A COUPLING-COMPLEXITY METRIC SUITE FOR PREDICTING
SOFTWARE QUALITY

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Christopher L. Gray

June 2008
AUTHORIZATION FOR REPRODUCTION OF MASTER’S THESIS

I reserve the reproduction rights of this thesis for a period of seven years from the date of submission. I waive reproduction rights after the time span has expired.

Signature

Date June 5th, 2008
TITLE: A Coupling-Complexity Metric Suite for Predicting Software Quality

AUTHOR: Christopher L. Gray

DATE SUBMITTED: June 2008

Dr. David Janzen
Advisor or Committee Chair

Dr. Aaron Keen
Committee Member

Dr. Gene Fisher
Committee Member
Abstract

A Coupling-Complexity Metric Suite for Predicting Software Quality

by

Christopher L. Gray

Coupling Between Objects and Cyclomatic Complexity have long been used to measure software quality and predict maintainability and reliability of software systems prior to release. In particular, Coupling Between Objects has been shown to correlate with fault-proneness and maintainability of a system at the class level. We propose a new set of metrics based on a fusion of Coupling Between Objects and Cyclomatic Complexity that can be superior to Coupling Between Objects alone at predicting class quality. The new metrics use Cyclomatic Complexity to 1) augment Coupling Between Objects counting to assign a strength of a coupling between two classes and 2) determine the complexity of a method invocation chain through the transitive relation of invocations involved in a coupling. This results in a measure that identifies objects that are coupled to highly complex methods or method invocation chains. The metrics were implemented as an Eclipse Plugin and an analysis of two industry Java projects, ConnectorJ and Hibernate, demonstrates the correlation between the new metrics and post-release defects identified in system change logs.
Acknowledgements

I would like to give a very special thanks to Dr. David Janzen for all of his support and guidance on this project. Without his help, this would never have come together. I would also like to thank the other members of my committee, Dr. Aaron Keen and Dr. Gene Fisher, for their help.
## Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Cyclomatic Complexity . . . . . . . . 2
1.2 Chidamber & Kemerer Metric Suite . . . . . 2
   1.2.1 CK Metric: Weighted Methods per Class (WMC) . 3
   1.2.2 CK Metric: Coupling Between Objects (CBO) 3
1.3 Goals and Direction . . . . . . . . . . . 4
1.4 Inspiration and Motivation . . . . . . . 5

2 Related Work 7

2.1 Cyclomatic Complexity . . . . . . . . . . 7
   2.1.1 Defining McCabe’s Number . . . . . . . . . 7
   2.1.2 Code Example . . . . . . . . . . . . . . . . . 8
   2.1.3 Extended Cyclomatic Complexity . . . . 9
   2.1.4 Modified Cyclomatic Complexity . . . . 9
   2.1.5 Common threshold values . . . . . . . . 10
   2.1.6 Empirical Results . . . . . . . . . . . . . 10
   2.1.7 Criticisms . . . . . . . . . . . . . . . . . 11
2.2 Coupling Between Objects . . . . . . . . . . 13
   2.2.1 Defining Coupling . . . . . . . . . . . . . 13
   2.2.2 Chidamber and Kemerer Metric Suite . . 13
   2.2.3 Evaluation . . . . . . . . . . . . . . . . . 14

vi
2.2.4 Code example ............................................. 14
2.2.5 Empirical Validation of WMC/CBO ................. 15
2.3 Established Coupling Metrics .............................. 17
  2.3.1 Dynamic-aspect ........................................ 18

3 Established and Proposed Metrics ............. 20
  3.1 Terminology and Notation ................................. 20
  3.2 A Unified Framework for Coupling .................. 21
    3.2.1 System ............................................. 21
    3.2.2 Methods ........................................... 22
    3.2.3 Method Invocations ................................. 22
    3.2.4 Attributes ......................................... 23
    3.2.5 Attribute References ............................... 24
    3.2.6 Predicates ......................................... 24
  3.3 Established Object-Oriented Metrics ............. 24
    3.3.1 Weighted Method per Class ....................... 25
    3.3.2 Coupling Between Objects ......................... 27
  3.4 Proposed Metrics ........................................ 28
    3.4.1 Transitive Cyclomatic Complexity ............... 29
    3.4.2 Coupling-Complexity Between Objects ............ 30
    3.4.3 Transitive Coupling-Complexity Between Objects . 32
    3.4.4 α Variation ......................................... 34

4 Evaluation .................................................. 36
  4.1 Source Control ............................................. 36
    4.1.1 Defect Log Example ................................ 37
  4.2 ConnectorJ ................................................ 38
  4.3 Hibernate .................................................. 38
  4.4 Eclipse Plug-in ........................................... 39
    4.4.1 Verification of Accuracy ......................... 39
    4.4.2 Cyclomatic Complexity ................................ 39
    4.4.3 Coupling Between Objects ......................... 40
4.4.4 Proposed Metrics ........................................... 40
4.5 Hypotheses .................................................... 41

5 Results & Analysis ........................................... 42
  5.1 Correlation .................................................. 42
  5.2 Linear Regression .......................................... 44
  5.3 Univariate Analysis ........................................ 46
    5.3.1 ConnectorJ 3.0.0 ...................................... 46
    5.3.2 ConnectorJ 3.1.0 ...................................... 47
    5.3.3 ConnectorJ 5.1.0 ...................................... 48
    5.3.4 Hibernate 3.0.0 ...................................... 50
  5.4 Threats to Validity ........................................ 50
  5.5 Analysis .................................................... 51

6 Conclusions & Future Work ................................ 53
  6.1 Contribution ................................................ 53
  6.2 Validation .................................................. 54
  6.3 Results ..................................................... 55
  6.4 Future Work ................................................ 56

Bibliography .................................................... 57
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Rules to compute CC</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Common threshold values from the Software engineering institute at Carnegie Mellon University [2]</td>
<td>11</td>
</tr>
<tr>
<td>5.1</td>
<td>Correlation of software metrics and class quality for ConnectorJ and Hibernate. Asterisk (*) represents which metrics are most correlated with predicting class quality in either the WMC or CBO modification metrics.</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Univariate analysis for CCBO-CC calculated for ConnectorJ 3.0.0.</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Univariate analysis for WMC-McCabe calculated for ConnectorJ 3.0.0.</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Univariate analysis for CCBO-CC calculated for ConnectorJ 3.1.0.</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Univariate analysis for WMC-McCabe calculated for Hibernate ConnectorJ 3.1.0</td>
<td>48</td>
</tr>
<tr>
<td>5.6</td>
<td>Univariate analysis for CCBO-CC calculated for ConnectorJ 5.1.0.</td>
<td>48</td>
</tr>
<tr>
<td>5.7</td>
<td>Univariate analysis for WMC-McCabe calculated for ConnectorJ 5.1.0.</td>
<td>49</td>
</tr>
<tr>
<td>5.8</td>
<td>Univariate analysis for CCBO-CC calculated for Hibernate 3.0.0alpha.</td>
<td>49</td>
</tr>
<tr>
<td>5.9</td>
<td>Univariate analysis for WMC-McCabe calculated for Hibernate 3.0.0alpha.</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Figures within boxes are used in the calculation of the CC: 3 .... 9
2.2 Control flow graph of Figure 2.1. $E = 10$, $N = 9$, $M = 10 - 9 + 2(P=1); M = 3$ ......................... 10
2.3 Identical CCs even with alternate structure. $CC = 3$ in both cases 12
2.4 What constructs to evaluate when computing the CBO value for a given class [33] ............................ 15
2.5 The CBO value for ClassF is 5. The syntax is Java. ............ 16

3.1 The WMC-CC value for ClassA is 4. The WMC-McCabe value is 12. ................................................... 26
3.2 CBO value is 2. .......................................................... 28
3.3 Example Java System. ................................................. 31

4.1 Example Defect Log ....................................................... 37

5.1 Linear regression for WMC-McCabe against Number of Defects per Class for ConnectorJ 3.0.0 .......................... 44
5.2 Linear regression for CCBO-CC against Number of Defects per Class for ConnectorJ 3.0.0 .......................... 45
Chapter 1

Introduction

Software engineering is a difficult and complex task. It can be successful up-front, but then be littered with faults and maintenance headaches if the system falters due to heavy defects and errors after deployment. Software metrics are one way to predict quality within a system, pointing to problem areas that can be addressed prior to software release. Metrics attempt to measure a particular aspect of a software system. These aspects can range from trivial measurements such as the number of lines of code to the relationships created between components in a system. Measurements are often output in a numerical representation which can then be transformed to be indicators of “reliability” [25]. Cook [13] states that if you know enough about something you can measure it numerically, while if the reverse occurs then you have a lack of knowledge for the subject matter. Two metrics that are fundamental to this research, Cyclomatic Complexity and Coupling Between Objects will be described in further detail.
1.1 Cyclomatic Complexity

Cyclomatic Complexity (CC), introduced by Thomas McCabe in 1976 [30], does not accurately measure the complexity of methods within the object-oriented paradigm. Object-oriented methods are much shorter than that of procedural languages. Procedural languages can take pages at a time while object-oriented methods typically have less than six lines [29]. In addition, well designed object-oriented systems will not have to make excessive use of case statements — which is common within procedural languages and a common complexity decision point.

Methods within an object-oriented system can also construct objects in their body of statements. With the creation of an object you not only get the same functionality of a structure, but also a number of methods of additional functionality upon that object instance. This is known as delocalization [14].

1.2 Chidamber & Kemerer Metric Suite

Chidamber and Kemerer introduced a metric suite to measure testability, maintenance, and reusability of a class but without any empirical validation. The suite consists of six metrics but the proposed metrics only use two of these metrics introduced in 1994 [12].

\[1\] The object-oriented paradigm is a fundamental way of designing systems that use “objects” and their interactions to design applications and computer programs. Common features include encapsulation, abstraction, polymorphism, and inheritance.
1.2.1 CK Metric: Weighted Methods per Class (WMC)

WMC is defined to be the summation of all method's complexity defined in a given class. Chidamber and Kemerer define this to be the “complexity” of the method leaving the intention ambiguous. However the CC will be in substitution for this “complexity”. Weighted Methods per Class (WMC) has been shown to be a good predictor of class quality in an object-oriented environment when compared to metrics within the CK [12], MOOD [15], and QMOOD [3] metric suites. This is the case, even though CC does not have any components that measure specific aspects of the object-oriented paradigm [11].

1.2.2 CK Metric: Coupling Between Objects (CBO)

Chidamber and Kemerer define Coupling Between Objects (CBO) for a class to be the count of the number of other classes to which it is efferently coupled. This number represents an object’s fan-out to external objects. The metric’s basis is in the fact that if an object is coupled to another it uses another’s methods or instance variables.

Defining Coupling

Stevens et al. [38] introduced the concept of coupling into structured design. He defined coupling to be “the measure of the strength of association established by a connection from one module to another.” This infers that highly coupled classes are not desired as it is considered bad design and can lead to difficulty understanding classes. As their degree of coupling increases so does the complexity of the class. This results with the module becoming increasingly dependent on external classes to implement its functionality and is bound to reflect any changes
the external classes may undergo in future maintenance.

Applicability

CBO gives insight into important aspects of software design. Excessive CBO does not promote reuse as an object relies too heavily on another to be reused independently. If excessive coupling is found between objects it is advised to reduce the amount of coupling through refactoring. If not, changes to the coupled objects in the future could result in relationships being modified that are unexpected. CBO has been shown to be correlated to class quality (defect or error-proneness of a class) [4, 21, 39, 40, 34, 20]. Creating mock objects to test within the coupled objects also provide additional complexities between objects further increasing the infinitely difficult problem of testing a system.

1.3 Goals and Direction

CBO is an Object-Oriented metric that measures the efferent coupling relationship between objects. This original metric assigns a measurement of one to each coupling relationship. However, when CBO is paired with the CC it can help measure the weight of a coupling connection. These connections modify the strength of the relationship and therefore must be taken into account when defining the coupling complexity between objects.

In recent years, many new coupling metrics have been introduced – with and without empirical validation [37, 1, 5]. The goal is to fuse two metrics together; CC and an Object-Oriented metric, CBO, and propose novel metrics which combine the two to measure class quality (defect or error-proneness of a
The proposed metrics will result in numerous variations of CC definitions and CBO fusions. This will generate multiple chances at experimenting with each variation to determine which is more accurate at identifying class quality. The object-oriented coupling-complexity metrics would be compared to CBO’s accuracy at identifying class quality, and the modifications to the CC would compete with WMC.

Improvements in software metrics will improve with evidence-based software engineering. The metrics will be implemented through the Eclipse Plug-in framework [18]. The proposed metrics will be empirically validated by applying them to three versions of ConnectorJ (JDBC driver for MySQL written in Java) and Hibernate (an object-relational mapping framework for Java).

1.4 Inspiration and Motivation

Creating software is complex and increasingly expensive to develop [26]. The maintenance phase of software is by far the most costly part in the software life cycle [35]. Being able to reduce potential defects as well as increasing ease of maintenance through software metrics creates a huge interest in the applicability of metrics.

The two metrics (CC and CBO) offer varying degrees of aspects measured within a software system. CBO is a measurement which can be interpreted to show the reusability of a component and its proneness to change in the future. This proneness to change is caused through its extensive coupling throughout the system. If one object is modified where the coupled object relied on the
preexisting behavior previously, then there is a subtle defect that is potentially introduced.

CC can be an indicator for potential refactoring of a method. A method that displays a high CC represents a method that may be too large and can be broken up into smaller more maintainable components. A web-advertising development team spent a good portion of their development refactoring methods that had a high CC [11]. This helped combat the problem of rising complexity as a project ages and grows in size.

With the fusion of these two metrics there is hope that combined their measured aspects will remain. This results in a measure that will be able to indicate an object that is coupled to highly complex methods. This can lead developers to rethink particular components that can be re-factored into more maintainable modules or indicate the complexity of reusing a component in an alien system.
Chapter 2

Related Work

In the field of software metrics there has been research on metrics to predict fault-proneness, change-proneness, identifying refactorable items, etc. Some of the more popular metrics will be discussed in detail as they are foundations for the proposed metrics. Other metric suites have been introduced and experiments at estimating their accuracy have been conducted. A brief insight into other work will be explored.

2.1 Cyclomatic Complexity

2.1.1 Defining McCabe’s Number

Thomas McCabe [30] defined Cyclomatic Complexity (CC) in 1976 to measure the structural complexity of a procedure. McCabe’s metric finds its foundation within graph theory. The metric transforms a procedure’s statements into a graph. Each node within the graph represents a different conditional statement. CC can therefore be represented as a function:
\[ M = E - N + 2P \]

Where \( E \) is the number of edges, \( N \) the number of nodes, and \( P \) the number of components within the graph [23].

An alternative way to define the metric is to take the summation of the total number of decision points plus one. These decision points can include various programming language semantics. For example in the popular C language some of these include:

<table>
<thead>
<tr>
<th>Construct</th>
<th>Effect</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>+1</td>
<td>An if statement is a decision point</td>
</tr>
<tr>
<td>else if</td>
<td>+1</td>
<td>An else if adds a new decision point</td>
</tr>
<tr>
<td>else</td>
<td>0</td>
<td>The original decision was made at the if</td>
</tr>
<tr>
<td>switch</td>
<td>+1 for each case</td>
<td>Each case branch is equivalent to an if</td>
</tr>
<tr>
<td>for/while</td>
<td>+1</td>
<td>A decision is made at each iteration of the loop</td>
</tr>
</tbody>
</table>

Table 2.1: Rules to compute CC

From a body of statements one can construct a control flow graph that represents the individual decision points within a procedure. The number of nodes within this control flow graph plus one is a popular alternative way to define CC.

2.1.2 Code Example

The code in Figure 2.1 illustrates how to determine decision points within a procedure to display a more concrete example.

In Figure 2.1 the language operators: if, &&, and || are considered decision points within the CC computation. To reinforce this code example Figure 2.2 demonstrates a control flow graph of the code in Figure 2.1.

This control flow graph represents the if/else scenario, except with an extra...
```
if ((value >= 0 && value < 100) || value == MAX) {
    foo = value;
} else {
    foo = -1;
}
```

Figure 2.1: Figures within boxes are used in the calculation of the CC: 3

component on the left side. This component is created through the boolean operators within the if guard. These operators effectively create another conditional branch which is represented by two additional paths.

2.1.3 Extended Cyclomatic Complexity

This variation adds one additional factor to the original definition, boolean operators. These can be used within an if guard to increase the complexity of the decision. However, these boolean operators can be re-factored out and turned into their own if statements. Therefore, this variation includes the complexity that boolean operators add if they were if constructs.

2.1.4 Modified Cyclomatic Complexity

In C based languages, switch statements are equivalent to a series of if/elseif constructs. In the case of a switch the guard is evaluated each time and then directs execution towards the correct case handler. This can be argued to be a single decision point instead of the alternative if/elseif.
Figure 2.2: Control flow graph of Figure 2.1. $E = 10$, $N = 9$, $M = 10$
- $9 + 2(P=1)$; $M = 3$

<table>
<thead>
<tr>
<th>Variation</th>
<th>Construct</th>
<th>Effect</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended CC</td>
<td>boolean operators: &amp;&amp;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modified CC</td>
<td>switch</td>
<td>+1 for entire switch</td>
<td>switch statements use a single boolean guard</td>
</tr>
</tbody>
</table>

2.1.5 Common threshold values

In Table 2.2 the Software engineering institute at Carnegie Mellon University gives threshold values to the CC metric. Depending on the resulting complexity there is a risk associated with that specific procedure’s complexity. Appropriate measures should be taken to reduce the complexity to avoid future maintenance.

2.1.6 Empirical Results

In a two-and-a-half year study of a commercially developed intelligent web-advertising program, CC was used to determine how agile-development affected
Cyclomatic Complexity  |  Risk evaluation  
---|---
1–10  |  a simple procedure, not much risk  
11–20  |  moderate complexity, moderate risk  
21–50  |  highly complex, high risk  
>50  |  untestable, very high risk  

Table 2.2: Common threshold values from the Software engineering institute at Carnegie Mellon University [2]  

the project. Comparing method complexities in a two week period, the observers were able to determine if there was refactoring or restructuring of the code; based on if complexities were reduced in the latter method’s complexity. The metric was also used to evaluate the system as a whole. There were only 2 methods in the entire system that were flagged as high complexity. They used a CC value of 15 [31] to determine the difference between a high and low complexity function. This was shocking as their results showed that other projects that were examined typically had 5–10% of their methods flagged at high complexity [11].

2.1.7 Criticisms

Differentiation between ANDed and Nested constructs

A complexity value for a function can be identical to another even though its structure is completely different. In Figure 2.3 there is a clear structural difference. The control flow graph on the left shows code that is structured with two decision point constructs. While the graph on the right contains a nested decision point construct within another. The complexities of these graphs are the same, but are different in their structuring of decision points. Davis and LeBlanc try to quantify these differences using Shannon and Weaver’s concept of (information) entropy. These unbiased quantifications [23] will not be used within the previously defined CC metric.
Figure 2.3: Identical CCs even with alternate structure. CC = 3 in both cases

Object-Oriented Programming

CC does not accurately measure the complexity of methods within the object-oriented paradigm. Object-oriented methods are much shorter than that of procedural languages. Procedural languages, such as C, can take pages at a time while object-oriented methods typically have less than six lines [29]. In addition, well designed object-oriented systems will not have to make excessive use of case statements — which is common within procedural languages and a common complexity decision point.

Methods within an object-oriented system can also construct objects within their body of statements. In procedural languages such as C one could construct a structure which is a mere illusion to segment bits into alternate primitive types. But with the creation of an object you not only get the same functionality of a structure but also a number of methods of additional functionality upon that object instance. This is known as delocalization [14].

Rajaraman and Lyu [37] attack the applicability of three widely used metrics (statement count, Halstead’s Software Science, and CC) as they do not address
OO design constructs such as inheritance and encapsulation. They propose equivalent metrics that account for these object-oriented constructs (different coupling metrics) and show correlation with change-proneness and testability.

2.2 Coupling Between Objects

2.2.1 Defining Coupling

Stevens et al. [38] introduced the concept of coupling into structured design, defined coupling to be "the measure of the strength of association established by a connection from one module to another." Highly coupled classes are considered bad design and low coupled components are advocated by Pfleeger and Atlee [36]. As the degree of coupling increases so does the complexity of the class. The module becomes dependent on external classes to implement its functionality and is bound to reflect any changes the external classes may undergo in future maintenance. It is ideal to modify a class without having to take into consideration other modules or components within a system.

2.2.2 Chidamber and Kemerer Metric Suite

Chidamber and Kemerer introduced a metric suite to measure testability, maintenance, and reusability of a class but without any empirical validation. They define Coupling Between Objects (CBO) for a class to be the count of the number of other classes to which it is coupled. The metric’s basis is in the fact that if an object is coupled to another it uses another’s methods or instance variables.
CBO gives insight into important aspects of software design. Excessive CBO does not promote reuse as an object relies too heavily on another to be reused independently. If excessive coupling is found between objects it is advised to reduce the amount of coupling through refactoring. If not, changes to the coupled objects in the future could result in relationships being modified that are unexpected. Creating mock objects to test within the coupled objects also provides additional complexities between objects further increasing the infinitely difficult problem of testing a system.

2.2.3 Evaluation

Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. A use is only counted once between two classes, but can go either way.

2.2.4 Code example

In Figure 2.5 ClassF has a CBO value of 5. This is calculated through those constructs that are highlighted within boxes. ClassF has a coupling relationship with Classes: A, B, C, D, and E. The multiple uses of ClassA are only counted once. However, note that there was no coupling relationship created because of the use of Constant.MAX_VAL. This is because the field is an immutable value. ClassF is coupled to ClassE through inheritance and is not coupled to the class java.io.Serializable. This is because the Java framework is considered to be stable through its widespread use and longterm installment.
• Counted:
  - Method Invocation
  - Variable Reference
  - Return Types
  - Formal Parameter Types
  - Field Accesses
  - Inheritance
  - Polymorphism
  - Exceptions

• Not counted:
  - Use of constants
  - Calls to API declares
  - Handling of events

Figure 2.4: What constructs to evaluate when computing the CBO value for a given class [33]

2.2.5 Empirical Validation of WMC/CBO

Work has been conducted to empirically validate the widely-used CBO metric. Basili et al. [4] find that through empirical work that the metric is a good predictor of fault-proneness. Harrison et al. [21] have investigated the use of CBO and Number of Associations (NAS) which resulted in a collinear relationship between the dimensions of software that they attempt to measure. The results were that the coupling metrics have no correlation with code understandability and error
public class ClassF extends ClassE implements Serializable {
    ClassB objB;
    public ClassC foo(ClassA objA) throws ClassD {
        int noConstants = Constant.MAX_VAL;
        if (objB.isGreaterThan(objA)) {
            noConstants = objA.getValue();
        } else {
            throw new ClassD();
        }
        return new ClassC();
    }
}

Figure 2.5: The CBO value for ClassF is 5. The syntax is Java.

density. Wilkie and Kitchenham [39] tried to investigate the correlation between CBO and change-ripples but fail to find any relationship. In addition to their previous research they attempted once again to extend the CBO metric with finer grain relationships of distinct forward and backward coupling but determine that the original CBO metric is superior and more accurate without the alterations [40].

Olague et al. [34] conducted a comparison of the Chidamber and Kemerer metric suite (CK), Abreu’s metrics (MOOD), and Bansiya and Davis’ metrics (QMOOD). They found that the CK suite (in particular WMC) is superior to the close second QMOOD, but MOOD was found to not be a good predictor of

2.3 Established Coupling Metrics

Binkley and Schach [6] propose a Coupling Dependency Metric which measures change and fault-proneness at a finer grain compared to other metrics that restrict their scope to the class level. Briand et al. [7] introduce a software metric suite which accounts for class-attribute, class-method, and method-method relationships and show that there is correlation between these coupling measurements with fault-proneness. In addition, they inject that the use of friend within C++ can increase the fault-proneness of classes even more than other types of coupling. Emam et al. [17] attempted to find the best metric at predicting fault-proneness among 24 metrics that were proposed by Chidamber and Kemerer and Briand et al. They found that the OCMEC (Other class-method export coupling) was the best among the 24 to be fault-proneness predictors. Liu and Xu [28] have proposed an object-oriented metric suite that measures the magnitude of coupling between classes and show that their suite offers a new dimension of measurement complementing other metrics.

Hitz and Montazeri [24] argue that coupling between two classes should be multi-faceted rather than being a singular relation. In other words there should be many aspects taken into account when measuring the coupling relationship between classes within a system. Briand et al. [9, 8] identify eighteen distinct aspects of coupling with each focusing on a different type of relationship. These relationships are finer-grained than previous approaches where they tend to only pay
attention to method-method, class-method, class-attribute, etc. Li and Henry [27] propose two additions to the existing CK suite of metrics. Message Passing Coupling (MPC) is the number of messages (method invocations) a class sends to other classes. Data Abstraction Coupling (DAC) is the number of attributes in a class that have other classes as their type.

Gui and Scott [19] propose metrics that satisfy three limitations with metrics they identify: CK suite [12], MOOD [15], Li and Henry’s [27] additions to the CK suite, Briand et al.’s [9, 8] eighteen finer-grain metrics, and considering Hitz and Montazeri’s [24] argument for multi-faceted coupling metrics. First, Gui and Scott argue that without regards for a specific variation of Response for Class (RFC) all of the static metrics identified treat coupling as an intransitive relation; they only consider direct coupling to classes and not indirect couplings. If Class A is coupled to Class B, and Class B is coupled to Class C, then there exists an indirect coupling from Class A to C. Second, some metrics like CBO treat coupling between a pair of classes as a binary relation—either they have one or not. There is no distinction between a strong and weak relation. Third, the metrics do not adjust for the complexity of the classes that are coupled.

2.3.1 Dynamic-aspect

To capture the run-time analysis of coupling which can change because of the polymorphism — an array of dynamic coupling metrics have been proposed. Beszedés et al. [5] proposed a Dynamic Function Coupling (DFC) metric to capture the proximity of functions on the run-time stack to predict change-proneness and impact sets. Hassoun et al. [22] prove the concept of dynamic coupling and that it is indeed a new dimension of coupling between objects within a run-time
environment. They find that polymorphic systems demonstrate less coupling overall compared to an identical system that does not use polymorphism. Arisholm [1] introduces a dynamic coupling suite to measure run-time behavior of a system and find similar results to Emam et al. in that the best metrics among the suite were those that dealt with export ¹ coupling metrics.

¹Locus of impact is defined in [7]
Chapter 3

Established and Proposed Metrics

The proposed metrics are modifications of established metrics. The established metrics will be discussed along with a unified way of representing these metrics. Each metric introduced will be accompanied with an example of how to calculate this metric. There is also a description of exactly which aspects of the system they attempt to measure.

3.1 Terminology and Notation

In many of the related software metrics the authors have given mathematical notation to concretely describe their metrics fully. However, many of these notations are merely created just for their specific metrics. Fortunately Briand et al. [10] proposed a standard notation that is used to describe software metrics so that all could readily understand the terminology.
3.2 A Unified Framework for Coupling

Briand et al. [10] introduce a unified framework for defining coupling measurement in object-oriented systems. They review three other previous attempts at defining such a framework and attempt to improve and unify the terminology. The previous frameworks have been proposed by Eder et al. [16], Hitz and Montazeri [24], and an earlier attempt by Briand et al. [7]. The framework utilizes mathematical notation to specifically define the different types of relationships. There are many definitions which are stated within the framework, for brevity, the definitions necessary for understanding the proposed metrics will be introduced.

3.2.1 System

A system is made of classes. These classes may be related to one another through either a parental, child, ancestral, or descendent nature.

System, Classes, Inheritance Relationships: An object-oriented system consists of a set of classes, $C$. There can exist inheritance relationships between classes such that for each class $c \in C$ let

- $Parents(c) \subseteq C$ be the set of parent classes of $c$.
- $Children(c) \subseteq C$ be the set of children classes of class $c$.
- $Ancestors(c) \subseteq C$ be the set of ancestor classes of class $c$.
- $Descendents(c) \subseteq C$ be the set of descendent classes of class $c$. 
3.2.2 Methods

A class has a set of methods. A method can be either virtual or non-virtual and either inherited, overridden, or newly defined, all of which have implications for measuring coupling.

Methods of a Class: For each class \( c \in C \) let \( M(c) \) be the set of methods of class \( c \).

The Set of all Methods: \( M(C) \) is the set of all methods in the system and is represented as

\[
M(C) = \bigcup_{c \in C} M(c)
\]

The Set of Methods Implemented in a Class: \( M_I(c) \subseteq M(c) \) be the set of methods implemented in \( c \), i.e., methods that \( c \) inherits but overrides or nonvirtual noninherited methods of \( c \).

Polymorphic Identification: \( P(m) \) is the function to identify which class the method \( m \) is dynamically bound to. \( P(m) = c \in C \) where \( m \in M(c) \)

3.2.3 Method Invocations

To measure coupling of a class \( c \), it is necessary to define the set of methods that \( m \in M(c) \) invokes and the frequency of these invocations. Method invocations can be either static or dynamic. For static invocations, the invoked method is determined by the type of the variable that references the object. For dynamic invocations, the invoked method is determined by a late-binding at run-time to the polymorphic type. One definition which is needed here but not defined in the unified framework is the notion of a transitive relation upon method invocations.
A method invocation may possibly invoke another method and so on. A proposed
addition to the framework will be defined to account for this behavior.

The Set of Statically Invoked Methods of \( m \): Let \( c \in C, \ m \in M_I(c) \), and
\( m' \in M(C) \). Then \( m' \in SIM(m) \iff \exists d \in C \) such that \( m' \in M(d) \) and
the body of \( m \) has a method invocation where \( m' \) is invoked for an object
of static type class \( d \).

The Set of Polymorphically Invoked Methods of \( m \): \( PIM(m) \) is the set
of all polymorphically invoked methods on \( m \). Let \( c \in C, \ m \in M_I(c) \),
and \( m' \in M(C) \). Then \( m' \in PIM(m) \iff \exists d \in C \) such that \( m' \in M(d) \)
and the body of \( m \) has a method invocation where \( m' \) may, because of
polymorphism and dynamic binding, be invoked for an object of dynamic
type \( d \).

The Transitive Closure on a Set of Invoked Methods \( m \): \( T(m) \) is the trans-
sitive closure on a set of invoked methods. Let \( m \) be a method, whether it
be statically or polymorphically invoked. Let \( m \) be defined to be \( m_0 \), where
\( m_0 \) can invoke \( m_1 \), \( m_1 \) can invoke \( m_2 \), and so on. Let

\[
T = \bigcup_{i \in M(C)} m_i
\]

3.2.4 Attributes

Classes have attributes which are either inherited or newly defined.

The Set of all Attributes: \( A(C) \) is the set of all attributes in the system and
is represented as

\[
A(C) = \bigcup A(c) \text{ where } c \in C
\]

23
3.2.5 Attribute References

Methods may reference attributes. These attributes may not be part of the encompassing class, therefore coupling it to the referenced encompassing class.

The Set of Attributes referenced by the method \( m \): For each \( m \in M(C) \)

let \( AR(m) \) be the set of attributes reference by method \( m \).

3.2.6 Predicates

To ensure proper usage between terms, a \( uses \) predicate must be defined.

Uses: Let \( c \in C, \ d \in C. \ uses(c, \ d) \leftrightarrow (\exists m \in M_I(c) : \exists m' \in M_I(d) : m' \in PIM(m)) \lor (\exists m \in M_I(c) : \exists a \in A_I(d) : a \in AR(m)) \)

A class \( c \) uses a class \( d \) if a method implemented in class \( c \) references a method or an attribute implemented in class \( d \).

3.3 Established Object-Oriented Metrics

These metrics presented will be used in comparison to the proposed metrics to see if there is an actual measurable improvement. The proposed metrics will be compared to these established metrics, because they derived from these established metrics and are additionally measuring another aspect of the system.
3.3.1 Weighted Method per Class

Class $C_1$, has methods $m_1, \ldots, m_n$ that are defined in the class. Let $c_1, \ldots, c_n$ be the complexity of the methods. Then:

$$WMC = \sum_{i=1}^{n} c_i$$

Chidamber and Kemerer [12] do not state a specific complexity function as it allows users of the metric the flexibility to choose whichever complexity function that applies to their needs. In this instance Thomas McCabe’s Cyclomatic Complexity metric will be used [30]. McCabe’s complexity function will be represented as $McCabe$ and when applied to $WMC$, will be referred to as $WMC - McCabe$. An alternate way to compute the CC arises from computing the number of decision points in a procedure plus one [32], this will also be used. This alternative method will be represented as $CC$ and when applied to $WMC$, will be referred to as $WMC - CC$.

Example

In Figure 3.1, an example is shown in Java how to evaluate $WMC$ on a particular class. $WMC - CC$ and $WMC - McCabe$ are both computed on the class and come out with different results. In method $foo$ the if/else is a decision branch which adds one to the complexity of the method. Likewise in method $bar$ the while contains a guard which is another decision point within the class.

For the method $foo$: $WMC - CC$ computes 2, 1 for the decision branch and plus one as the definition states, $WMC - McCabe$ computes 7, 1 for the decision branch plus 2 multiplied by the number of connected components or method invocations (there are 3). These invoked methods contain logistical complexity.
as well; but it is not accounted for directly, rather a flat value is supplied for outsourcing logic into separate modules.

For the method `bar`: $WMC - CC$ computes 2, 1 for the `while` and then plus one, $WMC - McCabe$ computes 5, 1 for the decision and then 2 multiplied by the two method invocations. Finally $WMC - CC$ and $WMC - McCabe$ compute 4 and 12 respectively for `ClassA`.

```java
public class ClassA {
    public void foo(ClassB refB, int y) {
        if (refB.getValue() > y) {
            refB.yWasLarger();
        } else {
            refB.yWasSmaller();
        }

    public void bar(ClassB refB) {
        while (refB.isLoop()) {
            refB.iterateLoop();
        }
    }
}
```

Figure 3.1: The WMC-CC value for `ClassA` is 4. The WMC-McCabe value is 12.
3.3.2 Coupling Between Objects

Coupling Between Objects (CBO) for a class is a count of the number of other classes to which it is coupled. This definition is flexible in three ways.

1. Which direction a class is coupled to another

2. How a class is actually coupled to another

3. The value to give a coupling relationship to distinguish its strength from another coupling

CBO will be strictly efferently coupling, in other words, only focusing on the outward coupling to foreign classes. The way two classes are coupled will follow the same definition as before in Chapter 2. The value that will be given to the coupling will be defaulted to one, but this research will experiment with various other values as well. These variations will be the novel part the proposed metric. Efferent coupling was chosen because it has been shown to be stronger at predicting class quality when compared to afferent coupling [17, 1].

Example

In Figure 3.3.2, an example of a Java system shows how Coupling Between Objects is computed for ClassA. ClassA has a CBO value of 2. ClassA has a coupling with ClassB through many of its method invocations on a reference variable to ClassB. ClassA then also has an implicit relationship to ClassC through polymorphism. ClassC derives from ClassB, so at runtime a ClassB reference could be referencing an instantiated ClassC object, therefore there is a possible coupling with Class A.
public class ClassA {
    public void foo(ClassB refB, int y) {
        if (refB.getValue() > y) {
            refB.yWasLarger();
        } else {
            refB.yWasSmaller();
        }
    }
    public void bar(ClassB refB) {
        while (refB.isLoop()) {
            refB.iterateLoop();
        }
    }
}

public class ClassB {
    public int getValue() {...}
    public void yWasLarger() {...}
    public void yWasSmaller() {...}
    public boolean isLoop() {...}
    public void iterateLoop() {...}
}

public class ClassC extends ClassB {
    ...
}

Figure 3.2: CBO value is 2.

3.4 Proposed Metrics

Ten metrics are proposed that measure different dimensions of a system when compared to CC and CBO. The metrics are grouped in a suite which we call Gray/Janzen Coupling Complexity Metrics Suite. Some of the variations on CC look more in depth on a procedure’s possible execution path rather then solely
focusing on a procedure's static nature. The variations on CBO are upon CBO's coupling weight to other classes. This weight can be different depending on exactly how a class is coupled to another. The different method complexities and possible method execution paths will be explored.

3.4.1 Transitive Cyclomatic Complexity

CC computes a complexity value over a procedure. By McCabe's definition the number of connected components (method calls) or $P$ is included within his complexity metric. It is defined as $M = E - N + 2P$ where the number of method calls is merely multiplied by two. Transitive Cyclomatic Complexity attempts to further this value. Instead of only using $2P$, TCC will inject the summation of all Cyclomatic Complexities computed on all methods that can possibly be executed on the static types invoked.

Class $C_1$, has methods $m_1, \ldots, m_n$ that are defined in the class:

\[
\begin{align*}
McCabe(m_i) &= E - N + 2P \\
CC(m_i) &= Decisions + 1 \\
TCC(C) &= \sum_{i=1}^{n} CC(T(SIM(m_i))) \\
TWMC(C) &= \sum_{i=1}^{n} McCabe(T(SIM(m_i)))
\end{align*}
\]

$E$ is the number of edges and $N$ is the number of nodes in the control flow graph of a method $m$. The plus one is used for when the CC of a method that has no decision points still maintains some complexity or more specifically a value of one.
3.4.2 Coupling-Complexity Between Objects

Coupling-Complexity Between Objects (CCBO) attempts to give a weight to a coupling between a pair of classes. This weight will be either McCabe or CC as defined in the established metrics. This is essentially a fuse between an object-oriented metric with a metric that does not measure within the object-oriented paradigm. When Class A is coupled to another through a method invocation then the CC of that method’s body is the value to be applied to the weight of the coupling between Class A and the dynamic type of the coupled class. This will generate multiple values for a coupling between two classes. The set for each pair is evaluated and is assigned the greatest value within the set. If a class is coupled to another through merely an attribute reference or a field access expression, then the value for a weight is instead given a value of one. This is justified because
public class ClassA {
    public void bar(ClassB refB) {
        while (refB.isLoop()) {
            refB.iterateLoop();
        }
    }
}

public class ClassB {
    private boolean mIsLoop;
    private ClassC mC;
    public void isLoop() {
        if (mIsLoop) {
            mC.foo();
        }
    }
    public void iterateLoop() {
        mIsLoop = !mIsLoop;
    }
}

public class ClassC {
    public void foo() {
        for (...) {
            if (...) {
                ...
            }
            else {
                ...
            }
        }
    }
}

Figure 3.3: Example Java System.

the class coupled is not utilizing anything as complex as a method, it is only accessing an attribute.
Class $C_a$, has methods $m_1, \ldots, m_n$ that couple it to Class $C_b$ or any of its descendants through polymorphism:

Let $\delta_1, \ldots, \delta_n = PIM(m_1) \ldots PIM(m_n)$

\[
CCBO - McCabe(C_a, P(\delta_i)) = \sum_{i=1}^{\mid \delta \mid} \text{Max}(McCabe(\delta_i))
\]

\[
CCBO - CC(C_a, P(\delta_i)) = \sum_{i=1}^{\mid \delta \mid} \text{Max}(CC(\delta_i))
\]

Example

Based on Figure 3.3 $CCBO - CC$ and $CCBO - McCabe$ are computed for ClassA. Class A is coupled to ClassB through method invocations upon ClassB. These two method invocations are ClassB.isLoop() and ClassB.iterateLoop().

The original definition of CBO only counts coupling to a single instance once, but in this case we can have multiple different values because the weight depends on the CC of the invoked method. Instead of counting a use of another class a single time, it will take the greatest value among the possible choices. So, the $CC$ value of isLoop() is 2 and iterateLoop() is 1, therefore it will use a value of 2 for the coupling weight for ClassA to ClassB. $CCBO - McCabe$ follows the same procedure for computation as $CCBO - CC$ except it uses McCabe instead of $CC$ to compute its $CC$ values. The $CCBO - McCabe$ value for ClassA to ClassB is 3.

3.4.3 Transitive Coupling-Complexity Between Objects

Class A can be coupled to Class B by a method invocation. However, Class B can potentially have a significant amount of efferent coupling. This can affect the reliability of Class A because it relies on the method and the stability of Class B.
Therefore the coupling between Class A and Class B is affected by the transitive relationship of the method invocation.

Class $C_a$, has methods $m_1, \ldots, m_n$ that couple it to Class $C_b$:

\[
\delta_{1..n} = \text{PIM}(m_1) \ldots \text{PIM}(m_n)
\]

\[
\text{CCBO} - \text{TWMC}(C_a, P(\delta_i)) = \sum_{i=1}^{\lvert \delta \rvert} \text{Max}(\text{TWMC}(\delta_i))
\]

\[
\text{CCBO} - \text{TCC}(C_a, P(\delta_i)) = \sum_{i=1}^{\lvert \delta \rvert} \text{Max}(\text{TCC}(\delta_i))
\]

**Example**

Based on Figure 3.3 $\text{CCBO} - \text{TCC}$ and $\text{CCBO} - \text{TWMC}$ are computed for Class A. Class A is coupled to Class B through method invocations upon Class B. These two method invocations are $\text{ClassB.isLoop()}$ and $\text{ClassB.iterateLoop()}$. The original definition of CBO only counts coupling to a single instance once, but in this case we can have multiple different values because the weight depends on the CC of the invoked method. Instead of counting a use of another class a single time, it will take the greatest value among the possible choices. So, the $\text{TCC}$ value of $\text{isLoop()}$ is 5 ($\text{isLoop()}$ makes an additional method invocation to $\text{ClassC.foo()}$) and $\text{iterateLoop()}$ it is 1, therefore it will use a value of 5 for the coupling weight for Class A to Class B. $\text{CCBO} - \text{TWMC}$ follows the same procedure for computation as $\text{CCBO} - \text{TCC}$ except it uses $\text{TWMC}$ instead of $\text{TCC}$ to compute its CC values. The $\text{CCBO} - \text{TWMC}$ value for Class A to Class B is 5.
3.4.4 \(\alpha\) Variation

A variation to the Transitive Cyclomatic Complexity is the \(\alpha\) variation. This variation takes into account the actual depth of the execution path a method invocation could potentially create. It is exactly the same as \(TCC\) except in one minor aspect. The transitive function \(T\alpha(m)\) computes the CC for each method in the transitive closure. However, at each computation of the CC it is multiplied by an \(\alpha\) value. This value is the current height of the call stack. For example, if method \(m_0\) invokes \(m_1\) (height is one), \(m_1\) invokes \(m_2\) (height is two), then \(m_2\) returns (height is one) and so on. Given this new CC definition it creates four new metrics:

\[
TCC\alpha(C_1) = \sum_{i=1}^{n} CC(T\alpha(SIM(m_i)))
\]

\[
TWMC\alpha(C_1) = \sum_{i=1}^{n} McCabe(T\alpha(SIM(m_i)))
\]

Let \(\delta_{1, n} = PIM(m_1) \ldots PIM(m_n)\)

\[
CCBO - TWMC\alpha(C_a, P(\delta_i)) = \sum_{i=1}^{|\delta|} \max(TWMC\alpha(\delta_i))
\]

\[
CCBO - TCC\alpha(C_a, P(\delta_i)) = \sum_{i=1}^{|\delta|} \max(TCC\alpha(\delta_i))
\]

Example

Based on Figure 3.3 \(TCC\alpha, TWMC\alpha, CCBO - TCC\alpha,\) and \(CCBO - TWMC\alpha\) are computed for ClassA. \(TCC\alpha\) for ClassA is calculated first computing the \(TCC\alpha\) value for ClassA.bar. ClassA.bar itself has a CC value of
2. `ClassB.isLoop()` has a `CC` value of 2 multiplied by the number of method invocations before reaching this body which is two, resulting in a value of 4. Which in turn invokes `ClassC.foo` which has a `CC` value of 3 multiplied by the number of method invocations before reaching this body which is 3. The final `CC` value for `ClassC.foo` is then 15. This results in 15 plus the `CC` value of `ClassB.iterateLoop` which is 1 multiplied by the `α` to get 2, so the `TCCα` value for `ClassA` is 17. The `TWMCα` value is similar except `McCabe` is used to compute the `CC` instead of `CC` resulting in a `TWMCα` value of 17 for `ClassA`.

    Class A is coupled to ClassB through method invocations upon ClassB. These two method invocations are `ClassB.isLoop()` and `ClassB.iterateLoop()`. The original definition of CBO only counts coupling to a single instance once, but in this case we can have multiple different values because the weight depends on the `CC` of the invoked method. Instead of counting a use of another class a single time, it will take the greatest value among the possible choices. The `TCCα` value of `isLoop()` is 8 (`isLoop()` makes an additional method invocation to `ClassC.foo()`). `isLoop()` has a value of 2 times the number of method calls required to get to this invocation which is one, so the value is still 2. In addition, `isLoop()` invokes `ClassC.foo()` which has a value of 3 but multiplied by 2 since it requires two method invocations to get to this method, resulting in a value of 6. Therefore, the `TCCα` value of `isLoop()` is 8. The value of `iterateLoop()` is 1, which is the smaller of the two so it will use a value of 8 for the coupling weight for `ClassA` to `ClassB`. `CCBO – TWMCα` follows the same procedure for computation as `CCBO – TCCα` except it uses `TWMCα` instead of `TCCα` to compute its `CC` values. The `CCBO – TWMCα` value for `ClassA` to `ClassB` is 7.
Chapter 4

Evaluation

The proposed metrics are empirically evaluated on specific versions of source code in open-source projects from industry. Two open-source Java projects are analyzed. ConnectorJ (http://www.mysql.org/products/connector/j) is a JDBC driver and is analyzed for versions 3.0.0(development), 3.1.0(alpha), and 5.1.0(alpha). The second project is Hibernate (http://www.hibernate.org), an object-relational mapping (ORM) library for the Java language, and is analyzed for version 3.0.0(alpha). The proposed metrics are calculated for each source code base and the results consider if a correlation exists between the metrics and the number of defects identified per class.

4.1 Source Control

The source control or Subversion (http://subversion.tigris.org) logs are searched for any bug or defect fixes up until the next release. Defects are identified using keywords commonly used in corresponding fix log messages. For example in Hibernate 3.0.0 alpha, log messages for defect fixes generally contained one of the
following keywords: fix, resolve, error, defect, bug, issue, or anything else that appears to be a pattern in the Subversion logs.

A web-utility called SVNsearch (http://www.svnsearch.org) was used to search the Subversion logs. SVNsearch has a database filled with searchable logs for a number of project repositories. For example if the source for version 1.0 is analyzed for Project X then the Subversion logs from version 1.0 until version 1.1 are searched for all bugs and defects. If a fix log is found, then the number of defects for the associated class is incremented by one, signifying that the class contained a defect.

### 4.1.1 Defect Log Example

5029: April 27, 2007: 20:10:09
Fix BUG#HIB379 Resolved issue regarding off-by-one error on structure X.

/trunk/Hibernate3/src/.../FunctionalClassA.java
/trunk/Hibernate3/src/.../FunctionalClassB.java
/trunk/Hibernate3/src/.../FunctionalClassD.java
/trunk/Hibernate3/src/.../FunctionalClassC.java

4079: April 01, 2007: 10:03:42
Fix BUG#HIB429 errored on startup
/trunk/Hibernate3/src/.../FunctionalClassA.java
/trunk/Hibernate3/src/.../FunctionalClassE.java

Figure 4.1: Example Defect Log
In Figure 4.1 an example defect log is shown. The example contains two defects that were fixed. The horizontal line signifies a separation of a revision. In each revision a set of Java files is associated with each defect fixed. In both of the revisions “FunctionalClassA.java” is involved with two defects, this would result in the “FunctionalClassA.java” having 2 defects while the other classes would only have a defect of 1. The remainder of the log is examined as follows which ultimately gives us a Defects per Class (DPC) metric to use in the analysis.

4.2 ConnectorJ

ConnectorJ is a JDBC driver for MySQL written in Java. It was chosen because it was open-source, has a publicly available source control system and is tied to a commercial product which demands reliability and features (databases). Three versions are analyzed, 3.0.0, 3.1.0 and the more recent at the time of this writing 5.1.0. Version 3.0.0 is an interesting release because in the change log it is given the tag of “dev” short for a development release, while the publicly available source is given a “beta” tag. Version 3.1.0 is given an “alpha” tag and 5.1.0 is given “alpha” in the change log but no suffix in the publicly available source code.

4.3 Hibernate

Hibernate is an object-relational mapping (ORM) library for the Java language. It is an open-source project that has publicly available source code and is developed for a system that demands reliability and features (databases). A single version is analyzed; version 3.0.0 is an alpha release according to the de-
development change logs.

4.4 Eclipse Plug-in

The proposed metrics were implemented using the Eclipse (http://www.eclipse.org) Plug-in framework. This framework was chosen because of the Java language parser available that was able to parse the Third Java Language Specification (JLS3). Eclipse is an Integrated Development Environment (IDE) which allows any of the proposed metrics to be run while a developer is modifying their code. This gives the developer immediate feedback on certain aspects or relationships that he/she may have created.

4.4.1 Verification of Accuracy

Defining metrics and implementing them are two different things. Since these metrics are not trivial metrics to measure it is possible for measurements to be incorrect. To first test and verify that the implemented metrics were correct the basis on which the proposed metrics are built upon must first be verified; these being CC and CBO. A series of comparisons is done between an open-source calculator and the implemented Eclipse plug-in. These comparisons consisted of example systems to test all aspects of each measurement.

4.4.2 Cyclomatic Complexity

There is an existing metrics plug-in for Eclipse that measures the Cyclomatic Complexity for a class. This plug-in is called Metrics (http://metrics.sourceforge.net). A comparison was done between the Metric’s calculations and $CC$ and $McCabe$. 
CC had a slight difference because it was found that CC counted expressions within `assert` statements when Metric's computation did not include these in the end result. McCabe was different from the Metric's complexity metric because it did not seem to include the connected components of the graph and multiply them by two. If McCabe doesn’t account for the $2P$ within McCabe’s original definition then it matches CC and Metric’s complexity metric perfectly. However, since the goal was to use both popular ways to compute Cyclomatic Complexity McCabe was left in its original state.

4.4.3 Coupling Between Objects

There is an open-source Java bytecode parser which uses the Apache Byte Code Engineering Library (http://jakarta.apache.org/bcel). The tool is called Chidamber Kemerer Java Metrics (ckjm) which can be found at (http://www.spinellis.gr/sw/ckjm). It computes the entire Chidamber and Kemerer metric suite along with the number of public methods per class and afferent coupling (similar to the efferent CBO). This tool was used to verify that the implemented CBO representation was accurate at measuring CBO correctly. However, there was a discrepancy between the two tools. It was discovered that the tool did not account for the possible polymorphic behavior of a reference in the Java language. Due to this possible coupling nature between a static reference type and any of its polymorphic counterparts it was left as is.

4.4.4 Proposed Metrics

The proposed metrics were then under the assumption that the underlying CC and Coupling Between Object metrics were accurate at measuring their aspects
of the system. The proposed metrics that were built on top of these base metrics were then verified by hand and by a series of acceptance tests. The acceptance tests are a hand calculated metric result for an example system which would then be compared to the metric calculating the result real-time.

4.5 Hypotheses

For this experiment there are two hypotheses that are formulated:

**H1:** The metrics CCBO-McCabe, and CCBO-CC predict reliability of objects better than CBO alone.

**H2:** The metrics TCC, TWMC, CCBO-TCC, CCBO-TWMC, and the $\alpha$ variations which account for the transitive relationships among method invocations predict reliability of objects better than CC or CBO alone.
Chapter 5

Results & Analysis

All of the measured metrics have some positive correlation with defects among classes. WMC-McCabe, TWMC, and CCBO-CC are the top three metrics at predicting class reliability from the ConnectorJ case study. The top three metrics show consistent performance compared to the other metrics for the analysis of the three versions of ConnectorJ. The WMC metrics were an interesting result as they are criticized for not being able to accurately measure complexity within an object-oriented environment [37]. This was shown not to be the case here where often a WMC-based metric was superior to a CBO-based one.

5.1 Correlation

Spearman’s rank correlation was applied to determine each metric’s ability to predict reliability. Spearman’s was chosen instead of Pearson because Pearson’s correlation assumes a linear relation with the data to begin with; this is not the case with software metrics and defects.
The metrics will be categorized by WMC-based and CBO-based metrics. In all versions of ConnectorJ WMC-McCabe and CCBO-CC are the most accurate at predicting class quality in their respective categories as shown in Table 5.1. However, the single analysis for Hibernate produced different results. It is a different code base which counts for the expected change, but the top performing metrics in either the WMC or CBO category change to CBO and TCC as being the top performers. The difference between CBO’s and CCBO-CC’s correlation with class quality for the Hibernate evaluation is small, the difference is 0.009 with TCC and WMC-McCabe being larger at 0.56. This could be caused by the fact that Hibernate 3.0.0 has significantly more classes than ConnectorJ did even in its largest code base (5.1.0), 669 versus 175.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Conn. J 3.0.0</th>
<th>Conn. J 3.1.0</th>
<th>Conn. J 3.1.0</th>
<th>Hibernate 3.0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC-McCabe</td>
<td>0.808*</td>
<td>0.757*</td>
<td>0.569*</td>
<td>0.393</td>
</tr>
<tr>
<td>WMC-CC</td>
<td>0.736</td>
<td>0.657</td>
<td>0.519</td>
<td>0.444</td>
</tr>
<tr>
<td>CCBO-CC</td>
<td>0.658*</td>
<td>0.479*</td>
<td>0.528*</td>
<td>0.356</td>
</tr>
<tr>
<td>CCBO-McCabe</td>
<td>0.621</td>
<td>0.423</td>
<td>0.488</td>
<td>0.352</td>
</tr>
<tr>
<td>CCBO-TCC</td>
<td>0.617</td>
<td>0.479</td>
<td>0.508</td>
<td>0.355</td>
</tr>
<tr>
<td>CCBO-TWMC</td>
<td>0.590</td>
<td>0.425</td>
<td>0.482</td>
<td>0.350</td>
</tr>
<tr>
<td>CCBO-TCCα</td>
<td>0.602</td>
<td>0.471</td>
<td>0.494</td>
<td>0.355</td>
</tr>
<tr>
<td>CCBO-TWMCα</td>
<td>0.580</td>
<td>0.421</td>
<td>0.471</td>
<td>0.350</td>
</tr>
<tr>
<td>CBO</td>
<td>0.598</td>
<td>0.377</td>
<td>0.509</td>
<td>0.365*</td>
</tr>
<tr>
<td>TCC</td>
<td>0.728</td>
<td>0.651</td>
<td>0.514</td>
<td>0.449*</td>
</tr>
<tr>
<td>TWMC</td>
<td>0.792</td>
<td>0.733</td>
<td>0.544</td>
<td>0.388</td>
</tr>
<tr>
<td>TCCα</td>
<td>0.715</td>
<td>0.658</td>
<td>0.513</td>
<td>0.446</td>
</tr>
<tr>
<td>TWMCα</td>
<td>0.778</td>
<td>0.730</td>
<td>0.530</td>
<td>0.387</td>
</tr>
</tbody>
</table>

Table 5.1: Correlation of software metrics and class quality for ConnectorJ and Hibernate. Asterisk (*) represents which metrics are most correlated with predicting class quality in either the WMC or CBO modification metrics.
5.2 Linear Regression

In Figure 5.1 and 5.2 linear regressions are done on WMC-McCabe and CCBO-CC against the number of defects per class for ConnectorJ 3.0.0 respectively. WMC-McCabe has a 0.808 correlation and a strong linear relationship with the number of defects per class. CCBO-CC has a 0.658 correlation and a strong linear relationship with the number of defects per class. Through the creation of a line plot it was immediately obvious that there was an outlier with a number of defects per class at the value of fourteen. The offending class is “StringUtils.java” which only has a few public static functions that are the target of several changes due to the nature of strings or merely for formatting and other issues.

Through linear regression we can use the standard equation for a line to be
able to predict future values of $y$ for any given $x$. However from these results we can state that using an equation of a line a mediocre predictor for CCBO-CC and WMC-McCabe. The mediocre estimation is due to the fact that there aren’t many points in the upper region (many are duplicated or close enough in proximity to another in the lower region) to provide much evidence for a strong linear relationship between the number of defects and either of the metrics. Although, CCBO-CC appears to be better suited for linear approximation. This is because its y-intervals are smaller and the higher points are better correlated with the best fit line when compared to WMC-McCabe’s.

![Simple Linear Regression](image)

**Figure 5.2:** Linear regression for CCBO-CC against Number of Defects per Class for ConnectorJ 3.0.0
5.3 Univariate Analysis

The metrics CCBO-CC and WMC-McCabe are analyzed individually because they tend to be the metrics best correlated with predicting class quality. Each metric is broken up into a set of intervals for its output values and it is used to illustrate different aspects of the data. The number of classes and defects are depicted for each interval. Finally a ratio of the number of defects per class is then computed by using the number of changes divided by the number of classes. This results in an average number of defects per class within this interval. This does not mean that every individual class within this range has the ratio; it is entirely possible for one class to have all the defects within a given interval. This number merely represents a reasonable expectation of the number of defects to find within any given class in this interval.

### 5.3.1 ConnectorJ 3.0.0

In Table 5.2, values above 40 have a high ratio of defects per class (10+). The 0–19 range has an additional 14 defects because of the anomaly discussed earlier related to the “StringUtils.java” class. In Table 5.3, there is a relationship where if the value of WMC-McCabe increases so does the ratio of defects per
### Table 5.3: Univariate analysis for WMC-McCabe calculated for ConnectorJ 3.0.0.

<table>
<thead>
<tr>
<th>WMC-McCabe</th>
<th>Classes</th>
<th>Defects</th>
<th>defects per class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–49</td>
<td>26</td>
<td>41</td>
<td>0.634</td>
</tr>
<tr>
<td>50–99</td>
<td>3</td>
<td>9</td>
<td>3.0</td>
</tr>
<tr>
<td>100–149</td>
<td>3</td>
<td>25</td>
<td>8.333</td>
</tr>
<tr>
<td>150–199</td>
<td>2</td>
<td>24</td>
<td>12.0</td>
</tr>
<tr>
<td>200–249</td>
<td>2</td>
<td>47</td>
<td>23.5</td>
</tr>
<tr>
<td>250–299</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>300+</td>
<td>3</td>
<td>66</td>
<td>22.0</td>
</tr>
</tbody>
</table>

### Table 5.4: Univariate analysis for CCBO-CC calculated for ConnectorJ 3.1.0.

<table>
<thead>
<tr>
<th>CCBO-CC</th>
<th>Classes</th>
<th>Defects</th>
<th>defects per class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–19</td>
<td>46</td>
<td>120</td>
<td>2.609</td>
</tr>
<tr>
<td>20–39</td>
<td>3</td>
<td>25</td>
<td>8.334</td>
</tr>
<tr>
<td>40–59</td>
<td>1</td>
<td>82</td>
<td>82.0</td>
</tr>
<tr>
<td>60–79</td>
<td>2</td>
<td>99</td>
<td>49.5</td>
</tr>
<tr>
<td>80–99</td>
<td>2</td>
<td>66</td>
<td>33.0</td>
</tr>
<tr>
<td>100–119</td>
<td>2</td>
<td>87</td>
<td>43.5</td>
</tr>
<tr>
<td>120+</td>
<td>1</td>
<td>75</td>
<td>75.0</td>
</tr>
</tbody>
</table>

class. There are 2 ranges where there are no classes within the range but have been kept for interval consistency. Values above 150 show a high ratio of defects per class.

#### 5.3.2 ConnectorJ 3.1.0

In Table 5.4 CCBO-CC is analyzed for ConnectorJ 3.1.0. Immediately the interval 40–59 stands out due to its extreme number of defects for one class. The class was investigated to be called “ResultSet.java”. This is no surprise as ResultSet is a very important class to database querying and it is likely for this class to be defect-prone. A value of 20+ for CCBO-CC and 125+ for WMC-McCabe show a significant decrease in class quality as each metric value rises
### Table 5.5: Univariate analysis for WMC-McCabe calculated for Hibernate ConnectorJ 3.1.0

<table>
<thead>
<tr>
<th>WMC-McCabe</th>
<th>Classes</th>
<th>Defects</th>
<th>$\frac{\text{defects}}{\text{class}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–124</td>
<td>47</td>
<td>129</td>
<td>2.745</td>
</tr>
<tr>
<td>125–249</td>
<td>3</td>
<td>66</td>
<td>22.0</td>
</tr>
<tr>
<td>250–374</td>
<td>1</td>
<td>20</td>
<td>20.0</td>
</tr>
<tr>
<td>375–499</td>
<td>3</td>
<td>113</td>
<td>37.666</td>
</tr>
<tr>
<td>500–624</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>625–749</td>
<td>2</td>
<td>118</td>
<td>59.0</td>
</tr>
<tr>
<td>750+</td>
<td>1</td>
<td>82</td>
<td>82.0</td>
</tr>
</tbody>
</table>

### Table 5.6: Univariate analysis for CCBO-CC calculated for ConnectorJ 5.1.0

<table>
<thead>
<tr>
<th>CCBO-CC</th>
<th>Classes</th>
<th>Defects</th>
<th>$\frac{\text{defects}}{\text{class}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–49</td>
<td>159</td>
<td>21</td>
<td>0.132</td>
</tr>
<tr>
<td>50–99</td>
<td>5</td>
<td>8</td>
<td>1.6</td>
</tr>
<tr>
<td>100–149</td>
<td>1</td>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>150–199</td>
<td>1</td>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>200–249</td>
<td>2</td>
<td>5</td>
<td>2.5</td>
</tr>
<tr>
<td>250–299</td>
<td>3</td>
<td>25</td>
<td>8.333</td>
</tr>
<tr>
<td>300+</td>
<td>4</td>
<td>20</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Table 5.6: Univariate analysis for CCBO-CC calculated for ConnectorJ 5.1.0.

beyond this threshold value.

### 5.3.3 ConnectorJ 5.1.0

In Tables 5.6 and 5.7 CCBO-CC and WMC-McCabe are analyzed for ConnectorJ 5.1.0 which has a significantly lower number of defects overall when compared to ConnectorJ 3.0.0. This could be due to ConnectorJ 3.0.0 being marked as a development release or perhaps 5.1.0 is a more sound system when compared to 3.0.0. A value of 100 or more for CCBO-CC and 600 or more show a high ratio of defects per class when compared to other intervals.
Table 5.7: Univariate analysis for WMC-McCabe calculated for ConnectorJ 5.1.0.

<table>
<thead>
<tr>
<th>WMC-McCabe</th>
<th>Classes</th>
<th>Defects</th>
<th>defects class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–199</td>
<td>156</td>
<td>21</td>
<td>0.135</td>
</tr>
<tr>
<td>200–399</td>
<td>7</td>
<td>10</td>
<td>1.429</td>
</tr>
<tr>
<td>400–599</td>
<td>2</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>600–799</td>
<td>4</td>
<td>13</td>
<td>3.25</td>
</tr>
<tr>
<td>800–999</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1000–1199</td>
<td>3</td>
<td>21</td>
<td>7.0</td>
</tr>
<tr>
<td>1200+</td>
<td>3</td>
<td>20</td>
<td>6.667</td>
</tr>
</tbody>
</table>

Table 5.8: Univariate analysis for CCBO-CC calculated for Hibernate 3.0.0alpha.

<table>
<thead>
<tr>
<th>CCBO-CC</th>
<th>Classes</th>
<th>Defects</th>
<th>defects class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–39</td>
<td>619</td>
<td>244</td>
<td>0.394</td>
</tr>
<tr>
<td>40–79</td>
<td>31</td>
<td>61</td>
<td>1.968</td>
</tr>
<tr>
<td>80–119</td>
<td>10</td>
<td>28</td>
<td>2.8</td>
</tr>
<tr>
<td>120–159</td>
<td>7</td>
<td>33</td>
<td>4.714</td>
</tr>
<tr>
<td>160–199</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>200–239</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>240+</td>
<td>1</td>
<td>6</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Table 5.9: Univariate analysis for WMC-McCabe calculated for Hibernate 3.0.0alpha.

<table>
<thead>
<tr>
<th>WMC-McCabe</th>
<th>Classes</th>
<th>Defects</th>
<th>defects class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–149</td>
<td>646</td>
<td>304</td>
<td>0.471</td>
</tr>
<tr>
<td>150–299</td>
<td>12</td>
<td>26</td>
<td>2.167</td>
</tr>
<tr>
<td>300–449</td>
<td>6</td>
<td>18</td>
<td>3.0</td>
</tr>
<tr>
<td>450–599</td>
<td>3</td>
<td>7</td>
<td>2.333</td>
</tr>
<tr>
<td>600–749</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>750–899</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>900+</td>
<td>2</td>
<td>17</td>
<td>8.5</td>
</tr>
</tbody>
</table>
5.3.4 Hibernate 3.0.0

In Tables 5.8 and 5.9 a similar analysis is conducted on Hibernate 3.0.0. Hibernate is a completely different code base compared to ConnectorJ. However, it still shows similar increases in defects per class as ConnectorJ does. Values over 40 for CCBO-CC and values of 150 or more for WMC-McCabe show a higher ratio of defects per class when compared to lower intervals.

5.4 Threats to Validity

The “Number of Defects” metric did not accurately represent the possibility of logic being moved to new files. If a defect was fixed by creating or moving logic into a newly created file this file would then exist after the strict source release. These discrepancies are not accounted for in the “Number of Defects” metric due to its complexity. This means that it is possible for there to be classes that are involved with defects unaccounted for.

The “Number of Defects” metric is constructed by searching SVN logs for a defect associated with a group of files. Then associating that group of modified files with a particular defect. The developer is expected to only fix exactly what they state within their change log. It is possible that a developer states a fix to a particular defect, but could fix multiple defects with a single commit to the repository, where this would result in counting multiple defect fixes as a singular defect fix. This is because merely traversing and verifying each log would take an enormous amount of time. This problem is ignored, but stated here for clarification.

In Java it is possible to declare inner-classes and these logistically can be
separate from the containing class' functionality. An inner-class may be declared because it is unnecessary for it to exist as its own class. If an inner-class is part of a defect change its outer most class will be the name of the file and thus be the one associated with the defect. In addition, CC values includes Java’s notion of anonymous inner-classes. Instead of declaring these separate from the container class, any additional value computed from the statements within the anonymous inner-class are added to the container class' CC value.

5.5 Analysis

The values are significantly different when comparing either ConnectorJ 3.0.0 or Hibernate 3.0.0 to ConnectorJ 5.1.0. The metric values might be different, but when put into intervals, an increase in defects per class can be witnessed at equivalent locations in both tables. This can lead to the notion that a specific threshold value is not applicable, but the entire range of values must be taken into account in order to identify class quality accurately.

Viewing from a higher perspective, as the value of both metrics grow greater the class quality decreases and defects per class rises. For CCBO-CC, most of the classes that are in the lower interval and the ratio per class is less than 1.0 which indicates that these classes are not highly connected within the overall system and are less prone for error. However, as the CCBO-CC metric value rises, the highly coupled classes show a trend to have a greater chance at containing defects compared to the lower coupled classes.

Two hypotheses were stated regarding ten new coupling-complexity metrics. **H1** is supported by showing that modifying CBO with CC has a better ability to predict class quality than the original unmodified CBO metric. Of the many
modified variations of CBO (e.g. CCBO-CC, CCBO-McCabe, etc.), CCBO-CC is the best among the variations. **H2** has only shown an inconsistent pattern of predicting class quality accurately when taking into account the transitive relationship among method invocations. In some instances one of the modified metrics of CC would either perform better or worse than the original definition. For example, with Hibernate 3.0.0 the transitive variation on CC (TCC) performs better. However, in all instances of ConnectorJ it is consistently worse than WMC-McCabe, but not by much. Therefore **H2** is rejected.
Chapter 6

Conclusions & Future Work

Developing software is complex and everyone is looking for a solution to help aid in this endeavor. A whole field has been dedicated to the search for software measurements to be able to accurately predict particular aspects of a system. Being able to predict a class’ quality has been one of the goals of software metrics. There have been several proposals [6, 7, 12, 17, 28, 27, 19, 9] and validations [4, 21, 39, 40, 34, 20] in order to provide accurate predictions of class quality. As of now there is no oracle, however the seminal metrics have been tried and true compared to other more recent additions.

6.1 Contribution

Ten metrics are introduced and collectively named the Gray/Janzen metric suite. These metrics are modified versions of Thomas McCabe’s [30] Cyclomatic Complexity (CC) and Chidamber and Kemerer’s [12] Coupling Between Objects (CBO). The modifications made to CC are accounting for the transitive relation among all possible method invocations beginning at a particular method. This
measures a different aspect compared to the original in that it accounts for the potential splitting of logic among objects in an object-oriented system. In an object-oriented system, logic is not in a single procedure, but modularized into components within the system. A programmer’s memory is also affected when logic is not in a single procedure, causing them to remember states between program flow across modules, increasing the complexity and burden on the programmer. The modifications made to CBO give a coupling between a pair of classes a weight. Originally they are given a fixed value of one, but instead this value is replaced with the CC of the corresponding coupled method. \( \alpha \) variations are also defined for each metric introduced. These \( \alpha \) variations are a minor alteration in that depending on the statically built invocation record of methods, the deeper a method call is from its originating invocation the more program states a programmer has to infer; this is taken into account when measuring these metric variations.

6.2 Validation

These metrics are implemented in the Eclipse plug-in framework and take advantage of the Java language parser that is built into the Java Development Tools. The metrics are verified by comparing them to an open-source Java project that computes the Chidamber and Kemerer metric suite called ckjm \(^1\) and a plug-in for Eclipse called Metrics \(^2\) which calculates the CC. The metrics are then evaluated on two industry open-source projects known as ConnectorJ and Hibernate. These are two projects that revolve around databases and databases have a high demand for reliability. Three versions are evaluated for ConnectorJ

\(^1\)http://www.spinellis.gr/sw/ckjm
\(^2\)http://metrics.sourceforge.net
Subversion logs are scraped for each project and compared to a set of substrings that is created from a sample of defect revision logs. For example for ConnectorJ 3.0.0 the set \{"fix", "defect", "bug", "resolve", "error", "issue"\} was used to scrape through the Subversion repository. For each revision that was marked as a defect its group of classes associated would all have their "Number of Defects" counter incremented. This variable would then be correlated with the output of each metric.

6.3 Results

Viewing from a higher perspective, as the value of both metrics grow greater the class quality decreases and defects per class rises. For CCBO-CC, most of the classes that are in the lower interval and the ratio per class is less than 1.0 indicates that these classes are not highly connected within the overall system and are less prone for error. However, as the CCBO-CC metric value rises, the highly coupled classes show a trend to have a greater chance at containing defects compared to the lower coupled classes.

Two hypotheses were stated regarding ten new coupling-complexity metrics. H1 is supported by showing that modifying CBO with CC has a better ability to predict class quality than the original unmodified CBO metric. Of the many modified variations of CBO (e.g. CCBO-CC, CCBO-McCabe, etc.), CCBO-CC is the best among the variations. H2 has only showed an inconsistent pattern of predicting class quality accurately when taking into account the transitive relationship among method invocations. In some instances one of the modified metrics of CC would either perform better or worse than the original definition.
An example, is for Hibernate 3.0.0 the transitive variation on CC (TCC) performs better. However, in all instances of ConnectorJ it is consistently worse than WMC-McCabe, but not by much. Therefore \textbf{H2} is rejected.

6.4 Future Work

Metrics are never fully validated until they have been tried and tested for many projects. These metrics have only been used on two industry projects totalling four different analyses across four versions. This is hardly a conclusive result for absolute proof of their accuracy. Future experiments can further investigate the accuracy of these metrics with other projects.

An experimental group could volunteer to use these metrics in developing software and supply feedback of their use. Analyses could be performed on the amount of work that is generated from the results of the metrics as well as another measurement judging the impact of this work on the overall system. Some examples could include showing the CC values of overall methods within the system and comparing them at a later date to see if the CC has decreased in particular classes. If this is the case it could be classified that this decrease was in response to a value that surpassed a particular threshold for a metric.

Threshold values for metrics are a hard goal. As discussed previously it is thought to be the case that the entire range of the metric’s values must be taken into account when creating a threshold value. An algorithm for figuring out the threshold value of a metric depending on its current range of values would be an interesting piece of research.
Bibliography


