In pursuit of code quality: Monitoring cyclomatic complexity

What to do when code complexity is off the charts

Andrew Glover

March 28, 2006

If complexity has been shown to correlate to defects, doesn't it make sense to monitor your code base's complexity values? Andrew Glover shows you how to use simple code metrics and Java™-based tools to monitor cyclomatic complexity.

View more content in this series

Every developer has an opinion about what code quality means and most have ideas about how to spot poorly written code. Even the term code smell has entered the collective vocabulary as a way to describe code in need of improvement.

Cyclomatic what?

For questions related to this article or anything on the subject of code quality, visit the Improve your Java Code Quality discussion forum, moderated by Andrew Glover.

One code smell that usually divides straight down the line with developers, interestingly, is the smell of too many code comments. Some claim judicious code commenting is a good thing, while others claim it only serves as a mechanism to explain overly complex code. Clearly, Javadocs™ serve a useful purpose, but how many inline comments are adequate to maintain code? If the code is written well enough, shouldn't it explain itself?

What this tells us about code smell as a mechanism for evaluating code is that it's subjective. What I might deem as terribly smelly code could be the finest piece of work someone else has ever written. Do the following phrases sound familiar?

Sure, it's a bit confusing (at first), but look how extensible it is!!

or

It's confusing to you because you obviously don't understand patterns.
What we need is a means to objectively evaluate code quality, something that can tell us, definitively, that the code we're looking at is risky. Believe it or not, that something exists! The mechanisms for objectively evaluating code quality have been around for quite a while, it's just that most developers ignore them. They're called code metrics.

**A history of code metrics**

Decades ago, a few super smart people began studying code hoping to define a system of measurements that could correlate to defects. This was an interesting proposition: by studying patterns in buggy code, they hoped to create formal models that could then be evaluated to catch defects before they became defects.

Somewhere along the way, some other super smart people also decided to see if, by studying code, they could measure developer productivity. The classic metric of lines of code per developer seemed fair enough on the surface:

> Joe produces more code than Bill; therefore, Joe is more productive and worth every penny we pay him. Plus, I noticed Bill hangs out at the water cooler a lot. I think we should fire Bill.

But this productivity metric was a spectacular disappointment in practice, mostly because it was easily abused. Some code measurement included in-line comments, and the metric actually favored cut-and-paste style development.

> Joe wrote a lot of defects! Every other defect is assigned to him. It's too bad we fired Bill -- his code is practically defect free.

Predictably, the productivity studies proved wildly inaccurate, but not before the metrics were widely used by a management body eager to account for the value of each individual's abilities. The bitter reaction from the developer community was justifiable, and for some, the hard feelings have never really gone away.

**Diamonds in the rough**

Despite these failures, there were some gems in those complexity-to-defect correlation studies. Most developers have long since forgotten them, but for those who go digging -- especially if you're digging in pursuit of code quality -- there is value to be found in applying them today. For example, have you ever noticed that long methods are sometimes hard to follow? Ever had trouble understanding the logic in an excessively deep nested conditional? Your instinct for eschewing such code is correct. Long methods and methods with a high number of paths are hard to understand and, interestingly, tend to correlate to defects.

I'll use some examples to show you what I mean.

**A sea of numbers**

Studies have shown that the average person has the capacity to handle about seven pieces of data in her or his head, plus or minus two. That is why most people can easily remember phone
numbers but have a more difficult time memorizing credit card numbers, launch sequences, and other number sequences higher than seven.

This principle also applies to understanding code. You've probably seen a snippet of code like the one in Listing 1 before:

### Listing 1. Numbers at work

```java
if (entityImplVO != null) {
    List actions = entityImplVO.getEntities();
    if (actions == null) {
        actions = new ArrayList();
    }
    Iterator enItr = actions.iterator();
    while (enItr.hasNext()) {
        entityResultValueObject arVO = (entityResultValueObject) enItr.
        .next();
        Float entityResult = arVO.getActionResultID();
        if (assocPersonEventList.contains(actionResult)) {
            assocPersonFlag = true;
        }
        if (arVL.getByName(AppConstants.ENTITY_RESULT_DENIAL_OF_SERVICE)
            .getID().equals(entityResult)) {
            if (actionBasisId.equals(actionImplVO.getActionBasisID())) {
                assocFlag = true;
            }
        }
        if (arVL.getByName(AppConstants.ENTITY_RESULT_INVOL_SERVICE)
            .getID().equals(entityResult)) {
            if (!reasonId.equals(arVO.getStatusReasonID())) {
                assocFlag = true;
            }
        }
    }else{  
        entityImplVO = oldEntityImplVO;
    }
```

Listing 1 shows up to nine different paths. The snippet is actually part of a 350-plus-line method that was shown to have 41 distinct paths. Imagine if you were tasked to modify this method for the purpose of adding a new feature. If you didn't write the method, do you think you could make the requisite changes without introducing a defect?

Of course, you'd write a test case, but do you think your test case could isolate your particular change in that sea of conditionals?

### Measuring path complexity

*Cyclomatic complexity*, pioneered during those studies I previously mentioned, precisely measures path complexity. By counting the distinct paths through a method, this integer-based metric aptly depicts method complexity. In fact, various studies over the years have determined that methods having a cyclomatic complexity (or CC) greater than 10 have a higher risk of defects. Because CC represents the paths through a method, this is an excellent number for determining how many test cases will be required to reach 100 percent coverage of a method. For example, the following code (which you might remember from the first article in this series) includes a logical defect:
Listing 2. PathCoverage has a defect!

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

In response, I can write one test, which achieves 100 percent line coverage:

Listing 3. One test yields full coverage!

```java
import junit.framework.TestCase;

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

Next, I run a code coverage tool, such as Cobertura, and get the report shown in Figure 1:

Figure 1. Cobertura reports

Well, that's disappointing. The code coverage report indicates 100 percent coverage; however, we know this is misleading.

Two for two

Note that the `pathExample()` method in Listing 2 has a CC of 2 (one for the default path and one for the `if` path). Using CC as a more precise gauge of coverage implies a second test case is
required. In this case, it would be the path taken by not going into the if condition, as shown by the testPathExampleFalse() method in Listing 4:

Listing 4. Down the path less taken

```java
import junit.framework.TestCase;

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

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

As you can see, running this new test case yields a nasty NullPointerException. What's interesting here is that we were able to spot this defect using cyclomatic complexity rather than code coverage. Code coverage indicated we were done after one test case, but CC forced us to write an additional one. Not bad, eh?

Luckily, the method under test in this case only had a CC of 2. Imagine if that defect were buried in a method with a CC of 102. Good luck finding it!

CC on the charts

A few open source tools available to Java developers can report on cyclomatic complexity. One such tool is JavaNCSS, which determines the length of methods and classes by examining Java source files. What's more, this tool also gathers the cyclomatic complexity of every method in a code base. By configuring JavaNCSS either through its Ant task or through a Maven plug-in, you can generate an XML report that lists the following:

- The total number of classes, methods, noncommenting lines of code, and varying comment styles in each package.
- The total number of noncommenting lines of code, methods, inner classes, and Javadoc comments in each class.
- The total number of noncommenting lines of code and the cyclomatic complexity per each method in the code base.

The tool ships with a few stylesheets that you can use to generate an HTML report summarizing the data. For example, Figure 2 demonstrates the HTML report that Maven generates:
This report's section labeled *Top 30 functions containing the most NCSS* details the largest methods in the code base, which incidentally *almost always* correlate to methods containing the highest cyclomatic complexity. For instance, the report lists the class `DBInsertQueue`'s `updatePCensus()` method as having a noncommenting line count of 283 and a cyclomatic complexity (labeled as CCN) of 114.

As demonstrated above, cyclomatic complexity is a good indicator of code complexity; moreover, it's an excellent barometer for developer testing. A good rule of thumb is to create a number of test cases equal to the cyclomatic complexity value of the code being tested. In the case of the `updatePCensus()` method seen in Figure 2, you would need 114 test cases to achieve full coverage.

**Divide and conquer**

When faced with a report indicating high cyclomatic complexity values, the first course of action is to verify the existence of any corresponding tests. If there are any tests, how many are there? All but the most rare code base would actually have 114 test cases for the `updatePCensus()` method (in fact, writing that many test cases for a method could take quite a long time). But even having a few is a great start toward reducing the method's risk for having a defect.

If there aren't any associated test cases, you obviously need to test the method. Your first thought could be that it's time to refactor, but doing so would break the first rule of refactoring, which is to write a test case. Writing test cases first lowers your risk in refactoring. The most effective way to reduce cyclomatic complexity is to pull out portions of code and place them into new methods. This pushes the complexity into smaller, more manageable (and therefore more testable) methods. Of course, you should then test those smaller methods.

In a continuous-integration environment, it's possible to evaluate the method's complexity *over time*. Having run the report for the first time, you can monitor the method's complexity value or any associated growth. If you see a growth in CC, you can take appropriate action.

If a method's CC value keeps growing, you have a couple of response options:
• Ensure a healthy amount of related tests are present to reduce risk.
• Evaluate the possibility of refactoring the method to reduce any long-term maintenance issues.

Also note that JavaNCSS isn’t the only tool for the Java platform that can facilitate complexity reporting. PMD, another open source project that analyzes Java source files, has a series of rules, one of which reports cyclomatic complexity. CheckStyle is another open source project with a similar cyclomatic complexity rule. Both PMD and CheckStyle also have Ant tasks and Maven plug-ins. (See Related topics for more about all of the tools discussed so far.)

Use complexity metrics

Because cyclomatic complexity is such a good indicator of code complexity, there is a strong relationship between test-driven development and low CC values. When tests are written often (note, I'm not implying first), developers have the tendency to write uncomplicated code because complicated code is hard to test. If you find that you're having difficulty writing a test, it's a red flag that the code under test may be complex. The short "code, test, code, test" cycle of TDD invites refactoring in these cases, which continually drives the development of uncomplicated code.

Measuring cyclomatic complexity is, therefore, particularly valuable in situations where you're working with a legacy code base. Moreover, it can be helpful to monitor CC values with distributed development teams, or even on large teams with various skill levels. Determining the CC of class methods in a code base and continually monitoring these values will give your team a head start on addressing complexity issues as they arise.
Related topics

- "In pursuit of code quality: Don't be fooled by the coverage report" (Andrew Glover, developerWorks, January 2006): Are your test coverage measurements leading you astray? Find out in this first article in the new series!
- JavaNCSS: A source measurement suite for the Java platform.
- PMD: This popular open source tool scans Java code for problems.
- CheckStyle: Another Java analysis tool from SourceForge.