How to use Test Resources

Joseph Pelrine
MetaProg GmbH

Version 1.0
July 12, 2005

What are Test Resources and why would you want to use them?

One of the practices of extreme Programming that contributes greatly to its success is the idea of test-driven programming using extremely (sic) short feedback cycles. Press a button, run your test, get your answer. The longer this feedback cycle takes, the more difficult it becomes to program in an extreme style, and the more thinking has to be done up-front to avoid wasting precious time.

There are usually a few test objects which can be expensive to instantiate, though:

- Database connections
- Extremely complex objects

and we don't want to have to do this for each Test Case. Keeping a test object alive over multiple Test Cases breaks one of the primary rules of unit testing, by not starting with a clean slate before each Test Case is run.

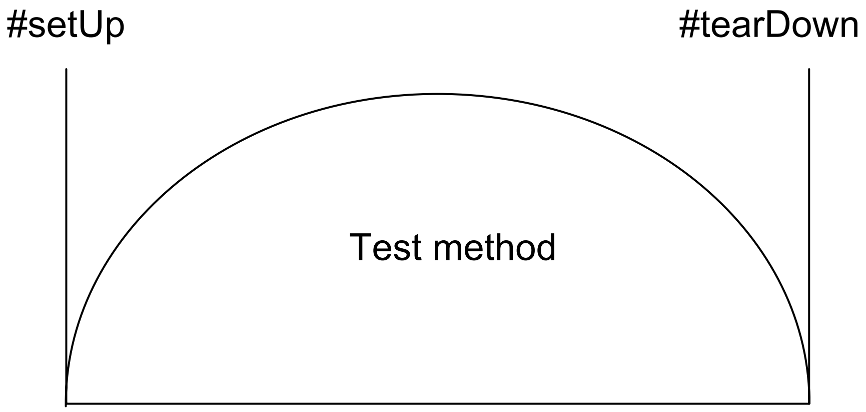
> *One final bit of philosophy. It is tempting to set up a bunch of test data, then run a bunch of tests, then clean up. In my experience, this always causes more problems than it is worth. Tests end up interacting with one another, and a failure in one test can prevent subsequent tests from running. 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.* [Beck95]

Nevertheless, this is desirable from the point of rapid feedback, and can be very effective is used responsibly. This is why we've developed Test Resources.

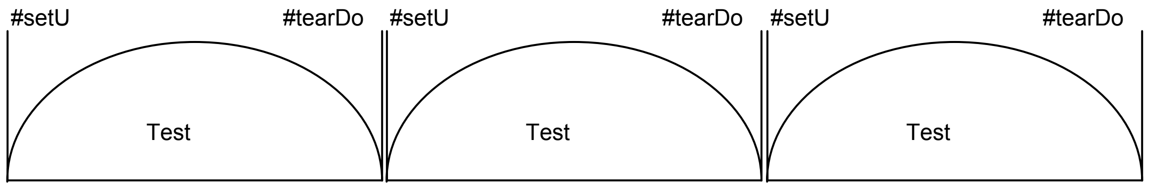
> A Test Resource is an object which is needed by a number of Test Cases, and whose instantiation is so costly in terms of time or resources that it becomes advantageous to only initialize it once for a Test Suite run.

Initialization and release of Test Cases

How does the initialization and release mechanism - #setUp and #tearDown - work in SUnit? It's pretty simple.
To run a test, SUnit calls the appropriate method of the TestCase. This call is not only wrapped in the SUnit exception-handling framework (to catch test failures and errors), but also surrounded by methods to set up and tear down any objects needed for the test. These methods are called – not surprisingly - #setUp and #tearDown.

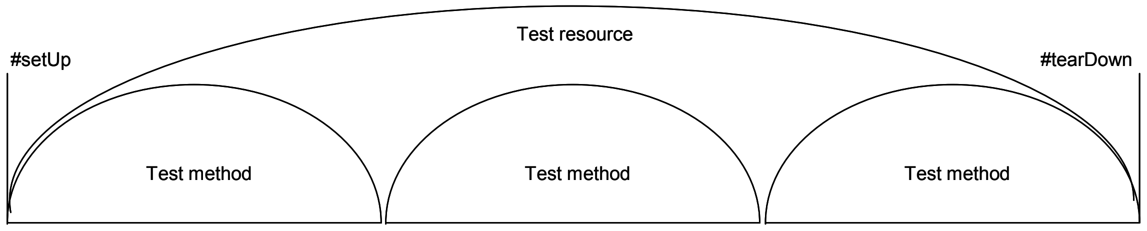


When a test suite is run, which is what happens when you use the SUnit TestRunner, many tests are called in succession.



As you can see, this may create problems. Calls to #setUp and #tearDown occur rapidly, possibly asynchronously, and the resulting opening and closing of things like database connections, ports, file and other i/o accessors, may get royally messed up.

A TestResource gets sent #setUp once, before any tests run, and gets sent #tearDown once, after all tests have run. If any errors occur during the TestResource #setUp, the test run is aborted with an error. This avoids the case of having e.g. 200 database-related test cases all fails because the database connection TestResource did not initialize properly.



Writing your first Test Resource

Let's start by writing a simple Test Resource. This one will tell us when it's been started and stopped, by printing something to the Transcript.

First, let's declare our Test Resource class.

```
TestResource subclass: #MyFirstTestResource
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Next, we'll implement #setUp and #tearDown methods.

MyFirstTestResource methods

setUp
```
Transcript cr; show: 'set up…'
```

tearDown
```
Transcript cr; show: tear down'
```

So, open a Test Runner, run your tests and see what happens…nothing. Why not? Well, a test resource is only instantiated when there is a need for it. Since it wasn't needed by and tests, nothing happened. Let's see what we need to do to use a resource.

To have a Test Resource managed by SUnit, we must tell SUnit that we need the resource (resources). We do that by declaring a method named (what else) #resources. Let's declare a Test Case class.

```
TestCase subclass: #MyFirstTestCase
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

On the class side, we declare the resources we need. We do this by returning a Collection with the names of all the resource classes we need.

MyFirstTestCase class methods

resources
```
^Array with: MyFirstTestResource
```

If we now run MyFirstTestCase, we still see that nothing happens. This is because SUnit only launches the resources for tests that will be run. So, let's implement a dummy test method.

MyFirstTestCase methods

testDummy
```
	self assert: true
```

If we now run the tests, we should see the following on the Transcript:

```
	set up…
	tear down
```

Conratulations! That's all you need to do to use Test Resources.

Capturing Test Results using Test Resources

For the next step, let's implement a Test Resource which does something useful. This one catches the results of a test run and displays them on the Transcript. (Note: SUnit 3.2 will contain a logging framework, but this implementation is an easy way to capture and post-process test results).

First we need to define the resource class.

```
	TestResource subclass: #ResultCaptureTestResource
	    instanceVariableNames: 'result'
	    classVariableNames: ''
	    poolDictionaries: ''
```

Notice that we've defined the instance variable <result>, which will hold the TestResult. We need to define accessors for it, so that we can set and get the result.

ResultCaptureTestResource methods

result
```
	^result
```

result: aTestResult
```
	result := a TestResult
```

Our TestResource doesn't need to do anything before the test run, so there's no need for a #setUp method. On the other hand, we do want to see the results when we're finished testing, so we'll implement #tearDown.

tearDown
```
	result printResultOn: Transcript
```

Now we need to define how we want the TestResult to print out its' data onto the Transcript. Since it is useful to be able to dump this data to any stream, we'll pass the stream – in this case, the Transcript – to the TestResult.

TestResult methods

printResultOn: aStream
```
      ^aStream
            nextPutAll: self runCount printString;
            nextPutAll: ' run, ';
            nextPutAll: self correctCount printString;
            nextPutAll: ' passed, ';
            nextPutAll: self failureCount printString;
            nextPutAll: ' failed, ';
            nextPutAll: self errorCount printString;
            nextPutAll: ' error'.

      self errorCount ~= 1 ifTrue: [aStream nextPut: $s].
      aStream flush
```

So, one big question still remains: how do we get hold of the TestResult to tell it to print itself? The easiest way to do so is to take advantage of an implementation detail in SUnit. In order for the test method to be run inside the SUnit exception handling framework, the current testResult instance is passed to the TestCase class, which then double-dispatches back to the TestResult. What we'll do is override this method, capture the result, and pass it on to the Resource for safekeeping.

First, we need a TestCase class.

```
      TestCase subclass: #ResultCaptureTestCase
          instanceVariableNames: ''
          classVariableNames: ''
          poolDictionaries: ''
```

Then we tell SUnit that we need the TestResource

ResultCaptureTestCase class methods

resources
```
      ^Array with: ResultCaptureTestResource
```

Next we override #run: and also implement a dummy test method.

ResultCaptureTestCase methods

run: aTestResult
```
      ResultCaptureTestResource current result: aTestResult.
      super run: aTestResult
```

testDummy
```
      self assert: true
```

So, we're finished. Press the "run all" button on the TestRunner and see what happens. If all is correct, you should see something similat to the following in your Transcript:

```
      42 run, 42 passed, 0 failures, 0 errors
```

A more useful example - storing state with Test Resources

One of the most important uses I have for TestResources is to run tests on big objects, or objects with very complex state. Since these objects take a lot of time to instantiate, I like to manage them as resources. On the other hand, they get changed by the tests, and I need a well-known, reproducible state in order to have valid, reproducible tests. The StoredStateTestResource class allows me to store the instantiated state of my test object in binary form outside of the image, and to reload the pristine state whenever I need it.

In this example, our data object will simply be a dictionary with 2 entries. We need a place to hold on to it, and will create an instance variable called <data> in our TestResource class to do just that.

```
      TestResource subclass: #StoredStateTestResource
          instanceVariableNames: 'data'
          classVariableNames: ''
          poolDictionaries: ''
```

StoredStateTestResource methods

data
```
      ^data
```

defaultData
```
      ^IdentityDictionary new
            at: #meaningOfLife put: 42;
            at: #illuminati put: 23;
            yourself
```

On starting the test run, we want to capture and save the initial state of our data object. Once we have it stored, we also need a method to allow us to reload it whenever we need it.

StoredStateTestResource methods

setUp
```
      self dumpData
```

reloadData
```
      data := self loadData
```

In this example, I use the native binary streaming and storage mechanism to dump the data object to a file. You can easily change and extend this example to store to a database, to ENVY user fields, etc. Since the following methods are dialect specific, you'll have to choose the code for the Smalltalk dialect you use.

StoredStateTestResource methods

dumpData

```
      "Dialect specific. Un-comment for your dialect"

      "Visual Age"

      ObjectDumper new unload: self defaultData intoFile: self filename

      "Visual Works – presumes you have BOSS loaded"
      | writeStream |
      writeStream := BinaryObjectStorage value
            onNew: self filename asFilename writeStream.
      [writeStream nextPut: self defaultData]
            ensure: [writeStream close]
```

loadData

```
      "Dialect specific. Un-comment for your dialect"

      "Visual Age"
      ^ObjectLoader new loadFromFile: self filename

      "Visual Works – presumes you have BOSS loaded"
      | readStream data |
      readStream := #{BinaryObjectStorage} value onOld:
            self filename asFilename readStream.
      [data := readStream next] ensure: [readStream close].
      ^data
```

fileName

```
      ^'storedStateData.bin'
```

Here's a small TestCase which exercises the stored-state mechanism.

```
      TestCase subclass: #StoredStateTestCase
          instanceVariableNames: ''
          classVariableNames: ''
          poolDictionaries: ''
```

Remember, we tell SUnit that we need the TestResource.

StoredStateTestCase class methods

resources

```
      ^Array with: StoredStateTestResource
```

In the #setUp method, we ensure that we have a fresh copy of the data loaded. I've also refactored the resource accessor out to a separate method.

StoredStateTestCase methods

setUp
```
      self resource reloadData
```

resource
```
      ^StoredStateTestResource current
```

Here are a few simple tests to prove it all works.

StoredStateTestCase methods

testExistingResource
```
      self assert: (self resource data at: #meaningOfLife) = 42
```

testChangeResource
```
      self resource data at: #meaningOfLife put: 39.
      self resource reloadData.
      self assert: (self resource data at: #meaningOfLife) = 42
```

Some implementation details

#needsMoreWork

Test Resources are currently implemented as an optional singleton:
- Follows standard singleton #current protocol
- But #new is not overridden to return an error

in order to give developers the option of using a sole instance of a resource, or multiple instances. Test Resources have a polymorphic syntax with TestCase, with #setUp and #tearDown messages.

Per default, all required resources are initialized before a TestSuite runs. This occurs non-deterministically by collecting the resources defined as required by the Test Cases into a Set, and sending them the message #isAvailable. The default implementation of #isAvailable checks to see if the variable holding the singleton is nil, lazy initialising it if it is not. It then tests whether the resource itself #isAvailable. This allows subclasses of TestResource to override #isAvailable to perform a different checking routine, or not to initialise the resource if the test environment allows manual initialisation of resources. TestCases optionally/preferably define required resources by overriding the class method #resources to return a Collection of resource class names.By not defining a resource in this method, its initialization becomes responsibility of the TestCase itself. In this case, the resource will probably not be accessed via the singleton, but rather held in a variable of the Test Case object.

When do you release a TestResource? This is a difficult question to answer, and the decision was made not to answer it. In the default implementation, TestRunner sends #reset to the resource class when it is finished. This invokes #tearDown on the resource and nils it out.

Required resources

#needsMoreWork

Some advanced examples

Here's a more advanced example of one of the things you can do with Test Resources. Many times when you get a test failure or error, you can understand it and fix it immediately. If not, it will often take you a while to diagnose and fix. It is inconvenient and time-consuming to tear down an expensive resource, just to build it again seconds later for the next test. In this case, it would be handy to have a test resource which waited a bit before tearing down – just in case. That's what a TimedTestResource does. This example comes from the open-source COAST framework. Analyze it yourself to understand how it works.

```
TestResource subclass: #COASTTimedTestResource
    instanceVariableNames: ''
    classInstanceVariableNames: 'tearDownProcess'
    classVariableNames: ''
    poolDictionaries: ''
```

COASTTimedTestResource class methods

current
```
self stopTearDownProcess.
^super current
```

stopTearDownProcess
```
tearDownProcess isNil ifFalse: [
      tearDownProcess terminate].
tearDownProcess := nil
```

reset
```
tearDownProcess := [
      (Delay forSeconds: self tearDownDelay) wait.
      super reset] fork
```

tearDownDelay
```
^0
```

hardReset
```
current notNil ifTrue: [
      current tearDown.
      current := nil]
```

Conclusion

#needsMoreWork

References

[Beck95] Kent Beck, 1995. *Simple Smalltalk Testing: with Patterns*. Available at http://www.xprogramming.com/testfram.htm