

## **Definitions and Benefits of Statement, Branch and Path Coverage**

May 7<sup>th</sup>, 2006



## 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 if 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% coverage. **While 100% coverage is an admirable goal, 100% of the wrong type of coverage can lead to problems.** A typical software development effort measures coverage in terms of the number of either statements or branches to be tested. Even with 100% statement or branch coverage, critical bugs may still be present in the logic of your code, leaving both developers and managers with a false sense of security.

How can 100% coverage be insufficient? Because statement and branch coverage does not tell you if the logic in your code was executed. Statement and branch coverage is 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 we discuss path coverage, let's 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 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 can easily get 100% coverage and still have glaring, uncaught bugs.

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

```

package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1, boolean condition2) {
        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        return x;
    }
}
    
```



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

```

/**
 * Test method for 'com.codign.sample.pathexample.PathExample.returnInput
 * @CoViewTest (coview methodundertest=com.codign.sample.pathexample.Path
 */
public void testReturnInputIntBooleanBoolean() {
    PathExample constructorInstance = new PathExample();

    // Method under test
    int methodReturn = constructorInstance.returnInput(5, true, true);
    assertEquals("input equals output",5, methodReturn);
}
    
```

There's an obvious bug in the returnInput() method. If one decision evaluates true and the other evaluates false, then the return value will not equal the method's input. An astute software developer will notice this right away, but the report of our statement coverage testing has led us to believe that we were done testing. If a manager sees 100% coverage on the daily coverage summary, the manager might decide that testing is complete and release this buggy code into production.

Recognizing that statement coverage doesn't fit the bill, our 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 us wanting more.

Determining the number of branches in a method is easy. Boolean decisions obviously have two outcomes, true and false, while 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.

In the example above, returnInput() has four branches – two true and two false. It looks like we can cover all four branches with two test cases:

```

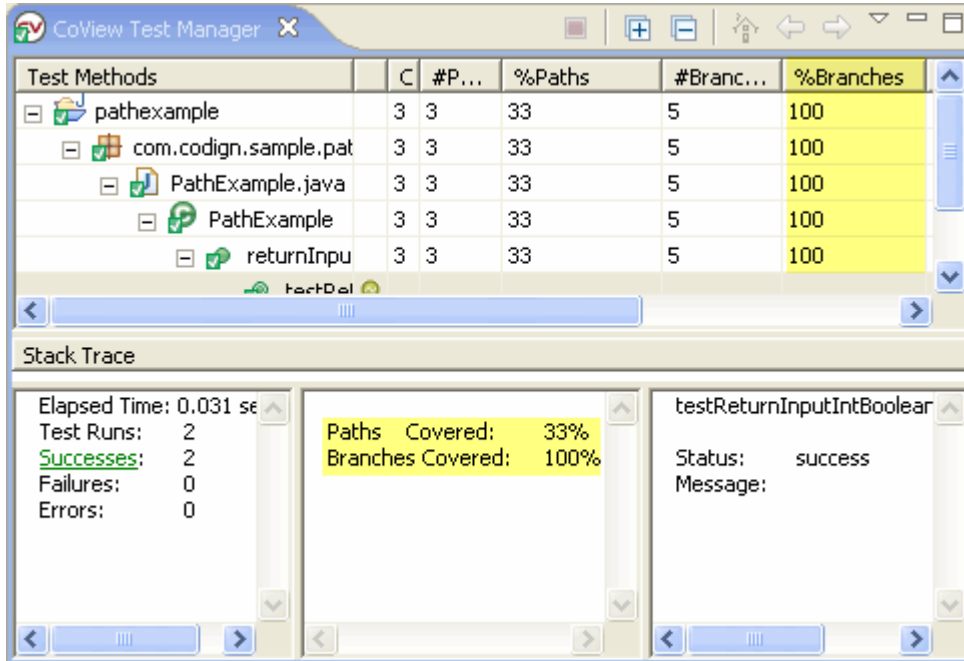
public void testReturnInputIntBooleanBoolean() {
    PathExample constructorInstance = new PathExample();
    // Method under test
    int methodReturn = constructorInstance.returnInput(5, true, true);
    assertEquals("input equals output",5, methodReturn);
}

Test method for 'com.codign.sample.pathexample.PathExample.returnInput(int, boole
public void testReturnInputIntBooleanBoolean__false() {
    PathExample constructorInstance = new PathExample();

    // Method under test
    int methodReturn = constructorInstance.returnInput(5, false, false);
    assertEquals("input equals output", 5, methodReturn);
}
    
```



Both tests verify our requirement (output equals input), and they generate 100% branch coverage results.



Even with 100% branch coverage, our tests missed finding the bug. And again, our developer's manager believes that testing is complete and that this method is ready for production.

Our savvy developer recognizes that we're missing some of the possible paths through the method under test. In the example above, we haven't tested the TRUE-FALSE or FALSE-TRUE paths, and we can check those by adding two more tests.

There are only two decisions in this method, so testing all four 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% statement and 100% branch coverage is obviously not 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, we can use a metric called cyclomatic complexity (<http://thediscoblog.com/?p=15>) to reduce the number of paths we need to test.

The cyclomatic complexity of a method is the number of unique decisions in the method plus one. Cyclomatic complexity helps us 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 paper, 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. See <http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/stoc.htm> for more information.

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 us to see how changing that one decision affects the method's behavior.

Let's use the same example to demonstrate.

```

package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1, boolean condition2) {
        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        return x;
    }
}
    
```

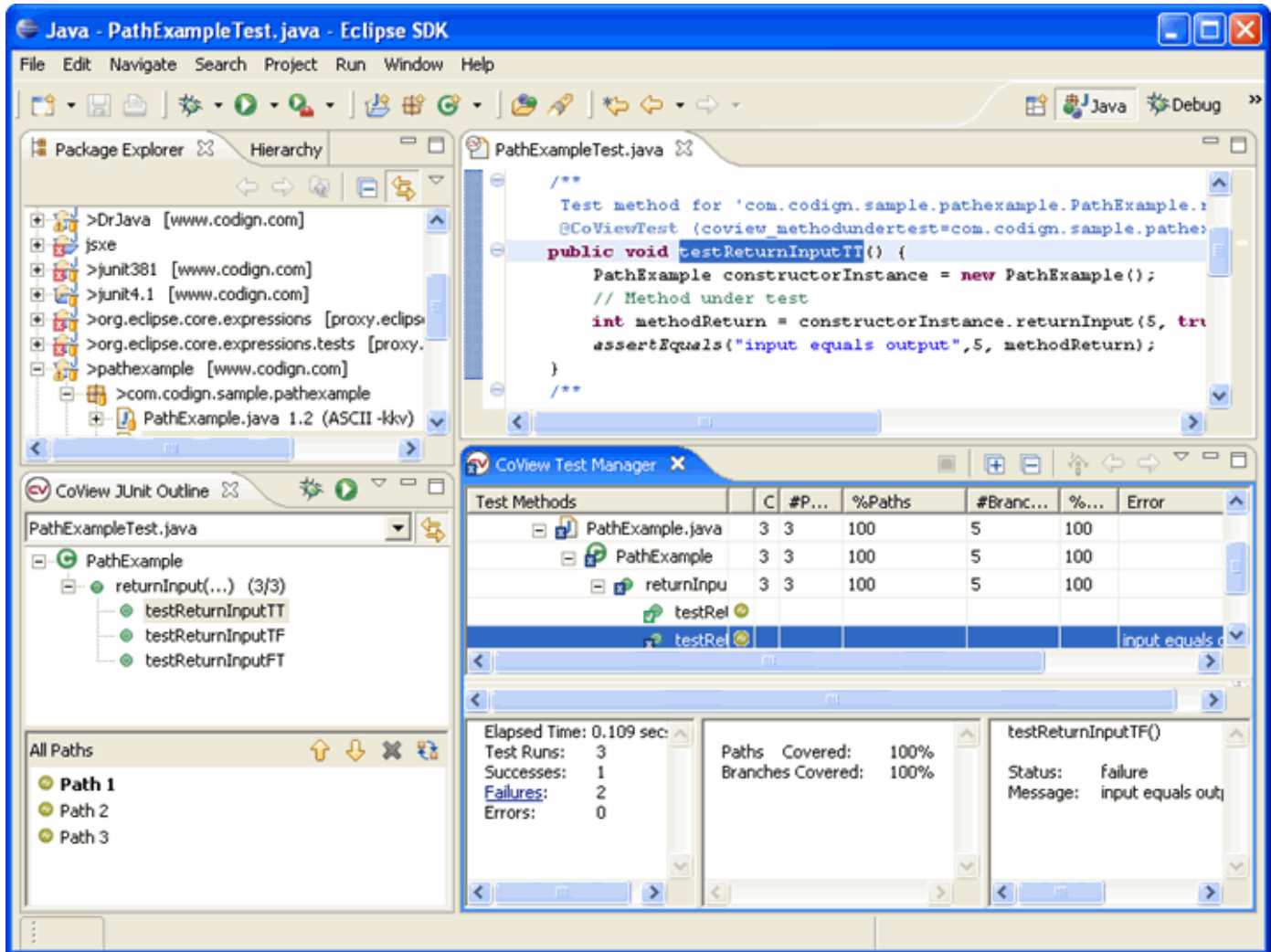
In order to achieve 100% basis path coverage, we need to define our basis set. The cyclomatic complexity of this method is three (two decisions plus one), so we need to define three linearly independent paths. To do this, we pick an arbitrary first path as a baseline, and then flip decisions one at a time until we have our basis set.

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

**Path 2:** To find the next basis path, we'll flip the first decision (only) in our baseline, giving us FT for our desired decision outcomes.

**Path 3:** Finally, we flip the second decision in our baseline path, giving us TF for our third basis path. In this case the first baseline decision remains fixed with the true outcome.

So our three basis paths are TT, FT, and TF. Let's make up our three tests and see what happens.



testReturnInputTF() and testReturnInputFT() found the bug that was missed by our 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. In fact, since basis path testing covers all statements and branches in a method, it effectively subsumes branch and statement coverage.

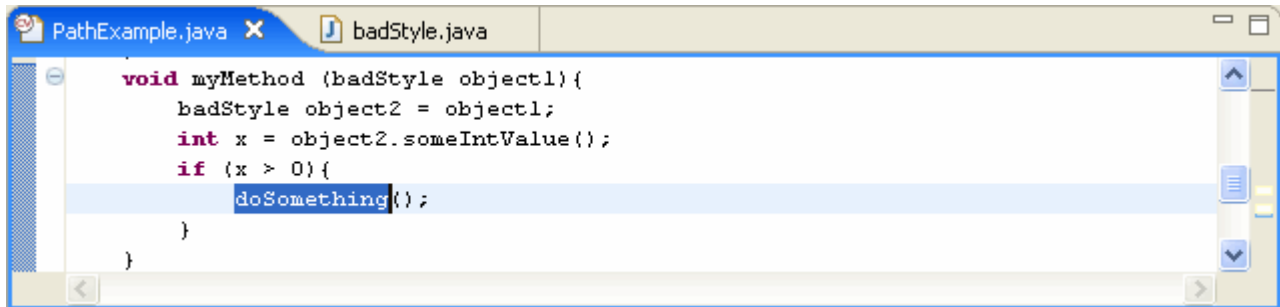
But why didn't we test that fourth path (FF)? Remember, the goal of basis path testing is to test all decision outcomes independently of one another. Testing the three basis paths achieves this goal, making the FF path extraneous. If you had started with FF as your baseline path, you'd wind up with the basis set of (FF, TF, FT), making the TT path extraneous. Both basis sets are equally valid, and either satisfies our independent decision outcome criterion.

### Creating Test Data

Achieving 100% basis path coverage is easy in this example, but fully testing a basis set of paths in the real world may prove 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.

- ✓ Keep your code simple. Avoid methods with cyclomatic complexity greater than ten. 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.

- ✓ Avoid duplicate decisions.
- ✓ Avoid data dependencies. Consider the following example:



```
PathExample.java x badStyle.java
void myMethod (badStyle object1){
    badStyle object2 = object1;
    int x = object2.someIntValue();
    if (x > 0){
        doSomething();
    }
}
```

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.

