, since much of the logic is left unspecified. SUnit is interesting because it allows you to structure tests, 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 why we test, then we present an example with SUnit and we go deep into the SUnit implementation.

2. Testing and Tests

Most developers believe that tests are a waste of time. Who has not heard: “I would write tests if I would have more time.”? If you write code that never changes, you should not write tests, but this also means that your 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 it saves 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 in a piece of functionality. They help you to quickly find the parts that break due to introduced changes. Finally, writing tests at the same time or even before writing code forces 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 ex-

pected 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 after writing a method, we would 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 the tests are not saved and run automatically. Moreover it often happens that the context of the tests is left unspecified so the reader has to interpret the results and assess if they are right or wrong.

It is clear that we cannot test 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 they can be left hidden waiting for an opportunity to damage your system. This is not a problem if you write a test that covers the bug as soon as you uncover it.

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

- Tests should be repeatable. You should be able to repeat a test as often as you want.
- Tests should run without human intervention. You should even be able to run them during the night.
- Tests should tell 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.
- Tests should have a change frequency lower than the one of the covered functionality. Indeed you do not want to change all your tests every time you modify your application. One way to achieve this property is to write tests based on the interfaces of the tested functionality.

The number of tests should be somewhat proportional to the number of tested functionalities. 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 writing tests before writing code. This may seem against our deep developer habits. Here are the observations we made while practicing up front tests writing. Up front testing helps you to know what you want

to code, it helps you know when you are done, and it helps 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 you should be using SUnit.

3. SUnit by Example

Before going into the details of SUnit, we will show a step by step example. We use an example that tests the class `Set`. Try entering the code as we go along.

3.1. Step 1

First you should create a `TestCase` subclass called `ExampleSetTest`. Right click on `TestCase` and select "Create Subclass...". Add two instance variable so your new class looks as follows.

```
XProgramming.SUnit defineClass: #ExampleSetTest
    superclass: #{XProgramming.SUnit.TestCase}
    indexedType: #none
    private: false
    instanceVariableNames: 'full empty'
    classInstanceVariableNames: ''
    imports: ''
    category: 'SUnit'
```

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

3.2. Step 2

The method `setUp` acts as a context definer method or initialize method. It is invoked before the execution of each 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. We define the method `setUp` as follows.

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

This method defines the context for each of the tests defined in our subclass of `TestCase` (`ExampleSetTest` in this example). In testing jargon it is called the *fixture* of the test.

3.3. Step 3

Lets create some tests by defining some methods in 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 `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 number of 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 no longer contains it.

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

3.4. 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`. After clicking on the **Run** button, you see a window similar to 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 include an expression in the comment of our tests that allows us to run them while browsing them as shown below.

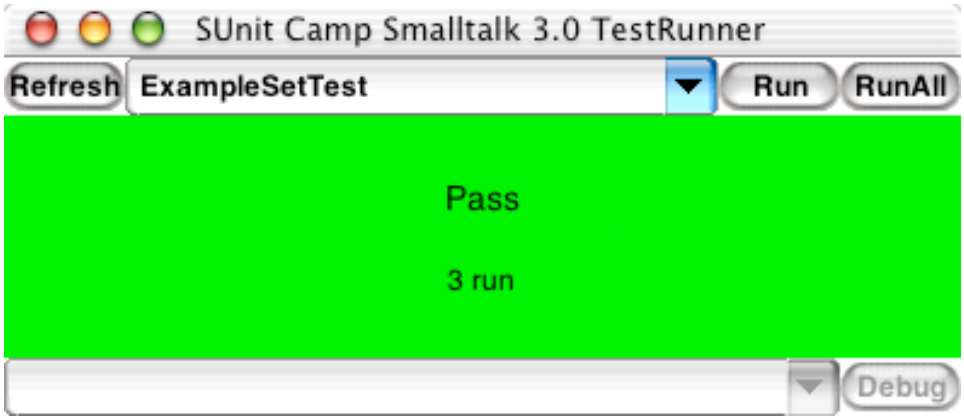

Figure 1: The user interface of SUnit in VisualWorks

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

To debug a test you may use the following expressions:

```
(ExampleSetTest selector: #testRemove) debug
```

or

```
ExampleSetTest debug: #testRemove
```

3.5. Explanation

The method `assert:` which is defined in the class `TestCase` requires a boolean 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 has two kinds of errors. A *failure* means that a test has failed. An *error* is something that has not been tested such as 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:` for testing exception handling. For example, you would use `(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. Enter and run this test.

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

SUnit is capable of running on all Smalltalk dialects. To accomplish this, the SUnit developers have factored out the dialect dependent aspects. The class method `TestResult>>error` provides the name of the error handler in a dialect independent fashion. Take a look at the code to see how this is done. If you had wanted to write tests that would work in several dialects, you could have written `testIllegal` as follows.

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

Give it a try.

4. Basic How To

This section will give you more details on how to use SUnit. If you are familiar with another testing framework such as Java's JUnit, much of this will be familiar since it has its roots in SUnit. Normally you will use SUnit's GUI to run tests, but there are situations where you may not want to use it.

4.1. Normal Operation

Normally, you will run your tests by using the test runner.

```
TestRunner open
```

4.2. How to Run a Single Test

Suppose you want to run a single test. You can run it as follows.

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

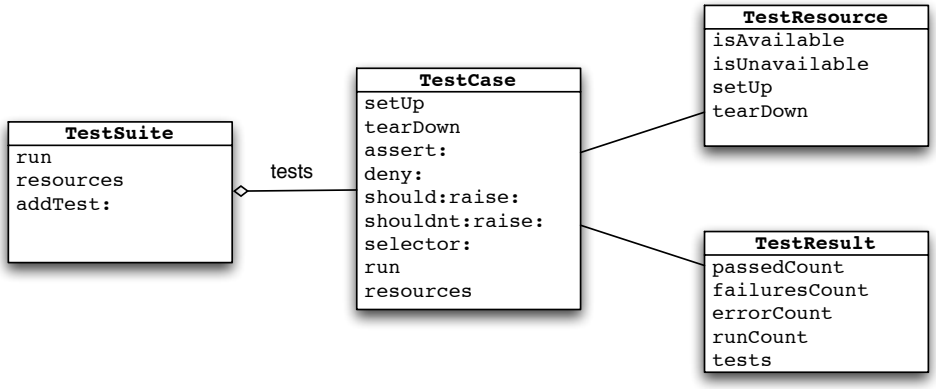

Figure 2: The four classes representing the core of SUnit

4.3. How to Run All 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. Here is how you turn all the `test*` methods into a `TestSuite`.

```
ExampleSetTest suite run
5 run, 5 passed, 0 failed, 0 errors
```

4.4. 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 a suite 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.

5. The SUnit Framework

SUnit 3.1 introduces the notion of resources that are necessary when building 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.

SUnit consists of four main classes. They are `TestCase`, `TestSuite`, `TestResult`, and `TestResource` as shown in Figure 2.

5.1. TestCase

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 TestCase and by the specialization of the method setUp which initializes the context in which the will be executed. The class TestCase also defines the method tearDown that is responsible for releasing any objects allocated during the execution of the method setUp. The method tearDown is invoked after the execution of each test.

5.2. TestSuite

The class TestSuite contains a collection of test cases. An instance of TestSuite is composed of instances 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.

5.3. TestResult

The class TestResult represents the results of a TestSuite execution. It contains the number of test passed, the number of tests failed, and the number of errors.

5.4. TestResource

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 each test.

A resource is run before a test suite is run. A resource is defined by overriding the class method resources as shown by the following example. By default, an instance of TestSuite assumes that its resources are the list of resources constructed from the TestCases that it contains.

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: ''

MyTestResource>>setUp
    "Set up resources here."
```

```
MyTestResource>>tearDown
    "Tear down resources here."

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.

6. Features of SUnit 3.1

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

6.1. 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". 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:
#shouldnt:description:
```

6.2. 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 these logging facilities will be available in release 3.2 of SUnit.

6.3. Resumable Test Failure

A resumable `TestFailure` has been added. This is a really powerful feature 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 accumulate failures, i.e., if the assertion fails 10 times in your test method, you'll still only see one failure.

7. Key Implementation Aspects

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

7.1. Running One Test

To execute one test, we evaluate the expression

```
(TestCase selector: aSymbol) run.
```

The method `TestCase>>run` creates an instance of `TestResult` that will contain the results of the executed tests, then it invokes the method `TestCase>>run:`.

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

Note that in a 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 any exception that may be raised during the execution of a test, invokes the execution of a TestCase by calling the method runCase and counts the errors, failures and passed tests.

```
TestResult>>runCase: 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 uses the calls to the methods setUp and tearDown as shown below.

```
TestCase>>runCase
    self setUp.
    [self performTest] sunitEnsure: [self tearDown]
```

7.2. Running a TestSuite

To execute more than one test, we invoke the method TestSuite>>run on a TestSuite. The class TestCase provides some functionality to get a test suite from its methods. The expression

```
MyTestCase buildSuiteFromSelectors
```

returns a 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 runs all the tests that compose the test suite. All the resources are then released.

```
TestSuite>>run
    | result |
    result := TestResult new.
    self areAllResourcesAvailable
        ifFalse: [^TestResult signalErrorWith:
                    'Resource could not be initialized'].
    [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 running and the resources are reset.

It is during the resource availability check that the resource is created if needed as shown in 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

TestResource class>>current
    current isNil ifTrue: [current := self new].
    ^current

TestResource>>initialize
    self setUp
```

8. Bits of Wisdom

Testing is difficult. Here is some advice on building tests.

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

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

Unit vs. Acceptance Tests Unit tests describe one functionality and as such make it easier to identify bugs. However, for certain deeply recursive or complex setup situations, 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.

9. Extending SUnit

In this section we will explain how to extend SUnit so that it uses a `setUp` and `tearDown` that are shared by all of the tests in a `TestCase` subclass. We will define a new sublass of `TestCase` called `SharingSetUpTestCase`, and a subclass of `SharingSetUpTestCase` called `SharedOne`. We will also need to define a new subclass of `TestSuite` called `SharedSetUpTestSuite`, and we will make some minor adjustments to `TestCase`.

Our tests will be in `SharedOne`. When we execute

```
Transcript clear.
SharedOne suite run
```

we will obtain the following trace.

```
SharedOne>>setUp
SharedOne class>>sharedSetUp
SharedOne>>testOne
SharedOne>>tearDown
SharedOne>>setUp
SharedOne>>testTwo
SharedOne>>tearDown
SharedOne class>>sharedTearDown
2 run, 2 passed, 0 failed, 0 errors
```

You can see that the shared code is executed just once for both tests.

9.1. SharedSetUpTestCase

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 (cleared) after a certain number of tests have been run. The class SharedSetUpTestCase defines one instance variable that indicates whether each test is run individually or in the context of a shared setUp and tearDown. There are also two class instance variables. One indicates the number of tests for which the shared setUp should be in effect, and the other indicates whether the shared setUp is in effect.

```
SharedSetUpTestCase
    superclass: TestCase
    instanceVariableNames: 'runIndividually '
    classInstanceVariableNames: 'numberOfTestsToTearDown
                                 sharedSetUp '
```

suiteClass is used by TestCase to determine the suite that is running.

```
SharedSetUpTestCase class>>suiteClass
    ^SharedSetUpTestSuite

SharedSetUpTestCase class>>sharedSetUp
    "A 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 initialized with the number of tests for which the shared setUp should be in effect.

```
SharedSetUpTestCase class>>armTestsToTearDown: aNumber
    self flushSharedSetUp.
    numberOfTestsToTearDown := aNumber.
```

Every time a test is run, the method anothertestHasBeenRun is invoked. Once the specified number of tests is reached the sharedSetUp is flushed and the sharedTearDown 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 and it then it calls the class method `privateSharedSetUp`. This method will only invoke the `sharedSetUp` if the `sharedSetUp` test indicates that it hasn't been done yet.

```
SharedSetUpTestCase class>>privateSharedSetUp
    sharedSetUp isNil
        ifTrue:
            [sharedSetUp := 1.
            self sharedSetUp]

SharedSetUpTestCase>>setUp
    self class privateSharedSetUp

SharedSetUpTestCase>>tearDown
    self class anotherTestHasBeenRun
```

When a test case is created we assume that it will be run once. We can change this later by invoking the method `executedFromASuite`.

```
SharedSetUpTestCase>>setTestSelector: aSymbol
    "Must do it this way because there is no initialize"

    runIndividually := true.
    super setTestSelector: aSymbol

SharedSetUpTestCase>>executedFromASuite
    runIndividually := false
```

The methods responsible for test execution are then specialized as follows.

```
runIndividually
    ^runIndividually
```

```
SharedSetUpTestCase>>armTearDownCounter
    self runIndividually
        ifTrue: [self class armTestsToTearDown: 1]

SharedSetUpTestCase>>runCaseAsFailure
    self armTearDownCounter.
    super runCaseAsFailure

SharedSetUpTestCase>>runCase
    self armTearDownCounter.
    super runCase
```

9.2. SharedOne

SharedOne is a new class which inherits from SharingSetUpTestCase as follows. We define two simple tests testOne and testTwo.

```
SharedOne
    superclass: SharingSetUpTestCase

SharedOne>>testOne
    Transcript
        show: 'SharedOne>>testOne';
        cr

SharedOne>>testTwo
    Transcript
        show: 'SharedOne>>testTwo';
        cr
```

Then we define the methods 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
    Transcript
        show: 'SharedOne>>setUp';
        cr.
    super setUp
```

```
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 assumes that the tests are not destructive to the shared fixture, but just query it.

```
SharedOne class>>sharedSetUp
    Transcript
        show: 'SharedOne class>>sharedSetUp';
        cr
    "My set up here."

SharedOne class>>sharedTearDown
    Transcript
        show: 'SharedOne class>>sharedTearDown';
        cr
    "My tear down here."
```

9.3. `SharedSetUpTestSuite`

The `SharedSetUpTestSuite` defines just one instance variable `testCaseClass` and redefines the two methods necessary to run the test suite `run:` and `run`. `checkAndArmSharedSetUp` initializes the number of tests to run before the shared `tearDown` is executed.

```
SharedSetUpTestSuite
    superclass: TestSuite
    instanceVariableNames: 'testCaseClass'

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

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

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

Finally the method `addTest:` is specialized so that it marks all its tests with the fact that they are executed in a `TestSuite` and checks 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]]
```

9.4. Changes to `TestCase`

In order for the above changes to work, you must make `TestCase` aware of your new test suite.

```
TestCase class>>buildSuite
    | suite |
    ^self isAbstract
        ifTrue:
            [suite := self suiteClass new.
            suite name: self name asString.
            self allSubclasses
                do: [:each |
                    each isAbstract
                        ifFalse: [suite addTest:
                          each buildSuiteFromSelectors]].
            suite]
        ifFalse: [self buildSuiteFromSelectors]
```

```
TestCase class>>buildSuiteFromMethods: testMethods
    ^testMethods
        inject: ((self suiteClass new)
                name: self name asString;
                yourself)
        into:
            [:suite :selector |
            suite
                addTest: (self selector: selector);
                yourself]
```

If you have made all the changes correctly, you should be able to run your tests and see the results shown in section 9.

10. Exercise

The previous section was designed to give you some insight into the workings of SUnit. You can obtain the same effect by using SUnit's resources.

Create new classes `MyTestResource` and `MyTestCase` which are subclasses of `TestResource` and `TestCase` respectively. Add the appropriate methods so that the following messages are written to the `Transcript` when you run your tests.

```
MyTestResource>>setUp has run.
MyTestCase>>setUp has run.
MyTestCase>>testOne has run.
MyTestCase>>tearDown has run.
MyTestCase>>setUp has run.
MyTestCase>>testTwo has run.
MyTestCase>>tearDown has run.
MyTestResource>>tearDown has run.
```

11. 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.

References

[1] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 253–263, 1997. [Online]. Available: http://st-www.cs.uiuc.edu/~droberts/tapos.pdf

[4] (2003) The XProgramming.com website. [Online]. Available: http://www.xprogramming.com/software.htm 2