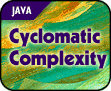
[[](http://www.onjava.com/)](http://www.onjava.com/)  
Published on [**ONJava.com**](http://www.onjava.com/) (<http://www.onjava.com/>)  
[See this](http://www.onjava.com/pub/a/general/print_code.html) if you're having trouble printing code examples

**Code Improvement Through Cyclomatic Complexity**

by [Andrew Glover](http://www.onjava.com/pub/au/1425)  
06/16/2004

Software metrics often receive negative criticism, as they are viewed as an exact science, uniformly applicable to all scenarios. True, software metrics are an unbiased, objective measurement of a particular aspect of code; however, a particular metric's applicability to a domain is usually subjective. For instance, highly coupled code may have been intentionally designed that way for performance reasons; consequently, a coupling metric that suggests problems with this code must be evaluated in the context of the overall application.

With this in mind, applying various software metrics to a code base can be an effective overall gauge of software quality. One such metric, *cyclomatic complexity*, can be helpful in ascertaining areas of code that may require additional attention to head off future maintenance issues. That attention, moreover, can take the form of unit testing and refactoring.

Pioneered in the 1970s by Thomas McCabe of *McCabe & Associates* fame, cyclomatic complexity essentially represents the number of paths through a particular section of code, which in object-oriented languages applies to methods. Cyclomatic complexity's formal academic equation from graph theory is as follows:

CC = E - N + P

where E represents the number of edges on a graph, N the number of nodes, and P the number of connected components.

If you've already given up on software metrics after reading that equation, there is an easier equation that will make sense. Cyclomatic complexity can be explained in layman's terms as follows: every decision point in a method (i.e., an if, for, while, or case statement) is counted; additionally, one is added for the method's entry point, resulting in an integer-based measurement denoting a method's complexity.

For instance, the following uncomplicated method will yield a cyclomatic complexity value of 3.

public int getValue(int param1) {

int value = 0;

if (param1 == 0) {

value = 4;

} else {

value = 0;

}

return value;

}

In the above method, there are two decision points: an if and an else. Remembering that a method's entry point automatically adds one, the final value equals 3.

Indeed, the example method is quite simple; however, one can imagine that if there were 10 additional decision points (yielding a cyclomatic complexity value of 13), the method may be perceived as complex. While one's opinion as to what construes code complexity is quite subjective, over the years the software industry has largely agreed that highly complex code can be difficult for engineers to understand and therefore harder to maintain. Moreover, highly complex code has a high probability of containing defects.

Various authors and studies have, in fact, suggested that a cyclomatic complexity value of 10 or higher for a particular method is considered complex. This is the key point. By determining the cyclomatic complexity of various methods found in objects and paying attention to outlier values, one can uncover code that most likely should become a candidate for a highly focused unit testing/refactoring effort.

**Determining Cyclomatic Complexity**

There are various commercial and open source software metrics tools on the market that can examine source code and report on the gamut of software metrics. One such tool, which happens to be open source, is [PMD](http://pmd.sourceforge.net/).

Running PMD against a code base is quite easy (see [this article by Tom Copeland](http://www.onjava.com/pub/a/onjava/2003/02/12/static_analysis.html)). Upon examining the code, PMD will produce a report in various forms, such as XML and HTML, that describes the various customizable infractions found.

For example, running PMD against a code base may produce a report that contains the following snippet of XML:

<file name="C:\com\dgg\web\AnomalyAction.java">

<violation line="1"

rule="CouplingBetweenObjectsRule">

A value of 27 may denote a high amount of coupling

within the class

</violation>

<violation line="103"

rule="CyclomaticComplexityRule">

The method 'updateAnomaly' has a

Cyclomatic Complexity of 22.

</violation>

</file>

As the XML demonstrates, *AnomalyAction.java* has a method, updateAnomaly, which has a rather high value of cyclomatic complexity. Interestingly enough, this class also appears to be quite coupled to various other objects within the code base, which only serves to complicate the code further. After viewing this report, red flags should start popping up, as clearly the AnomalyAction object's quality and relative stability have been called into question.

Software metrics, however, need to be evaluated in the context of the application against which they are being applied. While some red flags have now been raised, are there others? Reading the entire report may reveal even more egregiously high values of cyclomatic complexity. Additionally, it could turn out that every class examined possesses a high cyclomatic complexity value. The key to effective utilization of cyclomatic complexity and other metrics is to look for the outliers; i.e., those whose complexity is much higher than the rest of the code.

|  |
| --- |
| Related Reading  [http://covers.oreilly.com/images/9780596003876/cat.gif](http://shop.oreilly.com/product/9780596003876.do?CMP=ILL-4GV796923290)  [Java Extreme Programming Cookbook](http://shop.oreilly.com/product/9780596003876.do) **By** [**Eric M. Burke**](http://www.oreillynet.com/pub/au/110)**,** [**Brian M. Coyner**](http://www.oreillynet.com/pub/au/1078) |

The outliers indicate areas where one should focus. If there is a uniform pattern of complexity throughout a code base, then deciding where to make repairs becomes significantly more challenging. In this situation, other metrics, such as CM delta rates, can help indicate areas of needed attention.

Interestingly enough, there are multiple practices for discovering outliers. Tools like PMD report on individual methods' cyclomatic complexities. Other tools may report an object's aggregate cyclomatic complexity, or even a package's collective value. Unfortunately, aggregating cyclomatic complexity is rarely useful. As every concrete method found in an object will yield a minimum value of 1, any object following a JavaBean-like pattern (entity beans, DAOs, value objects, etc.) may produce a high cyclomatic complexity value. Large classes containing many methods may also produce elevated cyclomatic complexity scores.

**Taking Action**

When the determination has been made that a class is, in fact, complex, there are two steps available to mitigate the associated risk: unit testing, followed by refactoring.

As the cyclomatic complexity of a method is fundamentally the number of paths contained in that method, a general rule of thumb states that in order to ensure a high level of test coverage, the number of test cases for a method should equal the method's cyclomatic complexity. Unfortunately, this bold statement often is ignored.

Indeed, writing 22 unique test cases for updateAnomaly, while a noble goal, will probably not happen. Time constraints, developer attention spans, caffeine deprivation, etc. all work against even the most well-intentioned quality plans. While 22 test cases for updateAnomaly may be unattainable, surely zero should also be cause for concern.

As before, common sense needs to be applied. A further examination of the code may reveal that five test cases can effectively exercise all paths of the code under focus. Conversely, it may be determined that most of the paths in the code under focus are rarely executed edge cases; therefore, fewer test cases may suffice (for now). At this point in the process, one's tolerance for risk must be applied.

If five test cases can indeed assist in lowering the risk of the method's complexity, the next question becomes, "What is the most effective method for building the associated test cases?" By far, the most useful manner is unit testing with a framework like [JUnit](http://junit.org/index.htm). White-box unit testing is close to the code and ensures that the code under test fulfills its fundamental responsibilities at the lowest possible level. Unit testing is also quite easy as one can implement unit tests *before* and *after* actually writing the code. Higher-level testing usually involves other aspects of a system, such as other packages, objects, databases, containers, etc., that introduce dependencies and complexities that ultimately make it harder to effectively address low-level code details.

**Refactoring**

Unit testing helps mitigate risk by instilling developer confidence and facilitating rapid code change; however, unit testing will not make the code under test less complex. To simplify complex code, one must surgically remove the knots via refactoring. Be warned, however, that well-written unit tests are required before one attempts to refactor code.

The seminal book *Refactoring: Improving the Design of Existing Code*, by [Martin Fowler and friends](http://www.refactoring.com/catalog/index.html), provides a catalog of techniques that can produce code that is simpler and, consequently, easier to maintain. Additionally, the patterns found in *Design Patterns*, by Erich Gamma et al., can also help disentangle code.

|  |  |
| --- | --- |
|  |  |

**Case Study**

Imagine a business-to-business application that handles orders for a restaurant chain. These orders can be for beverages, food, napkins, silverware, etc. Given a need for additional beverages, the system creates an XML document and transmits the document to a beverage vendor, which then fulfills the order. Different vendors have varying XML formats and, as the system is such a success for the chain's business, additional vendors have been joining the network over time. This places additional requirements on the system, which can increase complexity.

A common pattern of code that usually triggers elevated values of cyclomatic complexity is a large method with many if/else conditions that each do a small unit of work on some input or in creating the method's output. We've probably all crafted these methods and have probably discovered, at some later point in time, that the method has grown and continues to grow, as new requirements come up, triggering additional conditionals to handle them.

For example, the following snippet of code returns a List of XML documents depending on the orderType property found in the containing class.

if(MessageConstants.ORDER\_TYPE\_FOOD.equals(

this.getOrderType())) {

messages.add(new StandardMessage(

this.getVendorId(),

this.getFoodOrderAcceptResponse()));

}else if(MessageConstants.ORDER\_TYPE\_BEVERAGE.

equals(this.getOrderType())) {

messages.add(new StandardMessage(

this.getVendorId(),

this.getBeverageOrderAcceptResponse()));

}

//many more to follow...

As you can imagine, as new types of Orders are added to the system, the conditional statements grow to handle the resulting new XML documents. Not surprisingly, running PMD against this code produced a report indicating that this method, processRequest(), contained an elevated cyclomatic complexity.

The first step in stabilizing the code indeed proves to be somewhat arduous, as in order to craft an effective JUnit test suite, we have to craft multiple XML documents. What's more, the original class containing the processRequest() method is part of an inheritance hierarchy, which contains state and depends on other classes and their respective states.

Looking deeper into the code reveals the base class's constructor obtains an instance of a PropertyUtility, which happens to require that System variables are set to facilitate finding a property file!

We finally develop a suite of test cases that exercise our class to a level of confidence. Below is an example test case.

public void testHandleResponseGeneration

throws Exception{

String baseXML = //...obtain XML

StandardMessageGenerator generator =

new StandardMessageGenerator(baseXML);

List messages = generator.processRequest();

Message mess = (Message)messages.get(0);

TestCase.assertNotNull("message was null",

mess);

}

Once we've attained a level of confidence in the code, we can move on to other tasks in other objects, or further refine this one. As we're aware that this code is one of the core components of the application that is continually pushed and morphed by various developers into a high level of chaos, we decide to refactor.

One of the most effective techniques for breaking up a large, complex group of conditionals utilizes polymorphism. Based upon [Replace Conditional with Polymorphism](http://www.refactoring.com/catalog/replaceConditionalWithPolymorphism.html) and a liberal interpretation of *Design Pattern*'s *Template Pattern*, *Factory Method Pattern*, and *Chain of Responsibility*, we can craft an elegant framework that pushes each conditional into a separate class. With this surgical maneuver, we can now easily isolate each conditional, which allows us to write effective unit tests.

As the original code flow went down the conditional stack adding XML documents to a List if a condition held true, we will create an API that takes an Order object and a List object, as shown below. The Order object embodies all of the properties of the original class that forced the previous code to reference this.

public abstract

class AbstractResponseMessageHandlerBase {

public AbstractResponseMessageHandlerBase() {

super();

}

abstract public void handleResponseGeneration(

final OrderMessageInfoVO orderMessage,

List messageList)

throws ResponseMessageException;

}

Each conditional will now embody a separate class, such as BeverageOrderAcceptResponseHandler, which matches a conditional testing whether the order is of type BEVERAGE.

With the abstract class defined, we simply push the corresponding conditional's logic into the template method. Below is BeverageOrderAcceptResponseHandler's implementation.

public void handleResponseGeneration(

final OrderMessageInfoVO orderMessage,

List messageList)

throws ResponseMessageException {

try{

if (MessageConstants.ORDER\_TYPE\_BEVERAGE.

equals(orderMessage.getOrderType())

&& (MessageConstants.

ORDER\_ACTION\_ORIGINAL.equals(

orderMessage.getOrderAction()))) {

final String orderXML =

ResponseContentDelegate.

getDefaultOrderResponseContent(

orderMessage);

messageList.add(new

StandardMessage(orderMessage.getVendorId(),

orderXML));

}

}catch(MessageFormatException e1){

throw new ResponseMessageException(e1);

}

}

We also define an *Abstract Factory*, IResponseHandlerAbstractFactory, which takes the Order object and returns a List; furthermore, we create an implementation of the factory.

Bearing in mind our liberal interpretation of various design patterns, we simulate a chain by adding all our ResponseHandler objects into a Collection and then iterating over the collection in the manufacture method, calling handleResponseGeneration and passing in our List and Order objects.

public class DefaultResponseHandlerFactory

implements IResponseHandlerAbstractFactory {

final private static

AbstractResponseMessageHandlerBase[] HANDLERS;

static{

Collection temp = new ArrayList( );

temp.add(new FoodOrderAcceptResponseHandler());

temp.add(

new BeverageOrderAcceptResponseHandler());

temp.add(

new NapkinOrderAcceptResponseHandler());

temp.add(new CancelOrderReponseHandler());

final AbstractResponseMessageHandlerBase[]

tmAr = new AbstractResponseMessageHandlerBase[

temp.size()];

HANDLERS =

(AbstractResponseMessageHandlerBase[])temp.

toArray(tmAr);

}

public List manufacture(

final OrderMessageInfoVO orderMessage) {

List messages = new ArrayList();

for(int x = 0; x < HANDLERS.length; x++){

AbstractResponseMessageHandlerBase iHandler =

HANDLERS[x];

try{

iHandler.handleResponseGeneration(

orderMessage, messages);

}catch(ResponseMessageException e1){

Log.logError(e1);

}

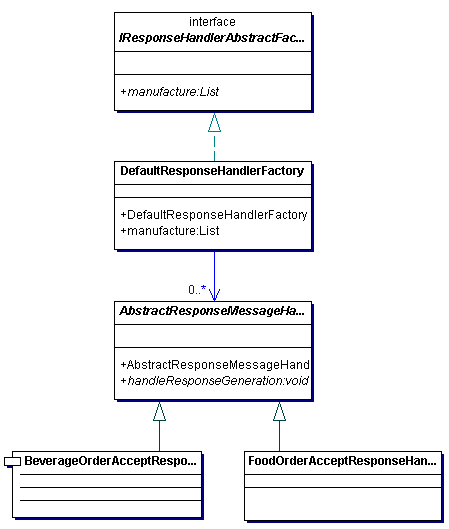
}

return messages;

}

}

Our resulting framework is actually quite simple, as shown below in Figure 1. Our DefualtResponseHandlerFactory contains a Collection of AbstractResponseMessageHandlers. As more vendors come online, we simply extend AbstractResponseMessageHandler and ensure the newly created class is part of DefualtResponseHandlerFactory's collection.

  
*Figure 1. Class diagram of refactored code.*

Testing each ResponseHandler is now trivial. We craft a desired Order object and pass it into the corresponding ResponseHandler; subsequently, we ensure that the resulting List contains our desired XML document(s).

public void testHandleResponseGeneration

throws Exception{

List messages = new ArrayList();

OrderMessageInfoVO vo = //...obtain sample VO

AbstractResponseMessageHandlerBase

handler = //...obtain desired Handler

handler.handleResponseGeneration(vo, messages);

Message mess = (Message)messages.get(0);

TestCase.assertEquals("vendor id should be 7908",

"7908", mess.getUserId());

}

Additionally, after we have test cases for each handler, we then move on to test the DefaultResponseHandlerFactory. Again, testing this object could not be easier, as we essentially perform the same steps as above. We create an Order object, pass it into the factory, and ensure that the desired number of XML documents can be confirmed in the returned List. What's more, the original test cases we crafted *before* refactoring serve as regression tests!

**Lessons Learned**

Clearly, a metric like cyclomatic complexity can facilitate the discovery of potentially dangerous code. Its simple and logical representation of complexity has shown, time and time again, to correlate directly to defects. Indeed, running code-metric tools against familiar code bases will often support your deepest suspicions on difficult and unmanageable code.

As you utilize this metric and begin a process of stringent unit testing and refactoring, you may start to see the value in [Test-Driven Development](http://www.objectmentor.com/writeUps/TestDrivenDevelopment). As adopters of this revolutionary methodology can confirm, test-case development before actually writing implementation code yields highly elegant and, indeed, simple code, which is quite easy to test. In fact, projects that carry out Test-Driven Development will find their code bases are rarely flagged for high cyclomatic complexity values. Give it a try -- it will save you time down the road!

[*Andrew Glover*](http://www.onjava.com/pub/au/1425) *is the founder and CTO of* [*Vanward Technologies*](http://www.vanwardtechnologies.com)*, a company specializing in building automated testing frameworks.*

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

Copyright © 2009 O'Reilly Media, Inc.