PMD Applied

Tom Copeland

PMD Applied

Tom Copeland

Abstract

Supplemental chapter to "PMD Applied".

Library of Congress Catalog Number: 2005907843

ISBN: 0-9762214-1-1

Editing: Elizabeth Joyce

Table of Contents

Chapter 11. Writing Better JUnit Tests With PMD..... 1

11.1. JUnit Overview..... 1
11.2. Case Study..... 2
11.3. Make good use of the JUnit API..... 4
11.3.1. JUnitTestsShouldIncludeAssert..... 4
11.3.2. UseAssertEqualsInsteadOfAssertTrue..... 5
11.3.3. UseAssertNullInsteadOfAssertTrue..... 6
11.3.4. UseAssertSameInsteadOfAssertTrue..... 6
11.4. Make tests informative..... 7
11.5. Write tests that can fail..... 7
11.6. Keep the test code clean..... 8
11.6.1. JUnitStaticSuite..... 8
11.6.2. JUnitSpelling..... 8
11.6.3. TestClassWithoutTestCases..... 9
11.7. Conclusion..... 9
References..... 11
Index..... 13

List of Tables

11.1. PMD JUnit ruleset report.....4

Chapter 11. Writing Better JUnit Tests With PMD

A test of what is real is that it is hard and rough. Joys are found in it, not pleasure. What is pleasant belongs to dreams.
—Simone Weil

11.1. JUnit Overview

Here's some background information for those unfamiliar with JUnit and unit testing in general. Of course, there are many more complete resources on this topic; I've listed the main JUnit web site and several excellent books on JUnit in the bibliography.

Perhaps the simplest way to test a program is by compiling and running it. You write some code, compile it, and all's well. Perhaps you run the compiler with warnings turned all the way up; that shakes out a few more problems. After that, you could use a tool like PMD to check for more indepth problems. Finally, you actually run the program, enter some data and click some buttons, and generally see if everything looks normal. This technique works well enough for small programs, but manual testing gets somewhat tedious with a program of any complexity. No one wants to enter the same boilerplate data over and over, and to retest everything after making a small change is no fun at all.

The next step, then, is to automate some of that testing. A simple step in this direction is to have a `main` method in each class that creates an instance of that class and calls a few methods. So, for example, a `Car` class might have a `main` method that creates a new `Car` object, calls the `start` method, ensures that the `engineRunning` field is now `true`, and perhaps prints `Tests pass` if all seems well. If something goes wrong, it can print `Engine not running after car.start!` or something to that effect.

This is a move in the right direction, but it's still rather clunky. The programmer has to run a separate test for each class, which gets old after two or three classes. So the test invocations will be placed in a shell script or batch file, which then gets filled up with a bunch of class names. Next someone will write a script to call the `main` method on each class, which triggers problems for those classes which don't have tests. Finally, when that's cleaned up and the tests are all run, the results fly by and the programmer is scrolling up and down to see if everything worked. Also, having all those `main` methods clutters up the code; it results in bigger source and class files, and from a purely aesthetic point of view, it's just ugly.

Enter JUnit - a unit testing framework. With JUnit, you can quickly put together a collection of tests to exercise the functionality in your code and report any problems. Originally written by Erich Gamma and Kent Beck in 1998, JUnit matured to the point that its most recent release was in September of 2002 (although rumors of an impending JUnit 4 release are rattling around), and it's still arguably the most popular Java unit testing tool. Instead of keeping test code in a `main` method, you place it in a dedicated test class; if you have a `Car` class in your code, you'll also create a `CarTest` class.

Here's a quick example. Consider a simple `Car` class. Being a dependable vehicle, after the `start` method is called the engine should be running, and the code reflects that:

```
1   package example;
2   public class Car {
3       private boolean running;
4       public void start() {
5           running = true;
6       }
7       public boolean engineRunning() {
8           return running;
9       }
10  }
```

To ensure that this behavior stays the way it is, you could write a JUnit test class, `CarTest`:

```
1   package test.example;
2   import junit.framework.*;
3   import example.Car;
4   public class CarTest extends TestCase {
5       public void testRunningAfterStart() {
6           Car car = new Car();
7           car.start();
8           assertTrue(car.engineRunning());
9       }
10  }
```

You can run this test either from the command line or from an Ant script. Here's what the Ant output looks like for a successful run of `CarTest`:

```
$ ant cartest
Buildfile: build.xml

requires-junit:

compile:

copy:

cartest:
    [junit] Running test.example.CarTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.011 sec

BUILD SUCCESSFUL
Total time: 1 second
```

A success! Now we can make further changes to our `Car` class, resting safely in the knowledge that our unit tests will ensure the basic behavior of this vehicle is preserved.

A final thought on unit testing: it's not a silver bullet. It's very difficult to write a unit test that tests for a good look and feel, or a nice color scheme; some things need to be evaluated by a person. But unit testing can get you quite a ways down the road to good code quality.

11.2. Case Study

To illustrate how PMD can help clean up the code in a test case we'll do a case study. We'll look at a simple `Truck` class alongside a `TruckTest` unit test written to verify its functionality. We'll then run the PMD JUnit rules on the `TruckTest` unit test and clean it up, making it smaller, more readable, and more informative.

Here, then, is our `Truck`. It can be started, it can be stopped, and it has wheels, and it has a name:

```
1   package example;
2   public class Truck {
3       private boolean running;
4       private String name;
5       public Truck(String name) {
6           this.name = name;
7       }
8       public String getName() {
9           return name;
10       }
11      public void start() {
12          running = true;
13      }
14      public boolean engineRunning() {
15          return running;
16      }
17      public void stop() {
18          running = false;
19      }
20      public int getWheelCount() {
21          return 4;
22      }
23      public boolean equals(Object otherTruck) {
24          return ((Truck)otherTruck).getName() == name;
25      }
26      public int hashCode() {
27          return name.hashCode();
28      }
29  }
```

Next, the `TruckTest`. It looks rather thorough, but we can make it much better:

```
1   package test.example;
2   import junit.framework.TestCase;
3   import example.Truck;
4   public class TruckTest extends TestCase {
5       public void setup() {
6           Truck truck = new Truck("jim");
7       }
8       public void testEngine() {
9           Truck t = new Truck("jim");
10           t.start();
11          t.stop();
12      }
13      public void testWheelCount() {
14          Truck t = new Truck("jim");
15          assertEquals(4, t.getWheelCount());
16      }
17      public void testSame() {
18          Truck t = new Truck("jim");
19          assertTrue("Strings should be the same", t.getName() == "jim");
20      }
21      public void testEquals() {
22          Truck t1 = new Truck("jim");
23          Truck t2 = new Truck("jim");
24          assertTrue("Trucks with the same name should be equal", t1.equals(t2));
25      }
26      public void testNotYetCreated() {
27          Truck t1 = null;
28          assertTrue("Truck is still null", t1 == null);
29      }
30      public void testOdd() {
31          assertTrue("True should be true", true);
32      }
33  }
```

Here is our unit test in action via an Ant task:

```
$ ant trucktest
Buildfile: build.xml

requires-junit:

compile:

copy:

trucktest:
```

```
    [junit] Running test.example.TruckTest
    [junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 0.02 sec

BUILD SUCCESSFUL
Total time: 1 second
```

Finally, here's the way to run the JUnit ruleset on `TruckTest` and also the results:

```
$ ./pmd.sh regress/example/TruckTest.java text junit
```

Table 11.1. PMD JUnit ruleset report

Item	Line	Description
1	5	You may have mispelled a JUnit framework method (setUp or tearDown)
2	8	JUnit tests should include assert() or fail()
3	15	JUnit assertions should include a message
4	19	Use assertSame(x,y) instead of assertTrue(x==y), or assertNotSame(x,y) vs assertFalse(x==y)
5	24	Use assertEquals(x,y) instead of assertTrue(x.equals(y))
6	28	Use assertNull(x) instead of assertTrue(x==y), or assertNotNull(x) vs assertFalse(x==null)
7	28	Use assertSame(x,y) instead of assertTrue(x==y), or assertNotSame(x,y) vs assertFalse(x==y)
8	31	assertTrue(true) or similar statements are unnecessary

A fine unit test! We've uncovered a number of targets of opportunity. In the next sections, we'll examine each of these warnings, determine what's wrong, and demonstrate how to fix the problem.

11.3. Make good use of the JUnit API

One of the ways in which TruckTest can be improved is by using the JUnit API to its full advantage. This means using the methods that JUnit provides for writing more concise and thorough tests. Four different rules in the JUnit ruleset fall into this category; we'll look at each one to see what it found.

11.3.1. JUnitTestsShouldIncludeAssert

When examining the `testEngine` unit test method:

```
1       public void testEngine() {
2           Truck t = new Truck("jim");
3           t.start();
4           t.stop();
5       }
```

`JUnitTestsShouldIncludeAssert` reports this message:

```
JUnit tests should include assert() or fail()
```

In other words, `testEngine` is a reasonable test in that it exercises some of the `Truck`code, but it will only fail if an exception is thrown. It would be a bit more effective if it were rewritten to test the state of the `Truck` object after we invoke `start` and `stop`. Here's an improved version:

```
1       public void testEngine() {
2           truck.start();
3           assertTrue("Engine should be running. ", truck.engineRunning());
4           truck.stop();
5           assertFalse("Engine should not be running. ", truck.engineRunning());
6       }
```

That's a more helpful test; now it actually tests some conditions and fails if the expected result is not obtained.

I've noticed that this rule will find cases where parts of a unit test have been commented out - usually because they were failing! Of course, that's a clear signal that those tests need some attention; either they're testing the wrong thing, or they're testing something that's broken and needs to be fixed.

11.3.2. UseAssertEqualsInstead-OfAssertTrue

Running `UseAssertEqualsInsteadOfAssertTrue` produced this warning:

```
Use assertEquals(x, y) instead of assertTrue(x.equals(y))
```

And the offending code is here:

```
1       public void testEquals() {
2           Truck t1 = new Truck("jim");
3           Truck t2 = new Truck("jim");
4           assertTrue("Trucks with the same name should be equal", t1.equals(t2));
5       }
```

This test is a step in the right direction. It tests a condition and it will fail if someone changes the `Truck.equals` method to do something unexpected. But it's not as clean as it could be. Instead of calling the somewhat generic framework method `assertTrue` and calling the `equals` method inside the test expression, we can save a bit of typing by calling `assertEquals`:

```
1       public void testEquals() {
2           Truck t2 = new Truck("jim");
3           assertEquals("Trucks with the same name should be equal. ", truck, t2);
4       }
```

Now the unit test is a bit more expressive. With the `assertTrue` method call, someone reading this code would have had to read to the end of the line to see what sort of assertion was being made; now it's clear at a glance that the purpose of the test is to check the equality of the two objects.

11.3.3. UseAssertNullInstead-OfAssertTrue

`UseAssertNullInsteadOfAssertTrue` is along the same lines of `UseAssertEqualsInsteadOfAssertTrue`. Here's the warning message:

```
Use assertNull(x) instead of assertTrue(x==null), or assertNotNull(x) vs assertFalse(x==null)
```

And here's the corresponding code:

```
1       public void testNotYetCreated() {
2           Truck t1 = null;
3           assertTrue("Truck is still null", t1 == null);
4       }
```

This is a rather silly test, but it makes the point. If we want to ensure something is null, we can `assertTrue` a comparison of that object reference to null. But as the rule warning indicates, there's a better way:

```
1       public void testNotYetCreated() {
2           Truck t1 = null;
3           assertNull("Truck is still null", t1);
4       }
```

Again, using a specific JUnit framework method yields a more expressive test. By using `assertNull`we're making it clear that the purpose of this assert is to ensure a given object is null; there's no need to read across into the test expression to figure this out.

11.3.4. UseAssertSameInstead-OfAssertTrue

`UseAssertSameInsteadOfAssertTrue` is another "use the framework" rule. Here's what it found:

```
Use assertSame(x, y) instead of assertTrue(x==y), or assertNotSame(x,y) vs assertFalse(x==y)
```

And here's the problem test:

```
1       public void testSame() {
2           Truck t = new Truck("jim");
3           assertTrue("Strings should be the same", t.getName() == "jim");
4       }
```

Fixing this isn't hard; it's just a matter of calling the API method. Note that there's a difference between `assertSame` and `assertEquals`. `assertSame` ensures that the two object references point to the same object, whereas `assertEquals` simply ensures that they are equal. Here's the improved version:

```
1       public void testSame() {
2           assertSame("Strings should be the same. ", truck.getName(), "jim");
3       }
```

There's a small wrinkle here. The `String` object "jim" was used previously when it was passed into the `Truck` constructor, and the JVM intern'd it to avoid having multiple copies of a literal `String` cluttering up memory. So declaring another literal `String` "jim" just points to the same `String` object in memory, thus, the `assertSame` test passes.

11.4. Make tests informative

Another principle of writing good JUnit tests is to make the tests fail in a helpful manner. When one of the tests fails, we should know exactly what went wrong and where it happened. Thus the reason for `JUnitAssertionsShouldIncludeMessage`; it finds places where a test failure will only result in a generic failure message as opposed to something more helpful.

Here's what it found in `TruckTest` on line 15:

```
JUnit assertions should include a message
```

The code in question is this:

```
1       public void testWheelCount() {
2           Truck t = new Truck("jim");
3           assertEquals(4, t.getWheelCount());
4       }
```

Seems like a straightforward test, but if it fails we'll just get a generic error message:

```
There was 1 failure:
1) testWheelCount(test.example.TruckTest) junit.framework.AssertionFailedError: expected:<4> but
was:<3>
        at test.example.TruckTest.testWheelCount(TruckTest.java:15)
```

It's reasonably clear, but we can do better by modifying the test case:

```
1       public void testWheelCount() {
2           assertEquals("Trucks should have 4 wheels. ", truck.getWheelCount(), 4);
3       }
```

With this new usage, if someone changes our `Truck` class so that it only has three wheels, we'll get the following failure message:

```
1) testWheelCount(test.example.BetterTruckTest) junit.framework.AssertionFailedError: Trucks should
have 4 wheels.  expected:<4> but was:<3>
        at test.example.BetterTruckTest.testWheelCount (BetterTruckTest.java:18)
```

Now it's clear what the problem is; and someone who is working on a bug may be able to track down the cause sooner.

11.5. Write tests that can fail

There are a few ways to write a test that is unlikely to fail. We've already talked about one possibility - `JUnitTestsShouldIncludeAssert` will catch places where a test doesn't include an assertion.

Another possibility is the case found by the `UnnecessaryBooleanAssertion` rule; it catches places where an assertion is being made that always fails. Here's the case that it found in `TruckTest`:

```
1       public void testOdd() {
2           assertTrue("True should be true", true);
3       }
```

Usually this rule turns up cases that are a slightly more plausible. For example, someone may have put an `assertTrue(true)` statement in as a placeholder and then forgotten to go back and fix it.

There's not a standard fix for the problems this rule finds; it all depends on what the rest of the test case does (if anything) and the state of the class that's being tested. Sometimes this test should simply be deleted, and sometimes it should be enhanced to make it actually do something. But either way, this rule will find those cases for you so that you can deal with them.

11.6. Keep the test code clean

The last two rules in the PMD JUnit ruleset are there to catch naming and structural problems which might crop up in a large test suite. These may seem obvious to folks who have been writing unit tests for a while, but they crop up occasionally and it doesn't hurt to check for them.

11.6.1. JUnitStaticSuite

The first rule, `JUnitStaticSuite`, requires some background information. JUnit has a concept of a "test suite", which is a set of related unit tests. For example, if you wanted to run a certain subset of the `Truck` tests, you could define a `suite` method which included several tests:

```
public static Test suite() {
 TestSuite suite = new TestSuite();
 suite.addTest("testEngine");
 suite.addTest("testWheelCount");
 return suite;
}
```

This can be a handy feature if some of the tests take a long time to run or are only available on certain platforms. If you don't supply a `suite` method, then JUnit will by default collect all the methods that start with `test` and run them.

`JUnitStaticSuite` examines the source code and reports any `suite` method declarations that are not `public` or `static`. Suppose we had declared the above method but had forgotten the `static` modifier, like this:

```
public Test suite() {
 TestSuite suite = new TestSuite();
 suite.addTest("testEngine");
 suite.addTest("testWheelCount");
 return suite;
}
```

In that case JUnit would collect and run all the test methods in your test case rather than just running `testEngine` and `testWheelCount`. In a large application, this might go unnoticed for a while. But `JUnitStaticSuite`, trusty sidekick that it is, will find and report this misspelling, and the fix is usually as simple as (after a quick smack to the forehead) adding the missing modifier.

11.6.2. JUnitSpelling

JUnit provides two methods for setting up a test case and tearing it down. This is handy if you need to perform a time-consuming operation before running your tests. For example,

suppose you wanted to parse a large XML file and then run a number of tests on it. Rather than parsing the file for each test, you can declare a `setUp` method in your test class which will be called before any other test method. Similarly, if you declare a `tearDown` method, it will be called after all the other tests in the class are called.

Occasionally someone will misspell one of these method names and will type it as `setup` or `teardown` or some other variation. It's easy to mistype a method name, and of course you'll get no compile-time or runtime warnings. Instead, your tests will simply fail if they had expected something to be done in the `setUp` method, and you'll waste precious seconds examining the failing test methods to see what went wrong.
`JUnitSpelling` to the rescue; it'll catch those mistakes.

Here's the example of this from `TruckTest`:

```
1       public void setup() {
2           Truck truck = new Truck("jim");
3       }
```

The warning for this problem is straightforward:

```
You may have misspelled a JUnit framework method (setUp or tearDown)
```

And the fix is easy as well:

```
1       public void setUp() {
2           truck = new Truck("jim");
3       }
```

That is, it's an easy fix once you find the problem! And in this case, there was another problem as well; the `truck` variable should have been a field, not a local variable. Of course, the PMD unused code ruleset would catch that problem, but generally, it's much easier for dead code to hide inside other dead code. Fixing the `setUp` method name helps make the whole test a bit tidier.

11.6.3. TestClassWithoutTestCases

The last rule in the JUnit ruleset is a very simple tidiness check; it searches for classes which end in `Test` but are not JUnit tests. It's a sign of the ubiquity of JUnit that this naming convention has become so common that violating it would cause confusion.

11.7. Conclusion

Here's the final product: a `BetterTruckTest` class, suitable for framing:

```
1   package test.example;
2
3   import junit.framework.TestCase;
4   import example.Truck;
5
6   public class BetterTruckTest  extends TestCase {
7       private Truck truck;
8       public void setUp() {
9           truck = new Truck("jim");
10       }
11      public void testEngine() {
12          truck.start();
13          assertTrue("Engine should be running. ", truck.engineRunning());
14          truck.stop();
15          assertFalse("Engine should not be running. ", truck.engineRunning());
16      }
```

```
17      public void testWheelCount() {
18          assertEquals("Trucks should have 4 wheels. ", truck.getWheelCount(), 4);
19      }
20      public void testSame() {
21          assertSame("Strings should be the same. ", truck.getName(), "jim");
22      }
23      public void testEquals() {
24          Truck t2 = new Truck("jim");
25          assertEquals("Trucks with the same name should be equal. ", truck, t2);
26      }
27      public void testNotYetCreated() {
28          Truck t1 = null;
29          assertNull("Truck is still null", t1);
30      }
31  }
```

This unit test is shorter than the test we started with, it does more actual testing, and if anything fails, the messages it produces will be more informative. In short, it's a much better unit test, and it's a good lesson on how to use the JUnit framework to its full advantage.

A few end notes are appropriate here. Most of these rules are only intended to be run on JUnit test classes; if you run them on other code you'll almost certainly get all sorts of spurious warnings. Thus these rules are most useful if your tests are separated into their own directory or package or at least have some sort of naming convention that allows them to be easily selected via a script or an Ant task. Also, note that all the code used in this chapter is on this book's web site ([A]) so you can download it and experiment with it as you see fit.

Hopefully this chapter has given you some information on how you can use PMD to keep your JUnit tests in good order. Naturally, if you have any suggestions for new rules, or if you have any improvements to the current rules, please share them with the rest of the community by posting to the PMD web site. Thanks for using PMD!

[A] http://pmdapplied.com

References

[1] *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2003.

[2] *Java Testing Patterns*, Wiley, 2004.

Colophon

This chapter was produced with the DocBook XSL stylesheets, xsltproc, and AltSoft Xml2PDF.

Index

A
Ant, 2

B
Beck, Kent, 1

G
Gamma, Erich, 1

J
JUnit, 1
JUnitAssertionsShouldIncludeMessage, 7
JUnitSpelling, 9
JUnitStaticSuite, 8
JUnitTestsShouldIncludeAssert, 4, 7

P
PMD, 1

U
UnnecessaryBooleanAssertion, 7
UseAssertEqualsInsteadOfAssertTrue, 5
UseAssertNullInsteadOfAssertTrue, 6
UseAssertSameInsteadOfAssertTrue, 6