JAVAWORLD

ent, relocatable, multithreaded JUnit tests

By Andy Schneider

JavaWorld |
DEC 21, 2000 12:00 AM PT

JUnit is a typical toolkit: if used with care and with recognition of its idiosyncrasies, JUnit will help to develop good, robust tests. Used blindly, it may produce a pile of spaghetti instead of a test suite. This article presents some guidelines that can help you avoid the pasta nightmare. The guidelines sometimes contradict themselves and each other -- this is deliberate. In my experience, there are rarely hard and fast rules in development, and guidelines that claim to be are misleading.

We'll also closely examine two useful additions to the developer's toolkit:

- A mechanism for automatically creating test suites from classfiles in part of a filesystem
- A new `TestCase` that better supports tests in multiple threads

When faced with unit testing, many teams end up producing some kind of testing framework. JUnit, available as open source, eliminates this onerous task by providing a ready-made framework for unit testing. JUnit, best used as an integral part of a development testing regime, provides a mechanism that developers can use to consistently write and execute tests. So, what are the JUnit best practices?

Do not use the test-case constructor to set up a test case

Setting up a test case in the constructor is not a good idea. Consider:

```
public class SomeTest extends TestCase
    public SomeTest (String testName) {
}
```

Imagine that while performing the setup, the setup code throws an `IllegalStateException`. In response, JUnit would throw an `AssertionFailedError`, indicating that the test case could not be instantiated. Here is an example of the resulting stack trace:

```
junit.framework.AssertionFailedError: Cannot instantiate test case: test1       at
junit.framework.Assert.fail(Assert.java:143)     at
junit.framework.TestSuite.runTest(TestSuite.java:178)  at
junit.framework.TestCase.runBare(TestCase.java:129)      at
junit.framework.TestResult.protect(TestResult.java:100)        at
junit.framework.TestResult.runProtected(TestResult.java:117)     at
junit.framework.TestResult.run(TestResult.java:103)      at
junit.framework.TestCase.run(TestCase.java:120)  at
junit.framework.TestSuite.run(TestSuite.java, Compiled Code)     at
junit.ui.TestRunner2.run(TestRunner.java:429)
```

This stack trace proves rather uninformative; it only indicates that the test case could not be instantiated. It doesn't detail the original error's location or place of origin. This lack of information makes it hard to deduce the exception's underlying cause.

Instead of setting up the data in the constructor, perform test setup by overriding `setUp()`. Any exception thrown within `setUp()` is reported correctly. Compare this stack trace with the previous example:

```
java.lang.IllegalStateException: Oops     at bp.DTC.setUp(DTC.java:34)   at
junit.framework.TestCase.runBare(TestCase.java:127)      at
junit.framework.TestResult.protect(TestResult.java:100)        at
junit.framework.TestResult.runProtected(TestResult.java:117)     at
junit.framework.TestResult.run(TestResult.java:103)
...
```

This stack trace is much more informative; it shows which exception was thrown (`IllegalStateException`) and from where. That makes it far easier to explain the test setup's failure.

Don't assume the order in which tests within a test case run

JAVAWORLD

ests will be called in any particular order. Consider the following

```
public class SomeTestCase extends TestCase {
   public SomeTestCase (String testName) {
      super (testName);
   }
   public void testDoThisFirst () {
     ...
   }
   public void testDoThisSecond () {
   }
}
```

In this example, it is not certain that JUnit will run these tests in any specific order when using reflection. Running the tests on different platforms and Java VMs may therefore yield different results, unless your tests are designed to run in any order. Avoiding temporal coupling will make the test case more robust, since changes in the order will not affect other tests. If the tests are coupled, the errors that result from a minor update may prove difficult to find.

In situations where ordering tests makes sense -- when it is more efficient for tests to operate on some shared data that establish a fresh state as each test runs -- use a static `suite()` method like this one to ensure the ordering:

```
public static Test suite() {
   suite.addTest(new SomeTestCase ("testDoThisFirst";));
   suite.addTest(new SomeTestCase ("testDoThisSecond";));
   return suite;
}
```

There is no guarantee in the JUnit API documentation as to the order your tests will be called in, because JUnit employs a `Vector` to store tests. However, you can expect the above tests to be executed in the order they were added to the test suite.

Avoid writing test cases with side effects

Test cases that have side effects exhibit two problems:

- They can affect data that other test cases rely upon without manual intervention

JAVAWORLD

vidual test case may operate correctly. However, if incorporated into a `TestSuite` that runs every test case on the system, it may cause other test cases to fail. That failure mode can be difficult to diagnose, and the error may be located far from the test failure.

In the second situation, a test case may have updated some system state so that it cannot run again without manual intervention, which may consist of deleting test data from the database (for example). Think carefully before introducing manual intervention. First, the manual intervention will need to be documented. Second, the tests could no longer be run in an unattended mode, removing your ability to run tests overnight or as part of some automated periodic test run.

Call a superclass's setUp() and tearDown() methods when subclassing

When you consider:

```
public class SomeTestCase extends AnotherTestCase {
   // A connection to a database
   private Database theDatabase;
   public SomeTestCase (String testName) {
      super (testName);
   }
   public void testFeatureX () {
      ...
   }
   public void setUp () {
      // Clear out the database
      theDatabase.clear ();
   }
}
```

Can you spot the deliberate mistake? `setUp()` should call `super.setUp()` to ensure that the environment defined in `AnotherTestCase` initializes. Of course, there are exceptions: if you design the base class to work with arbitrary test data, there won't be a problem.

Do not load data from hard-coded locations on a filesystem

from some location in the filesystem. Consider the following:

```
public void setUp () {
   FileInputStream inp ("C:\\TestData\\dataSet1.dat");
   ...
}
```

The code above relies on the data set being in the `C:\TestData` path. That assumption is incorrect in two situations:

- A tester does not have room to store the test data on `C:` and stores it on another disk
- The tests run on another platform, such as Unix

One solution might be:

```
public void setUp () {
   FileInputStream inp ("dataSet1.dat");
   ...
}
```

However, that solution depends on the test running from the same directory as the test data. If several different test cases assume this, it is difficult to integrate them into one test suite without continually changing the current directory.

To solve the problem, access the dataset using either `Class.getResource()` or `Class.getResourceAsStream()`. Using them, however, means that resources load from a location relative to the class's origin.

Test data should, if possible, be stored with the source code in a configuration management (CM) system. However, if you're using the aforementioned resource mechanism, you'll need to write a script that moves all the test data from the CM system into the classpath of the system under test. A less ungainly approach is to store the test data in the source tree along with the source files. With this approach, you need a location-independent mechanism to locate the test data within the source tree. One such mechanism is a class. If a class can be mapped to a specific source directory, you could write code like this:

e how to map from a class to the directory that contains the identify the root of the source tree (assuming it has a single root) by a system property. The class's package name can then identify the directory where the source file lies. The resource loads from that directory. For Unix and NT, the mapping is straightforward: replace every instance of "." with `File.separatorChar`.

Keep tests in the same location as the source code

If the test source is kept in the same location as the tested classes, both test and class will compile during a build. This forces you to keep the tests and classes synchronized during development. Indeed, unit tests not considered part of the normal build quickly become dated and useless.

Name tests properly

Name the test case `TestClassUnderTest`. For example, the test case for the class `MessageLog` should be `TestMessageLog`. That makes it simple to work out what class a test case tests. Test methods' names within the test case should describe what they test:

- `testLoggingEmptyMessage()`
- `testLoggingNullMessage()`
- `testLoggingWarningMessage()`
- `testLoggingErrorMessage()`

Proper naming helps code readers understand each test's purpose.

Ensure that tests are time-independent

Where possible, avoid using data that may expire; such data should be either manually or programmatically refreshed. It is often simpler to instrument the class under test, with a mechanism for changing its notion of today. The test can then operate in a time-independent manner without having to refresh the data.

es. One approach to creating dates would be:

```
Date date = DateFormat.getInstance ().parse ("dd/mm/yyyy");
```

Unfortunately, that code doesn't work on a machine with a different locale. Therefore, it would be far better to write:

```
Calendar cal = Calendar.getInstance ();
Cal.set (yyyy, mm-1, dd);
Date date = Calendar.getTime ();
```

The second approach is far more resilient to locale changes.

Utilize JUnit's assert/fail methods and exception handling for clean test code

Many JUnit novices make the mistake of generating elaborate try and catch blocks to catch unexpected exceptions and flag a test failure. Here is a trivial example of this:

```
public void exampleTest () {
   try {
      // do some test
   } catch (SomeApplicationException e) {
      fail ("Caught SomeApplicationException exception");
   }
}
```

JUnit automatically catches exceptions. It considers uncaught exceptions to be errors, which means the above example has redundant code in it.

Here's a far simpler way to achieve the same result:

```
public void exampleTest () throws SomeApplicationException {
   // do some test
}
```

In this example, the redundant code has been removed, making the test easier to read and maintain (since there is less code).

t methods to express your intention in a simpler fashion. Instead of writing:

```
assert (creds == 3);
```

Write:

```
assertEquals ("The number of credentials should be 3", 3, creds);
```

The above example is much more useful to a code reader. And if the assertion fails, it provides the tester with more information. JUnit also supports floating point comparisons:

```
assertEquals ("some message", result, expected, delta);
```

When you compare floating point numbers, this useful function saves you from repeatedly writing code to compute the difference between the result and the expected value.

Use `assertSame()` to test for two references that point to the same object. Use `assertEquals()` to test for two objects that are equal.

Document tests in javadoc

Test plans documented in a word processor tend to be error-prone and tedious to create. Also, word-processor-based documentation must be kept synchronized with the unit tests, adding another layer of complexity to the process. If possible, a better solution would be to include the test plans in the tests' `javadoc`, ensuring that all test plan data reside in one place.

Avoid visual inspection

Testing servlets, user interfaces, and other systems that produce complex output is often left to visual inspection. Visual inspection -- a human inspecting output data for errors -- requires patience, the ability to process large quantities of information, and great attention to detail:

attributes not often found in the average human being. Below are some basic techniques that will help reduce the visual inspection component of your test cycle.

When testing a Swing-based UI, you can write tests to ensure that:

- All the components reside in the correct panels
- You've configured the layout managers correctly
- Text widgets have the correct fonts

A more thorough treatment of this can be found in the worked example of testing a GUI, referenced in the Resources section.

XML

When testing classes that process XML, it pays to write a routine that compares two XML DOMs for equality. You can then programmatically define the correct DOM in advance and compare it with the actual output from your processing methods.

Servlets

With servlets, a couple of approaches can work. You can write a dummy servlet framework and preconfigure it during a test. The framework must contain derivations of classes found in the normal servlet environment. These derivations should allow you to preconfigure their responses to method calls from the servlet.

For example: