# CODE COVERAGE ISN'T QUALITY, IT ISN'T EVEN COVERAGE

**Wayne Roseberry**

wayner@microsoft.com

## Abstract

If anybody ever says "I know our code has high quality because we achieved 100% code coverage," smack them. There really is no relationship between code coverage and quality because code coverage does a dreadful job telling you whether or not tests have been thorough.

Good test coverage only comes from doing as many interesting things as possible that are good at exposing flaws in code. Code coverage reports, when used as a metric of quality, hide the useful tests.

This paper shows real examples of code coverage reports that showed 100% coverage that were completely, and entirely useless. Alternate tests will be presented that demonstrate how sometimes directing one's attention in the opposite direction implied by code coverage actually yields better test generation and more confidence in what the test suite addresses.

## Biography

Wayne Roseberry is a Principal Design Engineer in Test at Microsoft Corporation, where he has been working since June of 1990. His software testing experience ranges from the first release of The Microsoft Network (MSN), Microsoft Commercial Internet Services, Site Server, and all versions of SharePoint. Previous to testing, Wayne also worked in Microsoft Product Support Services, assisting customers of Microsoft Office.

Previous to working for Microsoft, Wayne did contract work as a software illustrator for Shopware Educational Systems.

In his spare time, Wayne writes, illustrates and self-publishes children's literature.

# 1 INTRODUCTION

Code coverage tools are one of the more useful devices in a tester's handbag. Knowing which blocks have been covered, or not covered, by a set of tests is very informative. This has led some people to use code coverage as a measure of quality in the product and test. The practice is simple. Run some tests, look at the coverage report. Modify the tests to hit blocks not covered and repeat until the code coverage numbers are high enough percentage to feel good.

The problem with this approach is that code coverage numbers don't tell you if testing was good. In fact, the very nature of code coverage reporting is such that it hides, obscures and misleads your impressions. You are likely led to believe that since a large percentage of blocks were hit during test that you have done a pretty good job looking for bugs, or that the equivalent set of tests when run again will do a thorough job of discovering regressions. You will likely be wrong in that belief.

The fact is code coverage reports miss several classes of bugs, and obscure the need for important and sometimes large sets of test. The proper way to use code coverage reporting to improve test coverage is not as a metric of quality, but instead as an inspiration for discovering what tests ought to be introduced.

My intent is to share some stories that demonstrate these points. The stories are real; the examples are real-world. I took the liberty to modify the code examples from the original, partly because the code is proprietary and partly for readability in the document, but the story is still the same as it played out in real life. In both stories, somebody had made a decision based on blindly following the numbers from a code coverage report and did the wrong thing with regard to tests.

# 2 CODE COVERAGE & TEST COVERAGE

Let's talk about what we mean by code coverage and specifically test coverage. The definition is important, because it helps us clarify our goals and intent.

**What is meant by code coverage?**

By "code coverage", I specifically refer to tools that can show exactly which parts of the product code, expressed as source code, were hit during a test run. The most common form of code coverage is referred to as "block coverage," which is discreet pieces of source code that have a single entry and a single exit point and no branches. For example, the following code…

```
if (a == true)
{
   return 1;
}
else
{
   return 0;
}
```

has three blocks. The first block is the `if` statement, the second block is the one that returns 1 if the statement resolves to true, and the third block is the one that returns 0 if the statement resolves to false. Assuming somebody were to test this code in the case where the variable a was true, that test would achieve 66% code coverage, two out of three blocks were hit by the test.

There are other ways to describe code coverage, path coverage, traversal coverage, line coverage. For sake of this document we are going to talk about block coverage as described above. The same principles generally apply across all forms.

**Classical Approach to Code Coverage**

The typical way to treat code coverage is to do the following:

1. Enable a code coverage tool, sometimes by instrumenting a build, sometimes not
2. Execute a series of tests
3. Look at the Percentage of blocks covered
4. Do a white-box inspection of the code to figure out how the blocks were missed
5. Continue building/executing new tests until the percentage of blocks covered reaches a desired goal (e.g. 80%, 100%, etc.)

Examples of this approach can be found in previous PNSQC sessions (Karwa & Panda). Teams have had varying success and value out of achieving different goals, with interesting results on diminishing returns going after the last few percentage points (Manu, Najpal, Amalo & Tan).

**What do we mean by test coverage?**

For sake of this document, I will use a very simplified definition of test coverage. Test coverage is a list of things that have been done to discover flaws in a product. When we talk about test coverage as a percentage, we are talking about the number of things that have been done divided by the number of things that ought to be done to discover flaws in a product. In this definition, we use the phrase "ought to be done" to acknowledge that the number of things which could be done is almost always impossible or impractical and that to accommodate we have already applied our best judgment to pick the ones which we believe increase the odds of finding the flaws we desire most to remove from the product.

Tests are typically generated by enumerating the product behaviors and exercising them under different conditions, with different inputs and in different sequences.

This simple definition makes it easy to draw a comparison between code coverage and test coverage. Code coverage describes the inner structures which control the product behaviors. It is a way of saying "here are all the blocks of code in our system," which ought to generate the question "what are the different tests we should make based on this list of blocks?" As I will demonstrate later, sometimes we answer that question the wrong way. All too often, the answer is "I need to do something to hit that block of code," when the real answer should be "What tests will find problems because of that block of code?"

**How does "coverage" relate to "quality"?**

Bypassing the debate on what we mean by quality, let's just say that quality is "the degree to which the intended customer thinks the product is good". I settle on that definition because it acknowledges the subjective nature of quality. I also chose it because it demonstrates that all our engineering metrics cannot truly identify quality. All they can do is attempt a prediction of the customer's opinion of the quality of the product. Better metrics help us make better predictions, and poor metrics, or poor use of metrics lead us to make bad predictions. Better coverage metrics usually the results of a rich set of test activities that are designed to effectively and efficiently uncover the flaws in the product that will reduce quality as perceived by the customer. When a rich set of tests are well executed and passing at a reasonably high rate, then we are more able to make an accurate prediction that the product is of a quality suitable for the customer's needs.

This is the point where people get into trouble with code coverage metrics.

# 3 MISTAKES, RULES AND PRINCIPLES

**Two mistakes**

This document is going to focus on two main mistakes that people make when dealing with code coverage numbers. The first is assuming that code coverage is an indication of product quality. The argument is that if a large percentage of blocks are hit by a given set of tests, and assuming those tests pass, then the product must be of high quality. In fact, code coverage percentage has a very weak relationship to product quality, several examples of which will be given later.

The next mistake derives from the first, that being to chase the code coverage numbers by using test activities that drive the numbers higher. This leads people to often indiscriminately take the easiest, most efficient or most expedient route to achieving those numbers. The problem with this approach is that the activity which is the most practical way to increase the code coverage numbers is very often a poor way of discovering flaws in a product. Examples will be given later in this document.

**The Rules**

I like to reduce the rules regarding interpretation of code coverage results to the following two simple statements:

- Missing Blocks in Code Coverage == Badness
- Hitting Blocks in Code Coverage != Goodness

It is very important to understand the subtle implications from these rules. The only qualitative statement that code coverage reports can state about a given test suite is whether or not tests have missed a block of code. We assume that all missed blocks of code carry some degree of badness because we desire, if possible, to cover every block we can. Previous works make the case for exactly how much coverage is worth achieving, typically calling the practical limit somewhere around 80% before results diminish dramatically, but it is at least logically sound to say that a missed block is something to investigate because we generally do not want to see that.

However, a code coverage report does not indicate whether or not testing is sufficient, or even good. In fact, a portion of code could show as 100% covered even if the test that achieved that end accomplished nothing useful with regard to finding flaws in the product. It is therefore unsafe to decide that since a code coverage report indicates high percentages that the quality of coverage must likewise be good.

This means that code coverage reports should not be used to assess product quality. The only thing they can measure is a negative, whether or not there is a block of code not covered yet. This is useful, but it should not be used for special means. We will talk about those later.

**Metaphors:**

- Assessing quality of code from a code coverage report is like assessing quality of food in a restaurant by counting the number of employees who show up for work when scheduled
- People behavior according to measurements: My uncle Jerry used to sit on his seat belt with it buckled underneath him to avoid the warning buzzer.

# 4 EXAMPLE 1: CHASING THE NUMBERS

My first example came while covering part of the list editing and management features in SharePoint Foundation. A tester working for me was using code coverage reports to choose how to improve the existing set of automated tests. The report showed some blocks not covered, so based on that he had decided to target those blocks. I asked him to show me an example of what he meant. The code he showed me looked something like the following:

```
private object returnfielddata(fieldobject fld)
  {
    switch (fld.type)
    {
      Case FLD_STRING:
        return ExtractStringFromField(fld);
      Case FLD_DATE:
        return ExtractDateFromField(fld);
      Else
        return null;
    }
  }
```

The code coverage report said that this function had 75% coverage. It divides into four basic blocks, shown below:

| | |
|---|---|
| Covered | `switch (fld.type)`<br>`{` |
| Covered | `Case FLD_STRING:`<br>`  return ExtractStringFromField(fld);` |
| Covered | `Case FLD_DATE:`<br>`  return ExtractDateFromField(fld);` |
| Not Covered | `Else`<br>`  return null;`<br>`}` |

In the code above, the variable `fld` is a class of type `fieldobject`. The `Type` property of this class is an enumerator of which there are around a dozen potential values (e.g. `FLD_INTEGER, FLD_FLOAT, FLD_CURRENCY`, etc.). The function is a private function, meaning only the application itself would be able to call the code.

I asked the tester what he intended to do. His response was the following:

1. Build his own version of the application in which he would put in a hook to write his own test code

2. Write a test method that would call the function directly, setting `fld.type` to some value other than `FLD_STRING or FLD_DATE`

He was seeking no other goal to than to ensure the code coverage report indicated 100% coverage. As far as he was concerned, the uncovered block, on its own, was interesting enough to merit the extra effort of writing special test hooks into the product just to allow test code to specifically call the function.

Let's look at that line of reasoning. Why is it so interesting to cover that block of code specifically? It is clear and obvious exactly what is going to happen if `fld.type` equals some value other than `FLD_STRING` or `FLD_DATE`. The function will return null. There is no other possibility. It is not like the C compiler is going to suddenly behave differently than it has for decades and cause a switch statement to evaluate differently. Testing the function in this way has no value or purpose. Yet, by assuming that the

goal of producing a code coverage report was to drive the coverage numbers as high as possible, that is precisely the kind of test this engineer was proposing. The tester was allowing the metrics to mislead him.

**Something Far More Important**

When I saw this uncovered block of code, I had a completely different set of concerns:

1. *Any value other than* `FLD_STRING` *and* `FLD_DATE` *were treated as an equivalence class by the code.*

   One of the biggest sources of bugs in product code, and something completely missed by code coverage reports is missing logic. What if one of the other possible values for `fld.type` needed to be addressed in the switch statement?

   Making it even more complicated, more than one test would hit exactly the same code block. Assume, for a moment that `FLD_INTEGER` should return null, `but FLD_COUNTRYCODE` should not. If during test, only the value `FLD_INTEGER` were used, the code coverage report would show the block had been hit, yet we would be no closer to discovering that the code does not properly handle `FLD_COUNTRYCODE` because from a pure code block perspective the two cases are equivalence classes, when in fact they should not be.

2. *Is* `null` *an appropriate return value for the other ranges on* `fld.type`*?*

   Again, code coverage will not tell you if the behavior in the code is correct, only if it has been executed or not. Perhaps `null` should be replaced with some other value.

3. *Our tests had never tried any values beyond* `FLD_STRING` *and* `FLD_DATE`

   As stated above, the enumerator data type for `fld.type` has about a dozen or so possible values. If the code coverage report indicated the final `return null;` block had never been hit, then that means the tests had not included any of those other values for `fld.type`. This says not so much about the function coverage itself as it says about coverage of the rest of the product code. The actual code coverage report is a hint that something is wrong, but the missed block is not the appropriate target of our attentions. The missing block is telling us there is something wrong somewhere else.

4. *The consuming code of the function "returnfielddata" had never processed a return value of null when under test*

   Imagine that the code calling this function looked like the following:

   ```
   fielddata fd = (fielddata) returnfielddata(field);
   addFieldToDocument (fd.value);
   ```

   We don't need to know what `fielddata` is, or what `addFieldToDocument` does to recognize that if returnfielddata returns a null object into `fd` that the next statement is going to fault when trying to access the `Value` property on `fd`.

5. *The switch statement where uncovered block was an else clause on an enumeration, and it returned a NULL*

   **PRINCIPLES:**

   - The uncovered block isn't interesting in itself

- Taking gyrations to run the block in isolation (e.g. special build to make the internal API callable and code block easily hittable)
- REAL PROBLEM: The test code isn't trying all value possibilities in the enumeration
- REAL PROBLEM: Hitting only one value would give illusion of 100% coverage
- REAL PROBLEM: The calling code has never been tested with the value returned in the else block – e.g. what if return was `null` and the caller immediately tries to use `result.member`

# 5 EXAMPLE 2: GOOD ISN'T GOOD

I experienced a similar issue several months later, around the same feature set. I assigned one of my testers a piece of code that was used for building queries to retrieve items from the database. The previous owner had written automation and achieved 100% coverage of a particular class. I had been told by the previous feature owner the coverage was in pretty good shape because of the numbers on the report. I asked the tester to evaluate the test automation to ensure it was good. Several days later, she came back to tell me she had found some substantial problems, in spite of the 100% coverage stated in the report.

The feature in question was for a class called SPQuery. It is mostly a container that is used to store an XML description of which items to retrieve. There are additional properties on the class that indicate how the retrieval should behave. On its own, the SPQuery class does not do anything. After the SPQuery object is constructed and its arguments set, it is passed to a method called "GetItems" that converts the SPQuery object into an SQL database query and performs the actual fetch. A typical pattern for using this class would look as follows:

```
// instantiate the object, set its properties
SPQuery qry = new SPQuery();
qry.Query = queryXMLString; // assume this was set prior…
qry.DatesInUTC = true; // we want date fields to be in UTC format
qry.AutoHyperlink = true; // we want HTML format anchor tags

// fetch the items from our list by passing in our query object
SPListItemCollection items = splist.GetItems(qry);

// once here, the code will iterate through results in items object
…
```

The tester assigned to the project pointed out to me that the high coverage numbers had been achieved simply by instantiating the class and checking that the properties were being set properly. Here is an example of the sort of test code she found:

```
// instantiate the object, set its properties
SPQuery qry = new SPQuery();
qry.Query = queryXMLString // assume this was set prior…;
if (qry.Query == queryXMLString)
{
   Log.Pass("Query matched expected value");
}
else
{
   Log.Fail("Query did not match expected value");
}
```

Likewise:

```
// instantiate the object, set its properties
SPQuery qry = new SPQuery();
qry.DatesInUTC = true; // assume this was set prior…
if (qry.DatesInUTC)
{
   Log.Pass("DatesInUTC set as expected");
}
else
{
   Log.Fail("DatesInUTC not set as expected");
}
```

By doing the above, the previous tester had very quickly and very efficiently achieved 100% code coverage of the entire SPQuery class. They had also accomplished very little. For example, in the above property `DatesInUTC`, the product code looked something like this:

```
public bool DatesInUTC()
{
   get {return m_DatesInUTC;}
   set {m_DatesInUTC = value;}
}
```

All that happens when the property is either assigned or retrieved is it is put into, or read from, an in memory member property of the class. This is an exceptionally uninteresting thing to test. There are no transformations, there is no parsing, the data does not get passed to another process or method during either assignment or retrieval, and the behavior does not change based on any condition or state.

The tests which had achieved 100% code coverage on this class were almost entirely useless. They were doing no physical harm, although they were doing a bit of psychological harm by creating the illusion that the class had been well tested and covered.

Fortunately, the tester I had asked to analyze the code realized this, and came up with the following conclusions:

1. **Using Wrong Test Pattern**

   The automation was not exercising the expected pattern for this class. The SPQuery class is a primary input to the `GetItems` method, and therefore it was really only interesting to test it by setting the class properties and subsequently calling `GetItems()`.

2. **Missing Properties That Drive Behavior**

   The automation was not testing how different properties on the SPQuery object affected behavior when subsequently calling `GetItems`. The `DatesInUTC` test stated above is a good example. The property affects whether the data returned from `GetItems` will come back in the format of a URL (e.g. http://www.contoso.com) or anchor tag (e.g. "<a href="http://www.contoso.com">http://www.contoso.com</a>"). In the very least, there should have been two tests for `DatesInUTC`, one where it was set true, and one where it was set false. Further, assume that the developer had forgotten to format results as an anchor tag and always returned hyperlinks in their raw format. A code coverage report of such a mistake would likely show the block completely covered, as such bugs are missing code blocks, something code coverage is incapable of detecting.

3. **Missing Large, Complex Existing Data Domain**

The SPQuery and `GetItems` methods return items from lists of data with customized column data types (string, date, integer, people, lookups on other columns, hyperlinks, etc.). Furthermore, those lists can have different settings that affect retrieval behavior (do they contain subfolders, do they items have multiple versions, security settings on this list, etc.). These different data domains represent a large number of conditions and states that affect the behavior of the `GetItems` method, and integrate very tightly with the state of the SPQuery object when `GetItems` is called. The test automation was missing tests that populated lists with different values, field types and settings.

4. **Missing Large, Complex Format and Behavior Domain**

   The SPQuery class takes as its primary property a string that describes in XML the items that ought to be retrieved from the list. This XML query is transformed into an SQL database query. The XML syntax for the query is moderately complex; the generated SQL query is extremely complex. The SQL query generation and execution is one of the most complicated features in the product overall, making the development team very nervous about fixing any bugs. Regressions in this feature are notoriously difficult to find, understand and fix. Yet, despite the complexity of this problem, the test automation was not trying variations on the XML query format.

   One of the riskiest and most difficult portions of the code was being virtually unexercised because a code coverage report had said a feature was 100% covered. One important point to note is that a large part of the code logic happens inside the generated SQL, something the code coverage tools we were using was unable to inspect. With regard to different XML string, as far as the code coverage was concerned they were all equivalent data, because the get and set methods on the Query property did so little.

Working together, the tester and I drew the following conclusions for what to do:

1. Enhance the test tools to more easily generate lists with the different range of data types
2. Build data generation tools that populated lists with interesting test data ranges
3. Build query XML generating tools that could build thousands of different queries in different structures
4. Build a simple query testing validation mechanism that combined the three components above to easily cover lots of different lists, with lots of different data and settings with lots of different types of queries

# 6 WHAT SHOULD WE DO?

So, what is our call action? If code coverage reports are a poor assessment of quality, and if using them to chase the numbers and achieve high coverage percentages the most efficient, easy way possible leads to insufficient and sometimes misleading results, then what should we do instead?

Fortunately there is an answer, and perhaps also unfortunately it isn't a simple, easy-as-pie thing to do. Like most aspects of software test engineering, the right thing to do is hard and requires skill and experience. But then again, that is largely what makes the job as rewarding as it is. Here are the recommendations that I would extend based on the experiences I have described so far:

- **Use code coverage reports like a probe**

  Code coverage reports are an essential tool. They help you see something that is otherwise unavailable to you. If feasible, run them as often as you can with the narrowest granularity possible. In a lot of ways, code coverage reports are like the gas gauge on a car, or the

thermometer outside. The gas gauge will not tell you if the trip you are driving on is successful no more than the thermometer will tell you if you are having a good day or a bad day. But both are critical tools for driving somewhere and having a picnic (or a snowball fight) respectively. Check the code coverage reports periodically and use them to help guide adjustments in your test approach.

To complement this, if you are using code coverage reports as a means of measuring product quality, replace it with something else, such as bug open and fix rates, customer feedback and incident reports, reliability reports and test case execution pass fail rates. Whatever it is you use, you want it to have more success predicting impact on the customer.

- **Consider every missed block bad**

How bad a missed block might be is up to you, but your default disposition ought to be that a missed block of code is innately undesirable. Maybe the best way to address this would be to state the report the opposite way we are used to. Most of the time, we state code coverage in terms of percent covered (e.g. “We achieved 75% coverage!”). Perhaps it is better to state in terms of percent uncovered (e.g. “We still have 45% of our code blocks uncovered in testing”).

Do not do the opposite and consider every covered block good. To that, see the next point.

- **Ask “What does this mean?” rather than “How do I hit this block?”**

Code coverage reports should more than anything else inspire imagination and new ideas. They help you understand things that are not happening, and it is in those gaps where we sometimes learn the most important things.

Ask two questions: 1> “What are the tests doing that hit the covered blocks?”, and 2> “What do the uncovered blocks imply regarding missing tests?” For each of these, force yourself to ignore the opportunity to “just make sure the block gets hit” and instead try to think of the following:

**Covered blocks:**

- Create Strong tests that cover many domains and protect the code from regression
- Identify and remove weak tests that achieve superficial depth and are poor at discovering bugs
- Target tests that may be selected when specific portions of the code change
- Develop opportunities to borrow between tests for things like shared data, configuration, and system state, sequences of operations, etc.

**Uncovered blocks:**

- Identify weak investments in larger test domains, such as heavily data dependent, system state dependent
- Call attention to weakly covered features that directly or indirectly integrate with the missing block and do not handle variations in data, return values, error states and faults or other implications of the missed block
- Correct or replace test patterns that do not exercise the product appropriately to force integration of components or usage the way intended
- Seek opportunities for tests in usage scenarios

All of the above may sound easier said than done, because in fact they are. This is an activity that improves with experience. It is the sort of thing where more senior engineers and leads ought to mentor others in how to come up with better ways to use the code coverage information.

# 7 CONCLUSION

Quality metrics are an artifact of our constant search for ways to manage our schedules and predict how well a product will do in the hands of a customer. Sometimes we come across a useful tool that makes us do our job better and surrender to the temptation to turn that tool into a quality metric. Sometimes that works really well, guiding us to more successful projects with better product, and sometimes it introduces problems that hinder us more than it helps. Code coverage reports are that sort of useful tool. They are an essential part of every test engineer's toolkit, but as demonstrated in the stories shared in this document, when used as a metric of product quality the unintended side effects are shortcuts and misleading coverage. For sake of better software engineering, code coverage reports are best treated as a tool carried in the engineer tool belt and not plastered on the wall of the executive boardroom.

## References

Manu, C., Najpal, P., Amalo, D. & Tan, R.P. “Code Coverage Case Study: Covering the Last 9%”. Paper presented at the Pacific Northwest Software Quality Conference, Portland, Oregon, October 2010.

Karwa, S. & Panda, S. “Streamlining test Automation using White-box Driven Approach”. Paper presented at the Pacific Northwest Software Quality Conference, Portland, Oregon, October 2010.

Microsoft. “SPQuery Class” Published 2010 http://msdn.microsoft.com/en-us/library/microsoft.sharepoint.spquery.aspx