In pursuit of code quality: Don't be fooled by the coverage report

Are your test coverage measurements leading you astray?

Andrew Glover January 31, 2006

Test coverage tools bring valuable depth to unit testing, but they're often misused. This month, Andrew Glover brings his considerable expertise in this area to his new series, *In pursuit of code quality*. This first installment takes a closer look at what the numbers on the coverage report really mean, as well as what they don't. He then suggests three ways you can use your coverage to ensure code quality early and often.

View more content in this series

Do you remember what it was like before most developers jumped on the code quality bandwagon? In those days, a skillfully placed `main()` method was considered both agile and adequate for testing. *Yikes*! We've come a long way since then. I, for one, am immensely grateful that automated testing is now an essential facet of quality-centric code development. And that's not all I'm grateful for. Java™ developers have a plethora of tools for gauging code quality through code metrics, static analysis, and more. Heck, we've even managed to categorize refactoring into a handy set of patterns!

Ensure your code quality

To get the answers to your questions related to code quality, visit the Code Quality discussion forum, moderated by Andrew Glover.

All these new tools make it easier than ever to ensure code quality, but you do have to know how to use them. In this series of articles, I'll focus on the sometimes arcane details of ensuring code quality. In addition to familiarizing you with the variety of tools and techniques available for code quality assurance, I'll show you how to:

- Define and effectively measure the aspects of your code that most impact quality.
- Set quality assurance goals and plan your development efforts accordingly.
- Decide which code quality tools and techniques actually meet your needs.
- Implement best practices (and weed out poor ones) so that ensuring code quality *early and often* becomes a painless and effective aspect of your development practice.

I'll start this month with a look at one of the most popular and easy additions to a Java developer's quality assurance toolkit: test coverage measurement.

Beware fools gold

It's the morning following a nightly build and everyone's standing around the water cooler. Developers and management alike are exchanging bold NFL-style pats on the backside when they learn that a few particularly well-tested classes have coverage rates in the high 90s! The collective confidence of the team is at an all-time high. "Refactor with abandon!" can be heard in the distance as defects become a distant memory and the responsibility of the weak and inferior. But there is one small, dissenting voice that says:

Ladies and Gentlemen: Don't be fooled by the coverage report.

Now, don't get me wrong: There's nothing foolish about using test coverage tools. They're a great addition to the unit testing paradigm. What's important is how you synthesize the information once you've got it, and that's where some development teams make their first mistake.

High coverage rates simply mean that a lot of code was exercised. High coverage rates do not imply that code was exercised *well*. If you're focusing on code quality, you need to understand exactly how test coverage tools work, as well as how they don't; then you'll know how to use these tools to obtain valuable information, rather than just settling for high coverage goals, as many developers do.

Test coverage measurement

Test coverage tools are generally easy to add into an established unit testing process, and the results can be reassuring. Simply download one of the available tools, slightly modify your Ant or Maven build script, and you and your colleagues have a new kind of report to talk about around the water cooler: *The Test Coverage Report*. It can be a big comfort when packages like `foo` and `bar` show astonishingly *high* coverage, and it's tempting to rest easy when you believe that at least a portion of your code is certifiably "bug free." But to do so would be a mistake.

There are different types of coverage measurements, but most tools focus on *line coverage*, also known as *statement coverage*. In addition, some tools report *branch coverage*. A test coverage measurement is obtained by exercising a code base with a test harness and capturing data that corresponds to code having been "touched" throughout the lifetime of the test process. The data is then synthesized to produce a coverage report. In Java shops, the test harness is commonly JUnit and the coverage tool is usually something like Cobertura, Emma, or Clover, to name a few.

Line coverage simply indicates that a particular line of code was exercised. If a method is 10 lines long and 8 lines of the method were exercised in a test run, then the method has a line coverage of 80%. The process works at the aggregate level as well: If a class has 100 lines and 45 of them were touched, then the class has a line coverage of 45%. Likewise, if a code base comprises 10,000 non-commenting lines of code and 3,500 of them were exercised on a particular test run, then the code base's line coverage is 35%.

Tools that report *branch coverage* attempt to measure the coverage of decision points, such as conditional blocks containing logical `AND`s or `OR`s. Just like line coverage, if there are two branches in a particular method and both were covered through tests, then you could say the method has 100% branch coverage.

The question is, how useful are these measurements? Clearly, all of this information is easy to obtain, but it's up to you to be discerning about how you synthesize it. Some examples clarify my point.

Code coverage in action

I've created a simple class in Listing 1 to embody the notion of a class hierarchy. A given class can have a succession of superclasses -- like `Vector`, whose parent is `AbstractList`, whose parent is `AbstractCollection`, whose parent is `Object`:

Listing 1. A class that represent a class hierarchy

```
package com.vanward.adana.hierarchy;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class Hierarchy {
  private Collection classes;
  private Class baseClass;

  public Hierarchy() {
    super();
    this.classes = new ArrayList();
  }

  public void addClass(final Class clzz){
    this.classes.add(clzz);
  }
  /**
   * @return an array of class names as Strings
   */
  public String[] getHierarchyClassNames(){
    final String[] names = new String[this.classes.size()];
    int x = 0;
    for(Iterator iter = this.classes.iterator(); iter.hasNext();){
       Class clzz = (Class)iter.next();
       names[x++] = clzz.getName();
    }
    return names;
  }

  public Class getBaseClass() {
    return baseClass;
  }

  public void setBaseClass(final Class baseClass) {
    this.baseClass = baseClass;
  }
}
```

As you can see, Listing 1's `Hierarchy` class holds a `baseClass` instance and its collection of superclasses. The `HierarchyBuilder` in Listing 2 creates the `Hierarchy` class through two overloaded `static` methods dubbed `buildHierarchy`:

Listing 2. A class hierarchy builder

```
package com.vanward.adana.hierarchy;

public class HierarchyBuilder {

  private HierarchyBuilder() {
    super();
  }

  public static Hierarchy buildHierarchy(final String clzzName)
    throws ClassNotFoundException{
      final Class clzz = Class.forName(clzzName, false,
          HierarchyBuilder.class.getClassLoader());
      return buildHierarchy(clzz);
  }

  public static Hierarchy buildHierarchy(Class clzz){
    if(clzz == null){
      throw new RuntimeException("Class parameter can not be null");
    }

    final Hierarchy hier = new Hierarchy();
    hier.setBaseClass(clzz);

    final Class superclass = clzz.getSuperclass();

    if(superclass !=
      null && superclass.getName().equals("java.lang.Object")){
       return hier;
    }else{
       while((clzz.getSuperclass() != null) &&
          (!clzz.getSuperclass().getName().equals("java.lang.Object"))){
             clzz = clzz.getSuperclass();
             hier.addClass(clzz);
       }
       return hier;
    }
  }
}
```

It's testing time!

What would an article about test coverage be without a test case? In Listing 3, I define a simple, sunny-day scenario JUnit test class with three test cases, which attempt to exercise both the `Hierarchy` and `HierarchyBuilder` classes:

Listing 3. Test that HierarchyBuilder!

```
package test.com.vanward.adana.hierarchy;

import com.vanward.adana.hierarchy.Hierarchy;
import com.vanward.adana.hierarchy.HierarchyBuilder;
import junit.framework.TestCase;

public class HierarchyBuilderTest extends TestCase {

  public void testBuildHierarchyValueNotNull() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
```

```
    assertNotNull("object was null", hier);
  }

  public void testBuildHierarchyName() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.Assert",
       "junit.framework.Assert",
         hier.getHierarchyClassNames()[1]);
  }

  public void testBuildHierarchyNameAgain() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.TestCase",
       "junit.framework.TestCase",
         hier.getHierarchyClassNames()[0]);
  }

}
```

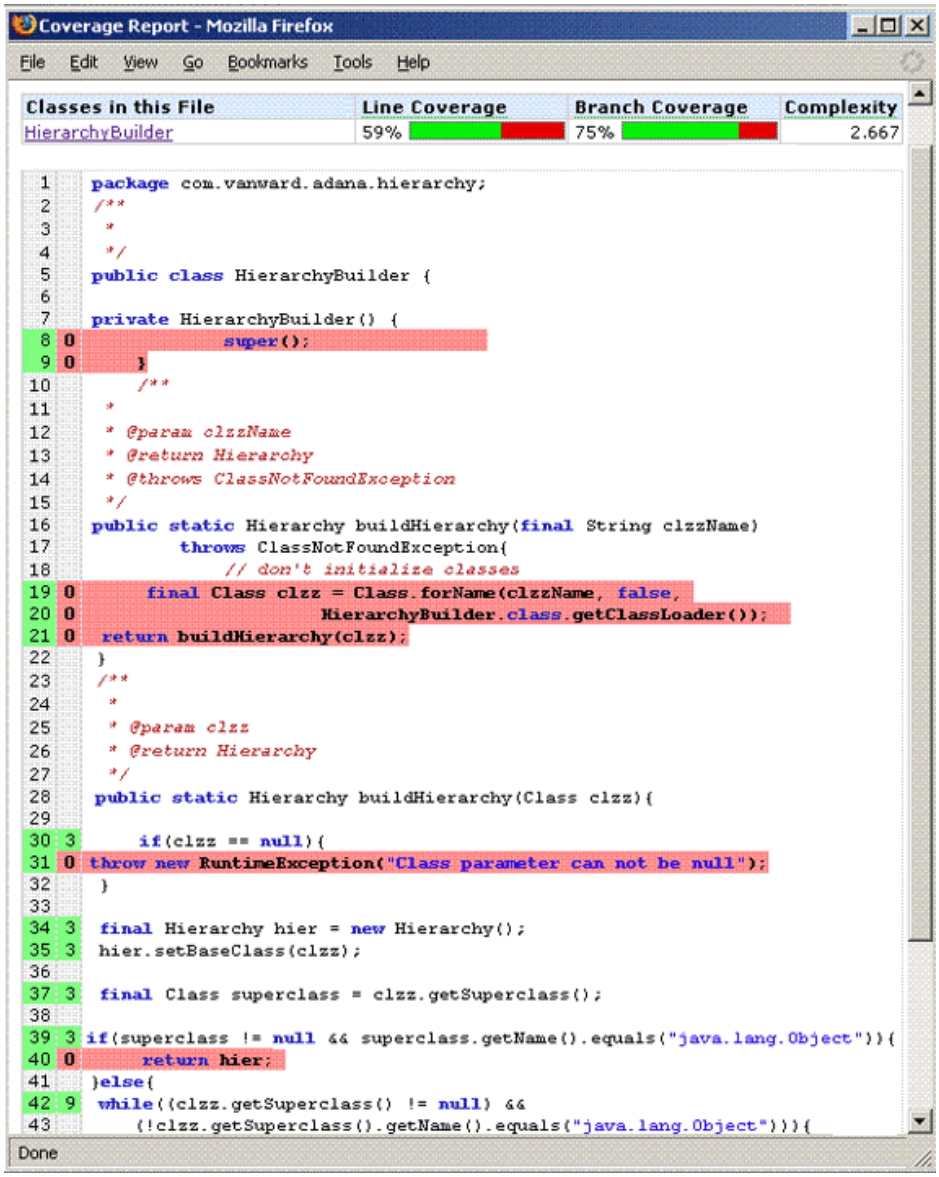
Because I'm an avid tester, I naturally want to run some coverage tests. Of the code coverage tools available to Java developers, I tend to stick with Cobertura because I like its friendly reports. Also, Cobertura is an open source project, which forked the pioneering JCoverage project.

The Cobertura report

Running a tool like Cobertura is as simple as running your JUnit tests, only with a middle step of instrumenting the code under test with specialized logic to report on coverage (which is all handled via the tool's Ant tasks or Maven's goals).

As you can see in Figure 1, the coverage report for `HierarchyBuilder` illustrates a few sections of code that *weren't* exercised. In fact, Cobertura claims that `HierarchyBuilder` has as 59% line coverage rate and a 75% branch coverage rate.

Figure 1. The Cobertura report



So, my first shot at coverage testing failed to test a number of things. First, the `buildHierarchy()` method, which takes a `String` as a parameter, wasn't tested at all. Second, two conditions in the other `buildHierarchy()` method weren't exercised either. Interestingly, it's the second unexercised `if` block that is a concern.

I'm not worried at this point because all I have to do is add a few more test cases. Once I've reached those areas of concern, I should be good to go. Note my logic here: I used the coverage report to understand what *wasn't* tested. Now I have the option to use this data to either enhance my testing or move on. In this case, I'm going to enhance my testing, because I've left a few important areas uncovered.

Cobertura: Round 2

Listing 4 is an updated JUnit test case that adds a handful of additional test cases in an attempt to fully exercise `HierarchyBuilder`:

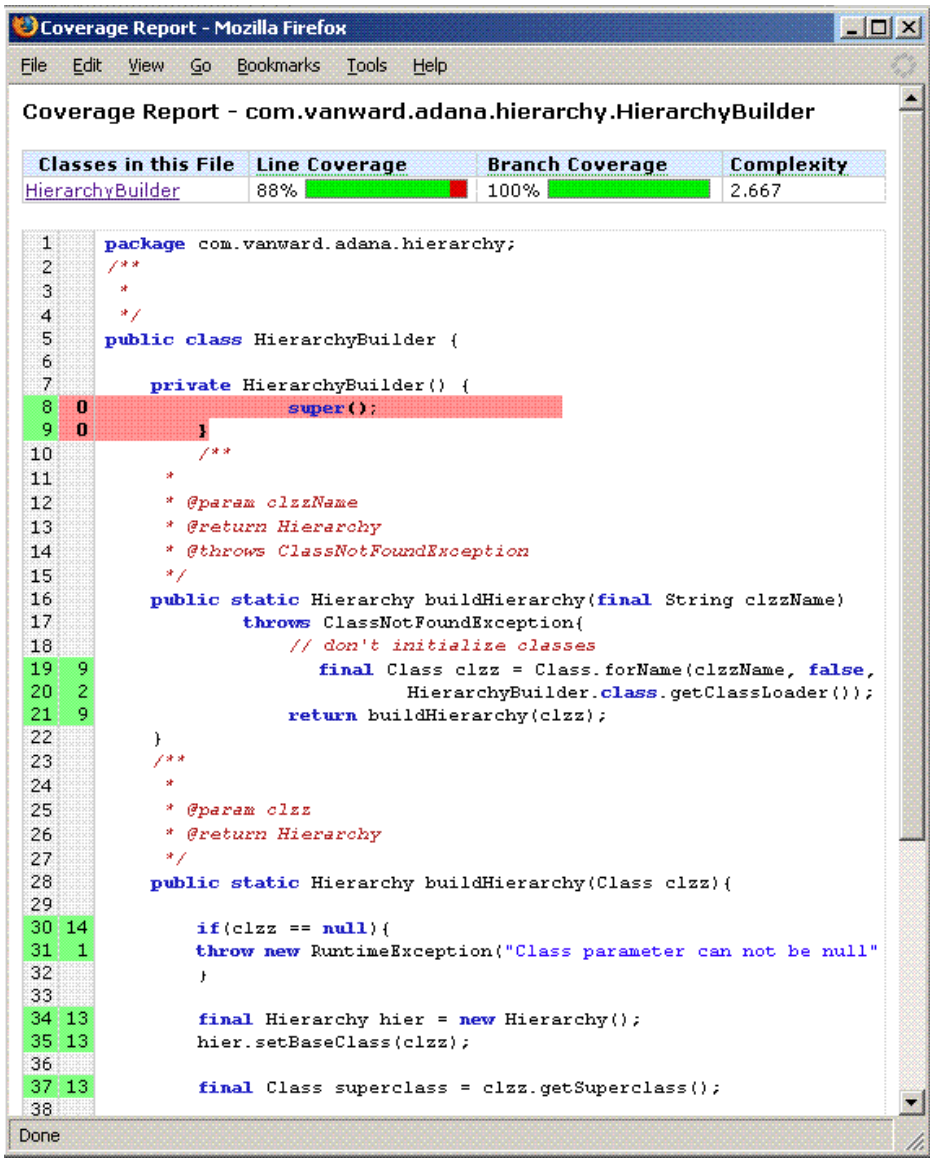
Listing 4. An updated JUnit test case

```
package test.com.vanward.adana.hierarchy;

import com.vanward.adana.hierarchy.Hierarchy;
import com.vanward.adana.hierarchy.HierarchyBuilder;
import junit.framework.TestCase;

public class HierarchyBuilderTest extends TestCase {

  public void testBuildHierarchyValueNotNull() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertNotNull("object was null", hier);
  }

  public void testBuildHierarchyName() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.Assert",
       "junit.framework.Assert",
         hier.getHierarchyClassNames()[1]);
  }

  public void testBuildHierarchyNameAgain() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.TestCase",
       "junit.framework.TestCase",
         hier.getHierarchyClassNames()[0]);
  }

  public void testBuildHierarchySize() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be 2", 2, hier.getHierarchyClassNames().length);
  }

  public void testBuildHierarchyStrNotNull() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertNotNull("object was null", hier);
  }

  public void testBuildHierarchyStrName() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.Assert",
      "junit.framework.Assert",
        hier.getHierarchyClassNames()[1]);
  }

  public void testBuildHierarchyStrNameAgain() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.TestCase",
      "junit.framework.TestCase",
        hier.getHierarchyClassNames()[0]);
  }

  public void testBuildHierarchyStrSize() throws Exception{
     Hierarchy hier =
```

```
        HierarchyBuilder.
        buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
     assertEquals("should be 2", 2, hier.getHierarchyClassNames().length);
  }

  public void testBuildHierarchyWithNull() {
     try{
       Class clzz = null;
       HierarchyBuilder.buildHierarchy(clzz);
       fail("RuntimeException not thrown");
     }catch(RuntimeException e){}
  }
}
```

When I run the test coverage process again with the new test cases, I get a much more complete report, as shown in Figure 2. I've now covered the untested `buildHierarchy()` method as well as hitting both `if` blocks in the other `buildHierarchy()` method. `HierarchyBuilder`'s constructor is `private`, however, so I can't test it via my test class (nor do I care to); therefore, my line coverage still hovers at 88%.

Figure 2. Who says there are no second chances?




```
 1    package com.vanward.adana.hierarchy;
 2    /**
 3     *
 4     */
 5    public class HierarchyBuilder {
 6
 7        private HierarchyBuilder() {
 8  0                 super();
 9  0             }
10            /**
11         *
12         * @param clzzName
13         * @return Hierarchy
14         * @throws ClassNotFoundException
15         */
16        public static Hierarchy buildHierarchy(final String clzzName)
17                throws ClassNotFoundException{
18                    // don't initialize classes
19   9                  final Class clzz = Class.forName(clzzName, false,
20   2                          HierarchyBuilder.class.getClassLoader());
21   9                  return buildHierarchy(clzz);
22        }
23        /**
24         *
25         * @param clzz
26         * @return Hierarchy
27         */
28        public static Hierarchy buildHierarchy(Class clzz){
29
30  14            if(clzz == null){
31   1            throw new RuntimeException("Class parameter can not be null"
32                }
33
34  13            final Hierarchy hier = new Hierarchy();
35  13            hier.setBaseClass(clzz);
36
37  13            final Class superclass = clzz.getSuperclass();
38
```

As you can see, using a code coverage tool *can* uncover important code that doesn't have a corresponding test case. The important thing is to exercise caution when viewing the reports (*especially* ones with high values), for they can hide nefarious subtleties. Let's look at a couple more examples of code issues that can hide behind high coverage rates.

The trouble with conditionals

As you may well know by now, many variables found in code can have more than one state; furthermore, the presence of conditionals creates multiple paths of execution. With these caveats in mind, I've defined an absurdly simple class with one method in Listing 5:

Listing 5. Do you see the defect below?

```
package com.vanward.coverage.example01;

public class PathCoverage {

  public String pathExample(boolean condition){
    String value = null;
    if(condition){
      value = " " + condition + " ";
    }
    return value.trim();
  }
}
```

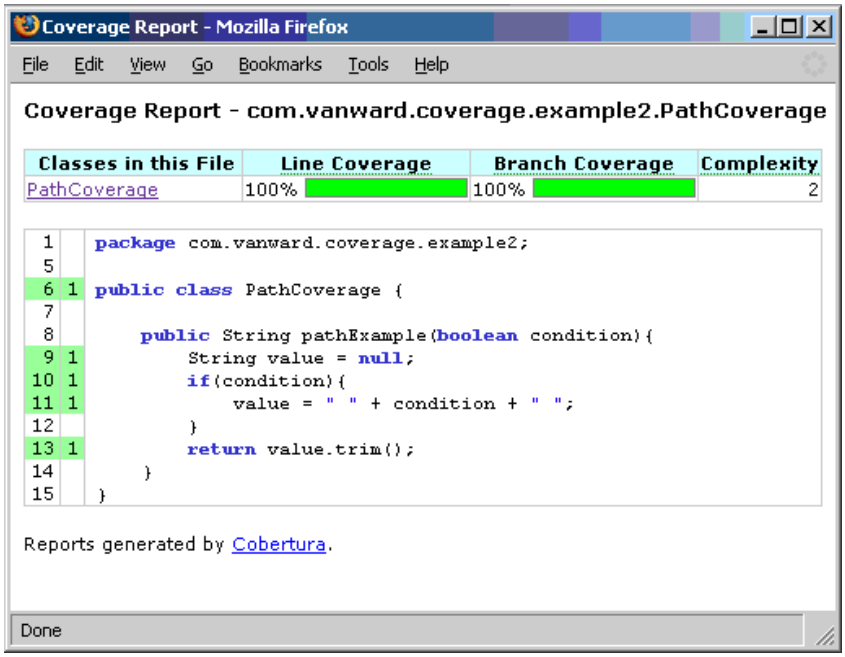
Listing 5 has an insidious defect in it -- do you see it? If not, no worries: I'll just write a test case to exercise the `pathExample()` method and ensure it works correctly in Listing 6:

Listing 6. JUnit to the rescue!

```
package test.com.vanward.coverage.example01;

import junit.framework.TestCase;
import com.vanward.coverage.example01.PathCoverage;

public class PathCoverageTest extends TestCase {

  public final void testPathExample() {
    PathCoverage clzzUnderTst = new PathCoverage();
    String value = clzzUnderTst.pathExample(true);
    assertEquals("should be true", "true", value);
  }
}
```

My test case runs flawlessly and my handy-dandy code coverage report (below in Figure 3) makes me look like a superstar, with 100% test coverage!

Figure 3. Rock star coverage, baby!



I'm thinking it's time to go hang out by the water cooler, but wait -- didn't I suspect a defect in that code? Closer examination of Listing 5 shows that Line 13 will indeed throw a `NullPointerException` if `condition` is `false`. *Yeesh*, what happened here?

It turns out that line coverage isn't such a great indicator of test effectiveness.

The horror of paths

In Listing 7, I've defined another simple example with an *indirect*, yet flagrant defect. Note the second half of the `if` conditional found in the `branchIt()` method. (The `HiddenObject` class is defined in Listing 8.)

Listing 7. This code is simple enough

```
package com.vanward.coverage.example02;

import com.acme.someotherpackage.HiddenObject;

public class AnotherBranchCoverage {

  public void branchIt(int value){
    if((value > 100) || (HiddenObject.doWork() == 0)){
      this.dontDoIt();
    }else{
      this.doIt();
    }
  }

  private void dontDoIt(){
    //don't do something...
  }

  private void doIt(){
    //do something!
  }
```

```
}
```

Yikes! The `HiddenObject` in Listing 8 is *evil*. Calling the `doWork()` method as I did in Listing 7 yields a `RuntimeException`:

Listing 8. Uh oh!

```
package com.acme.someotherpackage.HiddenObject;

public class HiddenObject {

  public static int doWork(){
    //return 1;
    throw new RuntimeException("surprise!");
  }
}
```

But surely I can catch the exception with a nifty test! In Listing 9, I've written another sunny-day test in an attempt to win my way back to rock stardom:

Listing 9. Risk avoidance with JUnit

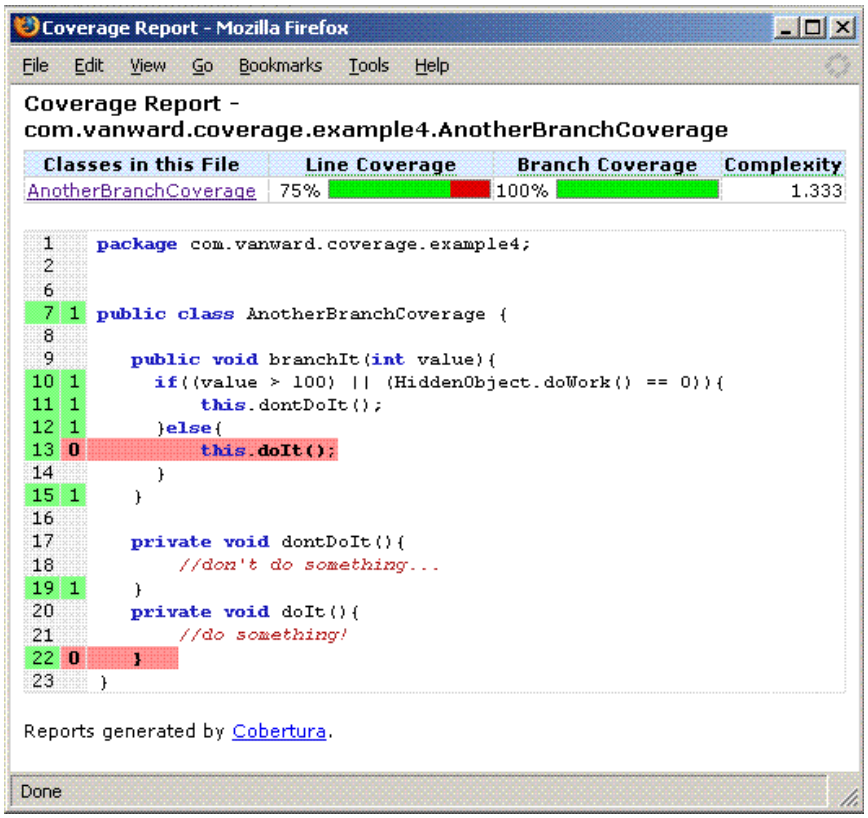
```
package test.com.vanward.coverage.example02;

import junit.framework.TestCase;
import com.vanward.coverage.example02.AnotherBranchCoverage;

public class AnotherBranchCoverageTest extends TestCase {

  public final void testBranchIt() {
    AnotherBranchCoverage clzzUnderTst = new AnotherBranchCoverage();
    clzzUnderTst.branchIt(101);
  }
}
```

What do you think of this test case? You probably would have written a few more test cases than I did, but imagine if that dubious conditional in Listing 7 had more than one short-circuit operation. Imagine if the logic in the first half was a bit more cerebral than a simple `int` comparison -- how many test cases would *you* write before you were satisfied?

Just give me the numbers

The results of my test coverage analysis of Listings 7, 8, and 9 shouldn't be any surprise to you by now. The report in Figure 4 shows that I've achieved 75% line coverage and 100% branch coverage. Most important, I've exercised line 10!

Figure 4. The fools reward



Boy am I proud, at least on first reflection. But do you see what's misleading about this report? A cursory look could lead you to believe the code was *well tested*. Based on that, you would probably assume the risk of a defect to be quite low. The report does little-to-nothing to help you ascertain that the second half of the `or` short-circuit is a ticking time bomb!

Testing for quality

I'll say it one more time: you can (and should) use test coverage tools as part of your testing process, but *don't be fooled by the coverage report*. The main thing to understand about coverage reports is that they're best used to expose code that *hasn't been* adequately tested. When you examine a coverage report, seek out the low values and understand why that particular code hasn't been tested fully. Knowing this, developers, managers, and QA professionals can use test coverage tools where they really count -- namely for three common scenarios:

- Estimating the time to modify existing code
- Evaluating code quality
- Assessing functional testing

Now that I've established some of the ways test coverage reports can lead you astray, consider these best practices for using them to your advantage.

1. Estimating the time to modify existing code

Writing test cases against code naturally increases the collective confidence of a development team. Tested code is easier to refactor, maintain, and enhance than code without corresponding

test cases. Test cases also serve as adept documentation because they implicitly demonstrate *how* the code under test works. Moreover, if the code under test changes, test cases usually change in parallel, unlike static code documentation, such as comments and Javadocs.

On the flipside, code without corresponding tests can be more challenging to understand and is harder to modify *safely*. Therefore, knowing whether code has been tested, and seeing the actual test coverage numbers, can allow developers and managers to more accurately predict the time needed to modify existing code.

A trip back to the water cooler should illustrate my point nicely.

> Linda from marketing: "We'd like to have the system do *x* when the user executes a trade. How long would it take? Our customers need this feature as soon as possible."
> Jeff the manager: "Let's see, that code was written by Joe a few months back. It will require changes to the business layer and a few changes to the UI. Mary should be able to do the work in a couple of days."
> Linda: "Joe? Who's that?"
> Jeff: "Oh Joe, yeah. We fired him because he didn't know what he was doing."

That situation sounds kind of ominous, doesn't it? Still, Jeff assigns the new feature to Mary, who also thinks she can have it done in just two days -- until she sees the code, that is.

> Mary: "Was Joe *asleep* when he wrote this code? This is the worst stuff I've ever seen. I'm not even sure this is Java code. There's no chance I can change this without breaking it! I quit."

Things don't look good for the water cooler team, do they? But let's see what happens when I rewind this series of unfortunate events, this time empowering Jeff and Mary with a coverage report. When Linda requests the new feature, one of the first things Jeff does is consult the handy-dandy coverage report from the previous build. Noticing that the particular packages requiring changes have little to no coverage, he then consults with Mary.

> Jeff: "The code Joe wrote is horrible, and most of it is untested. How long do you think you'll need add the changes to support Linda's request?"
> Mary: "That code is a mess. I'm afraid to even look at it. Can't Mark do it?"
> Jeff: "I just fired Mark because he doesn't write tests. I need you to test this code and then add the changes. Tell me how much time you need."
> Mary: "I'll need at least two days to write the tests, then I'd like to refactor the code, and then I'll add the features. I'm thinking four days total."

As they say, knowledge is power. Developers can use coverage reports to check code quality *before* attempting to modify it. Likewise, managers can use coverage data to better estimate the time a developer actually needs to do the work.

2. Evaluating code quality

Developer testing decreases the risk of defects in code, so many development teams now require that unit tests be written alongside newly developed or modified code. As the case of Mark

illustrates above, however, unit testing doesn't always run parallel to coding, which can result in lower quality code.

Monitoring coverage reports helps development teams quickly spot code that is growing *without* corresponding tests. For example, running a coverage report in the beginning of the week shows that a key package in the project has a coverage rate of 70%. If later in the week that package's coverage has slipped to 60%, then you can infer that:

- The package grew in terms of lines of code, but no corresponding tests were written for the new code (or that newly added tests do not effectively cover the new code).
- Test cases are being removed.
- Both things are happening at once.

The beauty here is being able to observe trends. Viewing the report regularly makes it easier to set goals (such as obtaining coverage rates, maintaining test cases to lines of code ratios, etc.) and then monitor their progress. If you happen to notice that tests are routinely not being written, you can take proactive action, such as setting up developers for training, mentoring, or buddy programming. Informed response is much better than pointing fingers when the customer discovers that "once in a lifetime" defect (which could have been exposed with a simple test months earlier), or the inevitable surprise (and anger) when management finds out unit testing wasn't being done.

Using coverage reports to ensure proper testing is a great practice. The trick is to do it in a disciplined manner. For example, try generating and viewing coverage reports nightly, as part of a *continuous integration* process.

3. Assessing functional testing

Given that a code coverage report is most effective at demonstrating sections of code *without* adequate testing, quality assurance personnel can use this data to assess areas of concern with respect to functional testing. Let's go back to the water cooler and see what Drew, the QA lead, has to say about Joe's code:

> Drew to Jeff: "We're drawing up test cases for the next release and we noticed large swaths of code with basically no code coverage. It appears it's the code relating to stock trades."
> Jeff: "Yeah, we're having some issues in that area. If I were a betting man, I'd pay special attention to that functional area. Mary is working on the other major changes to the application -- she's doing a good job of writing unit tests, but the code is by no means perfect."
> Drew: "Yeah, I'm determining resources and levels of effort and it looks like we're going to be short again. I guess I'll have the team focus on the stock trading stuff."

Again, knowledge is power. Working in careful coordination with other stakeholders in the software lifecycle, such as QA, you can use the insight provided by a coverage report to facilitate risk mitigation. In the above scenario, perhaps Jeff can give Drew's team an early release that may not

contain all of Mary's changes; however, Drew's team can focus on the stock-trading aspect of the application where risk of defects seems to be higher than the code with corresponding unit tests.

Where testing pays off

Test coverage measurement tools are a fantastic addition to the unit testing paradigm. Coverage measurement lends both depth and precision to an already beneficial process. You should, however, view code coverage reports with careful conjecture. High coverage percentages alone do not ensure the quality of your code. Highly covered code isn't necessarily free of defects, although it's certainly *less likely* to contain them.

The trick to test coverage measurement is to use the coverage report to expose code that *hasn't been* tested, on a micro level and on a macro level. You can facilitate deeper coverage testing by analyzing your code base from the top level as well as analyzing individual class coverage. Once you've integrated this principle you and your organization can use coverage measurement tools where they really count, such as to estimate the time needed for a project, continuously monitor code quality, and facilitate QA collaboration.

Related topics

- "Test your tests with Jester" (Elliotte Rusty Harold, developerWorks, March 2005): Jester excels at finding test-suite problems and provides unique insights into the structure of a code base.
- "The business value of software quality" (Geoffrey Bessin, The Rational Edge, June 2004): Describes the IBM Rational proposition for improving code quality and lists major tools used for quality assurance.