[[](http://www.onjava.com/)](http://www.onjava.com/)  
Published on [**ONJava.com**](http://www.onjava.com/) (<http://www.onjava.com/>)  
[See this](http://www.onjava.com/pub/a/general/print_code.html) if you're having trouble printing code examples  
  
**Statement, Branch, and Path Coverage Testing in Java**

by [Joe Ponczak](http://www.onjava.com/pub/au/2953)   
03/02/2007

**Introduction**

Code coverage is a way to measure the level of testing you've performed on your software. Gathering coverage metrics is a straightforward process: Instrument your code and run your tests against the instrumented version. This produces data showing what code you did—or, more importantly, did not—execute. Coverage is the perfect complement to unit testing: unit tests tell you whether your code performed as expected, and code coverage tells you what remains to be tested.

Most developers understand this process and agree on its value proposition, and often target 100 percent coverage. *Although 100 percent coverage is an admirable goal, 100 percent of the wrong type of coverage can still leave problems undiscovered.* A typical software development effort measures coverage in terms of the number of either statements or branches to be tested. Even with 100 percent statement or branch coverage, critical bugs still may be present in the logic of your code, leaving both developers and managers with a false sense of security.

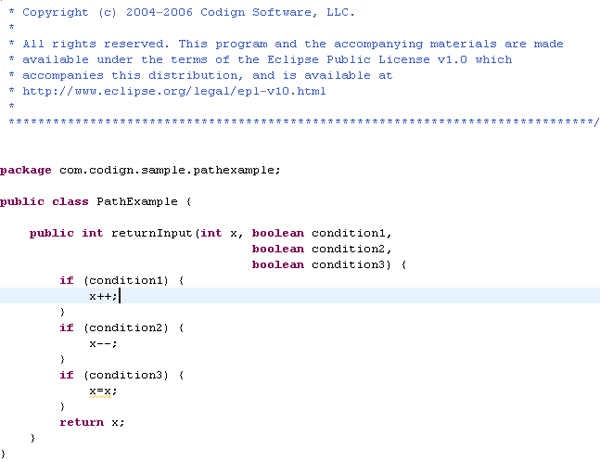
How can 100 percent coverage be insufficient? Because statement and branch coverage do not tell you whether the logic in your code was executed. Statement and branch coverage are great for uncovering glaring problems found in unexecuted blocks of code, but they often miss bugs related to both decision structures and decision interactions. Path coverage, on the other hand, is a more robust and comprehensive technique that helps reveal defects early.

Before you learn about path coverage, look at some of the problems with statement and branch coverage.

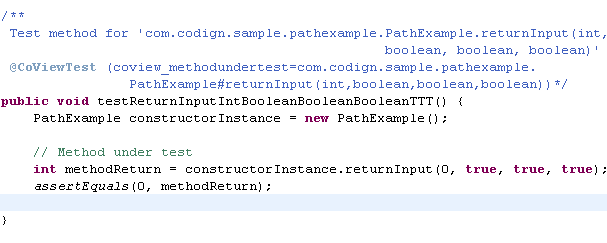
**Statement Coverage**

Statement coverage identifies which statements in a method or class have been executed. It is a simple metric to calculate, and a number of [open source products](http://www.java-source.net/open-source/code-coverage/emma) exist that measure this level of coverage. Ultimately, the benefit of statement coverage is its ability to identify which blocks of code have not been executed. The problem with statement coverage, however, is that it does not identify bugs that arise from the control flow constructs in your source code, such as compound conditions or consecutive switch labels. This means that you easily can get 100 percent coverage and still have glaring, uncaught bugs.

The following example demonstrates this. Here, the returnInput() method is made up of seven statements and has a simple requirement: *its output should equal its input*.

  
*Figure 1. Code sample*

Next, you can create one JUnit test case that satisfies the requirement and gets 100 percent statement coverage.

  
*Figure 2. Statement coverage*

There's an obvious bug in returnInput(). If the first or second decision evaluates true and the other evaluates false, the return value will not equal the method's input. An astute software developer will notice this right away, but the statement coverage report shows 100 percent coverage. If a manager sees 100 percent coverage, he or she may get a false sense of security, decide that testing is complete, and release the buggy code into production.

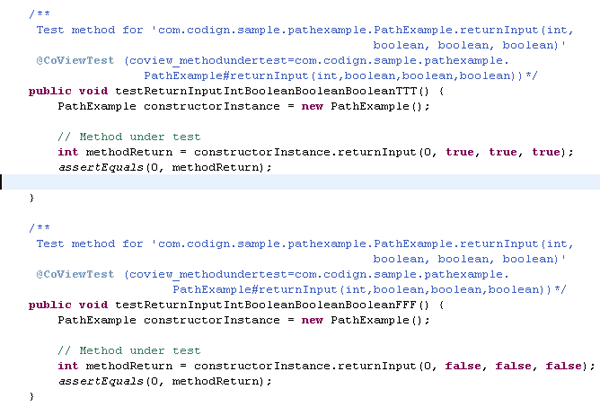
Recognizing that statement coverage may not fit the bill, the developer decides to move on to a better testing technique: branch coverage.

**Branch Coverage**

A branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested. This sounds great because it takes a more in-depth view of the source code than simple statement coverage, but branch coverage can also leave you wanting more.

Determining the number of branches in a method is easy. Boolean decisions obviously have two outcomes, true and false, whereas switches have one outcome for each case—and don't forget the default case! The total number of decision outcomes in a method is therefore equal to the number of branches that need to be covered plus the entry branch in the method (after all, even methods with straight line code have one branch).

In the example above, returnInput() has seven branches—three true, three false, and one invisible branch for the method entry. You can cover the six true and false branches with two test cases:

  
*Figure 3. Branch coverage*

|  |
| --- |
|  |

Both tests verify the requirement (output equals input) and they generate 100 percent branch coverage. *But, even with 100 percent branch coverage, the tests missed finding the bug.* And again, the manager may believe that testing is complete and that this method is ready for production.

A savvy developer recognizes that you're missing some of the possible paths through the method under test. The example above hasn't tested the *TRUE-FALSE-TRUE* or *FALSE-TRUE-TRUE* paths, and you can check those by adding two more tests.

There are only three decisions in this method, so testing all eight possible paths is easy. For methods that contain more decisions, though, the number of possible paths increases exponentially. For example, a method with only ten Boolean decisions has 1024 possible paths. Good luck with that one!

So, achieving 100 percent statement and 100 percent branch coverage may not be adequate, and testing every possible path exhaustively is probably not feasible for a complex method either. What's the alternative? Enter basis path coverage.

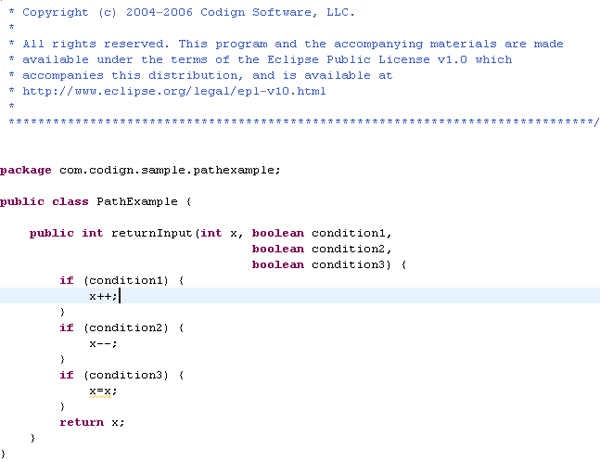
**Basis Path Coverage**

A path represents the flow of execution from the start of a method to its exit. A method with *N* decisions has *2^N* possible paths, and if the method contains a loop, it may have an infinite number of paths. Fortunately, you can use a metric called [cyclomatic complexity](http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html) to reduce the number of paths you need to test.

The cyclomatic complexity of a method is one plus the number of unique decisions in the method. Cyclomatic complexity helps you define the number of linearly independent paths, called the basis set, through a method. The definition of linear independence is beyond the scope of this article, but, in summary, the basis set is the smallest set of paths that can be combined to create every other possible path through a method.

Like branch coverage, testing the basis set of paths ensures that you test every decision outcome, but, unlike branch coverage, basis path coverage ensures that you test all decision outcomes *independently* of one another. In other words, each new basis path "flips" exactly one previously executed decision, leaving all other executed branches unchanged. This is the crucial factor that makes basis path coverage more robust than branch coverage, and allows you to see how changing that one decision affects the method's behavior.

I'll use the same example to demonstrate.

  
*Figure 4. Code sample*

To achieve 100 percent basis path coverage, you need to define your basis set. The cyclomatic complexity of this method is four (one plus the number of decisions), so you need to define four linearly independent paths. To do this, you pick an arbitrary first path as a baseline, and then flip decisions one at a time until you have your basis set.

**Path 1**: Any path will do for your baseline, so pick true for the decisions' outcomes (represented as *TTT*). This is the first path in your basis set.

**Path 2**: To find the next basis path, flip the first decision (only) in your baseline, giving you *FTT* for your desired decision outcomes.

**Path 3**: You flip the second decision in your baseline path, giving you *TFT* for your third basis path. In this case, the first baseline decision remains fixed with the true outcome.

**Path 4** : Finally, you flip the third decision in your baseline path, giving you *TTF* for your fourth basis path. In this case, the first baseline decision remains fixed with the true outcome.

So, your four basis paths are *TTT*, *FTT*, *TFT*, and *TTF*. Now, make up your tests and see what happens.

In the attached code, you can see that testReturnInputIntBooleanBooleanBooleanTFT() and testReturnInputIntBooleanBooleanBooleanFTT() found the bug that was missed by your statement and branch coverage efforts. Further, the number of basis paths grows linearly with the number of decisions, not exponentially, keeping the number of required tests on par with the number required to achieve full branch coverage. If fact, because basis path testing covers all statements and branches in a method, it effectively subsumes branch and statement coverage.

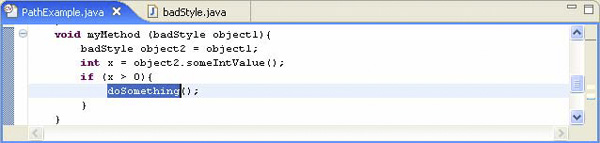
But, why didn't you test that the other potential paths? Remember, the goal of basis path testing is to test all decision outcomes independently of one another. Testing the four basis paths achieves this goal, making the other paths extraneous. If you had started with *FFF* as your baseline path, you'd wind up with the basis set of (*FFF*, *TFF*, *FTF*, *FFT*) making the *TTT* path extraneous. Both basis sets are equally valid, and either satisfies your independent decision outcome criterion.

**Creating Test Data**

Achieving 100 percent basis path coverage is easy in this example, but fully testing a basis set of paths in the real world will be more challenging, even impossible. Because basis path coverage tests the interaction between decisions in a method, you need to use test data that causes execution of a specific path, not just a single decision outcome, as is necessary with branch coverage. Injecting data to force execution down a specific path is difficult, but there are a few coding practices that you can keep in mind to make the testing process easier.

1. Keep your code simple. The lower the cyclomatic complexity of your methods, the better. Refactor when you can to lower your numbers. The higher the number, the harder it is to test, and numbers above 10 are essentially untestable. Not only does this reduce the number of basis paths that you need to test, but it reduces the number of decisions along each path.
2. Avoid duplicate decisions.
3. Avoid data dependencies.

Consider the following example:



*Figure 5. Basis path testing*

The variable *x* depends indirectly on the object1 parameter, but the intervening code makes it difficult to see the relationship. As a method grows more complex, it may be nearly impossible to see the relationship between the method's input and the decision expression.

**Summary**

Although statement and branch coverage metrics are easy to compute and achieve, both can leave critical defects undiscovered, giving developers and managers a false sense of security. Basis path coverage provides a more robust and comprehensive approach for uncovering these missed defects without exponentially increasing the number of tests required.

Return to [ONJava.com](http://www.onjava.com/).

Copyright © 2009 O'Reilly Media, Inc.