The Effects of Software Design Complexity on Defects

A Study in Open-Source Systems

THE AUSTRALIAN NATIONAL UNIVERSITY

A thesis submitted for the degree of Doctor of Philosophy of The Australian National University

Normi Sham Awang Abu Bakar
February 2011
I declare that the work in this thesis is entirely my own and that to the best of my knowledge it does not contain any materials previously published or written by another person except where otherwise indicated.

Normi Sham Awang Abu Bakar
28 February 2011
This thesis is dedicated to my late parents. May god bless you, always.
Acknowledgements

First and foremost, I would like to thank my family, especially my late parents for the love and support they had given me all my life. Their spirit and courage will always live in me and I am eternally grateful for all the times that we spent together. Also, to my siblings, Nor, Hisyam, Khairul, Ina, Anwar, Muni and Amir, thank you for always being there for me.

My special thanks goes to my supervisor, Dr. Clive Boughton, for the constant support that he has given me since the beginning of my PhD. Four years ago, I remember standing nervously outside his office before our first meeting, not knowing what to expect. Then, I discovered what a nice person he is, patiently guiding his students in learning the art of doing research. He is more like a father figure to us, his students, and I would like to thank him for all his help during the conduct of this research. He has contributed so much in this work and has given a lot of effort towards the completion of this thesis. This research is a collaborative work by both of us and any publications that resulted from this research are jointly written. I will always remember his guidance and kindness.

And not forgetting everyone in the Computer Science Department, especially Shayne for providing the template for this thesis, Srinivas for his help with various LaTeX 'issues', and for providing constant support and assistance for me, Agung for being a very helpful friend, Dr. Chris Johnson for his help in reviewing my thesis and all his constructive advice towards the completion of this thesis, not forgetting Lynette, Julie, Deanne, Pam, James and other TSG people, Ziyad, Alvin, Kshitij, Ian, Tim, Rainbow, Henry and Zoe. My appreciation also goes to Hwan-Jin Yoon for guiding me in doing the statistical analysis and everyone who has helped me during my study at ANU.

My gratitude is extended to Professor Barbara Kitchenham, whose help and advice have shed some light into the statistical analysis in this research. Also, to Dr. David Tegarden who has shared his knowledge in various aspects of software measurement.

To my friends: Syida, Kak Ana and family, Idham, Mosfi, Ifa, Su and Nadim, Siti, Khairil, Faie and family, thank you for the friendship and support you all have given me. Similarly, to Cheryl, Donna, Tracey, Preity, Sohela, Kim and Debbie, thank you for everything.

The journey from the beginning until the completion of this thesis has been a wonderful experience to me, and I owe this to everyone who has made it possible for me to complete this chapter of my life. The lessons learnt from this experience are invaluable and have paved the way towards embarking on future research career.
Abstract

The aim of this thesis is to investigate whether there is a general correlation between post-delivery defects and system design complexity by studying measures relating to Data, Structural and Procedural complexity in object-oriented systems and determining their effect on post delivery defects. A further aim is to determine whether, during the detailed design phase, measured Data Complexity can estimate measured Procedural Complexity and Class Size for the implemented system.

This research is based on prior work of Card and Glass, who introduced a System Complexity Model as a combination of Structural and Data Complexity. They applied their model to eight similar FORTRAN (RATFOR) systems. This research both investigates and extends the Card and Glass model for applying to the object-oriented environment. Several adjustments are made to accommodate important characteristics of object-oriented design and language, such as "inheritance" and "encapsulation". Based on these adjustments, a new System Complexity Model is proposed, which is then applied to 104 open-source systems to investigate its effectiveness in estimating post-delivery defects. The necessary data are extracted from the source code of systems maintained within SourceForge - a popular open-source repository. Included in the data are, Version Downloads and the Number of Developers considered as independent variables for predicting user reported defects.

The Spearman's rank correlation coefficient and Generalized Linear Model (GLM) with Poisson distribution are used to analyze the collected data. The results show that the newly proposed System Complexity (Structural + Data) is not significant for estimating the volume of post-delivery defects (Post-DD). When Structural and Data Complexity are analyzed separately, the results show that Structural Complexity is highly significant in estimating the number of post-DDs. Other important findings include:
1) Data Complexity can effectively estimate Procedural Complexity and Class Size, 2) The ratio of System Complexity and Procedural Complexity is useful for estimating the probability of Defect Density and Class Size. This ratio represents the mapping of metrics obtained during the detailed design phase with Procedural Complexity which is measurable during implementation (writing of the source code).
# Contents

Acknowledgements .......................... v
Abstract ....................................... vii

## I Research Overview

1 Introduction ................................ 3
   1.1 Background .............................. 4
   1.2 Research Objectives ...................... 4
   1.3 Research Questions ...................... 5
   1.4 Research Hypotheses ..................... 5
   1.5 Thesis Outline ........................... 5

2 Literature Review .......................... 9
   2.1 Introduction .............................. 10
   2.2 Software Engineering and Quality ........ 11
   2.3 Empirical Theory ......................... 14
   2.4 Measurement Theory and Framework ....... 17
   2.5 Open-Source Research ..................... 26
   2.6 Software Metrics ........................ 32
   2.7 Quality Prediction Modeling .............. 54
   2.8 Summary ................................. 59

## II Research Methodology and Implementation

3 Research Background ...................... 65
   3.1 Background .............................. 66
   3.2 Object-oriented Concepts in Java ......... 67
   3.3 Complexity in Java Systems .............. 68
   3.4 Proposed Model ......................... 72

4 Research Methodology ..................... 77
   4.1 Research Methodology .................... 78
   4.2 Data Collection .......................... 84
   4.3 Data Source ............................. 86
   4.4 Metric Extraction Tools ................. 89
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5 Data Collection Process                                             91</td>
</tr>
<tr>
<td>4.6 Summary                                                             108</td>
</tr>
<tr>
<td>III Results Analysis and Discussion                                     109</td>
</tr>
<tr>
<td>5 Data Analysis                                                         111</td>
</tr>
<tr>
<td>5.1 Introduction                                                        112</td>
</tr>
<tr>
<td>5.2 Statistical Modeling                                                112</td>
</tr>
<tr>
<td>5.3 Procedure for Data Analysis                                         113</td>
</tr>
<tr>
<td>5.4 Analysis Results                                                    117</td>
</tr>
<tr>
<td>5.5 Results Comparison with using Card and Glass Formula                144</td>
</tr>
<tr>
<td>5.6 Conclusion                                                          153</td>
</tr>
<tr>
<td>6 Results Discussion                                                    155</td>
</tr>
<tr>
<td>6.1 Introduction                                                        156</td>
</tr>
<tr>
<td>6.2 Results Discussion                                                  156</td>
</tr>
<tr>
<td>6.3 Findings in Relation to the Research Questions                      177</td>
</tr>
<tr>
<td>6.4 Limitations                                                         180</td>
</tr>
<tr>
<td>6.5 Potential Benefits to Industry                                      181</td>
</tr>
<tr>
<td>6.6 Challenges and Lessons Learnt                                       182</td>
</tr>
<tr>
<td>6.7 Threats to Validity                                                 183</td>
</tr>
<tr>
<td>7 Conclusions and Future Work                                           185</td>
</tr>
<tr>
<td>7.1 Conclusion                                                          186</td>
</tr>
<tr>
<td>7.2 Contribution of Results to Software Engineering Industry            188</td>
</tr>
<tr>
<td>7.3 Research Questions from the Results of This Research                190</td>
</tr>
<tr>
<td>7.4 Future Work                                                         190</td>
</tr>
<tr>
<td>A Research Data                                                         193</td>
</tr>
<tr>
<td>B Questions for First Survey                                            199</td>
</tr>
<tr>
<td>C Questions for Second Survey                                           203</td>
</tr>
<tr>
<td>D Metrics Extraction Tools                                              207</td>
</tr>
<tr>
<td>D.1 Logiscope                                                           209</td>
</tr>
<tr>
<td>D.2 Resource Standard Metrics (RSM)                                     209</td>
</tr>
<tr>
<td>D.3 JStyle                                                              210</td>
</tr>
<tr>
<td>D.4 Chidamber and Kemerer Java Metrics (CKJM)                           211</td>
</tr>
<tr>
<td>D.5 JHawk                                                               212</td>
</tr>
<tr>
<td>Bibliography                                                            213</td>
</tr>
<tr>
<td>x</td>
</tr>
</tbody>
</table>
List of Figures

3.1 Decomposition Model of System Design ........................................... 66
3.2 Package Representation in Java ....................................................... 69
3.3 Structural Complexity of Java Classes .............................................. 69
3.4 Class Representation in Java .......................................................... 70
3.5 Fan-out of Java Methods ................................................................. 70
3.6 Overall Complexity of Java Classes “Relationships” .............................. 71
3.7 Overall Complexity of Java Methods .................................................. 71

4.1 Data Collection Process Flow ............................................................ 92
4.2 Metrics Extraction Process Flow ......................................................... 94
4.3 Bug List in SourceForge ................................................................. 95
4.4 Bug Report Form in SourceForge ...................................................... 95
4.5 Fan-out of Java Classes ................................................................. 99
4.6 Fan-out Comparison between Tools ................................................... 101
4.7 CBO Comparison between Tools ....................................................... 104
4.8 Cyclomatic Complexity Comparison between Tools .............................. 106

5.1 Total Defects vs Total System Complexity .......................................... 122
5.2 Box Plot for Total Defects ............................................................... 122
5.3 Defect Density vs Average System Complexity ..................................... 123
5.4 Box Plot for Defect Density ............................................................. 124
5.5 Defect Density vs Average Data Complexity ....................................... 125
5.6 Defect Density vs Average Structural Complexity ................................ 125
5.7 Defect Density vs Average Procedural Complexity .............................. 126
5.8 Defect Density vs log10Version Download ......................................... 126
5.9 Average Data Complexity vs Average Procedural Complexity ................ 129
5.10 Average Data Complexity vs Class Size .......................................... 129
5.11 Defect Density vs Ratio System and Procedural Complexity ................ 130
5.12 Defect Density vs Number of Developers ........................................ 132
5.13 Defect Density for System Categories ............................................. 135
5.14 Team Member Profile ................................................................. 138
5.15 Defect Classification ................................................................. 139
5.16 Defects/Class and Defects/KLOC for Systems .................................. 140
5.17 System Size in KLOC ................................................................. 141
5.18 Number of Classes for System ....................................................... 142
5.19 Class Size for Systems ................................................................. 143
5.20 Structural Complexity Comparison .................................................. 146
LIST OF FIGURES

5.21 Data Complexity Comparison ........................................ 147
5.22 System Complexity Comparison .................................... 149

6.1 Scatter Plot of Class Size Clusters ................................. 158
6.2 Scatter Plot of Average Structural Complexity Clusters ........ 160
6.3 CBO Complexity Ratio ................................................. 162
6.4 Scatter Plot of Average Procedural Complexity Clusters ....... 165
6.5 Cyclomatic Complexity Ratio ......................................... 165
6.6 Scatter Plot of Average Data Complexity Clusters ............... 168
6.7 Average Data Complexity vs Average Procedural Complexity .. 170
6.8 Average Data Complexity vs Class Size .......................... 170
6.9 Scatter Plot of Average System Complexity Clusters .......... 172
6.10 Average System Complexity vs Defect Density by Categories 172
6.11 Defect Density vs System/Procedural Complexity .............. 174
6.12 Class Size vs System/Procedural Complexity .................... 175
6.13 Class Size vs Number of Classes ................................ 176
6.14 System/Procedural Ratio and Defect Density Shift for Different Versions ............................................ 177
## List of Tables

2.1 McCall’s Quality Factors [McCall et al., 1977] ......................... 13  
2.2 Summary of Measurement Scales and Statistics Relevant to Each Scale 22  
2.3 Orthogonal Defect Classification ........................................ 57  
2.4 Prediction Modeling Techniques .......................................... 60  
2.5 Comparison of Research Work .............................................. 61  
3.1 Comparison of Card and Glass’s Work with This Work .................... 67  
4.1 Research Goal, Questions and Metrics ..................................... 79  
4.2 Applicable metrics ......................................................... 81  
4.3 Metric Extraction Tools ...................................................... 89  
4.4 Comparison of Fan-out Properties in Different Tools ..................... 100  
4.5 Fan-out: Mann-Whitney U Test between Tools ........................... 102  
4.6 CBO: Mann-Whitney U Test between Tools ............................... 103  
4.7 Comparison of McCabe’s Cyclomatic Complexity (MCC) by Tools .... 107  
4.8 Cyclomatic Complexity: Mann-Whitney U Test between Tools .......... 108  
5.1 Anderson-Darling Test for Normality ..................................... 116  
5.2 Descriptive Statistics ....................................................... 118  
5.3 Spearman’s Rank Correlation Coefficient ................................ 120  
5.4 Univariate Analysis for Total Defects ................................... 121  
5.5 Defect Density Outlier Analysis .......................................... 124  
5.6 Univariate Analysis for Defect Density .................................. 127  
5.7 Multivariate Model for Defect Density .................................. 131  
5.8 Multivariate Model for Defect Density with Size Control ............. 131  
5.9 Results of Kruskal-Wallis Test ........................................... 134  
5.10 Defect Handling Survey Results ......................................... 137  
5.11 Results of Card and Glass Model ......................................... 144  
5.12 Results of the Proposed New Model .................................... 145  
5.13 Results of the Proposed Alternative Model .............................. 149  
5.14 Comparison of Results using Two Models ................................ 151  
6.1 Results of Cluster Analysis for Class Size ................................ 159  
6.2 Results of Cluster Analysis for Average Structural Complexity ....... 162  
6.3 CBO and Defects for JBoss Classes ....................................... 163  
6.4 Results of Cluster Analysis for Average Procedural Complexity ...... 166  
6.5 Cyclomatic Complexity and Defects for JBoss Classes .................. 167
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6</td>
<td>Results of Cluster Analysis for Average Data Complexity</td>
<td>169</td>
</tr>
<tr>
<td>6.7</td>
<td>Results of Cluster Analysis for Average System Complexity</td>
<td>173</td>
</tr>
<tr>
<td>A.1</td>
<td>Data for Systems in Games/Multimedia Category</td>
<td>195</td>
</tr>
<tr>
<td>A.2</td>
<td>Data for Systems in Internet/Communication Category</td>
<td>196</td>
</tr>
<tr>
<td>A.3</td>
<td>Data for Systems in Office/Programming/Database Category</td>
<td>197</td>
</tr>
<tr>
<td>A.4</td>
<td>Data for Systems in Scientific/Engineering/Operating Systems Category</td>
<td>198</td>
</tr>
</tbody>
</table>
Part I

Research Overview
Chapter 1

Introduction

Contents

1.1 Background ........................................ 4
1.2 Research Objectives .............................. 4
1.3 Research Questions ............................... 5
1.4 Research Hypotheses .............................. 5
1.5 Thesis Outline .................................... 5
1.1 Background

One of the most important objectives of software engineering is to improve the quality of software products. The quality of software can be defined in different ways but one of the most common definitions is the number of defects that arise in the final product [Fenton and Pfleeger, 1997], be it functional defects or programming defects, that can cause problems to users. The aim of this research is to identify whether there is any relationship between the number of discovered post-delivery defects and the design complexity of a system, and whether measured design complexity may provide an estimation mechanism for post-delivery defect discovery.

This thesis focuses on the measurement of several metrics in software products, which are open-source projects and are available freely from SourceForge.net. The projects chosen to be investigated were written in Java. This research is designed to investigate selected problems in a snapshot version of open-source systems rather than studying the evolution of systems.

This research is based on the earlier work of Card and Glass [Card and Glass, 1990] who argue that design measures are indicators/estimators of decision counts (cyclomatic complexity), module size (executable lines of code), and errors (discovered from system tests) [Card and Glass, 1990]. In a similar line, this thesis investigates how metrics available during the detailed design phase (Data Complexity) can be used to estimate the number of decisions (Procedural Complexity) during the implementation phase, and Class Size during the coding phase.

Although there is other work being conducted surrounding the issues of software measurement frameworks, open-source development, defect estimations, object-oriented metrics and estimation modeling as discussed in Chapter 2, this research is an important contribution to the software engineering field.

1.2 Research Objectives

The main objective of this research is to investigate whether there is a general correlation between post-delivery defects and system design complexity, by studying measures relating to data, structural and procedural complexity in object-oriented systems and comparing them with discovered, post-delivery defects. Another aim is to determine whether during the detailed design phase, measured Data Complexity can estimate measured Procedural Complexity and Class Size for the implemented system.
1.3 Research Questions

The research objective can be broken down further into other relevant questions that need additional investigation.

1. Is there a general correlation between system (data + structural) design complexity and various types of defects, and thus, can the level of defects be estimated early enough to undertake strategies to minimize them in the final product?

2. Between structural, procedural and data complexity, which has the most influence and/or is the most appropriate for estimating system defects?

3. Are there particular strategies for minimizing defects in open-source software?

4. Are there any differences in the correlation between various complexities and defect density for different system types?

5. Can the Card and Glass’s model be extended to object-oriented systems?

1.4 Research Hypotheses

1. H1 : Increasing values of the Average System Complexity (Average Structural Complexity plus Average Data Complexity) correlate with increasing post-delivery defect density.

2. H2 : Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Average Procedural Complexity (Cyclomatic Complexity), as in some non-object-oriented systems.

3. H3 : Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Class Size, as in some non-object-oriented systems.

4. H4 : There is a relationship between System Complexity and Procedural Complexity that will help minimize defects in the final system.

1.5 Thesis Outline

This thesis discusses the related theoretical framework, the application of the framework on this research, the methodology used to conduct the research, data
Chapter 1: Introduction
collection methodology, data analysis and results discussion. The chapters in this thesis are structured as follows:

Chapter 2 includes a review of the literature that is related to the issues covered in this thesis. The purpose of this chapter is to find out the theoretical basis or framework as a guideline to conduct this research and to explore the extent of work being done by other researchers in empirical software engineering area to ensure that the novelty of this research is protected. The literature being considered for review is mainly classified into several subject matters such as:

1. Software engineering and quality
2. Empirical theory
3. Measurement theory and framework
4. Open-source software research
5. Software metrics
6. Quality prediction modeling

Although an extensive amount of literature on software metrics and measurement was explored, only some relevant metrics are chosen to be included in this thesis since they are related to the previous work of Card and Glass [Card and Glass, 1990]. The metrics involved are known as design metrics, such as McCabe’s Cyclomatic Complexity [McCabe, 1976], Henry and Kafura’s Information Flow metrics [Kafura and Henry, 1982], and Chidamber and Kemerer’s metrics [Chidamber and Kemerer, 1994] to measure object-oriented properties of Java systems.

Chapter 3 discusses the idea of functional decomposition, the reason to use Card and Glass’s model as the basis of this research, the complexity in Java systems, as well as the discussion of the proposed model for object-oriented (OO) systems. This chapter explores the need for a new model for OO systems based on an older model for structured systems. As a result, a new model is proposed to make it more relevant to OO systems.

While Chapter 3 mainly discusses the theoretical part of this thesis, Chapter 4 explains the details of the actual work being conducted in the research. It extends the discussion on the data collection process in this research and illustrates how
1.5 Thesis Outline

open-source projects were selected, including the selection criteria of the projects, the choice of metrics that are applicable for this research, the measurement tools being used, which also includes the processes that were involved in getting the right tools and also validation of the results produced by the tools. Lastly, this chapter also describes the data collection procedures that were conducted during the progressive construction of this thesis.

In continuation, Chapter 5 gives details about the data analysis process, and this includes the process of choosing the statistical package to be used in the analysis, the explanation of data analysis methodology which consists of choosing appropriate statistical techniques to analyse the data that were obtained during data collection process and the process involved in building the estimation model based on the empirical model. Examples of the statistical analyses employed in this work are: correlation analysis, univariate analysis, multivariate analysis and analysis by system categories.

Chapter 6 consists of a lengthy discussion on the results obtained in the Chapter 5, including, discussion of the relationships between the independent variables on Defect Density, correlation of Data Complexity and Procedural Complexity and Class Size and also the ratio of System and Procedural Complexity. Moreover, this chapter also highlights the benefits of this research to the industry, challenges faced during the research and possible threats to the validity of results.

Chapter 7 concludes the thesis by revisiting the most important sections of this thesis including data collection and analysis, results discussion, the contribution of this research to the industry and also looks into the possible work to be expanded in the future.
Chapter 2

Literature Review

Be as simple as possible, but no simpler.

Albert Einstein

Contents

2.1 Introduction ................................................. 10
2.2 Software Engineering and Quality ............................. 11
2.3 Empirical Theory .............................................. 14
  2.3.1 Replications ........................................... 16
2.4 Measurement Theory and Framework ............................ 17
  2.4.1 Measurement Scales .................................... 20
  2.4.2 The Goal-Question-Metric Paradigm (GQM) .............. 24
  2.4.3 Related Work in Software Measurement ................. 25
2.5 Open-Source Research ......................................... 26
  2.5.1 Open-Source Software as a Research Base ............ 29
2.6 Software Metrics ............................................... 32
  2.6.1 Traditional Structure Metrics .......................... 32
  2.6.2 Object-oriented Metrics ............................... 49
2.7 Quality Prediction Modeling ................................... 54
2.8 Summary ...................................................... 59
Chapter 2: Literature Review

2.1 Introduction

This chapter includes the review of the literature which is useful as a guideline for this research. The theoretical foundations and frameworks related to the issues covered in this thesis are explored and discussed. Similarly, prior work by other researchers are investigated and reported in order to gain insight into the topics covered by this research. This chapter is structured as follows:

1. Software engineering and quality: This topic is discussed in this thesis because it provides various definitions of software quality from several resources including Crosby [Crosby, 1979], the IEEE [IEEEStandard1990, 1990], Juran [Juran, 1988], McCall [McCall et al., 1977] and Pressman [Pressman, 2000].

2. Empirical theory: The discussion on this topic is essential in order to grasp the background knowledge of the empirical theory in software engineering. Among issues explored in this topic include the approaches to empirical research in software engineering, as well as the importance of replications in building a body of knowledge in software engineering.

3. Measurement theory and framework: Due to the fact that this research involves a lot of measurement work, it is important to explore this topic to investigate the relevant issues and results from prior research. Among the issues discussed are: The measurement theory background, measurement scales and the Goal-Question-Metric paradigm (GQM).

4. Open-source research: This topic emphasizes the state of research in open-source software development as well as to investigate the potential of open-source software to be used as a research base.

5. Software metrics: This topic is explored to identify the metrics which have significant impact on this research. Among the metrics identified are: Henry and Kafura's information Flow Measure, McCabe's Cyclomatic Complexity Measure, Card and Glass's System Complexity Model, size measure, coupling, cohesion, Chidamber and Kemerer Object-oriented Metric Suite and Lorenz and Kidd Metrics.

6. Quality prediction modeling: The discussion of this topic has helped to shed some light on the issues surrounding the techniques used to build the quality prediction model of the systems under study. Results reported by prior research are examined and used as guideline to build the prediction model for this research.
2.2 Software Engineering and Quality

Software engineering describes the collection of techniques that apply an engineering approach to the construction and support of software products. Software engineering activities include managing, costing, planning, modeling, analyzing, specifying, designing, implementing, testing and maintaining. The term "engineering approach" means that each activity is understood and controlled, so there are few surprises as the software is specified, designed, built and maintained. Computer science provides the theoretical foundations for building software, and software engineering focuses on implementing the software in a controlled and scientific way [Fenton and Pfleeger, 1997].

The importance of software engineering cannot be understated, because software plays an extremely important role in our lives. The dependency on software is undisputed, from oven control to security equipment, from banking transaction to air traffic control, and from sophisticated power plants to sophisticated weapons, our life and quality of life depend on software. For such a young profession, software engineering has given a huge contribution in providing safe, useful and reliable functionality. However, there is room for a great deal of improvement. The literature is full of examples of projects that have overrun their budgets and schedules, and software failures can even put lives at risk [Conte et al., 1986], [Fenton and Pfleeger, 1997], [Galin, 2004], [Grady and Caswell, 1987], [Jarvis and Crandall, 1997], [Kan, 2003], [Laird and Brennan, 2006], [McGarry et al., 2002], [Putnam and Myers, 2003].

Therefore, it is imperative that the quality of software is taken seriously to avoid the calamity of software failures. The formal definitions of software quality were given by several sources, hence numerous definitions will be included here.

- IEEE definitions [IEEEStandard1990, 1990]:
  - The degree to which a system, component, or process meets specified requirements.
  - The degree to which a system, component, or process meets customer or user needs or expectations.

- Crosby's definition (founder of modern quality assurance) [Crosby, 1979]:
  - Quality means conformance to requirements.

- Juran's definitions (another founder of modern quality assurance) [Juran, 1988]:
Chapter 2: Literature Review

- Quality consists of those product features which meets the needs of customers and thereby provide product satisfaction.
- Quality consists of freedom from deficiencies.

- Pressman's definition [Pressman, 2000]:
  - Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

All of the quality definitions given include “conformance to requirements” and the non-conformances are regarded as defects, or product quality has not been met. In software, the narrowest sense of product quality is usually recognized as lack of “bugs” or “defects” in the product. It is also the most basic meaning of conformance to requirements, because if the software contains too many functional defects, the basic requirements of providing the desired function is not met.

Several models of software quality factors and their categorization in factor categories have been suggested over the years. The classic model of software quality factors introduced by McCall, consists of 10 factors [McCall et al., 1977]. Subsequent models, consisting of 12 to 15 factors, were suggested by Deutsch and Willis [Deutsch and Willis, 1988] and by Evans and Marciniak [Evans and Marciniak, 1987]. The alternative models do not differ substantially from McCall’s model. The McCall factor model, despite the quarter of a century of its “maturation”, continues to provide a practical, up-to-date method of classifying software requirements [Pressman, 2000]. The McCall factors are listed in Table 2.1 [McCall et al., 1977].

Table 2.1 lists a set of software-specific quality factors. McCall breaks each of these factors down into criteria, each of which has specific metrics associated. These metrics include both subjective judgments and objective measures. However, these quality factors capture characteristics of the final product rather than its design.

In his book, Juran, a noted industrial quality expert, defines product quality as “fitness for use” [Juran, 1988]. This concept includes all those features of the product recognized as beneficial to the users. Therefore, any attribute of the product that interferes with an intended use becomes a symptom of poor quality. When we look at design quality as fitness for use, there are two principal uses of design on which its fitness must be assessed. First, the software design defines the functionality and performance of the final product for the operational user. Second, the software design provides a template for the production of the operational
2.2 Software Engineering and Quality

<table>
<thead>
<tr>
<th>Factor</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>Extent to which a program satisfies its specifications and fulfills the user’s mission objectives</td>
</tr>
<tr>
<td>Reliability</td>
<td>Extent to which a program can be expected to perform its intended function with required precision</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Amount of computing resources and code required by a program to perform a function</td>
</tr>
<tr>
<td>Integrity</td>
<td>Extent to which access to software or data by unauthorized persons can be controlled</td>
</tr>
<tr>
<td>Usability</td>
<td>Effort required to learn, operate, prepare input and interpret output of a program</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Effort required to locate and fix an error in an operational program</td>
</tr>
<tr>
<td>Testability</td>
<td>Effort required to test a program to ensure that it performs its intended function</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Effort required to modify an operational program</td>
</tr>
<tr>
<td>Portability</td>
<td>Effort required to transfer a program from one hardware configuration and/or software system environment to another</td>
</tr>
<tr>
<td>Reusability</td>
<td>Extent to which a program can be used in other applications, related to the packaging and scope of the functions that programs perform</td>
</tr>
</tbody>
</table>

product by programmers and maintainers. If a design achieves both of these goals, then it is said to represent a “feasible” solution [Card and Glass, 1990].

Therefore, the essential components of software design fitness for use include:

1. Service in operation - the degree to which the designed system meets the specified needs of the operational user. These may include both functional and performance requirements. From the software design point of view, ‘service in operation’ equates to ‘satisfaction of requirements’. This means showing traceability from requirements to design.

2. Ease of production - the ease and accuracy with which the design can be implemented and the product maintained. This includes aspects of technical
Chapter 2: Literature Review

excellence like modularity and simplicity. Poor produceability means a high error rate, low productivity and difficult maintenance.

2.3 Empirical Theory

Empirical analysis of software engineering is an important research method that can add new knowledge to areas related to process and product improvement. Since it is based on observation, and reflects our actual experience with methods, tools and techniques, empirical research is closer to the real world than analytical or theoretical research [Harrison et al., 1999]. Empirical software engineering research offers the opportunity to build and verify theories for software engineering [Lehman and Belady, 1976]. Therefore, it provides steps towards a better understanding of our discipline.

An empirical approach to assessing software engineering technology, including industrial collaboration, began on large scale in the 1970s with the work of Victor Basili and his group at the University of Maryland [Basili et al., 2002], [Boehm et al., 2005]. Since then, there has been an increased focus on the importance of and approaches to applying empirical methods in software engineering research [Basili et al., 1986], [Basili et al., 2007a], [Perry et al., 2000], [Rombach et al., 1993], [Tichy et al., 1995], [Tichy, 1998] and [Zelkowitz and Wallace, 1998].

Empirical Science concerns the acquisition of knowledge by empirical methods. However, what constitutes knowledge and the methods for acquiring it, rests on basic assumptions regarding ontology (what we believe to exist) and epistemology (how beliefs are acquired and what justifies them) [Sjoberg et al., 2007]. Empirical research seeks to explore, describe and explain natural, social or cognitive phenomena by using evidence based on observation or experience. It involves obtaining and interpreting evidence by experimentation, systematic observation, interviews or surveys, or by careful examination of documents or artifacts.

An empirical body of evidence in empirical software engineering can be described as a set of studies, each performed under certain explicit conditions, for which both quantitative and qualitative, subjective and objective data have been collected and based on which certain conclusions and interpretations have been provided [Briand, 2007].

There are two approaches to empirical research, qualitative and quantitative methods and they are used for collecting and analyzing data. Quantitative methods collect numerical data and analyze it using statistical methods, while qualitative methods collect material in the form of text, images or sounds drawn
from observations, interviews and documentary evidence, and analyze it using methods that do not rely on precise measurement to yield their conclusions [Hardy and Bryman, 2004], [Hove and Anda, 2005], [Lethbridge et al., 2005] and [Seaman, 1999].

Although different approaches to research suggest different steps in the process of acquiring knowledge, most empirical methods require that the researcher specify a research question, design the study, gather the data or evidence, analyze and interpret the data. According to Sjöberg et al., there are four most common primary approaches to research in software engineering [Sjöberg et al., 2007]:

1. Experimentation - An experiment is an empirical inquiry that investigates causal relations and processes. The identification of causal relations provides an explanation of why a phenomenon occurred. Experiments are conducted when the investigator wants control over the situation, with direct, precise and systematic manipulation of the behaviour of the phenomenon to be studied.

2. Survey - A survey is a retrospective study of a situation that investigates relationships and outcomes. It is useful for studying a large number of variables using a large sample size and rigorous statistical analysis. Surveys are especially well-suited for answering questions about what, how much, and how many, as well as questions about how and why. They are used when control of the independent and dependent variables is not possible or not desirable, when the phenomena of interest must be studied in their natural setting and when the phenomena of interest occur in current time or the recent past.

3. Case studies - A case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident. Yin [Yin, 2003] noted that a case study has a distinct advantage when a “how” or “why” is being asked about a contemporary set of events over which the investigator has little or no control.

4. Action research - Action research focuses particularly on combining theory and practice. It attempts to provide practical value to the client organization while simultaneously contributing to the acquisition of new theoretical knowledge.

This thesis uses experimentation as the main approach to answer the research questions. There are several resources that present the guidelines of conducting
empirical studies in software engineering, such as, [Basili, 1993], [Kitchenham et al., 2002], [Kitchenham et al., 2008], [Lott and Rombach, 1996], [Perry et al., 2000], [Pfleeger, 1994], [Pfleeger, 1995a], [Pfleeger, 1995b], [Pfleeger, 1995c], [Pfleeger, 1995d], [Sjöberg et al., 2007], [Shull et al., 2008], [Tichy, 2000] and [Wohlin et al., 2003].

2.3.1 Replications

Replications play a key role in Empirical Software Engineering by allowing the community to build knowledge about which results or observations hold under different conditions. Hence, not only can a replication that produces results similar to the original experiment be viewed as successful, but a replication that produces results different from the original experiment can also be viewed as successful [Shull et al., 2008]. They also argue that the success of a replication must be judged relative to the knowledge it contributes to the body of knowledge, for example, identifying possible new variables that have an influence on the response variables. They highlight the important role of replications and suggest that the usage of exhaustive documentation of the original experiment, known as lab packages, to be used as guidelines to conduct replications of studies that were previously carried out by other researchers. According to Shull et al. [Shull et al., 2008], replications that reuse the original procedures, (e.g. the study design and the experimental steps) but modify the subject pool (e.g. number and types of software projects) or other experimental conditions (e.g. multiple software versions, metric types or artifacts) in order to gain more insight into the original results fall into the category of exact replications. This thesis is a replication of Card and Glass's work with several adjustments in the research environment and the details are discussed in Section 3.1.1.

In the past few years, there has been a growing awareness in the empirical software engineering community of the importance of replicating studies [Basili et al., 1999], [Basili et al., 2007b], [Brooks et al., 1996], [Lott and Rombach, 1996], [Shull et al., 2002] and [Shull et al., 2008]. Researchers realize that the true goal of empirical research should be, not the running of individual studies, but developing a better understanding of software development, the cost and benefits of various techniques and at the very end consolidating a body of knowledge and establishing software development models. They argue that too many uncontrollable sources of variation exist from one environment to another for the results of any study, no matter how well run, to be extrapolated to all possible software development environments. Most researchers accept that no one study on a technology should be considered definitive. A result of this realization is an increased commitment
2.4 Measurement Theory and Framework

to run more studies in a variety of environments [Shull et al., 2002]. Basili et al. support the importance of replications in their work [Basili et al., 1999], where they suggest that more validation work in the form of replications should be conducted to build a body of software engineering knowledge. They argue that building families of experiments and studies is necessary to protect the integrity of software engineering theories and research work.

Briand et al. [Briand et al., 1999a] also highlight the significance of replications in empirical software engineering. They propose that the introduction of new product measures should be stopped for a while, and the investigation and validation of existing measures should take place, as well as thorough, rigorous and complete analysis procedures within the context of well-designed empirical studies. Furthermore, they suggest that replicated studies across many environments should be conducted in order to establish a solid body of empirical knowledge, so that more general conclusions can be drawn in this context. Similarly, Jeffery and Scott [Jeffery and Scott, 2002] suggest that more replication and theory revision, more explicit theory statements and independent evaluation, and more reflection and development of empirical software engineering methods can help to improve empirical software engineering.

In order to support the replication research in empirical software engineering, several guidelines and frameworks have been published, including the sharing of data and artifacts for the purpose of replications by other researchers [Basili et al., 2007b], [Selby, 2007], [Shull et al., 2004] and [Sjoberg, 2007]. According to Miller [Miller, 2005], the need and role for replications or repeated re-examinations of research hypotheses is extremely demanding in software engineering, therefore, more families of experiments need to be done.

2.4 Measurement Theory and Framework

You cannot control what you cannot measure

DeMarco, 1982

In empirical software engineering, measurement theory plays a considerably important role in understanding the science of software development. During the past decade, measurement theory has been proposed and extensively discussed [Fenton and Pfleeger, 1997], [Zuse, 1991] as a means to evaluate the software engineering measures that have been proposed in the literature, and to establish criteria for the statistical techniques to be used in data analysis. Measurement theory is a considerably convenient theoretical framework to explicitly define the
underlying theories upon which software engineering measures are based. This means that measures are not defined out of context and that the theories on which they are based can be discussed, adapted and refined [Briand et al., 1996b].

In any measurement activity, there are rules to be followed that will help to ensure consistency in the measurement process. These rules also provide basis for interpreting data. Measurement theory provides the rules, laying the groundwork for developing and reasoning about all types of measurement [Fenton and Pfleeger, 1997]. Formally, the definition of measurement as given by Fenton and Pfleeger [Fenton and Pfleeger, 1997] is, “Measurement is a mapping from the empirical world to the formal, relational world”. Therefore, a measure is the number or symbols assigned to an entity by this mapping in order to characterize an attribute. According to Fenton and Pfleeger [Fenton and Pfleeger, 1997], there are five main stages of formal measurement and they are:

1. Identify attribute for some real-world entities.
2. Identify empirical relations for attribute.
3. Identify numerical relations corresponding to each empirical relation.
4. Define mapping from real world entities to numbers.
5. Check that numerical relations preserve and are preserved by empirical relations.

Measurement shows the ways processes, products, resources, methods and technologies of software development relate to each other. Measurements can help answer questions about the effectiveness of techniques or tools, the productivity of development activities, the quality of product and much more. Furthermore, measurement allows the definition of a baseline for understanding the nature and impact of proposed changes [Fenton and Pfleeger, 1997]. Hence, measurement is useful for:

- understanding
- establishing a baseline
- assessing and predicting

Even though software measurement is potentially diverse in nature, there are five basic stages of on-going activity [Roche et al., 1994], and they are:
2.4 Measurement Theory and Framework

1. Formulation: Involves setting measurement goals, identifying the metrics required and defining them in terms of the particular measurement environment.

2. Collection: Concerned with setting up the actual measurement processes and any tool development/selection that might be necessary.

3. Analysis: The stage dealing with measurements once they have been obtained. Statistical analysis maybe important to help uncover patterns, discriminate between software components and identify anomalies.

4. Interpretation: The assignment of meaning to the collected values, determining the causes of the values, distinguishing which cause was responsible and identifying the appropriate corrective action to be taken.

5. Validation: Validation should be conducted throughout the measurement process. During the formulation stage, measurement theoretic, axiomatic and algebraic approaches to validation are possible. Validation in the collection stage includes procedures to assess the validity of the actual measurements. Validation during analysis stage is usually based upon the search for associations between the metric and other measures of the attribute in question, using historical data.

Specifically, there are two main approaches in providing a definition of measurement [Briand et al., 1996b]:

1. Measurement Theory
   - Specifies the general framework in which measures should be defined. First, an Empirical Relation System should be specified, to define the relations among the entities as far as the studied attribute is concerned. Then a Numerical Relation System is defined, to provide values for the measures of the attribute and relations among these values.

2. Axiomatic approaches
   - Formally define desirable properties of the measures for a given software attribute. These properties are properties of the Numerical Relation System of measures. However, they indirectly affect the Empirical Relation System. Hence, axioms can be used as guidelines for the definition of a measure.

Several proposals have been introduced by using either Measurement Theory or Axiomatic approaches. For example, Zuse [Zuse, 1991] concentrates on the complexity of flowgraphs, where he introduces the notion of atomic modification (a change in the entity being measured) and partial property (relation between the
value of the metric before and after the modification). Weyuker introduces a set of software complexity axioms [Weyuker, 1988]. Briand at al. define a more general framework, with a number of sets of axioms to differentiate the measures for several software attributes, such as size, length, complexity, cohesion and coupling [Briand et al., 1996c]. The theoretical soundness of measure, which is the fact that it really measures the software characteristic it is supposed to measure, is an obvious prerequisite for its acceptability and use [Briand et al., 1996c].

### 2.4.1 Measurement Scales

Another important aspect of measurement theory is the measurement scale used and the selection of measurement scale will also determine which statistical analysis should be used to analyze data. The measurement scale is classified into five major types [Fenton and Pfleeger, 1997]:

1. **Nominal scale** - Usually classes and categories are defined, based on the value of the attributes. Nominal scale has two major characteristics:
   - The empirical relation system consists only of different classes, there is no notion of ordering among the classes
   - Any distinct numbering or symbolic representation of the classes is an acceptable measure, but there is no notion of magnitude associated with the numbers or symbols.

2. **Ordinal scale** - Often useful to augment the nominal scale with information about an ordering of the classes or categories. The ordering leads to analysis not possible with nominal measures. The ordinal scale has the following characteristics:
   - The empirical relation system consists of classes that are ordered with respect to the attribute.
   - Any mapping that preserves the ordering (monotonic function) is acceptable
   - The numbers represent ranking only, so addition, subtraction and other arithmetic operations have no meaning.

3. **Interval scale** - This scale captures information about the size of the intervals that separate the classes. Therefore, an interval scale can be characterized as follows:
   - An interval scale preserves order, as with the ordinal scale.
2.4 Measurement Theory and Framework

- An interval scale preserves differences but not ratios, which means the difference between any two of the ordered classes in the range of the mapping is known.
- Addition and subtraction are acceptable on the interval scale, but not multiplication and division.

4. Ratio scale - The key feature that differentiates ratio from nominal, ordinal and interval scales is the existence of empirical relations to capture ratios. A ratio scale has the following characteristics:

- It is a measurement mapping that preserves ordering, the size of intervals between entities, and ratios between entities
- There is a zero element, representing total lack of the attribute
- The measurement mapping must start at zero and increase at equal intervals (unit)
- All arithmetic can be meaningfully applied to the classes in the range of the mapping

5. Absolute scale - The absolute scale has the following properties:

- The measurement for an absolute scale is made simply by counting the number of elements in the entity set.
- The attribute always takes the form "number of occurrences of x in the entity."
- There is only one possible measurement mapping, that is the actual count
- All arithmetic analysis of the resulting count is meaningful

The scales previously discussed are summarized in Table 2.2 [Fenton and Pfleeger, 1997]:

In measurement theory, it is important to identify the measurement scales of the metrics under study. However, in software engineering, there is no clear choice on the scales most suitable to be used to measure the attributes of software [Briand et al., 1996b], [Kaner and Bond, 2004]. Most of the time, the decisions on which scales should be used to map the empirical relation system to some numerical relation system lie in the subjective judgment of the researchers themselves.

Briand et al. [Briand et al., 1996b] discuss in detail the concept of measurement theory and measurement scale as well as the application of both theory
Chapter 2: Literature Review

Table 2.2 – Summary of Measurement Scales and Statistics Relevant to Each Scale

<table>
<thead>
<tr>
<th>Scale Type</th>
<th>Defining relations</th>
<th>Examples of appropriate statistics</th>
<th>Appropriate statistical test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>Equivalence</td>
<td>Mode, Frequency</td>
<td>Non-parametric</td>
</tr>
<tr>
<td>Ordinal</td>
<td>Equivalence, Greater than</td>
<td>Median, Percentile, Spearman $r$, Kendall $t$, Kendall $W$</td>
<td>Non-parametric</td>
</tr>
<tr>
<td>Interval</td>
<td>Equivalence, Greater than, Known ratio of any intervals</td>
<td>Mean, Standard deviation, Person product-moment correlation, Multiple product-moment correlation</td>
<td>Non-parametric</td>
</tr>
<tr>
<td>Ratio</td>
<td>Equivalence, Greater than, Known ratio of any intervals, Known ratio of any two scale values</td>
<td>Geometric mean, Coefficient of variation</td>
<td>Non-parametric and parametric</td>
</tr>
</tbody>
</table>

and scale in software measurement. They disagree with Zuse [Zuse, 1991], who advocates the need for a complexity measure to be additive and for the underlying empirical system to “assume an extensive structure”. Instead, they suggest that complexity measures do not have to be additive, under some kind of composition operation between program segments or subsystems. In order to put their point across, they argue against Zuse’s theory using the Modified Extensive Structure Theory as well as Weyuker’s set of properties for complexity measures [Weyuker, 1988]. Furthermore, they also discuss the measurement scales in software engineering and state that in software engineering, it is difficult to determine the scale type of a measure, and it is easy to get confused with the most appropriate scale types to measure the properties of software. The choice between parametric and nonparametric tests is also difficult to make, and researchers need to be careful with making the choice because each test has its own strengths and weaknesses, especially the danger of making Type I and Type II errors if a wrong test is chosen. Type I error is incorrectly rejecting the null hypothesis and Type II error is accepting the null hypothesis when it is actually false [Fenton and Pfleeger, 1997].

In [Briand et al., 1996c], Briand et al. discuss the properties for a set of measurement concepts that are relevant for the definition of measures of internal software attributes, such as size, length, complexity, cohesion and coupling. They stated that the measurement scale for size, length and complexity is of ratio scale. Zuse also suggests that the scale for cyclomatic complexity should be the ratio scale [Zuse, 1996].
2.4 Measurement Theory and Framework

Measurement programs are not only conducted by researchers, but also adopted at large by several organizations to improve and control their software development program. One program that has been widely discussed and received great attention is the 10-step metric program developed at Hewlett Packard by Grady and Caswell [Grady and Caswell, 1987]. This program concentrates on production processes, advocates use of a standard set of metrics (i.e. people/time/cost, size and defect metrics) and improving processes through statistical control, and relies upon established and well proven relationships. Another example of metric program practice in industry is at Microsoft [Nagappan and Ball, 2005], [Nagappan et al., 2006a], [Nagappan et al., 2006b], [Nagappan and Ball, 2007]. In particular, Nagappan and Ball [Nagappan and Ball, 2005] study the relationship between relative code churn with software defect density and concluded that relative code churn is a good predictor of software defect density. Another study by Nagappan et al. [Nagappan et al., 2006b] analyzed five Microsoft projects to identify metrics that can predict post-release failures and reported the ways to systematically build predictors for post-release defects from historical data.

Before any statistical analysis can be conducted on the data, the measurement scales need to be determined based on the measurement theory discussed previously. According to Briand et al., the measurement scale for size, length and complexity should be a ratio scale [Briand et al., 1996c]. Likewise, Zuse stated that the scale for cyclomatic complexity should be a ratio scale [Zuse, 1996]. However, the scales for other metrics like Fan-out, CBO and defect count is difficult to determine. The decision of which scale to use depends on the distribution of normality of the data itself. There are several techniques for assessing whether a distribution is normal, some are described by Kitchenham [Kitchenham, 1992]. If the distribution of the data is not known, there are several approaches for dealing with this lack of knowledge [Fenton and Pfleeger, 1997]:

- Robust statistics and non-parametric methods can be used. Robust statistical methods are descriptive statistics that are resilient to non-normality. That means, regardless of whether the data are normally distributed or not, robust methods yield meaningful results. On the other hand, non-parametric statistical techniques take into account the fact that the data are not normal, thus allow to test various hypotheses about the data set without relying on the properties of the normal distribution. In particular, non-parametric techniques often use properties of the ranking of the data.

- Attempt to transform basic measurements into a scale in which the measurements conform more closely to the normal distribution. For example, when investigating relationships between project effort and product size, it is quite
common to transform to the logarithmic scale. Whereas the original data are not normally distributed, the logarithmic of the data are.

- Attempt to determine the true underlying distribution of the measurements and use statistical techniques appropriate to that distribution.

### 2.4.2 The Goal-Question-Metric Paradigm (GQM)

The Goal Question Metric (GQM) approach was originally defined for evaluating defects for a set of projects in the NASA Goddard Space Flight Center environment. This approach is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects. Then it must trace those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals [Basili et al., 1994]. A GQM model is a hierarchical structure starting with goals that specifies the purposes of measurement, the objects to be measured and the viewpoints from which the measure is taken. The goal later is refined into several questions that usually break down the issue into its major components. Then each question is refined into metrics, the same metric can be used in order to answer different questions under the same goal.

GQM enables definition of a set of metrics through establishing several goals and questions and from them deriving quantifiable questions and metrics. The ability to tailor the method to specific measurement goals enables measures to be defined, collected, analyzed, validated and interpreted within the context of the project [Roche et al., 1994]. The advantages of using GQM include [Roche et al., 1994]:

1. Covers the constructive, analytical, learning and feedback aspects of measurement.
2. Consists of measurement goals that give a purpose to the collection of measures, enabling targeting of questions, from which measurements are defined.
3. Enables the definition of a set of metrics for more effective capture of software attributes.
4. The metrics are tailored to a specific project, therefore, enabling validation and interpretation of the data within context of the project. The GQM offers the potential for reuse of metrics in similar projects having similar goals.
2.4 Measurement Theory and Framework

Metrics or measurement programs to be undertaken by any individuals or organizations should be carefully planned. A systematic plan is essential to ensure the measurement process can be conducted successfully. A metric plan consists of the reasons for the measurement program, what is to be measured, where and when the measurement process should take place, what tools to be used for data capturing and the people responsible for the activities [Fenton and Pfleeger, 1997].

Many metrics programs commence by measuring what is convenient or easy to measure, rather than by measuring what is needed. Such programs often fail because the resulting data are not useful to the developers and maintainers of the software [Fenton and Pfleeger, 1997]. An approach that is based on goals of the project was introduced by Basili [Basili et al., 1994], who provided a framework involving three steps:

1. List the major goals of the development or maintenance project.
2. Derive from each goal the questions that must be answered to determine if the goals are being met.
3. Decide what must be measured in order to be able to answer the questions adequately.

When measurement is derived in this manner, it becomes clear how to use the resulting data. The GQM approach combines in itself most of the current approaches to measure and generalizes them to incorporate processes and resources as well as products. This approach has been widely used in different environments and has been applied in several organizations, for example, NASA, Hewlett Packard, Motorola and Coopers & Lybrand, to name a few [Basili et al., 1994].

2.4.3 Related Work in Software Measurement

Since its inception, there have been a lot of publications discussing the theoretical aspects of the measurement framework, among them, [Basili et al., 1994], [Briand et al., 1996b], [Briand et al., 1996c], [Card and Jones, 2003], [Lindvall et al., 1997], [Mendonca and Basili, 2000], [Morasca and Briand, 1997], [Morasca, 2001], [Munson, 1995], [Roche et al., 1994] and [Wohlin, 1996]. In particular, Briand et al. [Briand et al., 1996b] discussed the basic concept of measurement theory, usage and interpretation of measurement theory in software engineering, the application of measurement theory in terms of measurement scale and the corresponding statistical techniques most suitable for the scale.
Much of the work being done in this topic is related to the application of measurement theory in empirical software engineering. As such, Briand et al. attempted to measure the relationships between design measures and software quality in object-oriented systems using \textit{Univariate Analysis} and \textit{Multivariate Logistic Regression Model}. Their results show that many of the measures capture similar dimensions in the dataset, thus reflecting the fact that many of them are based on similar principles and hypotheses [Briand et al., 1998c]. In their work, Basili et al. discussed the application of measurement and metrics as quality indicator in object-oriented systems [Basili et al., 1996]. In addition, there are many examples of the application of measurement theory in empirical software engineering, to name a few, [Arisholm et al., 2004], [Basili and Perricone, 1984], [Basili and Weiss, 1984], [Basili et al., 2002], [Belady and Evangelisti, 1981], [Briand et al., 1997], [Ebert, 1996], [El-Emam and Carleton, 2004], [Grady and Caswell, 1987], [Goldensen et al., 1999], [Kemphens et al., 2000], [Lott and Rombach, 1996], [McGarry et al., 2002], [Tian et al., 1997] and many more.

\section{2.5 Open-Source Research}

The open-source software (OSS) development community has grown enormously over the past decade. Open-source systems are widely accepted and successfully adopted/adapted into many organizations, and some of these systems have been used for mission-critical purposes. For example, open-source systems have been adopted in the US Department of Defense (DoD) as reported in the 2003 Mitre Corporation study: Use of Free and Open Source Software within the U.S. Department of Defense [Corporation, 2003], which identified more than 100 different open-source programs and more than 250 instances of their usage within DoD systems.

Another example of the open-source adoption within DoD systems was documented through a series of Cooperative Research and Development Agreements (CRADA) between the Naval Meteorology and Oceanography Command and the Open-Source Software Institute (Navy CRADA-08-001 and Navy CRADA-05-11). The goal of the initial joint study (NCRADA-08-001) was to assess the use of open-source software (2001 - 2003) at the Naval Oceanographic Office (NAVOCEANO) and to identify additional opportunities for further implementation of open-source software within NAVOCEANOs computing environment. The CRADAs findings reported extensive use of open-source within NAVOCEANOs existing infrastructure, particularly as mission critical applications within the ISS60, UNISIPS, Network Attached Storage Servers and QA/Monitoring workstations [Weathersby, 2007].
2.5 Open-Source Research

Furthermore, there has been an extensive use of OSS by the government of various countries all over the world [Lin]. Therefore, it is extremely important to assess and validate the reliability and performance of these systems to help ensure that they fulfill their purposes.

The OSS community has legal and pragmatic arrangements to ensure the source code for an OSS development will be generally available. Open-source developments usually have a central person or body that selects some subset of the developed code for the “official” release(s) and makes it/these available for distribution [Mockus et al., 2002].

These basic arrangements to ensure freely available source code led to a development process that is radically different from the usual industrial style of development. The main differences most often mentioned are the following [Raymond, 2000]:

- Open-source systems are built by potentially large numbers of volunteers. However, there are some exceptions to this statement where some OSS projects are supported by companies and some participants are not volunteers.
- Work is not assigned, people undertake the work they choose to undertake.
- There is generally no explicit system level design, or even detailed design.
- There is generally no project plan, schedule or list of deliverables.

These differences imply an extreme case of geographically distributed development, where developers work in arbitrary locations, rarely or never meet face to face, and coordinate their activities usually by email and bulletin boards. Most of the OSS projects lack many of the traditional mechanisms used to coordinate software development, such as plans, system-level design, schedules and defined processes [Mockus et al., 2002]. According to Mockus et al. [Mockus et al., 2002], the results from OSS development are often claimed to be equivalent, or even superior to software developed more traditionally. It is claimed, for instance, that defects are found and fixed incredibly fast because there are “many eyeballs” looking for the problems [Raymond, 2000]. Raymond calls this “Linus’s Law” and code is written with more care and creativity, because developers are working only on things for which they have a real passion [Raymond, 2000].

A study by Zhou and Davis [Zhou and Davis, 2005] of 8 open-source projects demonstrated that open-source projects show similar reliability growth patterns as with proprietary software projects. This means that even though open-source
development methodologies are usually seen as different from the proprietary software development methodologies, they have the same properties that can be used as indicators of software quality. In addition, Zhao and Elbaum [Zhao and Elbaum, 2003] conducted a study to investigate the quality assurance practice in projects under the open-source development model and they found that many quality assurance activities in open-source projects are still evolving and improving compared to previous years. They found that user participation has helped project developers to discover field defects that later increase the quality of these OSS projects. Furthermore, their findings show that most of the projects make good use of the configuration and bug tracking tools to manage the projects and some even use the tools more than in traditional projects.

A study by Koru and Tian [Koru and Tian, 2004] was conducted to investigate defect handling mechanisms for 119 projects in the open-source community and they have found that more than 70 percent of the respondents reported defects discovered in their system consistently by using a bug tracking system. Paulson et al. [Paulson et al., 2004] have conducted a study to compare several aspects of system development between open-source and closed-source projects. They have found that creativity is more widespread in open-source projects and defects are found and fixed more rapidly in open-source projects compared to closed-source projects. Another study was conducted by Mockus et al. [Mockus et al., 2002] to investigate the claim that open-source style software development has the capability to compete successfully and in most cases, even displace traditional commercial development methods. They had looked into the aspects of developer participation, core team size, code ownership, productivity, defect density and problem resolution intervals in order to understand the methods used for software development in open-source projects.

Several researchers conducted empirical work on open-source systems maintenance, for example, [Chen et al., 2004], [Feitelson et al., 2006], [Koru and Tian, 2005], [Kozlov et al., 2007], [Lee et al., 2007] and [Zhou and Davis, 2005]. In particular, Zhou and Davis [Zhou and Davis, 2005] studied the bug arrival pattern in eight open-source projects using a Weibull distribution. Interestingly, they also looked at the correlation between bug arrival rate and downloads and page view and they found generally low correlation between the variables. On a similar note, Ferenc. et al. [Ferenc et al., 2004] studied several releases of the Mozilla project by measuring the object-oriented metrics of the project in order to confirm the hypothesis given by Basili et. al [Basili et al., 1996]. Furthermore, they used the same metrics to predict fault-proneness of seven versions of Mozilla. The work of Chen et al. [Chen et al., 2004] discussed the usage of ChangeLog files to investigate changes in open-source software, especially from the development
2.5 Open-Source Research

and maintenance perspective.

In addition, Porter et al. [Porter et al., 2006] discussed the key challenges of OSS and described how their quality assurance techniques called Skoll (distributed continuous quality assurance (DCQA) techniques and processes) help to resolve key challenges in developing and validating open-source software. Meanwhile, Hahsler and Koch [Hahsler and Koch, 2005] discussed in detail a methodology for collecting repository data on a large number of open-source software projects from a single project hosting and community site. Another data collection methodology was presented by Massey, who used uniform software tools such as CVS/RCS with open formats and interfaces to collect data from open-source repository [Massey, 2005].

2.5.1 Open-Source Software as a Research Base

The emergence of the open-source software development paradigm has stirred interest among software developers and everyday software users in terms of how software is developed and also in terms of general usage of the software. According to the definition given by Open-Source Initiative [www.opensource.org], open-source software (OSS) allows users to have access to the source code of the software, the freedom to use the software as they see fit, modify the software to create derived work, and redistribute the derivative software for free or at a charge. The users of the software could modify or use the software according to their own needs.

One of the main strength of OSS is user participation in reporting bugs in the software. Some professional users can diagnose problems, suggest fixes and help improve the code faster than having the developers figure out everything by themselves [Raymond, 2000]. Raymond discusses the philosophy of open-source development and emphasizes several points pertaining to the concept of development methodology of open-source software [Raymond, 2000]. Particularly, he mentions several “mantra” based on Linux development that have been widely adopted by open-source community in present day. One of them is “Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging”, which stresses the importance of user participation in software development. Another one is “Release early. Release often. And listen to your customers”, follows Linus Torvald's development style that encourages developers to release their software early enough to attract participation of other developers or users to assist in improving the software. One famous “mantra” that has become synonymous with open-source development is “Given enough eyeballs, all bugs are shallow”, this contributes to the tradition of open-source development
which treats users as co-developers, as practiced by Linus Torvald during the
development of Linux. This proves to be successful in the development of major
open-source software nowadays including OpenBSD, Mozilla, Apache, MySQL and
many more. This success is due to the fact that finding or detecting problems
as well as fixing them can be done rapidly as users can help report and fix the
problems.

Generally, research conducted on commercial product development is ob-
structed by restricted access to the development process and selectively released
data. Companies that commercialize their software products are in most cases not
interested in sharing the product’s source code due to the risk of code spilling over
to competitors or “software pirates”. On the contrary, due to their development
practices, open-source software projects show considerably high transparency
of data for research. The software’s source code is generally available from
repositories that host the project, for instance, SourceForge and Freshmeat, as
well as from the websites of the open-source projects themselves, such as Apache
[www.apache.org], Mozilla [www.mozilla.org] and OpenBSD [www.openbsd.org].

This enables researchers to investigate the inner, technical workings of the
software and sometimes product development process [von Krogh and Spaeth,
2007]. This transparency of technical data allows great opportunity to study
certain issues such as functionality, software architecture, file size, language,
software component reuse, application protocol interfaces, bug identification and
fixing and individual contribution levels [MacCormack and J. Rusnak, 2006].
For example, in SourceForge, information about the projects, such as activity
percentile, rank, number of downloads, number of developers, bug tracking reports,
development status and many more are provided for public reference.

Furthermore, most projects host mailing lists dedicated to various aspects
of the software product and the project. Some lists may focus on technical
development issues, while others may deal with user assistance, general user
feedback, or discussions regarding the “philosophy” of the project [von Krogh and
Spaeth, 2007]. These lists represent exceedingly valuable data for researchers
because the make discussions available that can be used to examine multiple issues
in open-source development. The retrieval of both technical data and mailing
lists data is in many cases possible for all researchers, and not only restricted
to those who have exclusive relationships with developers. For many open-source
projects, such data are extensive, covering millions of lines of code and thousands
of messages and, therefore, are useful for various forms of quantitative analysis.

Critics of open-source software often question the quality of open-source
software. Research using bug report data [Kuan, 2001] showed that flaws in
open-source software did not exceed those of commercial software that performed the same functions. A study by Franke and von Hippel [Franke and von Hippel, 2003] also showed that users in general were more satisfied with the software product which they could change to fit their specific needs. Furthermore, a study by von Krogh et al. revealed that software product quality was secured by allowing only a selected group of developers the possibility to implement changes to the software code, although the ideas, bug fixes, or software patches might have come from a large number of contributors [von Krogh and von Hippel, 2003].

Many researchers [Binkley and Schach, 1998], [Chen et al., 2004], [Hahsler and Koch, 2005], [Kozlov et al., 2007], [Rainer and Gale, 2005], [Subramaniam et al., 2009], [von Krogh and von Hippel, 2006], [Wedel et al., 2008] have used SourceForge as the place to obtain data for their work, thus, enhancing its credibility as an open-source repository fit for research data collection. Further details of this repository will be discussed in Chapter 4.

Feitelson et al. [Feitelson et al., 2006] conducted a research study on the success of open-source software based on number of downloads. They use Zipf's Law to characterize downloads as a measure of success for OSS projects. They categorize the projects as Superprojects (more than 1.1 million downloads), Successful (more than 1680 downloads) and Struggling (1680 downloads or fewer). Recently, other research was conducted by Subramaniam et al. [Subramaniam et al., 2009] to investigate the determinants of open-source software project success. They concluded that the success of open-source projects, especially those that obtained from SourceForge, can be measured by using three factors: developer interest, user interest and project activity levels. Lee at al. conducted another study to measure the success of OSS and they found that the key determinant of OSS success is user satisfaction which is influenced by software quality and community service [Lee et al., 2009].

The main reason open-source software is chosen as empirical data in this research is because it allows researchers to access project source code, artifacts and other details of the software through several online repositories which offer publicly available data source of a size, diversity and complexity not previously available. In the past, acquiring data for research purposes has been difficult since not all organizations are willing to share their confidential data for public access to protect the confidentiality of the data from competitors or were even afraid that the data will be misused by the people who acquire the data themselves. In contrast, open-source software is readily available online to be collected, analyzed, shared and used by any interested party. As mentioned in a publication by Yu et al. [Yu et al., 2005], an editorial in Empirical Software Engineering commented [Harrison, 2001]:
"As empirical software engineers, we should embrace this development (opensource software). Suddenly one of the greatest obstacles in the way of empirical software engineering has been cleared! Not only is source code available, but also defect reports, update logs, etc. For a change, we can now focus on the analysis rather than the data collection”.

2.6 Software Metrics

Software metrics refers to a broad range of measurements for computer software. Although the terms “measure”, “measurement” and “metrics” are often used interchangeably, it is important to note the subtle differences between them. Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimensions, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The IEEE Standard Glossary of Software Engineering Terms [IEEEStandard1990, 1990] defines metric as “a quantitative measure of the degree to which a system, component, or process possesses a given attribute.” In this thesis, the metrics selected are divided into two main groups: traditional structure metrics and object-oriented metrics.

2.6.1 Traditional Structure Metrics

Structure metrics look into the interactions between modules in a system and quantify such interactions. Several frameworks for structure metrics have been proposed, for example, control-flow structure or flowgraphs by McCabe [McCabe, 1976], information flow metrics by Henry and Kafura [Kafura and Henry, 1982], system partitioning measures by Belady and Evangelisti [Belady and Evangelisti, 1981], and the most common design structure metrics are the Fan-in and Fan-out metrics introduced by Myers [Myers, 1978] and Yourdon and Constantine [Yourdon and Constantine, 1979] and the system complexity model by Card and Glass [Card and Glass, 1990]. However, many of those metrics still need more verification using empirical data from software development projects.

2.6.1.1 Henry and Kafura’s Information Flow Measure

Henry and Kafura’s information flow measure is a well-known approach to measuring the total level of information flow between individual modules and the rest of a system [Kafura and Henry, 1982]. A local direct flow exists if either:

1. a module invokes a second module and passes information to it
2.6 Software Metrics

2. the invoked module returns a result to the invoker

A local indirect flow exists if the invoked module returns information that is subsequently passed to a second invoked module. A global indirect flow exists if information flows from one module to another via a global data structure. Using these notions, there are two particular attributes of the information flow.

- Fan-in: The number of local flows that terminate at the given module plus the number of data structures from which information is retrieved by the module
- Fan-out: The number of local flows that emanate from the given module plus the number of data structures that are updated by the module

In most cases, modules with a high Fan-in are relatively small and simple, and are usually located at the lower layers of the design structure [Yourdon and Constantine, 1979]. On the contrary, modules that are large and complex are likely to have low Fan-in. Generally, modules with high Fan-out indicate high span of control and usually translate to having bad design, and need refactoring. Therefore, modules or components that have a high Fan-in and high Fan-out may indicate poor design. From the complexity and defect point of view, modules with a high Fan-in are expected to have inverse or insignificant correlation with defect levels, and modules with a high Fan-out are expected to have a positive correlation with defect levels [Kan, 2003]. Henry and Kafura's information flow measure is defined as:

\[ C_p = (\text{Fan-in} \times \text{Fan-out})^2 \]  

Henry and Selig [Henry and Selig, 1990] attempted to incorporate the module complexity and structure complexity by defining a hybrid form of their information-flow metric as:

\[ HC_p = C_{ip} \times (\text{Fan-in} \times \text{Fan-out})^2 \]  

where \( C_{ip} \) is the internal complexity of procedure \( p \), which can be measured by any module complexity metrics such as McCabe's cyclomatic complexity.

2.6.1.2 McCabe's Cyclomatic Complexity Measure

The McCabe's Cyclomatic Complexity metric [McCabe, 1976] was designed to indicate a program's testability and understandability (maintainability). It is the
Chapter 2: Literature Review

classical graph theory cyclomatic number, indicating number of regions in a graph. As applied to software, it is the number of linearly independent paths that comprise the program and can be used to indicate the effort required to test a program. To determine the paths, the program/module procedure is represented as a strongly connected graph with unique entry and exit points. For a program with flowgraph G, the general formula to compute cyclomatic complexity is:

\[ V'(G) = e - n + 2p \] (2.3)

where

\[ V(G) = \text{Cyclomatic number of flowgraph G} \]
\[ e = \text{Number of edges} \]
\[ n = \text{Number of nodes} \]
\[ p = \text{Number of unconnected parts of the graph} \]

The cyclomatic complexity metric is additive. The complexity of several graphs considered as a group is equal to the sum of the individual graphs’ complexities. However, it ignores the complexities of sequential statements. The metric does not distinguish different kinds of control flow complexity such as loops versus IF-THEN-ELSE statements or CASE statements versus nested IF-THEN-ELSE statements [Kan, 2003].

In order to have good testability and maintainability, McCabe recommended that no program module should exceed a cyclomatic complexity of 10. Since the complexity metric is based on decisions and branches, which is consistent with the logic pattern of design and programming, it appeals to software professionals. Since its introduction, cyclomatic complexity has become an active area of research and practical applications.

Since its inception in 1976, this metric has received mixed reviews from several researchers in software engineering. Many tried to observe the relationship between McCabe’s cyclomatic complexity and various variables and variety of results have been reported.

For example, Troster [Troster, 1992] conducted a software metrics study of a large software product written in SQL that consisted of about 1300 modules. He found a relatively good correlation between McCabe’s cyclomatic complexity index and the number of test defects \( r = 0.48 \) with p-value = 0.0001. Other studies found that the complexity index also correlates strongly with program size, i.e. lines of code. The question is, will the correlation between complexity and defect remain significant after program size is controlled? In his work, Troster observed that the LOC count also correlated with the number of test defects quite strongly \( r = \)
2.6 Software Metrics

0.49) and p-value = 0.001. To investigate the effect of program size, he calculated the Pearson correlation between McCabe's complexity index with testing defect per KLOC. He found that the correlation totally disappeared with \( r = 0.002 \) and p-value = 0.9415. He also computed the Spearman's rank-order correlation coefficient and found a considerably satisfactory correlation between McCabe's complexity and defect rate (Spearman's correlation = 0.27) with p-value = 0.0001.

Another study by Gill and Kemerer [Gill and Kemerer, 1991], explored the relationship between McCabe's Cyclomatic Complexity and software maintenance productivity using software modules written in Pascal and FORTRAN. Their paper proposed the use of a transformed version of metric, known as "complexity density", where the ratio of the cyclomatic complexity of the module to its length in non-comment single lines of code (NCSLOC) is calculated. This ratio is meant to represent the normalized complexity of a module and its likely level of maintenance task difficulty. In addition, Lind and Vairavan [Lind and Vairavan, 1989] conducted a study on the relationship between several complexity measures with program development effort in Pascal and FORTRAN systems. In order to further understand the linkage between the measurement metrics and program development effort, they first tried to correlate the metrics and the density of program changes. The results they obtained indicate that program change density declines with increasing metric values up to a certain minimum value, beyond this minimum value, the program change density actually increases with an increase in the value of the metrics.

However, this metric has received criticism from several researchers, such as, Kitchenham et al. [Kitchenham et al., 1990], Shepperd [Shepperd, 1988] and Shepperd and Ince [Shepperd and Ince, 1994]. In particular, Shepperd and Ince [Shepperd and Ince, 1994] exposed a number of theoretical concerns with this metric as listed below:

1. McCabe was originally concerned with the metrification of FORTRAN programs in which the mapping from source code to directed graph was clear. Such a mapping is not clear for other languages such as fourth-generation languages and concurrent languages such as Ada, in which, for example, it is not clear how its exception handling construct can be handled within McCabe's framework.

2. Cyclomatic complexity value of 1 will be generated by any length of linear code. Therefore, the metric is insensitive to complexity contributed by, for example, a large number of interdependent assignment statements. Hence, function-bound software represents a major class of system for which the metric is a poor predictor.
3. Cyclomatic complexity is insensitive to the structuring of software. A number of studies have pointed out that application of structure-improving heuristics can lead to an increase of cyclomatic complexity, and many rules of good programming do not lead to a decrease of the metric. The probable reason for this anomaly is that McCabe takes a lexical view of program code, rather than a structural view.

4. It has been shown that cyclomatic complexity increases when a designer uses the technique of factoring out duplicate code in order to increase modularization [Shepperd, 1988]. This is at variance with current software engineering ideas on design.

5. The metric ignores factors that are becoming increasingly important, such as data and functional complexity.

Besides the theoretical objections, there exist arguments on the empirical evidence of this metric [Shepperd, 1988] and [Shepperd and Ince, 1994]. Both papers discussed several empirical research conducted by prior researchers. In particular, Shepperd [Shepperd, 1988] presented a list of results of the empirical validations of cyclomatic complexity and concluded that the results are not extremely compelling either at the program level or for the studies on individual modules. The major exception is the study by Henry et al. [Henry et al., 1981] of 165 procedures from the UNIX operating system, where the results show a strong correlation between $v(G)$ and module error rates. However, this result may be slightly artificial since they seem to have filtered out all error-free modules.

Based upon the observation that large modules tend to contain more errors than small modules, a study by Basili and Perricone [Basili and Perricone, 1984] uses error density (errors per thousand lines of code (LOC)) as a size-normalized metric of software error-proneness. Surprisingly, their finding showed that error density diminishes with increasing cyclomatic complexity. Work by Shen et al. [Shen et al., 1995] provides support to this result, although there is disagreement as to whether error density is an appropriate means of size normalization since module size and error density do not appear to be independent.

The clearest result from the empirical studies is the strong relationship between cyclomatic complexity and LOC. For example, the study of Henry et al. [Henry et al., 1981] showed a fairly strong correlation between those variables. They suggest that software can be characterized as either decision or computation bound. In cases of decision-bound software such as UNIX, cyclomatic complexity closely correspond to LOC. In computation-bound software, with sizeable portions of linear code this correspondence will be extremely marginal.
2.6 Software Metrics

Similarly, Table 2 of the paper by Shepperd and Ince [Shepperd and Ince, 1994] shows 18 experiments conducted to find correlation between cyclomatic complexity and various dependent variables. Out of the 18 experiments, only five reported cyclomatic complexity as a strong predictor, six claimed it as a weak predictor and the remaining seven studies reported that it is not a predictor at all.

In another study, Kitchenham et al. [Kitchenham et al., 1990] investigated the relationships between these metrics:

- Design metrics: informational fan-out (IFO), informational fan-in (IFI) and information flow complexity (IFC)
- Codes metrics: size in lines of code (LEN) and control flow in branches (CF)
- Metrics to assess the final characteristics of a procedure: number of known errors (KE), number of planned changes (CHNG) and subjective complexity (SC)

The results show that although there is evidence of an overall trend for high values of informational fan-out, control flow and size metrics to coincide with high values of errors, changes and subjective complexity, the relationships are weak.

Although this metrics has received mixed responses by the software engineering community, it is still relevant in the context of this research. Calculation of cyclomatic complexity for each method in object-oriented (OO) systems correspond to the use of the metric on each module of structured systems. The work of Tegarden et al. [Tegarden et al., 1992] suggests that the use of inheritance and/or polymorphism decrease the cyclomatic complexity value in the OO systems compared to non-OO systems. Even though this metrics does not support the properties of the OO systems, such as encapsulation, inheritance and polymorphism, it is useful in measuring the application programming interface (API) calls, assignments, binary expressions, keyword messages, nested expressions, parameters, primitive calls, temporary variables, and unary expression [Lorenz and Kidd, 1994].

2.6.1.3 Card and Glass’s System Complexity Model

This work is based on that of Card and Glass at the Software Engineering Laboratory (SEL), sponsored by NASA Goddard Space Flight Center (GSFC). Their findings will be discussed in detail throughout this thesis. Card and Glass hypothesized that the complexity of a system can be broken down into 3 main components: data, structural and procedural complexity (all established as part
of design). According to Card and Glass, system design prescribes the strategy for implementing the requirements. The difficulty of that strategy results in the possibility of the developers making errors. Hence, system design tends to be an early source of errors [Card and Glass, 1990]. Due to the fact that coding is largely a translation process once design is complete, most software complexity resides in the design. Card and Glass carried out a study of 8 similar projects taken from the Software Engineering Laboratory database and all projects were “ground-based attitude determination systems for spacecraft in near-earth orbit”, written in FORTRAN (RATFOR). Some projects did not provide a complete set of design materials, therefore, many parts of the study relied on design product data extracted from software source code.

Based on various approaches to structure complexity and module complexity measures, Card and Glass [Card and Glass, 1990] developed a system complexity model.

$$C_t = S_t + D_t$$  \hspace{1cm} (2.4)

where

- $C_t$ = Total System complexity
- $S_t$ = Total Structural complexity
- $D_t$ = Total Data complexity

They defined average system complexity as

$$C = C_t/n = S_t/n + D_t/n$$  \hspace{1cm} (2.5)

where

- $C$ = Average system complexity
- $C_t$ = Total system complexity
- $S_t$ = Total structural complexity
- $D_t$ = Total data complexity
- $n$ = Number of modules in system

Structural complexity is further defined as

$$S = \frac{\sum f^2(i)}{n}$$  \hspace{1cm} (2.6)

where

- $S$ = Average Structural complexity
- $f(i)$ = Fan-out of module $i$
2.6 Software Metrics

\( n = \text{Number of modules in system} \)

They further defined data complexity as

\[ D_i = \frac{V(i)}{f(i) + 1} \]  \hspace{1cm} (2.7)

where

- \( D_i \) = Data complexity of module \( i \)
- \( V(i) \) = I/O variables in module \( i \)
- \( f(i) \) = Fan-out of module \( i \)

Finally, the overall data complexity is defined as the average of data complexity of all new modules. In Card and Glass's model, only new modules enter the formula because most often, the entire system consists of reused modules, which have been designed, used, aged, stabilized in terms of reliability and quality.

According to Card and Glass [Card and Glass, 1990], system complexity is a sum of structural complexity and overall data complexity. Structural complexity is defined as the mean (per module) of squared values of Fan-out. According to the findings in the literature [Card and Glass, 1990], the systems they studied mostly have either low or no Fan-in value, therefore, Fan-in is not considered as an important complexity indicator. Data complexity of a module is defined as a function that is directly dependent on the number of I/O variables and inversely dependent on the number of Fan-outs as shown in Equation 2.7. The rationale is that the more I/O variables in a module, the more functionality needs to be accomplished by the module and therefore, the higher internal complexity. In contrast, more Fan-out means that functionality is deferred to modules at lower levels, therefore, the internal complexity of a module is reduced. Finally, the overall data complexity is defined as the average of data complexity of all new modules. In Card and Glass's model, only new modules enter the formula because most often, the entire system consists of reused modules, which have been designed, used, aged, stabilized in terms of reliability and quality.

In a study of eight software projects, Card and Glass found that the system complexity measure was significantly correlated with subjective quality assessment by a senior development manager and with the development error rate.
Chapter 2: Literature Review

Specifically, the correlation between system complexity and development defect rate was 0.83, with complexity accounting for fully 69 percent of the variation in error rate. Hence, the regression formula derived was

$$\text{Error rate} = 0.4 \times \text{Complexity} - 5.2 \quad (2.9)$$

In other words, each unit increase in system complexity increases the error rate by 0.4 (errors per thousand lines of code). In his book, Kan [Kan, 2003] stated that the Card and Glass model appears quite promising and has an appeal to software development practitioners because they provide guidelines on achieving a low complexity design. Therefore, he suggested that more validation studies being conducted on Card and Glass's model so that the model and its related methods may gain greater acceptance in the software development industry.

While Card and Glass's model is for the system level, the system values of the metrics in the model are averages of module level data. Therefore, it is possible to correlate these metrics to defect rate at the module level as shown in Equation 2.5.

Troster [Troster, 1992] has conducted a study to validate Card and Glass's system complexity model. He attempted to study the correlation between defects rate and several metrics including Card and Glass's Structural Complexity, Data Complexity and System Complexity measured from the high level and module level design perspectives, and also McCabe's Cyclomatic Complexity. The system under study was a software product written in SQL/DS. He analysed the data using Pearson correlation analysis and found that when the product is taken as a whole, the linear correlations between the metrics and defects are not so strong. However, when he ran the data through Spearman correlation analysis, he found that the rank-order correlation coefficients for these metrics are very similar to that of McCabe's (0.27). Specifically, the coefficients are 0.28 for $D_i$, 0.19 for $S_i$, and 0.27 for $C_i$. Another interesting finding in Troster's study is that the relationships between the design metrics and McCabe's cyclomatic complexity with defect rate are not linear. Kan [Kan, 2003] also stated that more research in this area will yield more insights into the relationships of various design and module metrics and their predictive power in terms of software quality. The details of Card and Glass's work will be discussed further in other chapters of this thesis.

2.6.1.4 Size

A software product can be described in terms of its size. Fenton and Pfleeger [Fenton and Pfleeger, 1997] suggest that software size can be described using three
2.6 Software Metrics

attributes:

- Length - physical size of the product
- Functionality - measures the functions supplied by the product to the user
- Complexity - includes measuring efficiency and problem complexity

1. Length

One major measurement property of a system is the length, which can be divided into three components: the specifications, the design, and the code. The measurement of specification length is a useful indicator of how long the design could be, and later the design length can be used to predict code length [Fenton and Pfleeger, 1997].

The most commonly used measure of source code program length is the number of lines of code (LOC). However, there are several different methods to measure lines of code. For instance, many programmers use spacing and blank lines to make their programs easier to read. If lines of code are being used to estimate effort, then a blank code does not contribute the same amount of effort as a line implementing a difficult algorithm. It is the same case as comment lines which can improve a program’s understandability, but the effort taken to write comments is less than the effort taken to write the code itself. Therefore, the lines of code measurement should be precisely defined in order to eliminate ambiguity in the calculation.

There are several general suggestions that blank lines and comments should not be counted. For instance, Conte et al. define a line of code as any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This definition includes all lines containing program headers, declarations, and executable and non-executable statements [Conte et al., 1986]. Meanwhile, Grady and Caswell report that Hewlett-Packard defines a line of code as a non-commented source statement, which is any statement in the program except for comments and blank lines [Grady and Caswell, 1987]. In Java, a statement is a line of code that ends with a semicolon [Deitel and Deitel, 1997]. A recent study by Zhang and Tan [Zhang and Tan, 2007] investigates the distribution of class sizes in a large Java system and they use the lines of code (LOC) as the measure of size.

The most widely accepted definition is the Hewlett-Packard definition of a line of code. This definition is also referred to as effective lines of code (eLOC). This research use eLOC as the measure of lines of code or executable statements.
2. Functionality

Another attribute used to measure size is functionality and one of the approaches used to measure the amount of functionality in a system is Albrecht's function points [Albrecht and Gaffney, 1983]. To compute the number of function points (FP), first, an unadjusted function point count (UFP) is calculated. The UFP value is obtained by calculating these following items:

- **External inputs**: Items provided by the user that describe distinct application-oriented data, e.g., file names and menu selections.
- **External outputs**: Items provided to the user that generate distinct application-oriented data, e.g., reports and messages.
- **External inquiries**: Interactive inputs requiring a response.
- **External files**: Machine-readable interfaces to other systems.
- **Internal files**: Logical master files in the system.

The adjusted function point (FP) can be calculated by multiplying UFC by a technical complexity factor, TCF, which involves 14 contributing factor. The formula to calculate TCF is:

\[
TCF = 0.65 + 0.01 \sum_{i=1}^{14} F(i)
\]  

(2.10)

This factor varies from 0.65 (if each \( F_i \) is set to 0) to 1.35 (if each \( F_i \) is set to 5). The final calculation of function points multiplies the unadjusted function-point count by the technical complexity factor:

\[
FP = UFC \times TCF
\]  

(2.11)

3. Complexity

There is considerably little work on measuring complexity, especially in regard to efficiency and problem complexity [Fenton and Pfleeger, 1997]. Complexity can be interpreted in different ways:

- **Problem complexity**: measures the complexity of the underlying problem.
- **Algorithmic complexity**: reflects the complexity of the algorithm implemented to solve the problem.
- **Structural complexity**: measures the structure of the software used to implement the algorithm.
2.6 Software Metrics

- Cognitive complexity - measures the effort required to understand the software.

In this research, length is used as a measure of size because it is straightforward and can be consistently applied to all systems being studied.

2.6.1.5 Coupling

Coupling is the degree of interdependence between modules [Yourdon and Constantine, 1979]. Generally, coupling is an attribute of pairs of modules, rather than of the design as a whole. The entire sets of module in the design exhibits global coupling, which can be derived from the coupling among the possible pairs. Constantine and Yourdon [Yourdon and Constantine, 1979] have outlined several factors that influence coupling:

1. Type of connection between modules: So-called minimally connected systems have the lowest coupling, and normally connected systems have lower coupling than those with pathological connections.

2. Complexity of the interface: This is approximately equal to the number of different items being passed (not the amount of data), the more items, the higher the coupling.

3. Type of information flow along the connection: Data-coupled systems have lower coupling than control-coupled systems, which have lower coupling than hybrid-coupled systems.

4. Binding time of the connection: Connections bound to fixed referents at execution time result in lower coupling than binding that takes place at loading time, which results in lower coupling than binding that takes place at linkage-edit time, which in turn results in lower coupling than binding that takes place at compilation time - all of which result in still lower coupling than binding that takes place at coding time.

There are no standard measures of coupling [Fenton and Pfleeger, 1997]. However, from the viewpoint of measurement theory, coupling satisfies some of the basic prerequisites for measurement. For instance, there are several well-established empirical relations involving coupling that suggest at least an ordinal scale of measurement. The types of coupling as given by Constantine and Yourdon [Yourdon and Constantine, 1979], and extended by Page-Jones [Page-Jones, 1988] are:
• **Data coupling:** Two modules are data coupled if they communicate by parameters, each parameter being an elementary piece of data. Data coupling or also known as input/output coupling is the necessary communication of data between modules. Modules cannot function as a single system performing an overall purpose, unless the outputs of some modules become the inputs of others. All elements are required to be communicated. Since modules must communicate, data coupling is unavoidable and is quite harmless as long as it’s kept to a minimum.

• **Stamp coupling:** Two coupled modules are stamp coupled if one passes to the other a composite piece of data with meaningful internal structure. This type of coupling may cause interdependency between otherwise unrelated modules.

• **Control coupling:** Two modules are control coupled if one passes to the other a piece of information intended to control the internal logic of the other. Control coupling covers all forms of connection that communicate elements of control. This may involve actual transfer of control, such as activation of modules, or it may involve the passing of data that change, regulate, or synchronize the target module.

• **Common coupling:** Two modules are common coupled if they refer to the same global data area. This type of coupling is undesirable, if the format of the global data must be changed, then all common coupled modules must also be changed.

• **Content coupling:** Two modules exhibit content coupling if one refers to the inside of the other in any way, for example, if one module branches or falls through into another, if one modules refers to or changes data within another, or if one module alters a statement in another.

Kramer and Kaindl applied the coupling theory in knowledge-based systems [Kramer and Kaindl, 2004]. They used the coupling and cohesion metrics to measure modularity, in terms of relations induced between slots of frames through their common references in rules. The work of Xia [Xia, 2000] discussed the concept of coupling, its modeling and measurement in detail where he derived a mathematical model for measuring module coupling. AlGhamdi et al. [AlGhamdi et al., 2002] introduced and discussed a new tool to measure inheritance coupling in object-oriented systems.

Other researchers also reported new findings in coupling measurement [Briand et al., 1997], [Briand et al., 1999d], [Briand et al., 1999e], [Briand et al., 1999b], [Hall et al., 2005]. In particular, Briand et al. [Briand et al.,
2.6 Software Metrics

1999d] recommended that coupling metric is an important structural dimension to consider in building quality models for object-oriented design. They suggested strong emphasis should be put on method invocations and import coupling because they found that these properties are strong indicator of fault-proneness. In a similar line, Briand et al. [Briand et al., 1999b] discussed in detail several coupling frameworks proposed by Briand et al. [Briand et al., 1997], Eder et al.[Eder et al., 1994] and Hitz and Montazeri [Hitz and Montazeri, 1995b]. Based on the various frameworks, they proposed a new unified framework for coupling in object-oriented systems [Briand et al., 1999b]. The framework consists of six criteria, each criterion determining one basic aspect of the resulting measure. The six criteria of the framework are:

1. Type of connection - choosing a type of connection implies choosing the mechanism that constitutes coupling between two classes.

2. Locus of impact - It has to be decided whether to count import or export coupling:
   - Import coupling analyzes the attributes, methods, or classes in their role as clients of other attributes, methods, or classes.
   - Export coupling analyzes the attributes, methods, or classes in their role as servers to other attributes, methods, or classes.

3. Granularity - The granularity of the measure is the level of detail at which information is gathered. The granularity of the measure is determined by two factors:
   - The domain of the measure, i.e., what components are to be measured.
   - How exactly the connections are counted.

4. Stability of server - Two different categories of class stability are defined:
   - Unstable classes - these are classes which are subject to development or modification in the project at hand. Unstable classes are problem domain classes which are being developed exclusively for the system, or are being adapted from other systems.
   - Stable classes - classes that are not subject to change in the project at hand. Stable classes are classes imported from libraries, or classes reused verbatim from other systems.

5. Direct or indirect connections - One needs to decide whether to count direct connections only or also indirect connections. For example, if a method $m_1$
invokes a method $m_2$, which in turn invokes a method $m_3$, this also means that $m_1$ indirectly invokes $m_3$. Methods $m_1$ and $m_3$ are indirectly connected.

6. Inheritance - Three aspects need to be considered with respect to inheritance:

- Is there a need to distinguish between inheritance based coupling and non-inheritance based coupling?
- How do we assign methods and attributes to classes?
- For method invocations: shall we consider static or polymorphic invocations?

In conclusion, they discussed and compared several coupling measures as a guideline for other researchers.

A recent study by Offut et al. [Offut et al., 2008] discussed several coupling measures based on Java source code for open-source systems. They examined four coupling types:

1. Parameter coupling - Any method call, including parameters. Focuses on the occurrence of an invocation of a call to a method or constructor through an object or class.

2. External/file coupling - Refers to classes that access the same external medium, including external files.

3. Inheritance coupling - Occurs when one class is a subclass or descendant of another. The coupling is made through inherited but not re-defined data members of a superclass by its subclass.

4. Global coupling - Refers to variables that are defined in one class and used in others.

In addition, they presented techniques for measuring couplings in object-oriented relationships between classes, specifically focusing on types of couplings that are not available until after the implementation is finished. Furthermore, they presented a static analysis tool that measures couplings among classes in Java packages called Java Code Analysis Tool (JCAT).

2.6.1.6 Cohesion

The idea of cohesion was introduced by Constantine, and the term "cohesion" means the degree of functional relatedness of processing elements within a single
2.6 Software Metrics

Module [Yourdon and Constantine, 1979]. According to Yourdon and Constantine, module cohesion can be conceptualized as the cement that holds the processing elements of a module together. In other words, a high degree of module cohesion is an indication of close approximation of inherent problem structure.

In a sense, cohesion and coupling are interrelated. The greater the cohesion of individual modules in the system, the lower the coupling between modules will be [Yourdon and Constantine, 1979]. Cohesion is a way to tell how well a system is partitioned into modules. Ensuring that all modules have good cohesion is the best way to minimize coupling between the modules [Page-Jones, 1988].

From the early studies and later refinements, Stevens et al. [Stevens et al., 1972] developed a level/scale of cohesion as a measure of black boxness of a module, which later turns out to be a good measure of the maintainability of a module [Page-Jones, 1988]. There are seven levels of cohesion distinguishable by seven associative principles [Yourdon and Constantine, 1979]:

1. **Coincidental cohesion**: Coincidental cohesion occurs when there is little or no constructive relationship among the elements of a module.

2. **Logical cohesion**: Logical cohesive module is one whose elements contribute to activities of the same general category in which the activity of activities to be executed are selected from outside the module.

3. **Temporal cohesion**: Temporal cohesive module is one whose elements are involved in activities that are related in time.

4. **Procedural cohesion**: Procedurally cohesive module is one whose elements are involved in different and possibly unrelated activities in which control flows from each activity to the next.

5. **Communicational cohesion**: Communicational cohesive module is one whose elements contribute to activities that use the same input or output data.

6. **Sequential cohesion**: Sequentially cohesive module is one whose elements are involved in activities such that output data from one activity serves as input data to the next.

7. **Functional cohesion**: Functional cohesion is whatever is not sequential, communicational, procedural, temporal, logical, or coincidental.

Quite a number of cohesion validation using empirical data have been conducted by Briand et al. [Briand et al., 1998a], [Briand et al., 1998b], [Briand et al.,
1999c], [Briand and Wüst, 2001] and [Briand et al., 2001]. In particular, Briand et al. presented a unified framework for cohesion measurement in object-oriented systems, where they studied various cohesion measures and provided detailed guidelines to compare, evaluate, and use the cohesion measures [Briand et al., 1998b]. They discussed the frameworks proposed by Bieman and Kang [Bieman and Kang, 1995], Briand et al. [Briand et al., 1993], [Briand et al., 1994], Chidamber and Kemerer [Chidamber and Kemerer, 1994], Eder et al. [Eder et al., 1994], Henderson-Sellers [Henderson-Sellers, 1996b], Hitz and Montazeri [Hitz and Montazeri, 1995a] and Lee et al. [Lee et al., 1995]. In the end, they proposed a unified framework for cohesion measurement in object-oriented systems based on the following criteria:

1. Type of connection - The mechanism that makes a class cohesive. A connection within a class is a link between elements of the class (attributes, methods, or data declarations).

2. Domain of the measure - Specifies the objects to be measured i.e., methods, classes, set of classes and system.

3. Direct or indirect connections - The decision whether to count direct connections only or indirect connections need to be made. For example, consider a method $m_1$ which is similar to a method $m_2$ (connection type 3), which in turn is similar to method $m_3$. Then methods $m_1$ and $m_2$ are directly connected through a connection through a connection of type 3, as are methods $m_2$ and $m_3$. Methods $m_1$ and $m_3$ are indirectly connected.

4. Inheritance - Two aspects are to be considered regarding inheritance:
   - How do we assign methods and attributes to classes?
   - For method invocation: Shall we consider static or polymorphic invocations?

5. Access methods and constructors - Access methods and constructors may artificially increase or decrease the values for cohesion measures.

In another investigation, Briand et al. explored the relationships between object-oriented coupling, cohesion and inheritance measures and the probability of fault detection in system classes during testing. They found that cohesion measures do not have a significant impact on fault-proneness of a class [Briand et al., 1998a].
2.6 Software Metrics

2.6.2 Object-oriented Metrics

The measures of object-oriented systems are derived from traditional design techniques, for example, coupling and cohesion and then interpreted for object-oriented approaches. Chidamber and Kemerer have suggested measures for object-oriented systems [Chidamber and Kemerer, 1994], also known as Chidamber and Kemerer Object-Oriented metric suite (CK OO metrics). The basis for the empirical relations systems in the Chidamber and Kemerer's work is the set of "ontological principles" proposed by Bunge [Bunge, 1979] and later applied to object-oriented systems by Yand and Weber [Yand and Weber, 1990]. In the latter work, the world is viewed as being composed of substantial individuals that possess a finite set of properties. Collectively, a substantial individual and its properties constitute an object. A class is a set of objects that have common properties, and a method is an operation on an object that is defined as part of the declaration of the class [Fenton and Pfleeger, 1997].

Attributes such as coupling, cohesion, object complexity, and scope of properties are then defined in Bunge's "ontological" terms. For instance, in Bunge's terminology, two objects are coupled if and only if one of them acts upon the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a substantial individual traverses in time [Fenton and Pfleeger, 1997].

Lorenz and Kidd introduced several metrics to quantify software quality assessment for object-oriented systems [Lorenz and Kidd, 1994]. Eleven metrics introduced by Lorenz and Kidd are applicable to class diagrams and are classified into three metrics categories:

1. Class size metrics, which deal with quantifying an individual class:

   - Number of Public Methods (NPM): This is a count of the number of public methods in a class. It is used to help estimating the amount of work to develop a class.
   - Number of Methods (NM): The total number of methods in a class counts all public, private and protected methods. This metric is a useful indication of the classes which may be trying to do too much work themselves; i.e., they provide too much functionality.
   - Number of public Variables per class (NPV): This metric counts the number of public variables in a class. Lorenz and Kidd consider the number of variables in a class to be one measure of its size.
   - Number of Variables per class (NV): The total number of variables including public, private and protected variables.
Chapter 2: Literature Review

- Number of Class Variables (NCV).
- Number of Class Methods (NCM).

2. Class Inheritance metrics, which look at the quality of the classes use of inheritance:

- Number of Methods Inherited (NMI): This metric measures the number of methods inherited by a subclass.
- Number of Methods Overridden (NMO): A large number of overridden methods indicates a design problem, indicating that those methods were overridden as a design afterthought. It is suggested that a subclass should really be a specialization of its superclasses, resulting in new unique names for methods.
- Number of New Methods (NNM): The normal expectation for a subclass is that it will further specialize (or add) methods to the superclass object. A method is defined as an added method in a subclass if there is no method of the same name in any of its superclasses.

3. Class Internals metrics, which look at general characteristics of classes:

- Average Parameters per Method (APM): Defined as Total Number Parameters in a Class/Total Number of Methods. Lorenz and Kidd argue that APM should not exceed 0.7.
- Specialization Index (SIX): This metric looks at the quality of the classes use of inheritance. The specialization index measures to what extent subclasses redefine the behavior of their superclasses.

In a detailed treatment of software metrics for object-oriented (OO) systems, Whitmire [Whitmire, 1997] describes nine distinct and measurable characteristics of an OO design:

1. Size: Size is defined in terms of four views: population, volume, length and functionality. Population is measured by taking a static count of OO entities such as classes or operations. Volume measures are identical to population measures but are collected dynamically, at a given instance of time. Length is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of length). Functionality metrics provide an indirect indication of the value delivered to the customer by an application.

2. Complexity: Like size, there are many differing views of software complexity. One of the views of complexity is in terms of structural characteristics by examining how classes of an OO design are interrelated to one another.
3. Coupling: The physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects) represent coupling within an OO system.

4. Sufficiency: Sufficiency is defined as the “degree to which an abstraction possesses features in its abstraction, from the point of view of the current application.” In essence, a design component (e.g., a class) is sufficient if it fully reflects all properties of the application domain object that is modeling, which is, that the abstraction (class) possesses the features required of it.

5. Completeness: Completeness considers multiple points of view, asking the question: “What properties are required to fully represent the problem domain object?” Because the criterion for completeness considers different points of view, it has an indirect implication about the degree to which the abstraction or design component can be reused.

6. Cohesion: Like its counterpart in conventional software, an OO component should be designed in a manner that has operations working together to achieve a single, well-defined purpose. The cohesiveness of a class is determined by examining the degree to which “the set of properties it possesses is part of the problem or design domain”.

7. Primitiveness: A characteristic that is similar to simplicity, primitiveness (applied to both operations and classes) is the degree to which an operation is atomic, that is, the operation cannot be constructed out of a sequence of other operations contained within a class. A class that exhibits a high degree of primitiveness encapsulates only primitive operations.

8. Similarity: The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose is indicated by this measure.

9. Volatility: Volatility of an OO design component measures the likelihood that a change will occur.

Alternatively, Chidamber and Kemerer (CK) use the object-oriented concepts to define a number of metrics that are claimed to relate to some of the attributes of Bunge’s ontology [Chidamber and Kemerer, 1994]:

**Metric 1: Weighted methods per class (WMC)** This metric is intended to relate to the notion of complexity. For a class C with methods \(M_1, M_2, \ldots, M_n\), weighted respectively with ”complexity” \(c_1, c_2, \ldots, c_n\), the measure is calculated as
Chapter 2: Literature Review

\[ WMC = \sum_{i=1}^{n} c_i \]  

(2.12)

Metric 2: Depth of inheritance tree (DIT) In an object-oriented, the application domain is modeled as hierarchy of classes. This hierarchy can be represented as a tree, called the inheritance tree. The nodes in the tree, called the inheritance tree. The nodes in the tree represent classes, and for each such class, the DIT metric is the length of the maximum path from the node to the root of the tree. This measure relates to the notion of scope of properties. DIT is a measure of how many ancestor classes can potentially affect this class.

Metric 3: Number of children (NOC) This metric relates to a node (class) of the inheritance tree. It is the number of immediate successors of the class.

Metric 4: Coupling between object classes (CBO) For a given class, this measure is defined to be the number of other classes to which the class is coupled.

Metric 5: Response for class (RFC) This measure captures the size of the response set of a class. The response set of a class consists of all the methods called by local methods. RFC is the number of local methods plus the number of methods called by local methods.

Metric 6: Lack of cohesion metric (LCOM) The cohesion of a class is characterized by how closely the local methods are related to the local instance variables in the class. LCOM is defined as the number of disjoint (non-intersecting) sets of local methods.

However, the CK suite of metrics did not account for potential complexity that arises from certain other OO design factors such as encapsulation and polymorphism. Subsequent researchers proposed extensions and modification to the initial set of CK metrics highlighting these gaps. For instance, Abreu [Abreu and Melo, 1996] proposed extensions to measure encapsulation via metrics such as the Method Hiding Factor (MHF) and the Attribute Hiding Factor (AHF), which denote the information hiding aspects of a class. Abreu also proposed a measure of polymorphism, the Polymorphism Factor (PF), which denotes the ability of OO objects to take different forms based on their usage context. Another addition was also made to the CK metrics by Briand et al. [Briand et al., 1999e] where they proposed a comprehensive suite of measures to quantify the level of class coupling during the design of object-oriented systems. These metrics are complementary to Chidamber and Kemerer’s measures as quality predictors. In his paper, Zuse presented foundations of the properties of object-oriented software measures. The measures such as Dempster-Shafer Function of Belief, the Kolmogoroff axioms and the DeFinetti axioms are introduced [Zuse, 1996].
The correctness and the usefulness of the CK metrics suite have also been questioned. Churcher and Shepperd criticized CK metrics on the grounds that there is at present no consensus on the underlying attributes, for instance they argued that even the notion of “methods per class” is ambiguous [Churcher and Shepperd, 1995], CBO is criticized for naively treating all kinds of coupling as equal [Hitz and Montazeri, 1996] and LCOM is criticized for being counter-intuitive [Hitz and Montazeri, 1996]. Despite the criticism, the CK metrics have been widely cited and adopted since they are simple and intuitive to use and they have shown their usefulness in constructing prediction systems for size and number of defects. The main advantage of the CK metrics is their availability in the design stage of the software development process, when the classical size measure is not yet available.

Sherif and Sanderson [Sherif and Sanderson, 1998] applied the CK OO metrics to measure the metrics for two projects, UGC and SEQGEN which were written in C++. They found that the metrics gave good results and insights into comparing the complexity of the two projects and also the complexity of individual classes within a project. Tang et al. [Tang et al., 1999] attempted to correlate the CK metrics with defects and they found WMC is a good predictor of faulty classes and RFC is a good indicator for faults. Later, Wilkie and Harmer [Wilkie and Harmer, 2002] introduced a tool, Extensible Metrics toolBench for Empirical Research (EMBER) to measure the complexity of object-oriented software systems using CK metrics. Li and Henry used CK OO metrics plus several others including some size metrics (such as the number of attributes plus the number of local methods), to determine whether object-oriented metrics could predict maintenance effort. For the weights in WMC, they used cyclomatic numbers. On the basis of an empirical study using regression analysis, they concluded that these measures are useful. In addition, they claim that the Chidamber-Kemerer metrics “contribute to the prediction of maintenance effort over and beyond what can be predicted using size metrics alone” [Li and Henry, 1993]. El-Emam et al. have investigated the confounding of class size on the validity of object-oriented metrics. They studied a large C++ telecommunications framework using the CK metrics and a subset of Lorenz and Kidd metrics [Lorenz and Kidd, 1994] and they found that before controlling for size, the results indicate there are correlations between the metrics and fault proneness. However, after controlling for size, none of the metrics they studied associated with fault-proneness anymore [El-Emam et al., 2001]. This study was supported by Subramanyam and Krishnan [Subramanyam and Krishnan, 2003].

Moreover, Binkley and Schach [Binkley and Schach, 1998] conducted an investigation of several coupling measures, including the CBO and NOC metrics.
of the CK suite in two university software applications and they found that the coupling measure was associated with maintenance changes made in classes due to field failures. Besides the studies mentioned above, there are other validation studies that have been conducted by other researchers, for example, Basili et al. [Basili et al., 1996] conducted a study of eight medium-sized systems developed by students and they found that five (WMC, DIT, RFC, NOC and CBO) out of six CK OO metrics appear to be useful to predict class fault-proneness during the high and low level design phases of the life-cycle. In a commercial setting, Chidamber et al. [Chidamber et al., 1998] observed that higher values of coupling and the cohesion metrics in the CK suite were associated with reduced productivity and increased rework/design effort. In their work, Cartwright and Shepperd [Cartwright and Shepperd, 2000] analyzed a medium-sized telecommunication system written in C++. They studied the inheritance measures from the CK suite (DIT, NOC) and found that both these measures were associated with defect density of classes. On similar lines, Subramanyam and Krishnan [Subramanyam and Krishnan, 2003] reported their findings in their research based on industry data from software developed in two popular object-oriented programming languages, C++ and Java. They investigated the role of CK metrics in determining software defects and they discovered that even after controlling for the size of the software, these metrics are significantly associated with defects. Furthermore, they found that the effects of these metrics on defects vary across the samples from two programming languages, C++ and Java. Additionally, Rosenberg et al. [Rosenberg et al., 1991] of NASA published guidelines specifying that any class that meets at least two of the following criteria needs to be flagged and investigated for possible refactoring:

- Response for class more than 100
- Coupling between object classes more than 5
- Response for class more than 5 times the number of methods in the class
- Weighted methods per class more than 100
- Number of methods more than 40

2.7 Quality Prediction Modeling

Software quality has become critically important to many software products. The quality of software can be defined in various ways, but one of the most common definitions is the number of defects that arise in the final product [Fenton and Pfleeger, 1997]. However, defects cannot be measured considerably early in the
development process. As a result, various internal properties of products are measured and historical data are used in various models to predict quality.

In order to achieve an early indication of software quality, software is subjected to measurement. It would be of great benefit to predict the software system components that are likely to have a high error rate or that need high development effort. The terms bug, defect, failure, fault and error are often used to describe problems in software. According to the standard IEEE definitions \cite{IEEEStandard1990}, these terms are defined as:

- **Error**: A defect in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools.
- **Fault**: A concrete manifestation of errors within the software.
- **Failure**: A departure of the operational software system behavior from users’ expected requirement.

Koru and Tian took the initiative to look at defect handling mechanisms in open-source development and they reported that defect handling involves the activities of recording, tracking and resolving defects, with the realization that the way defects are handled can vary significantly from project to project \cite{KoruTian2004}. In addition, several research work embarked on the classification of defects in software development process \cite{BasiliShull2005}, \cite{Chillaregeetal1991}, \cite{Chillaregeetal1992}, \cite{HenningssonWohlin2004}, \cite{Nakamurawal2006}, \cite{Waliaetal2006} and \cite{Seamanetal2008}. Particularly, Chillarege et al. discuss in detail the concept of Orthogonal Defect Classification (ODC), which is the categorization of defects into classes that collectively point to the part of the process that needs attention \cite{Chillaregeetal1992}. They classified defects into eight types and then associate each defect type with the phase of the development process in which it occurs. The defect types and their associative development phase are illustrated in Table 2.3. According to Chillarege, the choice of defect types have evolved over time from the original five types \cite{Chillaregeetal1991}, to eight. As an extension to this concept, Seaman et al. \cite{Seamanetal2008} used ODC as a starting point for developing their own defect categorization scheme for historical data in NASA. Although their categorization scheme is NASA-specific and only addresses NASA-specific concerns, the processes that they have gone through can be used as guidelines to improve the process of defects categorization in the future, since they include several recommendations for categorizing defects, based on their experience.
One of the internal property of software products, complexity, can be measured using many techniques applied on source code, design and several other software artifacts [Card and Glass, 1990], [Kafura and Henry, 1982] and [McCabe, 1976]. It is commonly observed and intuitively believed that there is a positive correlation between complexity and defect count. According to Munson and Khoshgoftaar [Munson and Khoshgoftaar, 1992], although the defect-failure relationship is not straightforward, the common intuition is that a high complexity also leads to high number of failures, thus having a negative effect on quality and reliability. A study was conducted by Koru and Tian [Koru and Tian, 2003] to investigate the correlation between complexity and defect, where they compared and characterized the similarities and differences between the high defect (HD) and high complexity (HC) modules. Interestingly, they found that the most complex modules often have an acceptable quality and HD modules are not typically the most complex ones. Basili and Perricone [Basili and Perricone, 1984] examined FORTRAN module with fewer than 200 lines of code for the most part and found higher defect density in the smaller modules. They also found that the relationship between module size and cyclomatic complexity is strong ($r^2 = 0.94$), however, their findings do not support the belief that more complex modules are more error prone than less complex ones. Card and Glass reported similar finding, that most smaller modules have higher defect density and concludes that no relationship exists between defect density and module size [Card and Glass, 1990].

Furthermore, there has been much other research work conducted to learn more about the correlations between complexity metrics and defects. In particular, Khoshgoftaar and Munson [Khoshgoftaar and Munson, 1990] investigated some aspects of the relationship between program complexity measures and program errors which occur during development. They measured the relationships between the variables using regression analysis and they found a strong relationship between program errors and the complexity domains of program structure and size. Hence, they concluded that studied complexity domains can be used to predict program errors. Another study conducted by Ohlsson and Alberg [Ohlsson and Alberg, 1996] at Ericsson Telecom AB, supports the assumption of using design and complexity measures to predict the most fault-prone modules early during development.

In the context of building quantitative models of software faults, it has been argued that considering faults causing field failures is a more important question to address than faults found during testing [Binkley and Schach, 1998]. It has been argued that it is the ultimate aim of quality modeling to predict post-release fault-proneness [Fenton and Neil, 1999b]. In at least one study, it was found that pre-release fault-proneness is not a good surrogate measure for post-release
### Table 2.3 - Orthogonal Defect Classification

<table>
<thead>
<tr>
<th>Defect Type</th>
<th>Description</th>
<th>Process Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Affects significant capability, end-user interfaces, product interfaces, interface with hardware architecture, or global datastructure(s) and should require a formal design change.</td>
<td>Design</td>
</tr>
<tr>
<td>Interface</td>
<td>Corresponds to errors in interacting with other components, modules or device drivers via macros, call statements, control blocks, or parameter lists.</td>
<td>Low Level Design</td>
</tr>
<tr>
<td>Checking</td>
<td>Addresses program logic that has failed to properly validate data and values before they are used.</td>
<td>Low Level Design or code</td>
</tr>
<tr>
<td>Assignment</td>
<td>Indicates a few lines of codes, such as the initialization of control blocks or datastructure.</td>
<td>Code</td>
</tr>
<tr>
<td>Timing/Serialization</td>
<td>Errors that are corrected by improved management of shared and real-time resources.</td>
<td>Low Level Design</td>
</tr>
<tr>
<td>Build/Package/Merge</td>
<td>Describes errors that occur due to mistakes in library systems, management of changes, or version control.</td>
<td>Library tools</td>
</tr>
<tr>
<td>Documentation</td>
<td>Affects both publications and maintenance notes.</td>
<td>Publications</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Includes efficiency or correctness problems that affect the task and can be fixed by implementing algorithm or local data structure without the need for requesting a design change.</td>
<td>Low Level Designs</td>
</tr>
</tbody>
</table>

fault-proneness, the reason posited being that pre-release fault-proneness is a function of testing effort [Fenton and Ohlsson, 2000]. Prior work has identified important predictors for field defects and has predicted field defects for commercial software systems [Khoshgoftaar et al., 1996], [Mockus et al., 2005], [Raaschou and Rainer, 2008] and [Ostrand et al., 2004]. Also, Li et al. [Li et al., 2004], [Li et al., 2005b] and [Li et al., 2005a] discuss field defect predictors in open-source projects, for instance, OpenBSD and Tomcat. A recent study by Li et al. [Li et al., 2005a] embarked on exploring the existence of relationships between several predictors such as product metrics, development metrics, deployment and usage metrics, software and hardware configuration metrics, and field defects as the dependent variable. Their results show that deployment and usage metrics are the best predictor for field defects in OpenBSD.

This section will discuss the application of measurement theory in this research, the statistical package selected, the statistical analysis performed and other issues related to data analysis. The analysis of previously collected data
were carried out to validate the design complexity model introduced by Card and Glass [Card and Glass, 1990], as well as to produce the defect prediction models for open-source systems.

In this research, the defect prediction models are produced at system level. For this purpose, the statistical package GenStat 11.1 is used to carry out the statistical analysis for defect prediction based on several design metrics as discussed in Section 2.5.1.3. The number of independent variables or predictors selected for this research is limited, because the aim is to produce a simple model because simple models are preferable to more complicated alternatives since they involve less effort, not only in calculation, but more importantly, in metrics collection. Moreover, they tend to be more robust since they have fewer problems of collinearity [Cartwright and Shepperd, 2000]. This statement is supported by Iannino et al. [Iannino et al., 1984], who produced a set of criteria for model assessment and comparison, listed as follows:

1. **Predictive validity**: The capability of the model to predict failure behavior or the number of defects for a specified time period based on the current data in the model.

2. **Capability**: The ability of the model to estimate with satisfactory accuracy quantities needed by software managers, engineers, and users in planning and managing software development projects or controlling change in operational software systems.

3. **Quality of assumption**: The likelihood that the model assumptions can be met, and the assumptions' plausibility from the viewpoint of logical consistency and software engineering experience.

4. **Applicability**: The model's degree of applicability across different software products (size, structure, functions, etc.)

5. **Simplicity**: A model should be simple in three aspects:
   - simple and inexpensive to collect data
   - simple in concept and does not require extensive mathematical background for software development practitioners to comprehend
   - readily implemented by computer programs

The goal of statistical defect modeling, which includes what is commonly referred to as software reliability growth, has been to predict the reliability of a software product. Typically, this may be measured in terms of the number of
defects remaining in the field, the failure rate of the product, the short term defect
detection rate, etc [Chillarege et al., 1992], [Goel, 1985], [Musa et al., 1987] and
[Ramamoorthy and Bastani, 1982]. Statistical analysis methods are an important
component in empirical software engineering and mainly used to [Fenton and
Pfleeger, 1997]:

1. Confirm a theory - the usual analysis approach used:
   - Student's t-test
   - F statistic
   - Kruskal-Wallis

2. Explore a relationship - the usual analysis approach used:
   - Box plot
   - Scatter diagram
   - Correlational analysis
     - Pearson analysis
     - Not tied Spearman Kendall
     - Tied chi-squared
     - Linear regression
     - Multivariate regression
     - Logarithmic
     - Transformation
     - Thiel

Previous research [Khoshgoftaar et al., 2000] has shown that software quality
models based on software metrics can give predictions with useful accuracy.
Software quality prediction models can predict quantities like the number of faults
and software development effort [Khoshgoftaar and Seliya, 2003]. Over the last
few years, many software quality modeling techniques have been developed and
used in real life software quality predictions. The most commonly used modeling
techniques for software quality estimation are depicted in Table 2.4. Table 2.5
presents the comparison of several items of prior work with this thesis. All prior
work shown in the table has been discussed earlier in this chapter.

2.8 Summary

This chapter explores several topics which are used as the theoretical foundation
for this research. The materials presented help to provide a deeper understanding
Chapter 2: Literature Review

of the methodologies used in previous research and act as guidelines for conducting this research. For instance, Sections 2.3 and 2.4 provide the insight into the foundation of empirical and measurement theory which include replications, measurement scales and the Goal-Question-Metric paradigm which are used to plan the research methodology of this work.

Moreover, Section 2.5 discusses the current state of the open-source research which helps in identifying the potential systems to be included in this research as well as the suitable methodologies for data collection and analysis. Section 2.6 includes a lengthy discussion on a range of software metrics available and is useful in identifying the metrics most relevant in this research. Lastly, Section 2.7 explains the possible modeling techniques for quality prediction using various statistical analyses and this helps to identify the data analysis techniques to be used in this research.

<table>
<thead>
<tr>
<th>Modeling techniques</th>
<th>Prior work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification and Regression Trees (CART)</td>
<td>[Tian and Zelkowitz, 1995] [Gokhale and Lyu, 1997] [Khoshgoftaar et al., 1999] [Khoshgoftaar and Allen, 2001] [Khoshgoftaar and Seliya, 2002] [Takahashi et al., 1997] [Troster and Tian, 1995] [Li et al., 2005b]</td>
</tr>
<tr>
<td>Artificial neural networks</td>
<td>[Finnie et al., 1997] [Khoshgoftaar and Lanning, 1995] [Khoshgoftaar and Seliya, 2003] [Li et al., 2005b] [Kastro and Benner, 2008]</td>
</tr>
<tr>
<td>Case-based reasoning</td>
<td>[Ganesan et al., 2000] [Kolodner, 1993] [Khoshgoftaar and Seliya, 2003]</td>
</tr>
<tr>
<td>Multiple Linear Regression</td>
<td>[Cartwright and Shepperd, 2000] [Khoshgoftaar and Seliya, 2003] [Li et al., 2005b]</td>
</tr>
<tr>
<td>Logistic regression</td>
<td>[Khoshgoftaar et al., 2000] [Briand and Wüst, 2001]</td>
</tr>
<tr>
<td>Principle component analysis</td>
<td>[Briand et al., 1998a] [Briand et al., 1999d] [Briand and Wüst, 2001] [Li et al., 2005b] [Arisholm and Briand, 2006]</td>
</tr>
<tr>
<td>Univariate regression analysis</td>
<td>[Briand et al., 1998a] [Briand et al., 1999d] [Briand et al., 1999c] [Briand and Wüst, 2001] [Arisholm and Briand, 2006]</td>
</tr>
<tr>
<td>Multivariate regression analysis</td>
<td>[Briand et al., 1999c] [Briand and Wüst, 2001] [Arisholm and Briand, 2006]</td>
</tr>
<tr>
<td>Factor analysis</td>
<td>[Khoshgoftaar and Munson, 1990] [Munson and Khoshgoftaar, 1992]</td>
</tr>
<tr>
<td>Discriminant analysis</td>
<td>[Munson and Khoshgoftaar, 1992]</td>
</tr>
<tr>
<td>Cox Model</td>
<td>[Koru et al., 2008] [Wedel et al., 2008]</td>
</tr>
</tbody>
</table>
Table 2.5 – Comparison of Research Work

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>Industrial</td>
<td>Industrial</td>
<td>University</td>
<td>University</td>
<td>Open source</td>
<td>Open source</td>
</tr>
<tr>
<td>Programming language</td>
<td>Fortran (RATFOR)</td>
<td>Java</td>
<td>C++</td>
<td>C++</td>
<td>N/A</td>
<td>Java</td>
</tr>
<tr>
<td>Application domain</td>
<td>Scientific (SEL)</td>
<td>Legacy system in telecom company</td>
<td>Information systems</td>
<td>Information systems</td>
<td>Apache web server and Mozilla Internet browser</td>
<td>Different variety of open source projects</td>
</tr>
<tr>
<td>No. of projects studied</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>104</td>
</tr>
<tr>
<td>Variables</td>
<td>Structural, data and procedural complexity, errors</td>
<td>Various measures of class size, inheritance, coupling, cohesion and fault-proneness of a class</td>
<td>WMC, DIT, NOC, CBO, RFC, LCOM, probability of fault</td>
<td>Coupling, cohesion, inheritance and measure of fault-proneness</td>
<td>Developer participant, core team size, code ownership, productivity, defect density and problem resolution interval</td>
<td>Structural, data and procedural complexity, CBO, user reported defects and size</td>
</tr>
<tr>
<td>Findings</td>
<td>System complexity can predict error rates for development project</td>
<td>Build a multivariate prediction model to predict the probability of fault correction across classes using logistic regression on log-transformed variables</td>
<td>Found that WMC, DIT, RFC, NOC and CBO are good predictors of class fault proneness</td>
<td>Size of classes, frequency of method invocations and DIT influence fault-proneness</td>
<td>Open source development fosters faster system growth and generally have fewer defects than closed source projects</td>
<td>System complexity is not a good predictor of defect density, however, structural complexity is significant as a predictor</td>
</tr>
<tr>
<td>Defect measures</td>
<td>Errors used were errors found during integration and system testing</td>
<td>Faults found during acceptance test</td>
<td>Faults found during acceptance test</td>
<td>Faults found during acceptance test</td>
<td>Faults found during acceptance test</td>
<td>Defects used are from user reported defects</td>
</tr>
</tbody>
</table>
Part II

Research Methodology and Implementation
Chapter 3

Research Background

When you can measure what you are speaking about and express it in numbers, you know something about it: but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.

Lord Kelvin

Contents

3.1 Background ......................................................... 66
  3.1.1 Functional Decomposition .................................. 66
3.2 Object-oriented Concepts in Java .............................. 67
3.3 Complexity in Java Systems .................................... 68
3.4 Proposed Model .................................................. 72
3.1 Background

This chapter discusses the theoretical background that is used as the basis of this work and the application of those various theories in the context of this research. The topics explained in Chapter 2 are further explored in this chapter to better understand the research background and to describe the structure of this thesis.

3.1.1 Functional Decomposition

Functional decomposition is one common approach to system design. It results in a hierarchical network of units or modules. For any module, workload consists of input and output items to be processed, and these input/output items form the "data coupling" for the module. At each level of decomposition, the designer must decide whether to implement the indicated functionality in the current module or defer some of it to a lower level by invoking one or more other modules [Card and Glass, 1990]. Card and Glass based their system complexity model on this approach.

The reasons I use Card and Glass's work as the basis of this research are:

1. Their complexity model is extremely suitable for applying to industry since many software development projects go through the design process before...
3.2 Object-oriented Concepts in Java

being translated into code.

2. There has been little validation of this design complexity model has been performed, therefore it is necessary to test this model using empirical data for validation [Kan, 2003].

The main contribution of this research to the software engineering body of knowledge is the extension of structured metrics using the design complexity model of Card and Glass and its application to object-oriented systems to test whether this model can be used to predict defects in object-oriented systems as well. Actually, there are several differences in the research environment between my work and the work of Card and Glass. These differences are summarized in Table 3.1.

<table>
<thead>
<tr>
<th>Research environment</th>
<th>Card and Glass</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects</td>
<td>8 similar projects</td>
<td>104 open source projects of different types</td>
</tr>
<tr>
<td>Programming language</td>
<td>FORTRAN (RATFOR)</td>
<td>Java</td>
</tr>
<tr>
<td>Metrics used</td>
<td>Structural, data and procedural complexity and pre-release defects for structured design models</td>
<td>Object-oriented metrics corresponding to structured metrics and post-release defects</td>
</tr>
</tbody>
</table>

3.2 Object-oriented Concepts in Java

Object-oriented programming (OOP) is a programming paradigm using objects (data structures consisting of data fields and methods together with their interactions) to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance [Meyer, 1997].

Meyer discusses a set of criteria of object-orientation [Meyer, 1997], as follows:
1. Seamlessness: An object-oriented language and environment, together with the supporting method, should apply to the entire lifecycle, in a way that minimizes the gaps between successive activities.

2. Classes: The method and the language should have the notion of class as their central concept.

3. Assertions: The language should make it possible to equip a class and its features with assertions (preconditions, postconditions and invariants), relying on tools to produce documentation out of these assertions.

4. Information hiding: It should be possible for the author of a class to specify that a feature is available to all clients, to no client, or to specified clients.

5. Exception handling: The language should provide a mechanism to recover from unexpected abnormal situations.

6. Inheritance: Inheritance is one of the central concepts of the object-oriented methods and has profound consequences on the software development process. A class will be a child of another if it incorporates the other's features in addition to its own.

7. Polymorphism: Polymorphism is the ability for an entity to become attached to objects of various possible types, under the control of the inheritance-based type system.

8. Dynamic binding: Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call.

Java programming language complies to the aforementioned criteria, and considered as a type of object-oriented language. Since the systems selected in this research are written in Java, it is assumed that they follow the standards of object-orientation to some degree. The in-depth analysis of the level of object-orientation in each system is outside the scope of this thesis.

3.3 Complexity in Java Systems

The fundamental concepts surrounding high-quality design apply to all types of software implementations, whether they be conventional or modern, such as object-oriented (OO) methods. Thus, to achieve effective modularity, object-oriented software must apply good data and procedural abstractions in a similar way to the more traditional structured modularity. A class is an object-oriented concept
that encapsulates data and procedural abstractions that are required to describe the content and behavior of some real world entity, and also represents a module in OO systems [Pressman, 2000].

Figure 3.2 – Package Representation in Java

At a higher level, Java contains many predefined pieces called classes that are grouped by categories of related classes called packages (common structured package name) [Deitel and Deitel, 1997]. In Java, there are two types of packages, Java application programming interfaces (Java API) and also packages that are developed for specific systems. An example of packages in Java are illustrated in Figure 3.2. Relationships exist between the packages so that they can work
Figure 3.4 – Class Representation in Java

Internal procedure of method may be complex.

Figure 3.5 – Fan-out of Java Methods

together to solve a problem and these relationships are called “dependencies”. Inter-package dependencies are inter-class relationships that cross a package boundary. These dependencies reflect part of the structural complexity of the system.

Java packages comprise a group of classes which interact with each other, whether with classes within the same package or classes of other packages through dependency and inheritance. These relationships may be classified as “uses” relationships and inheritance or implementing relationships. Figure 3.3 illustrates the example of relationships between Java classes, and measured using Coupling between Objects (CBO) metric. When a Java class depends on too many other classes then it tends to have high structural complexity.

Figure 3.2, 3.4 and 3.5 illustrated the relationships between Java class instances, package instances and method instances, respectively, and adapted from a presentation document by Boughton et al. [Boughton et al., 2001]. In Figure 3.4, the data abstractions (attributes) that describe the class instances are enclosed by
3.3 Complexity in Java Systems

a “wall” of procedural abstractions, such as operations, methods or services that are capable of manipulating the data in some way. The only way to reach the attributes and use them is through one or more of the methods that comprise the “wall”. Hence, the class encapsulates data inside the wall and the processing that manipulate the data. This achieves information hiding and reduces the impact of side effects associated with change [Pressman, 2000].

In other words, a class is a generalized description of a collection of similar objects. This means all instances of a class inherit its attributes and the operations that are available to manipulate the attributes. Figure 3.6 shows the overall complexity of Java classes, which comprise number of parameters (data complexity), CBO/number of uses/accesses (structural complexity) and number of internal decisions (procedural complexity).

![Figure 3.6 - Overall Complexity of Java Classes “Relationships”](image)

The number of parameters indicates the level of data complexity.

The number of uses/accesses for a class is a measure of structural complexity.

The number of decisions within the class is a measure of procedural complexity.

![Figure 3.7 - Overall Complexity of Java Methods](image)

The number of parameters indicates the level of data complexity.

The number of calls for a method is a measure of structural complexity.

The number of decisions within the procedure of a method is a measure of procedural complexity.
Likewise, at method level, dependencies do exist between methods within the same class or methods in other classes. This is due to the fact that Java classes communicate with each other through method calls. This relationship is depicted in Figure 3.4. When a method calls other methods, this is known as Fan-out or structural complexity of the method, and having high Fan-out indicate a lot of dependencies on other methods. An example of method level Fan-out is illustrated in Figure 3.5.

The complexity of methods in Java comprise data complexity, procedural complexity and structural complexity. Data complexity is indicated by the number of parameters to and from the method, structural complexity is measured by the number of calls for the methods (CBO) and procedural complexity is represented by the number of decisions within the procedure of a method. These different types of complexity are shown in Figure 3.7. Card and Glass [Card and Glass, 1990] introduced this design complexity model in FORTRAN, a structured programming language, and I plan to apply this model to systems built using object-oriented programming language, to be specific, Java systems.

3.4 Proposed Model

In this research, the concept of “class” is used as an alternative to “module” since in object-oriented language, a class is a self-contained and independent unit as with a module.

The Structural Complexity in Card and Glass’s model, as presented in Equation 2.6 only considers one dimension of coupling, which is Fan-out (based on calls/invocation). In object-oriented (OO) systems, coupling measures should also include “Inheritance” which is an important characteristic of OO systems [Coad and Yourdon, 1991]. Furthermore, the value of Fan-out squared may not explain the complexity in OO systems as it might do in structured systems. Therefore, it is essential to find a substitute measure for coupling in OO systems.

There are several measures of coupling in OO systems discussed in the literature [Briand et al., 1996a], [Briand et al., 1999e], [Chidamber and Kemerer, 1994], [Hitz and Montazeri, 1995a], [Li and Henry, 1993]. Measures differ according to several criteria, the most important ones being: the types of connection/dependency contributing to coupling, the domain of the measure, its level of granularity, for example, how connections are counted, and so on.

Fundamentally, in OO systems, two types of dependencies are introduced by the existence of strongly connected classes. The first type of dependency emerges...
from the simple sharing of services between the classes. The second type of dependency between classes occurs from the inheritance hierarchy of classes.

In this research, the Coupling between Object classes (CBO) measure proposed by Chidamber and Kemerer [Chidamber and Kemerer, 1994] is chosen as a possible alternative to CG Structural Complexity in measuring the Structural Complexity of OO systems. The reason for choosing CBO to measure Structural Complexity is that it captures both kinds of dependencies by incorporating the “accesses”/“uses” from the given class (Fan-out) and other classes (Fan-in), including “inheritance”.

This suggests an alteration of Card and Glass’s formula for Structural Complexity to make it more relevant to OO systems, that being:

\[ S = \frac{\sum CBO(i)}{n} \]  

(3.1)

where
- \( S \) = Average Structural Complexity
- \( CBO(i) \) = CBO of class i
- \( n \) = Number of classes in system

In relation to the Data Complexity formula, it is also adjusted to better relate to the object-oriented environment. Card and Glass’s Data Complexity in Equation 2.8 divides the I/O variables with Fan-out to represent work in a module being deferred to lower level modules. In OO systems, this is not necessarily happening because of the “encapsulation” or “information hiding” concept, which is part of the OO programming principles. This concept states that the “state” and “implementation” of an object or class should be private to that object or class and only accessible through its public interface [Graham, 2001]. The question is: Is work being deferred in the way that Card and Glass claim or is the idea of deferment by activities like “data gathering”, “self containment” and “separation of concerns”, all of which when compared to a structured design (call tree) indicate that class is a considerably different “module”. To account for these differing concepts between OO and structured design systems, it is suggested that “deferment” is potentially unnecessary in OO systems. Thus, the formula for Card and Glass’s Data Complexity (Equation 2.8) is altered to:

\[ D = \frac{\sum v(i)}{n} \]  

(3.2)

where
- \( D \) = Average Data Complexity
\[ v(i) = \text{I/O variables in class } i \]
\[ n = \text{Number of new classes in system} \]

In essence, the concept of input/output variables (I/O variables) is altered to relate to the OO characteristics. At class level, I/O variables are considered as:

- **Input variables** - The number of parameters/arguments passed to the class and the number of messages that return an object to the class.
- **Output variables** - The number of arguments/parameters that the class must pass/return to other classes as part of sending messages

From this point onwards, the term Structural Complexity or Average Structural Complexity refers to the \textbf{Equation 3.1} and Data Complexity or Average Data Complexity refers to the \textbf{Equation 3.2}.

Once the function and input/output data of a class have been identified, they can be used to write an algorithm to perform the function and process the data. This implementation complexity is known as Procedural Complexity. Some parts of the complexity of this algorithm is due to the intrinsic difficulty of the problem, or the work that needs to be done by the class. McCabe’s Cyclomatic Complexity (as described in \textbf{Section 2.5.1.2}) [McCabe, 1976] is used as Procedural Complexity in this research. It is used as a simple count of the number of decisions in an implemented artifact such as a subroutine, method or class. The Average Procedural Complexity is calculated using this formula:

\[ P = \frac{\sum V(G)_{(i)}}{n} \]  \hspace{1cm} (3.3)

where
\[ P = \text{Average Procedural Complexity} \]
\[ V(G) = \text{Cyclomatic Complexity of class } i \]
\[ n = \text{Number of new classes in system} \]

A designed class will eventually be implemented in code. It is extremely useful to be able to predict the size of the class based on early design metrics. The Class Size metric is obtained using this formula:

\[ CS = \frac{\sum eLOC}{n} \]  \hspace{1cm} (3.4)

where
\[ CS = \text{Class Size} \]
3.4 Proposed Model

LOC = Effective lines of code in system (defined in Section 2.5.1.4)

\[ n = \text{Number of classes in system} \]

Card and Glass have demonstrated that CG Average Data Complexity is a good predictor for Average Procedural Complexity and Class Size. Thus, this research also investigates whether the proposed Average Data Complexity as in Equation 3.2 is a good predictor of Average Procedural Complexity and Class Size. The results are going to be discussed in Chapter 5 and 6.

A significant point to note is that the research plan of this work was presented at the International Doctoral Symposium of Empirical Software Engineering 2007 [Awang Abu Bakar and Boughton, 2007], and the suggestions by the reviewers were taken into consideration in conducting the research. Furthermore, both David Card and Robert Glass have contributed some input into this work.
Chapter 4

Research Methodology

Good estimation is based on the understanding and use of a range of tools and techniques and the expert judgment as to which combinations are most appropriate in each situation.

F. Wellman, 1992

Contents

4.1 Research Methodology ........................................ 78
  4.1.1 The Goal Question Metric Approach ....................... 78
  4.1.2 Research Design ............................................. 80
4.2 Data Collection .................................................. 84
4.3 Data Source ..................................................... 86
4.4 Metric Extraction Tools ......................................... 89
4.5 Data Collection Process ......................................... 91
  4.5.1 Project Selection Phase ................................... 92
  4.5.2 Metrics Extraction Phase ................................ 93
  4.5.3 Metrics Validation Phase ................................. 96
  4.5.4 Validation Process ........................................ 98
4.6 Summary ....................................................... 108
4.1 Research Methodology

This section will discuss the application of the theoretical background and framework that has been presented in Chapter 2 and previous sections of this chapter. The research design is planned according to the empirical and measurement theory and framework in empirical software engineering. This research is also in line with the work of Card and Glass which is used as the baseline of this work. The differences between this work and their work are depicted in Table 3.1 in Section 3.1.1, and one major difference is that, the systems being studied in this work are object-oriented systems written in Java whereas they studied structured systems developed using FORTRAN (RATFOR). Other main difference is, my work study post-delivery defects as opposed to the pre-delivery defects used in Card and Glass's work. Therefore, there are some differences in the research variables which will be explained in later sections. One of the motivations of this work is to validate the design complexity model introduced by Card and Glass, since there is a need for validation work using empirical data from other environments to enhance the credibility of aforementioned model [Kan, 2003]. The validation result is intended to produce new defect prediction models for object-oriented systems.

In addition, the discussion of research objectives, questions and hypotheses delivered in Chapter 1 will be extended by explaining the methodology used to collect and analyze data to achieve the stated objectives.

4.1.1 The Goal Question Metric Approach

The background and benefits of using GQM have been discussed in Section 2.4.2, this section will discuss the application of GQM in this research work. It was found that, using GQM is useful in mapping the main goal of this research with the questions that need to be answered, which are represented by the hypotheses and the metrics that are most relevant to this research. The mappings of the goal, questions and metrics are shown in Table 4.1. The metrics relevant to this research are further discussed in Section 4.1.3.1 and 4.1.3.2.

Based on the research questions, the research hypotheses are formulated for the purpose of answering the questions:

1. H1: Increasing values of the Average System Complexity (Average Structural Complexity plus Average Data Complexity) correlate with increasing post-delivery defect density.
### Table 4.1 – Research Goal, Questions and Metrics

**Goal**
Predict defects in the final implementation at an early stage of development from the viewpoint of the software engineers performing the work in the context of studying the quality of open-source systems.

**Questions**

1. Is there a general correlation between system (data + structural) design complexity and various types of defects, and thus, can the level of defects be predicted early enough to undertake strategies to minimize them in the final product?

2. Between structural, procedural and data complexity, which has the most influence and/or is the most appropriate for predicting system defects?

3. Are there particular strategies for minimizing defects in open source software?

**Metrics**

- Effective Lines of Code (eLOC)
- 1000 Lines of Code (KLOC)
- Structural Complexity (CBO)
- Procedural Complexity (Cyclomatic Complexity)
- Data Complexity (Number of Parameters)
- Number of Defects
- Defect Density (Number of Defects/KLOC)
- Ratio of Average System Complexity to Average Procedural Complexity

2. H2 : Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Average Procedural Complexity (Cyclomatic Complexity), as in some non-object-oriented systems.

3. H3 : Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Class Size, as in some non-object-oriented...
Chapter 4: Research Methodology

4. H4 : There is a relationship between System Complexity and Procedural Complexity that will help minimize defects in the final system.

4.1.2 Research Design

For the purpose of answering the research questions and testing the hypotheses, this research uses the experimental approach for data collection and analysis. Experiment is an empirical inquiry that investigates causal relationships and processes [Yin, 2003]. The investigation of causal relations provides an explanation of why a phenomenon occurred, while the identification of causal processes yields an account of how a phenomenon occurred.

While experiments can help to provide inductive support for hypotheses, their most important application is in testing theories and hypotheses.

Software engineering experiments are usually used to explore relationships among data points describing one variable or across multiple variables, to evaluate the accuracy of models, or to validate measures [Sjöberg et al., 2005].

This research aims at investigating selected problems in a snapshot version of open-source systems rather than studying the evolution of systems.

In a similar line, this research aims to verify Card and Glass's System Complexity Model [Card and Glass, 1990] and the results of this research are compared with their work to validate their model.

The details of the experimental design are covered in the data collection, independent variables, dependent variable and data analysis sections.

4.1.2.1 Independent Variables

The independent variables selected for this research are a collection of design metrics that can be used to measure system complexity at the design phase of the software development. They are based on the system complexity model by Card and Glass [Card and Glass, 1990], and are depicted in Table 4.2.

From Table 4.2, Structural Complexity is represented by CBO, Data Complexity is represented by Number of Parameters and Procedural Complexity is represented by McCabe's Cyclomatic Complexity. Specifically, the metrics used in this work are measured at the system and class level. Measures at the method level could not be validated, since the post-delivery defects are reported at system
4.1 Research Methodology

Table 4.2 – Applicable metrics

<table>
<thead>
<tr>
<th>Metrics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Card and Glass’s metrics</td>
<td>Structural Complexity</td>
</tr>
<tr>
<td></td>
<td>Data Complexity</td>
</tr>
<tr>
<td></td>
<td>Procedural Complexity</td>
</tr>
<tr>
<td>McCabe’s metrics</td>
<td>Cyclomatic Complexity</td>
</tr>
<tr>
<td>Chidamber and Kemerer’s object-oriented</td>
<td>Coupling Between</td>
</tr>
<tr>
<td>metrics suite</td>
<td>Object Classes (CBO)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of Parameters</td>
</tr>
<tr>
<td></td>
<td>Class Size (eLOC/Class)</td>
</tr>
</tbody>
</table>

level for most systems used in the dataset and considerably few systems report post-delivery defects at class level, while none reported defects at method level.

The advantage of using design metrics as estimators is that, they can be measured quite early in the development process and this can help developers/teams to estimate defects before the coding process starts and thus take action to change the design if the level of predicted defects is too high. Design is the earliest stage in which the system structure is clearly defined. Design generally refers to architectural design, that is, the process of partitioning the required functionality and data of a software system into parts that work together to achieve the full mission of a system [Blundell et al., 1997]. Design first concentrates on perfecting the software architecture and then focuses on the details within architectural components. The structured design methodology has had a great influence on product quality [Yourdon and Constantine, 1979].

Software metrics that provide feedback to programmers during development should be available early enough in the life cycle to guide software testing and maintenance, also possible changes to design itself. They should help partition a design to minimize coupling between modules/classes, be able to provide design hints and facilitate a logical choice of algorithms [Baker and Zweben, 1979]. Blundell et.al attempted to derive design metrics from software quality factors [Blundell et al., 1997]. They identified several measurable software attributes, such as, inter-modular and intra-modular complexity, cohesion, coupling and data structures as potential design metrics. Weiss and Basili have shown that design errors far outweigh other classes of errors [Weiss and Basili, 1985].

Besides the metrics described above, Version Downloads and the Number of
Developers are also considered as independent variables to predict user reported defects. The Version Downloads measure represents the number of downloads for the version used in this research while the Total Downloads measure (> 50,000) represents the number of accumulated downloads for systems. The Version Downloads measure is a usage measure for the selected version instead of Total Downloads because it is more relevant to this research. This metric is obtained from the “Download traffic statistics” page in SourceForge.

4.1.2.2 Dependent Variable

In spite of recent advances in programming technology, it is not yet possible for developers to produce error-free code consistently. Consequently, a significant amount of effort is usually allocated to testing and correcting software before delivery. Even after delivery, defects are still discovered by the users throughout the period of usage of the software. A software product is considered defective when it does not perform its functions according to the user’s expectations [Conte et al., 1986]. The definitions of error, fault and failure were given in Section 2.6 but are repeated here for convenience [IEEE Standard 1990, 1990]:

- Error: A software defect in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools
- Fault: A concrete manifestation of errors within the software
- Failure: A departure of the operational software system behavior from users’ requirements

Defects can manifest themselves in various stages of software development, for example, requirements analysis phase, design phase, coding phase, testing phase and maintenance phase. However, defects that appear during the design phase are the easiest (in terms of both thought and actual effort) to rectify by checking the design against the specifications of the system [Conte et al., 1986]. Conte et al. [Conte et al., 1986] list three typical metrics for assessing the defects in software:

1. Number of changes required in the design - this metric results from faulty understanding of specifications and appear beginning at the design phase. Design changes can occur anywhere in the life cycle from the design phase on, whenever a defect is discovered. The method of computing this metric is subject to a count based on the analyst’s assessment of the number of separate items changed. It is most efficient to find and correct design defects during design and not at later phases of development.
4.1 Research Methodology

2. **Number of errors** - when defects are discovered from the coding phase on and especially during the testing phase, they can be counted. After a program reaches the point where it is written, any errors discovered through hand-checking, walk-throughs or testing can be counted in this metric. In many settings, there are formal error reports filed whenever errors are found, and a count of different error reports can be used as a defect metric.

3. **Number of program changes** - this algorithmic defect metric is based upon the fact that defects are usually fixed by means of program code changes. The measure “one program change” is concerned with a contiguous set of statements that represent a single abstract action [Dunsmore and Gannon, 1980].

   - One or more changes to a single statement
   - One or more statements inserted between existing statements
   - A change to a single statement followed by the insertion of new statements

In this research, *defects* refer to *faults* in the system, that later will cause *failures* in the system. The dependent variable used in this research is post-delivery defects collected from the bug tracking report in SourceForge.

The total defect counts then need to be normalized by the system size. In this research, the effective lines of code (eLOC) is used as the size measure. This measure is then divided by 1000 to derive another measure, KLOC, which is later used in the formulation of Defect Density:

\[
DD = \frac{\sum \text{Defects}}{KLOC}
\]  

(4.1)

where
- DD = Defect Density
- Defects = Total defects collected
- KLOC = The number of thousands (1000) of lines of code in system

According to Jones [Jones, 1991], there are two general rules for customer reported defects:

1. The number of customer reported defects found correlates directly with the number of users.
Chapter 4: Research Methodology

2. The number of customer reported defects found correlates inversely with the number of defects found prior to shipment.

Even though these general rules seem to be quite the opposite of each other, they are actually interrelated. They are both based on the concept that the more the software is used, the more defects that will be discovered. If the software has many users, it will have more exposure and therefore, more defects will be found. On the contrary, when the software is shipped with lots of potential defects (that will cause either erroneous behaviour or failure) then users will quickly become frustrated as more defects are exposed and then give up using it.

The first rule will be tested and presented in the data analysis section in Chapter 5. The second rule will not be tested since it is difficult to obtain pre-delivery defect reports for most open-source systems.

4.2 Data Collection

Data speak a thousand words. The backbone of empirical theory is data, either qualitative or quantitative, and without data, measurement work cannot be carried out. In empirical software engineering, data play an important role in theory validation, hypothesis testing, model building, etc. Due to their importance, it is necessary to make sure that data collected are valid. Data collection is a rigorous exercise especially during the planning phase when several decisions concerning data collection issues need to be made, such as the type of data to collect, data collection methods, the source of data or population, sample size and the applicable statistical analysis to be applied to the collected data. Thus, the data collection process has to be done using a systematic approach to ensure that measures are defined unambiguously, that collection is consistent and complete, and that data integrity is protected.

According to Fenton and Pleeger [Fenton and Pfleeger, 1997], in order to collect quality data, there are several guidelines to follow:

1. **Correctness**: means that data are collected according to the exact rules of definition of a metric. For example, if a lines of code (LOC) count is supposed to include everything but comments, then a check for correctness will ensure that no comments were counted.

2. **Accuracy**: refers to the difference between the data and the actual value. For instance, time measured using an analog clock may be less accurate than time measured using a digital one.
4.2 Data Collection

3. **Precision**: deals with the number of decimal places needed to express the data. For example, it is not necessary to calculate the mean cyclomatic complexity to several decimal places, since cyclomatic complexity is the number of decisions in a module, fractions of a decision to several decimals places are meaningless.

4. **Consistent**: data should be consistent from one measuring device or person to another, without large or unexplained differences in values. Therefore, two evaluators should calculate the same or similar cyclomatic complexity values from the same source. Similarly, when the same data value is computed repeatedly over time, the data should be captured in the same way.

5. **Data collection period**: if data collected are associated with a particular activity or time-period, the data should be time-stamped, so that it’s known when they were collected. This also helps in tracking trends and comparing activities.

6. **Possible replication**: data are often collected to support surveys, case studies and experiments. These investigations are frequently repeated under different circumstances, and the results compared. Data collected should be able to be replicated to be used for any purposes.

Since the open-source software for this research is being downloaded from SourceForge, it is difficult to obtain the design documents from the repositories because design documentation was not generally available. Therefore, the only artifact available to be used for metric collection is versions of source code. However, the practice of deriving design metrics from source code is common in software research. Reverse engineering of quality measures has been used in many studies [Antoniol et al., 2001], [Antoniol et al., 2002], [Briand et al., 2006], [CanforaHarman and Penta, 2007], [Jiang et al., 2008], [Tonella and Potrich, 2005], [Troster, 1992].

Extracting design metrics from the source code does have the possibility of introducing the Hawthorne effect where early software design might change during the implementation and coding phase. Therefore, the design metrics values produced from code extraction might differ from using detailed design values from design documents, depending on how closely the development team follows the detailed design. Typically, the detailed design is inclusive of all data attributes, methods and incorporates state handling [Shlaer and Mellor, 1992], but since it is not possible to get the design documents from the project development teams,

---

1The Hawthorne effect is a form of reactivity whereby subjects improve an aspect of their behavior being experimentally measured simply in response to the fact that they are being studied, not in response to any particular experimental manipulation. [Adair, 1984]
the reverse engineering method has to be used to obtain the design metrics in this work.

4.3 Data Source

During the initial phase of this research, several possible sources were explored for collecting the data to test the hypotheses and answer the research questions. Due to the fact that quite a number of systems need to be tested, there's a need for a methodology to identify an appropriate number of systems that are in production and have post-release defect reports. Choosing the right data sources is extremely important to ensure the collection process can be completed successfully, and also to ensure correct data are collected.

Based on previous empirical work [Chen et al., 2004], [Crowston et al., 2004], [Feitelson et al., 2006], [Hahsler and Koch, 2005], [Li et al., 2005a], the possibility of using open-source software for data collection became apparent. After doing some search, three possible open-source repositories were found to be used as the data source, i.e., SourceForge, Freshmeat and Savannah. Out of those three, SourceForge has more detailed information about the projects hosted including, a comprehensive bug report, number of downloads, number of developers, developers' contact information (email) and more project choices from various categories. Hence, SourceForge was selected as the data source to carry out this research.

Open-source software repositories are a valuable source of information for empirical studies in software engineering. Most projects in the repositories, especially the active ones, contain all archived communications among project participants and record the rationale for the various decisions throughout the life of a project, as opposed to most commercial repositories that miss information verbally exchanged between developers. This relative completeness is contributed to by the fact that email archives, problem tracking systems, and version control systems represent the only way to exchange information among project participants located in different countries.

The selected repository, SourceForge, is the world's largest open source development web site. As of February 2009, more than 230,000 software projects have been registered in the repository, and 2 million users registered to use the services provided by SourceForge. Furthermore, the merger with Freshmeat makes SourceForge the largest collection of open-source tools and applications on the internet.
4.3 Data Source

SourceForge provides a real-time statistics system for tracking and trending project activity, for instance, user visits, posts to Tracker, downloads, subversion activity, CVS activity, etc. Project statistics are the basis of project ranking, which allow the activity and vitality of the projects to be compared. SourceForge provides ranking of top projects based on several categories:

- Most active projects
- Top downloads
- Top project pageviews
- Top project hits
- Top forum post counts
- Top tracker activity

The projects in SourceForge were grouped into several categories based on functionality, such as communications, database, education, games, internet, multimedia, office/business, security, software development and many more. For the purpose of this research, 104 projects from four different categories were chosen, namely:

- Games/Multimedia
- Internet/Communication
- Office/Programming/Database
- Scientific/Engineering/Operating Systems

These categories were selected because they represent different types of systems, and there are a large number of systems within these categories. This provides the opportunity to randomly choose systems from a wide range of selection, based on a Python script which was written to identify systems that fulfill the following selection criteria:

1. Active projects (Activity percentile more than 90 percent) - SourceForge ranking based on the overall project activity (combination of project traffic, development and communication). Only active projects were selected in this research because these projects provide the most current defect reports and number of downloads, while inactive projects do not publish such information [www.sourceforge.net].
Chapter 4: Research Methodology

2. Developed using Java programming language - Projects developed using Java are chosen because Java is a mainstream object-oriented development language, therefore the projects are more likely to have followed object-oriented development methodology.

3. Development status is: Production/Stable or Mature - Only projects that are in production and are stable have been chosen because they are being used and have most of the information needed for this research. SourceForge includes project development status information ranging from 1-6 where 1 is the planning stage and 6 is the mature phase.

4. High number of total downloads (number of downloads more than 50,000) - This criterion is chosen because projects with a high number of downloads tend to have a similar usage frequency by users and this will be reflected in the post-release defect report.

5. Availability of error reports (error reports shown in bug tracking system) - This criterion is important because the error reports are essential for using as the dependent variable in this work. They represent the post-release defects for the systems.

From the list of systems/projects in Table A.1, A.2, A.3 and A.4 in Appendix A, two projects are listed in the top 100 Most Active Projects of all time, and they are: jEdit and Galleon, while Azureus, JBoss, Freemind and jEdit are listed in the 100 Top Downloads projects. Several projects have been chosen as Project of the Month, in particular: JBoss (April 2003), Azureus (September 2004), JasperReports (July 2005), FreeMind (February 2006), Art of Illusion (April 2007), ehcache (February 2008) and recently, ZK (February 2009). These achievements have put these projects in the “hall of fame” in SourceForge record as top projects. Furthermore, of all 230,000 projects hosted in SourceForge (as of March 2009), ZK is ranked 2, Sweet Home 3D is ranked 9, FreeMind is ranked 10, Azureus is ranked 12, Opentaps is ranked 47, RapidMiner is ranked 52, Saxon is ranked 59, iText is ranked 82, aTunes is ranked 91, DataCrow is ranked 97, respectively, and other projects selected in this research are all active projects and have number of total downloads of more than 50,000.

The rationale in selecting open-source software as the source of data for this research was given in 2.5.1. Open-source software has become a reliable source of data for many research projects, since most of the software is in production and has a high success rate among users.
4.4 Metric Extraction Tools

Measurement accuracy is an important requirement for all engineering disciplines, including software engineering. A large body of software quality metrics have been introduced, researched, discussed and quite a number of tools have been developed to collect metrics from software artifacts including source code. There are different types of tools available in the market, some are free and some come at a price. Users can choose one of a number of tools that best suits their needs. In this research, several tools have been selected to acquire the relevant metrics for each chosen system. The list of tools used and the metrics that they collect are summarized in Table 4.3.

Table 4.3 – Metric Extraction Tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fan-out</td>
</tr>
<tr>
<td>Chidamber and Ke-merer Java Metrics (CKJM)</td>
<td></td>
</tr>
<tr>
<td>Resource Standard Metrics (RSM)</td>
<td></td>
</tr>
<tr>
<td>JStyle</td>
<td></td>
</tr>
<tr>
<td>JHawk</td>
<td></td>
</tr>
</tbody>
</table>

However, users of these various tools need to be careful in their selection. It is considerably convenient to assume that all tools gather and interpret the same metrics in the same way. But that is not usually the case. When using a combination of several tools to acquire the metrics relevant to my research, it was discovered that chosen tools actually calculated the same metrics differently. Therefore it is extremely important to carefully verify each metric produced by a tool in order to be confident with the results. This issue has been discussed in a paper presented at the Software Engineering and Applications (SEA '08) Conference in Orlando, Florida, USA [Awang Abu Bakar and Boughton, 2008].

The findings regarding the dissimilarity of results produced by different tools
have also been discovered by Lincke et al. where they compare the results produced by ten metrics tools for similar metrics [Lincke et al., 2008]. In their investigations, they tested the tools that can produce object-oriented metrics, such as Coupling between Object Classes (CBO), Depth of Inheritance Tree (DIT), Lack of Cohesion of Methods (LCOM), Lines of Code (LOC), Number of Children (NOC), Number of Methods (NOM), Response for a Class (RFC) and Weighted Methods per Class (WMC). They evaluated a set of tools based on three open-source projects and reported that the values for similar metrics differ between tools. The results reported by different tools were compared for minimum, maximum and average values, and for all projects tested, all tools produced different values. Their findings support my findings that the results obtained by using multiple tools should be carefully verified by the researchers to ensure that the validity of their data is intact. Lincke et al. concluded that, metrics results are tool dependent and metrics based results cannot be compared without checking when using different metric extraction tools. This unexpected situation makes metrics validation difficult and exacerbated by the fact that every metric acquired by each tool must be checked for correctness.

One of the possible reasons the tools produce different values for similar metrics is the ambiguity of metrics definitions. Most metrics are defined in a considerably general way without precise examples, and these unclear definitions open up the possibility for different interpretations and implementations. Thus, the tool developers most likely calculate the metrics based on their interpretations, which results in differences in metrics calculations between tools.

After the open-source systems had been downloaded, the process of extracting the relevant metrics began. Since the projects in the repository do not make design documents accessible (if they exist at all), another method to get the values of the design metrics is through reverse engineering.

Several tools are identified to extract the relevant metrics from the Java software artifacts, namely: Logiscope, Resource Standard Metrics (RSM), Chidamber and Kemerer Java Metrics (CKJM), JStyle and JHawk. These tools have been chosen because they can analyze Java programs and produce the metrics needed for this research. Except for CKJM, all tools being used require a license to operate. The detailed descriptions for all tools considered in this research are given in Appendix E.

It seems that JHawk may be the answer to all the requirements surrounding this research. However, given that there seemed to be potentially significant differences in the results between tools, any tools selected for this research need to be verified. The discovery of the dissimilarities in the results reported by
the aforementioned tools led to another issue: most results reported in many publications are actually tool dependent, and if other researchers attempt to measure the same metrics data for the same software using different tools, there is a significant possibility that the new results will be different, and even more importantly, that could lead to different conclusions [Lincke et al., 2008].

Thus, it is imperative that the results produced by different tools are verified rigorously in order to explain the reasons behind any differences of results. It is necessary to compare the results reported by these different tools with the original definition given for the applicable metrics. After a comprehensive comparison, the tools that most closely produce results that follow the original definitions of the metrics being collected have been chosen for this research. The details of the verification process will be discussed in Section 4.5.4.

4.5 Data Collection Process

The data collection stage usually follows the formulation of research questions and research hypotheses, since ultimately, the collected data should be able to contribute to answering the research questions that were raised at the beginning of the research.

Based on the research questions and hypotheses discussed in Section 1.3 and 1.4 of this thesis, the data collection plan was formulated to identify suitable software that can be used to test the hypotheses. The decisions on the sample size and population of software that should be used were made during this stage.

Initially, 20 proprietary systems were to be used, a number which was decided after taking into account that data collection will be incredibly challenging because of the need to contact several software companies to ask permission to use their data. However, after browsing numerous papers on empirical research, where several papers discussed data collection from open-source repositories [Chen et al., 2004], [Crowston et al., 2004], [Dinkelacker et al., 2002], [Ferenc et al., 2004], [Glass, 2003a], [Glass, 2004], [Hahsler and Koch, 2005], [Harrison, 2001], [Koru and Tian, 2004], [Mockus et al., 2002], it became apparent that the possibility of using data from open-source repositories should be considered.

As mentioned earlier, the open-source repository chosen for this research is SourceForge and 104 Java systems are selected to be included in this research. The reason to collect 104 systems as the sample size is to ensure that the results of this research are more generalizable to the software engineering field.

The data collection process started in December 2006 and concluded in June
Figure 4.1 – Data Collection Process Flow

2008. This period covers the project selection process, software downloading process, tools selection process, metrics extraction using selected tools and metrics validation by comparing values produced by different tools. The data collection phases are illustrated in Figure 4.1 and discussed in detail in the following sections.

4.5.1 Project Selection Phase

As mentioned above, instead of collecting data from several commercial systems which could possibly hide certain information about their projects and might not be so willing to share all relevant information with researchers, I choose to collect data from open-source software repositories instead. This choice has been made based on the liberty of data collection from a large open-source repository which hosted hundreds of thousands of software/systems and is used by millions of people worldwide.
4.5 Data Collection Process

The process of selecting suitable software was done using a Python script which was written to automate data collection based on all requirements specified in Section 4.3. The steps taken in selecting relevant projects are:

1. Identify active projects
2. Identify systems that are developed using the Java programming language
3. Choose projects that have "development status" of Production/Stable or Mature
4. Choose projects with a high number of total downloads
5. Ensure error reporting in a bug tracking system

Once all software have been identified according to the steps above, downloading of the current version started in December 2006.

4.5.2 Metrics Extraction Phase

Once the software and tools have been selected, the next step is the metrics extraction phase. The metrics extraction process is depicted in Figure 4.2 and carried out in this order:

1. First, the downloaded systems were divided into different directories/folders. Then two subdirectories are created to hold the Java source files and class files.
2. The directory which contains the source files were applied to JStyle, JHawk and RSM, while the class files were applied to CKJM.
3. The results obtained from these tools were kept in separate directories, one for each tool.
4. Results reported in a spreadsheet format were analyzed
5. Metric values obtained by various tools were compared and validated against the original definition of the metrics, the metrics validation exercise is further discussed in Section 4.5.4.

The defects reported by users (post-delivery) defects were obtained from the bug tracking system in SourceForge. An example of this tracker page is illustrated in Figure 4.3. In addition, users can report the bugs or defects they find in the system using a bug report form in SourceForge as depicted in Figure 4.4.
Some systems release new versions more frequently than others. To ensure consistency across all systems, defects collected are within the first 6 months of the release of the version included in this research. However, there is still a possibility that defects are continuously reported by users after that period, this issue will be discussed in Section 6.7.3.

To strengthen the confidence of defect data collection, I took the initiative to contact the developers of the projects through email to learn more about the number of post-delivery defects for their systems. Out of developers who were contacted for the defect information, 50% replied with useful information about defect reports for their systems. Their input was used to verify the defects reported by users as listed in SourceForge. The questions included in this survey mainly aimed at investigating the number of defects reported for the systems, and also to collect other information, such as the reporting mechanism used by the project teams and whether they publish "known defects" in the project website. The list of questions are given in Appendix B and the analysis result for this survey will be

Figure 4.2 – Metrics Extraction Process Flow
4.5 Data Collection Process

**Figure 4.3** – Bug List in SourceForge

**Figure 4.4** – Bug Report Form in SourceForge
Chapter 4: Research Methodology

discussed in Section 5.4.7.2. In addition to that survey, another online survey was conducted using StatPac online survey software [www.statpac.com] to investigate the defect handling mechanism in open-source projects. The questionnaires for this survey are shown in Appendix C. The developers for the selected projects were invited to participate in this survey. However, the number of responses received from this survey were low (around 20%). The results from this survey are discussed in Section 5.4.7.2.

4.5.3 Metrics Validation Phase

This phase is essential because the collected data must be screened for any possible mistakes during data collection and metrics extraction. Since multiple tools have been used to extract metrics, it is extremely important to verify the values produced by the combination of tools to ensure that data integrity is preserved.

The results produced by the four chosen tools are compared to ensure that they produce correct values according to the original definitions (given in Section 2.5). This exercise is in line with the measurement quality criteria of reliability and validity [Kan, 2003]. Furthermore, Ostrand recommends that the collected data should be validated by checking for consistency and validity [Ostrand, 2007]. He suggests that if the amount of data is small, individual data should be examined for consistency, but if the dataset is too large for individual examination, then it is reasonable to examine a randomly selected subset of the dataset.

- Reliability - refers to the consistency of a number of measurements taken using the same measurement method on the same subject. If repeated measurements are exceedingly consistent or even identical, then the measurement method or the operational definition has a high degree of reliability, and vice versa. The measurement of any phenomenon has a certain amount of chance of error. The goal of error-free measurement, is never accomplished in any discipline of scientific investigation [Kan, 2003]. The amount of measurement error may be large or small, and the goal is to achieve the best possible reliability.

- Validity - refers to whether the measurement or metric really measures what is intended. In other words, it refers to the extent to which an empirical measure reflects the real meaning of the concept under consideration. Validity is classified into several types:
  - Construct validity - refers to the validity of the operational measurement or metric representing the theoretical construct.
4.5 Data Collection Process

- Criterion-related validity - also referred to as predictive validity.
- Content-validity - refers to the degree to which a measure covers the range of meanings included in the concept.

As mentioned before, it is difficult to achieve error-free measurement. The amount of measurement error may be large or small, but it is universally present. There are two types of measurement error, systematic and random. Systematic measurement error is associated with validity, and random measurement error is associated with reliability. In the general case, the measurement equation is:

\[ M = T + s + e \]  

(4.2)

where \( M \) is the observed/measured value, \( T \) is the true value, \( s \) is systematic error, and \( e \) is random error. The presence of \( s \) (systematic error) makes the measurement invalid. When \( s \) is not in the Equation 4.2 and assuming that measurement is valid, a new equation is formed:

\[ M = T + e \]  

(4.3)

The Equation 4.3 states that any observed measurement is not equal to the true measurement because of random disturbance or the random error, \( e \). The disturbance means there is variability in the measurement result, where on different occasions, the observed measurement may be higher or lower than the true measurement. However, since the disturbances are random, it means that positive errors are just as likely to occur as the negative errors and these errors are expected to cancel each other. In other words, the average of these errors in the long run, or the expected value of \( e \) is zero: \( E(e) = 0 \) [Kan, 2003].

In this research, the dissimilarities in results produced by the different tools are considered as systematic errors, not random ones.

Thus, based on the concept of reliability and validity discussed previously, it is important to ensure that the collected data are reliable and valid, so that the data analysis stage will produce correct results and, consequently, arrive at correct conclusions. To achieve the goal of ensuring that the collected data are reliable and valid, a thorough data validation exercise was conducted. The data validation process is described below:

1. The source code and Java byte code files for each system were applied to each tool for analysis. Java source files were used as input for RSM, JStyle and
JHawk. Since CKJM can only read byte codes, Java byte code files were used as input.

2. Once results were obtained from all tools, they were compared. By doing so, I was able to investigate whether all tools produce the same values for similar metrics. First, I chose a considerably simple metric which I expected to be the same independent of tools, e.g., Number of Classes, so as to determine the extent of the possible different interpretations of the various metrics.

3. After thorough checking of the results reported by those tools, it was found that for certain metrics like Effective Lines of Codes (eLOC) or Executable Statements, Number of Classes, Number of Methods and Number of Parameters/Arguments are the same across all tools. On the other hand, calculations for metrics such as Fan-out, CBO and Cyclomatic Complexity are different. Fan-out metrics are produced by JStyle and JHawk, CBO is produced by CKJM and JHawk while Cyclomatic Complexity is produced by CKJM, Jstyle and JHawk. The differences in values of metrics provided by those tools required further investigation and will be discussed in the next section.

4.5.4 Validation Process

This section discusses the application of several metrics such as Fan-out, CBO, Cyclomatic Complexity and Number of Parameters in Java systems, as well as the validation of metric values reported by the metric extraction tools.

For the purpose of validating the metrics calculation by different tools, a mechanism was devised to calculate the relevant metrics and compare the results with the ones reported by the tools. Hence, I made several interpretations on the methods to calculate those metrics. The interpretations were made based on several references, including the original definitions given by the founders of those metrics [Chidamber and Kemerer, 1994], [Henry et al., 1981], [McCabe, 1976]. Other sources of information include [Gill and Kemerer, 1990], [Henderson-Sellers, 1996a], [Henderson-Sellers, 1996b], [Rosenberg et al., 1991], [Rosenberg and Hyatt, 1995], [Tegarden et al., 1992], [Tegarden et al., 1995], [Wilkie and Harmer, 2002] and [Wilkie and Hylands, 1998].

4.5.4.1 Fan-out

The Fan-out measure was introduced in structured systems as part of the information flow measure by Henry and Kafura [Henry et al., 1981]. The definition of Fan-out has been given in Section 2.6.1.1, and this section continues the
discussion of Fan-out in the object-oriented context. The concept of Fan-out in object-oriented (OO) systems is discussed in detail by Tegarden et al. [Tegarden et al., 1995], where they presented the application of Fan-out in various levels of components in OO systems, such as variable, method, object/class and system level. In particular, Fan-out at object/class level measure is defined as: “The number of unique messages that the object sends to all other objects”.

![Diagram of Java Classes](image)

**Figure 4.5 – Fan-out of Java Classes**

The Fan-out metric considered in this research was calculated at class level, and focused on inter-class dependencies, as depicted in **Figure 4.5**. After comparing the Fan-out values provided by both JStyle and JHawk, I found that they differ, and required further investigation to validate which value should be used in this research.

Based on the definitions given by Henry and Kafura and Tegarden et al., the Fan-out value was manually calculated through source code inspection. The primary focus of this research is the source code developed for the application, therefore, the Standard API classes are excluded from the calculation.

The Java source files were randomly selected from several applications, and code inspection was conducted to derive the Fan-out value from the code. Several types of dependencies were considered to calculate Fan-out, such as:

- Method calls - Only unique calls to another method in another class are included, which means, multiple calls are considered as one.
- *New* operators declared for another class.
- Object declarations for another class.
- Interface implementation of another class
- Cast operators - Only include arithmetic and conversion operators.
- Exception handlers.
Chapter 4: Research Methodology

The next step is to check the Fan-out values provided by JHawk and JStyle to investigate the Java properties included in the calculations by both tools.

Table 4.4 – Comparison of Fan-out Properties in Different Tools

<table>
<thead>
<tr>
<th>Fan-out Calculations</th>
<th>Manual</th>
<th>JHawk</th>
<th>JStyle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>New operators</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Cast operators</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Java library classes</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Exception handler</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Inheritance</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Interface</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Object declarations</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

The Fan-out properties included in manual inspection, JHawk and JStyle are summarized in Table 4.4, while Figure 4.6 shows the comparison of Fan-out values provided by both tools and manual calculation. From Figure 4.6, the Fan-out values provided by JStyle are higher than both JHawk and manual calculation's values, mainly because JStyle includes the following properties in the calculation:

- Java Standard API classes.
- Multiple occurrences of method calls are calculated as multiple Fan-out.
- The calling class (the class being studied) is also considered in Fan-out calculation.
- Inheritance

On the other hand, the Fan-out values provided by JHawk are closer to those arrived through manual calculation, essentially because it counts similar properties to those considered during code inspection, as illustrated in Table 4.4.

In order to test whether there is any statistically significant difference between the results produced by the tools and manual calculation, a Mann-Whitney U (Wilcoxon rank-sum) test was conducted. The non-parametric test was chosen because the data have non-normal distribution: Anderson-Darling value (A) = 2.17
Figure 4.6 – Fan-out Comparison between Tools
Further discussions on the Anderson-Darling test will be given in Chapter 5.

The Mann-Whitney U test is a non-parametric two-sample test which tests the equality of two populations' medians. It assumes independent samples and almost equal variances [Hoaglin et al., 1983].

The results for the Mann-Whitney U test are summarized in Table 4.5. The results show that there is no statistically significant difference between manual calculation and JHawk (p-value = 0.75). However, there is a statistically significant difference between manual calculation and JStyle (p-value = <0.001). Based on these findings, the Fan-out values provided by JHawk are used in this research.

<table>
<thead>
<tr>
<th>Tools</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual calculation - JHawk</td>
<td>0.75</td>
</tr>
<tr>
<td>Manual calculation - JStyle</td>
<td>&lt; 0.001</td>
</tr>
</tbody>
</table>

4.5.4.2 Coupling between Object Classes (CBO)

This metric is part of the Chidamber and Kemerer object-oriented metrics suite (CK metrics) [Chidamber and Kemerer, 1994]. In Java systems, the existence of interconnections between classes introduce two kinds of dependencies. The first kind of dependency surfaces from the simple sharing of services across classes, which could be contributed to by method calls to other classes, message passing and so on. Another kind of dependency between classes emerges from the inheritance hierarchy of classes in the design. The CBO metric captures both kinds of dependencies by considering any invocation of a method or instance variable of another class as a coupling. These dependencies are illustrated in Figure 3.4 in Section 3.3.

Initially, the CBO metric is calculated using two tools, CKJM and JHawk and the results provided by those tools are dissimilar. Essentially, the main reason for the differences is CKJM reads byte code as input whereas JHawk uses Java source code for processing. The generation of byte code basically depends on the compiler behaviour. For example, the javac compiler does not perform simple optimizations such as loop unrolling, algebraic simplification, strength reduction, and so on. However, other compilers like Jikes, Expresso and GCJ, to name
a few, may perform differently. In addition, some program properties like type information is not included in the byte code [Baxter et al., 2006]. This means the CBO results produced using byte code as provided by CKJM is not comparable to the results provided by JHawk.

Based on the definition of CBO by Chidamber and Kemerer as given in Section 2.5.2, the code inspection is performed on the selected Java files. The CBO calculation includes inter-class dependencies such as Fan-in, Fan-out and inheritance. For instance, if a class A is referred by B, C and D (Fan-in = 3) and class A refers to D, E and F, CBO for this class is 5 (B, C, D, E, F). These dependencies or couplings exclude Java API packages. JHawk calculates CBO in a similar way, while CKJM includes all classes in its calculation, including the classes in the API packages.

The comparison of CBO calculations between tools is demonstrated in Figure 4.7. It seems that the CBO values achieved by manual calculation are closer to the values produced by JHawk than to CKJM. However, in the case of JRBaseObjectFactory, JRFillObjectFactory and Controller, the differences between the values obtained using the manual calculation and JHawk are due to the fact that the manual calculation includes “Cast operators” and “object declarations” while JHawk excludes the calculations.

Another Mann-Whitney U test was conducted to investigate whether there is any statistically significant difference between the results provided by manual calculation and given by the tools. The results of the test are shown in Table 4.6. The results show that there is no statistical significant difference between manual calculation and JHawk (p-value = 0.82), as well as between manual calculation and CKJM (p-value = 0.52).

Although there is no statistically significant difference between the results of manual calculation and CKJM, the fact that CKJM reads byte code instead of source code makes it unsuitable for this research. Hence, CBO values provided by JHawk are used in this research.

Table 4.6 – CBO: Mann-Whitney U Test between Tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual calculation - JHawk</td>
<td>0.82</td>
</tr>
<tr>
<td>Manual calculation - CKJM</td>
<td>0.52</td>
</tr>
</tbody>
</table>
Figure 4.7 – CBO Comparison between Tools
4.5 Data Collection Process

4.5.4.3 Cyclomatic Complexity

McCabe's Cyclomatic Complexity (MCC) metric uses graph theory cyclomatic numbers for the complexity measure. It is used to calculate the number of decisions in a system. Since MCC was initially used to measure the complexity of structured systems, it does not handle certain constructs of OO systems, such as, inheritance, polymorphism, and data binding [McCabe, 1976]. However, several researchers report that since MCC calculates complexity at the method level, it is still useful in the context of counting the control flow paths in OO systems [Wilkie and Hylands, 1998], [Lee et al., 1993], [Rosenberg and Hyatt, 1995] and [Tegarden et al., 1992]. In the context of this research, MCC is used to evaluate the complexity of an algorithm in a method. A similar approach was employed by Rosenberg and Hyatt in Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center, where they applied MCC to calculate the complexity of OO systems developed at the center [Rosenberg and Hyatt, 1995].

For the purpose of applying this metric in this research, several assumptions are made, and they are:

1. The MCC calculations only consider the control paths at method level
2. The calculation of MCC values follow the suggestion by McCabe, as listed below.

The tools used to calculate the MCC metric in this research are RSM, JHawk and JStyle. There are some differences in the values reported by those tools which need to be checked. Another code inspection is conducted based on the definition in Section 2.5.1.2, taking into account the following decision counts [McCabe, 1976]:

- Sequence of flow - base $v(G) = 1$
- If-then-else - base $v(G) = 2$
- While - base $v(G) = 2$
- Until - base $v(G) = 2$
- Case - base $v(G) = 2$
- For - base $v(G) = 2$

The interpretation of this metric for the manual calculation was based on the assumptions given and the definitions by McCabe. However, I do not make any
Figure 4.8 - Cyclomatic Complexity Comparison between Tools
4.5 Data Collection Process

Table 4.7 – Comparison of McCabe’s Cyclomatic Complexity (MCC) by Tools

<table>
<thead>
<tr>
<th>MCC Calculations</th>
<th>Manual</th>
<th>JHawk</th>
<th>JStyle</th>
<th>RSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>If-then-else</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>While</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Until</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Case</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>For</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

claims that the manual calculations are better than the calculations provided by other tools.

The comparison of the decision types calculated by the tools is presented in Table 4.3, and it shows that JStyle does not include decisions related to Sequence of flows and Until. The bar chart in Figure 4.8 illustrates the comparison of Cyclomatic Complexity values provided by each tool and manual calculation. The values of Cyclomatic Complexity in RSM are usually higher than manual calculation and results obtained from JHawk and JStyle because it adds extra calculations of logical operations.

For the purpose of selecting the most suitable tool to be used in this research, the Mann-Whitney U test was conducted between manual calculation and the tools: JHawk, JStyle, and RSM. The results in Table 4.8 show that there is no statistical significant difference between manual calculation with all tools: JHawk (p-value = 0.85), JStyle (p-value = 0.73), and RSM (p-value = 0.73).

Between the three tools, JHawk gives the value closest to the manual calculation, which makes it the most suitable for this research. On the other hand, since RSM is unable to provide calculations for Fan-out and CBO, it is excluded from this research. In addition, JStyle has a significant limitation in its performance, it only supports applications developed using JDK 1.4, and is unable to run applications written using later versions of Java Development Kit, thus, excluded from this research.
### 4.5.4.4 Number of Parameters

This metric represents the input/output parameters that "flow" into or out of a class as defined in Section 3.1.1. The tools that are able to produce this metric are RSM and JHawk. The results provided by those tools are similar, however, the metric is reported at method level. This poses a problem, due to several issues such as redundancy and inclusion of "Private" methods in the calculation.

Thus, it is important to extract this metric at class level. Firstly, "Private" methods are removed from the calculation. Then, a script is written to calculate this metric at class level by eliminating redundancies in parameter/variable counting. Since the results provided by the tool are similar, it does not matter which tool is selected. JHawk is used to extract this metric because it can also produce other metrics like Fan-out, CBO and Cyclomatic Complexity.

### 4.6 Summary

This chapter describes the research methodology applied in this thesis. It covers the discussion of the necessary details on the Goal-Question-Metric approach, research design, dependent and independent variables, data collection process, data source, metrics extraction tools, as well as tools validation method.

Moreover, this chapter represents a critical boundary in this thesis. The discussions in this chapter and previous chapters involve the background and the preparation for the data analysis phase, while the chapters that follow emphasize the presentation and interpretation of the results.
Part III

Results Analysis and Discussion
No software system of any realistic size is ever completely debugged—that is, error free.

Edward Yourdon and Larry Constantine
Chapter 5: Data Analysis

5.1 Introduction

This chapter presents the findings of this research obtained using several statistical tests. It lays out the important work of the hypotheses testing, as well as investigating the possible methodologies to answer the research question. While the previous chapters mainly discuss the research background and data collection methodologies, this chapter elaborates on the statistical analysis on the collected data for the purpose of understanding the underlying phenomenon.

Among the issues explored within this chapter are: the correlation between the dependent variable and the independent variables, the outlier analysis, the defect handling mechanism in open-source systems, and the comparison between the proposed model and Card and Glass's model. Another general issue is the choice between parametric and non-parametric analysis. If distribution of variables can be identified, appropriate parametric tests will be more powerful than non-parametric tests. However, if the distribution of variables is unknown, non-parametric methods are usually more appropriate, most are exceedingly efficient relative to their parametric counterparts, and they are effective with small sample sizes [Kitchenham et al., 2002].

5.2 Statistical Modeling

The relationships between software metrics (as independent variables) and defects (as a dependent variable) can be investigated and studied using various statistical analysis methods. When/if the relationships between these variables are established, the next step is to produce a mathematical model to estimate defects in the system based on the tested software metrics.

A quantitative model is an equation or algorithm where the dependent variable is a function of one or more independent variables [Khoshgoftaar and Munson, 1990]. If the values for the independent variables are supplied to the model, then the value of the dependent variable can be calculated. A software quality model has independent variables that may be measured earlier in the life cycle than the dependent variable. Hence, the calculated dependent variable value is an estimation of what the actual value is expected to be.

The goal of this work is to develop models that will estimate quality factors related to defects at class and system level. Total defects are directly measurable after software has become operational. However, software product metrics and process metrics can be measured during development, even as early as during the design phase. A suitable software quality model can make predictions when it is
5.3 Procedure for Data Analysis

still not too late to find a way to minimize defects occurring in the finished product.

The datasets used in most software engineering research often have non-normal characteristics, such as skewness, unstable variance and outliers, and sometimes the combination of all these [Pickard et al., 1999]. Pickard et al. attempted to investigate three main statistically-based data analysis techniques, Residual Analysis, Multivariate Regression, and Classification and Regression Trees (CART) using a simulated dataset. They found that no single analysis technique is the best in all circumstances, and the best analysis techniques are determined by the nature of the dataset [Pickard et al., 1999].

Prior work has used several statistically-based data analysis techniques as shown in Table 2.4 in Section 2.7. The choice of the most suitable prediction techniques depends on the kinds of information to be predicted, for example, defect predictions could be for defect count, defect density or defect thresholding [Li, 2006]. This work is focused on investigating the relationship between system design complexity as a predictor of total post-delivery defects at system level, and defect density at class level. Henceforth, the data analysis discussions will mainly involve the process of identifying the effects of the independent variables on post-delivery defects.

5.3 Procedure for Data Analysis

Much literature in statistics [Hoaglin et al., 1983], [Hoaglin et al., 1985], [Caulcutt, 1991], [Harrell, 2001] discusses various (statistical) methods that can be used to analyze data. In particular, Harrell [Harrell, 2001] elaborates in detail the data analysis procedure using regression modeling. In addition, much literature in empirical software engineering [Basili et al., 1996], [Briand et al., 1998a], [Briand et al., 1999c], [Briand et al., 1999e], [El-Emam and Wieczorek, 1998], [Fenton and Ohißson, 2000], [Khoshgoftaar and Munson, 1990], [Khoshgoftaar et al., 1996], [Khoshgoftaar et al., 2000], [Kitchenham et al., 2002], [Pfleeger, 1994], [Pfleeger, 1995a], [Pfleeger, 1995b], [Pfleeger, 1995c], [Pfleeger, 1995d], [Pickard et al., 1999], [?] discusses several data analysis procedures commonly used in studying software engineering data. Since software engineering data are often “messy”, and most of the time exhibit non-normal characteristics such as skewness, unstable variance and extreme outliers, it is important to choose the most appropriate analysis technique so that the results from the analysis will help researchers to understand the underlying phenomenon being investigated.

The objective of this research is to firstly (and if possible) determine whether the findings of Card and Glass [Card and Glass, 1990] as mentioned in Chapter 2
and 3 are also applicable for object-oriented systems. For this part of the research, several metrics are considered, including Structural Complexity, Data Complexity and Procedural Complexity as independent variables to predict post-delivery defects for the selected systems. Another objective is to try and establish whether there are different estimators of post-delivery defects in open-source systems, for example, Version Downloads, Number of Developers and so on. The analysis procedure for the collected data is described in two stages:

1. Data distribution and outlier analysis

2. Regression model construction

### 5.3.1 Data Distribution and Outlier Analysis

During data collection, only non-library or classes that are determined to have been developed as part of a separate application are included in the calculations. This is consistent with Card and Glass's work (Card and Glass, 1990), where they only included newly developed modules in their calculations. The collected data was analyzed using a statistical package, GenStat 11.1 and supplemented by Microsoft Excel 2002. Analysis of data distributions and outliers are carried out in two steps:

1. The distribution and variance of data was studied to identify which analysis techniques should be used. It is important to identify the distribution of the data, whether it is normally distributed or not. For example, if the data are normally distributed, the parametric statistical technique can be used (Fenton and Pfleeger, 1997), (Harrell, 2001).

   However, if the data are not normally distributed, they can still be transformed to another scale that conforms more closely to the normal distribution. For example, in this work, Version Downloads data are not normally distributed (skewed to the left), so it is common to transform to a logarithmic scale. Even if the original data are not normally distributed, the logarithms of the data usually are.

   Nonetheless, in the situation when the data are not normally distributed, non-parametric analysis techniques are usually employed. For instance, non-parametric techniques often use properties for the ranking of the data (Fenton and Pfleeger, 1997), (Conte et al., 1986), (Harrell, 2001).

In this research, the Anderson-Darling test is used to investigate the distribution of data. The results show that both the dependent and independent variables are not normally distributed. Hence, the most appropriate analysis
5.3 Procedure for Data Analysis

techniques are the non-parametric ones. The details of the test are discussed in Section 5.3.1.1.

2. Outliers are data points which are located in an otherwise empty part of the sample space. Outliers should be identified to ensure that conclusions drawn are not solely dependent on a few outlying observations. Inclusion and exclusion of outliers can have a large influence on the analysis results and prediction models. Hence, it is important to identify outliers, test their influence, explain them and possibly remove them [Briand et al., 2000]. The definition of univariate outliers is:

- Univariate outliers: A class/system that has an outlying value in the distribution of any one of the measures used in the study. The influence of the identified data point is tested, if the significance of the relationship between the measure and defects depends on the absence or presence of the outlier, then the outlier point is considered influential.

5.3.1.1 Anderson-Darling Test

The Anderson-Darling test [Anderson and Darling, 1952], [Anderson and Darling, 1954] and [Stephens, 1974] is used to test if a sample of data came from a population with a specific distribution, i.e., normal, lognormal, exponential, Weibull, extreme value type I, and logistic distributions. It is a modification of the Kolmogorov-Smirnov (K-S) test and gives more weight to the tails than does the K-S test. The K-S test is distribution free in the sense that the critical values do not depend on the specific distribution being tested. The Anderson-Darling test makes use of the specific distribution in calculating critical values. This has the advantage of allowing a more sensitive test and the disadvantage that critical values must be calculated for each distribution.

The Anderson-Darling test is an alternative to the chi-square and Kolmogorov-Smirnov goodness-of-fit tests. The Anderson-Darling test is defined as:

\[ H_0: \] The data follow a specified distribution
\[ H_a: \] The data do not follow the specified distribution

The critical values for the Anderson-Darling test are dependent on the specific distribution that is being tested. Tabulated values and formulas have been published [Stephens, 1974] for a few specific distributions (normal, lognormal, exponential, Weibull, logistic, extreme value type 1). The test is a one-sided test and the hypothesis that the distribution is of a specific form is rejected if the test statistic is greater than the critical value \( \alpha = 0.05 \). In this research, the critical
value at $\alpha = 0.05$ is 0.787 and the Anderson-Darling values for all variables exceed the critical value as shown in Table 5.1. Based on those values, it can be concluded that all variables are from non-normal distributions.

Table 5.1 – Anderson-Darling Test for Normality

<table>
<thead>
<tr>
<th>Variables</th>
<th>Anderson-Darling Test</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Data Complexity (ADC)</td>
<td>6.25</td>
<td>Non-normal</td>
</tr>
<tr>
<td>Average Structural Complexity (ASC)</td>
<td>11.04</td>
<td>Non-normal</td>
</tr>
<tr>
<td>Average Procedural Complexity (APC)</td>
<td>3.82</td>
<td>Non-normal</td>
</tr>
<tr>
<td>Version Downloads (VD)</td>
<td>25.87</td>
<td>Non-normal</td>
</tr>
<tr>
<td>Number of Developers (Dev)</td>
<td>14.32</td>
<td>Non-normal</td>
</tr>
<tr>
<td>Class Size (CS)</td>
<td>5.05</td>
<td>Non-normal</td>
</tr>
<tr>
<td>Total Defects (TD)</td>
<td>15.21</td>
<td>Non-normal</td>
</tr>
<tr>
<td>Defect Density (DD)</td>
<td>10.71</td>
<td>Non-normal</td>
</tr>
</tbody>
</table>

5.3.2 Estimation Model Construction

There are two steps involved in constructing the estimation model for this work:

- Univariate analysis
- Regression analysis

1. Univariate analysis is executed for each independent variable against the dependent variable to determine if the predictor is statistically related to post-delivery defects. This analysis is conducted to test the hypotheses in Chapter 1. There are three methods usually used to establish an estimator as important:

   (a) Show high correlation between estimator and the dependent variable. This method is recommended by IEEE [IEEE Standard 1998, 1998] and is applied by Ohlsson and Alberg [Ohlsson and Alberg, 1996] and Ostrand and Weyuker [Ostrand and Weyuker, 2002].

   (b) Show that the estimator is selected using a model selection method. This method is applied by Jones et al. [Jones et al., 1999] and Mockus et al. [Mockus et al., 2005].
5.4 Analysis Results

(c) Show that the accuracy of estimations improves with the estimator included in the model. This method is applied by Khoshgoftaar et al. [Khoshgoftaar et al., 1995] and Jones et al. [Jones et al., 1999].

2. Regression analysis is used to build a estimation model for Defect Density of the Java software downloaded from SourceForge. This analysis is performed to determine how well the Defect Density can be predicted when the independent variables are used in combination. In order to select the metrics to be used in the model, a strategy must be used that:

- Minimizes the number of independent variables in the model. Using too many independent variables can have the effect of increasing the estimated standard error of the model’s prediction, making the model more dependent on the data set, in other words, less generalizable [Briand et al., 2000].
- Reduce multicollinearity or independent variables which are highly correlated. This makes the model more interpretable.

To build the regression model, a stepwise selection process is used, where regression models are built in a stepwise manner, at each step one variable enters or leaves the model. The two major stepwise selection processes used in linear regression are forward selection and backward elimination [Harrell, 2001]. The general forward selection procedures begin with a model that includes the intercept only. Based on certain statistical criteria, variables are selected one at a time for inclusion in the model, until a “stopping” criteria is fulfilled. Likewise, the general backward elimination procedure starts with a model that includes all independent variables. Variables are selected one at a time to be deleted from the model, until a stopping criteria is fulfilled [Hoaglin et al., 1983]. The regression model in this work are built using the backward elimination procedure.

5.4 Analysis Results

Based on the analysis procedure described above, the data analysis was carried out using the statistical package Genstat 11.1. The analysis procedure carried out in this research and the results are reported in this section.
5.4.1 Descriptive Statistics

The descriptive statistics for the independent variables are shown in Table 5.2. The number of observations or sample size for this work is 104 systems. Columns 'Mean', 'Std. Dev.', 'Median', 'Min', 'Max' represents the mean value, standard deviation, median, minimum and maximum values for each metric considered, respectively. The metrics Average Data Complexity (ADC), Average Structural Complexity (ASC), Average Procedural Complexity (APC) and Average System Complexity (ASyC) are derived from the metrics Total Data Complexity, Total Structural Complexity, Total Procedural Complexity and Total System Complexity divided by Number of Classes for each system. The formula for Class Size has been given by Equation 3.4 in Section 3.3.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Data Complexity</td>
<td>7.08</td>
<td>2.37</td>
<td>6.71</td>
<td>2.88</td>
<td>13.69</td>
</tr>
<tr>
<td>Average Structural Complexity</td>
<td>4.63</td>
<td>2.11</td>
<td>4.24</td>
<td>0.90</td>
<td>14.03</td>
</tr>
<tr>
<td>Average Procedural Complexity</td>
<td>18.19</td>
<td>7.35</td>
<td>17.20</td>
<td>5.98</td>
<td>42.46</td>
</tr>
<tr>
<td>Average System Complexity</td>
<td>11.71</td>
<td>3.63</td>
<td>10.94</td>
<td>5.64</td>
<td>25.47</td>
</tr>
<tr>
<td>Average CG Data Complexity</td>
<td>2.30</td>
<td>1.01</td>
<td>2.00</td>
<td>6.90</td>
<td>2.92</td>
</tr>
<tr>
<td>Average CG Structural Complexity</td>
<td>24.15</td>
<td>20.50</td>
<td>18.81</td>
<td>1.43</td>
<td>142.10</td>
</tr>
<tr>
<td>Average CG System Complexity</td>
<td>26.45</td>
<td>20.16</td>
<td>21.28</td>
<td>4.26</td>
<td>143.50</td>
</tr>
<tr>
<td>Total Data Complexity</td>
<td>3720</td>
<td>4576</td>
<td>2400</td>
<td>183</td>
<td>37813</td>
</tr>
<tr>
<td>Total Structural Complexity</td>
<td>2923</td>
<td>4074</td>
<td>1414</td>
<td>37</td>
<td>30392</td>
</tr>
<tr>
<td>Total Procedural Complexity</td>
<td>9340</td>
<td>11373</td>
<td>5945</td>
<td>508</td>
<td>86933</td>
</tr>
<tr>
<td>Total System Complexity</td>
<td>6643</td>
<td>8501</td>
<td>3769</td>
<td>220</td>
<td>68205</td>
</tr>
<tr>
<td>Total CG Data Complexity</td>
<td>1110</td>
<td>1445</td>
<td>733.5</td>
<td>125.8</td>
<td>12653</td>
</tr>
<tr>
<td>Total CG Structural Complexity</td>
<td>16934</td>
<td>25412</td>
<td>8388</td>
<td>57</td>
<td>136968</td>
</tr>
<tr>
<td>Total CG System Complexity</td>
<td>18044</td>
<td>26441</td>
<td>9328</td>
<td>204.8</td>
<td>149621</td>
</tr>
<tr>
<td>Class Size</td>
<td>61.79</td>
<td>27.63</td>
<td>53.5</td>
<td>21.73</td>
<td>168.7</td>
</tr>
<tr>
<td>Version Downloads</td>
<td>209152</td>
<td>599327</td>
<td>83058</td>
<td>7520</td>
<td>4783246</td>
</tr>
<tr>
<td>log10 Version Downloads</td>
<td>4.76</td>
<td>0.73</td>
<td>4.92</td>
<td>2.88</td>
<td>6.68</td>
</tr>
<tr>
<td>Number of Developers</td>
<td>11.30</td>
<td>13</td>
<td>5.00</td>
<td>1.00</td>
<td>158</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>587.2</td>
<td>855</td>
<td>325</td>
<td>32</td>
<td>7453</td>
</tr>
</tbody>
</table>

Two other metrics that are not directly related to the system design are also collected and analyzed to investigate other influential factors on the post-delivery
defects of open-source systems. The metrics are Version Downloads and Number of Developers.

The Version Downloads metric provides the number of downloads of the relevant version for all open-source systems under study, whereas Number of Developers represents the developers involved in constructing the individual systems. The range of minimum and maximum values for the Version Downloads metric is incredibly large, and after doing the diagnostic plot to test for normality, it was found that the data for this metric exhibited non-normality characteristics such as skewness. Hence, the appropriate solution is to transform this variable to a logarithmic scale [Caulcutt, 1991], [Harrell, 2001], [Hoaglin et al., 1983], [Fenton and Pfleeger, 1997]. After the transformation, the Log10 of VersionDownloads exhibits a normal distribution.

5.4.2 Correlation Analysis

Due to the non-normality of the data, the non-parametric analysis was used in this research. The correlation analysis was conducted using the Spearman's rank correlation coefficient. The rank correlation coefficient is intended to be resilient both to atypical values and to non-linearity of the underlying relationship, as well as not being susceptible to the influence of exceedingly large values [Fenton and Pfleeger, 1997].

The Spearman's rank correlation coefficient \( r_s \) of the independent variables with Total Defects and Defect Density are presented in Table 5.3. The correlation between Average System Complexity and both defect metrics are not shown in the table because it is a composite or derived metric from Average Data Complexity and Average Structural Complexity. From the analysis results, the correlation between Average System Complexity and Total Defects (TD) and Defect Density (DD) are \( r_s = -0.029 \) and \( r_s = -0.17 \) respectively.

Most of the correlations in Table 5.3 are weak. The correlations between all independent variables with Total Defects are generally weak except for Number of Developers \( (r_s = 0.30) \).

Likewise, the correlations between Average Procedural Complexity (APC), log10Version Downloads (VD) and Number of Developers (Dev) with Defect Density are weaker than with Total Defects, as shown in Table 5.3. The relationship between Average Structural Complexity with Defect Density is slightly higher than other independent variables. In particular, the negative correlation between Average Structural Complexity and Defect Density means, as Average Structural Complexity increases, Defect Density decreases. This relationship is depicted in
Table 5.3 – Spearman’s Rank Correlation Coefficient

<table>
<thead>
<tr>
<th>Metrics</th>
<th>ADC</th>
<th>ASC</th>
<th>APC</th>
<th>VD</th>
<th>Dev</th>
<th>CS</th>
<th>TD</th>
<th>DD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Data Complexity (ADC)</td>
<td>1.00</td>
<td>0.24</td>
<td>0.77</td>
<td>0.11</td>
<td>-0.14</td>
<td>0.71</td>
<td>-0.007</td>
<td>0.03</td>
</tr>
<tr>
<td>Average Structural Complexity (ASC)</td>
<td>1.00</td>
<td>0.18</td>
<td>-0.10</td>
<td>0.11</td>
<td>0.24</td>
<td>0.05</td>
<td>-0.33</td>
<td></td>
</tr>
<tr>
<td>Average Procedural Complexity (APC)</td>
<td>1.00</td>
<td>0.07</td>
<td>-0.15</td>
<td>0.95</td>
<td>0.04</td>
<td>0.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log10Version Downloads (VD)</td>
<td>1.00</td>
<td>-0.12</td>
<td>0.10</td>
<td>0.16</td>
<td>-0.02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Developers (Dev)</td>
<td>1.00</td>
<td>0.04</td>
<td>0.30</td>
<td>-0.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class Size (CS)</td>
<td>1.00</td>
<td>-0.01</td>
<td>-0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A scatter plot Figure 5.6. For visualization of the relationships between the variables, refer to Figure 5.5 to 5.9, respectively.

Interestingly, the results in Table 5.3 show that there is a considerably strong correlation between Average Data Complexity and Average Procedural Complexity ($r_s = 0.77$) and between Average Data Complexity and Class Size ($r_s = 0.71$). Furthermore, the correlation between Average Procedural Complexity and Class size is also extremely strong ($r_s = 0.95$).

5.4.3 Univariate Analysis

The next step in data analysis is to perform univariate analysis on the data, where the relationships of individual independent variables with Total Defects and Defect Density are measured. This analysis was carried out using the Spearman’s rank correlation coefficient. The results of the analysis are summarized in Table 5.4 and Table 5.6, respectively, where 'p-value' represents the statistical significance. The definition of 'p-value' is given as:

- Statistical significance - the probability that the coefficient is different from zero by chance Hoaglin et al. [1983], Harrell [2001] and usually referred to as $\alpha$ level. The $\alpha$ level used in this research is 0.05.

5.4.3.1 Univariate Analysis for Total Defects

The univariate analysis for Total Defects with the variables at system level is conducted to investigate the influence of these variables on defects for the system.
5.4 Analysis Results

From Table 5.4, we can see that:

- Total Data Complexity (p-value = 0.001), Total Structural Complexity (p-value = 0.008), Total System Complexity (p-value = 0.003) and Number of Developers (p-value = 0.002) are highly significant in correlation to Total Defects.

- Total Procedural Complexity is very highly significant in correlation to Total Defects (p-value < 0.001).

- The effect of the log10Version Downloads is not significant (p-value = 0.107).

### Table 5.4 – Univariate Analysis for Total Defects

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Coeff</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Data Complexity (TDC)</td>
<td>0.32</td>
<td>0.001</td>
</tr>
<tr>
<td>Total Structural Complexity (TSC)</td>
<td>0.26</td>
<td>0.008</td>
</tr>
<tr>
<td>Total Procedural Complexity (TPC)</td>
<td>0.35</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Total System Complexity (TSysC)</td>
<td>0.29</td>
<td>0.003</td>
</tr>
<tr>
<td>log10Version Downloads (VD)</td>
<td>0.16</td>
<td>0.107</td>
</tr>
<tr>
<td>Number of Developers (Dev)</td>
<td>0.30</td>
<td>0.002</td>
</tr>
</tbody>
</table>

At system level, Total Defects correlates well with the independent variables, with the exception of Version Downloads. The correlations between variables are moderate and significant, as shown in Table 5.4. These results are useful in investigating the overall effects of the various complexities at the system level in relation to the total defects found (in this case, after delivery).

**Figure 5.1** illustrates the plot between Total Defects and Total System Complexity. The figure shows that there are a few outliers (points where Total Defects > 80). Since non-parametric analysis is resilient to outliers, the outlying points are kept for the analysis.

A box plot of the data distribution of Total Defects is illustrated in **Figure 5.2**. It shows that there are nine outliers in the dataset and skewed to the left (shown by the crossbar in the interior of the box which represents the median of the dataset). The numbers correspond to the particular systems included in
Figure 5.1 – Total Defects vs Total System Complexity

Figure 5.2 – Box Plot for Total Defects
5.4 Analysis Results

the dataset as shown in Appendix B. The abbreviations used in the tables in Appendix B are defined in the glossary section. The systems are: Tvbrowser, HSQL Database Engine, Jabref, Findbugs, soapUI, Freemind, Robocode, VASSALEngine and OpenbravoPOS. Further details on box plot interpretation are explained by Hoaglin [Hoaglin et al., 1983].

5.4.3.2 Univariate Analysis for Defect Density

The analysis results for Total Defects need to be verified by removing the effect of size from the correlations between defects and other independent variables. This is important in order to conclude that there are genuine influences of the independent variables with post-delivery defects in the systems. This can be achieved by normalizing Total Defects with Lines of Code (LOC); to obtain defects per thousand Lines of Code (KLOC), in other words, Defect Density as the dependent variable.

![Defect Density vs Average System Complexity](image)

**Figure 5.3** – Defect Density vs Average System Complexity

During the initial analysis, it was found that there are two outliers that highly influence the relationships between Defect Density and other independent variables. The scatter plots in Figure 5.3 show the relationship of Defect Density with Average System Complexity. After checking why the outlier systems behave differently, it was discovered that both of the systems, Cewolf and StrutsTestCase, are from the earliest versions of any of the systems, very highly downloaded, small systems and having quite a number of defect reports. The profiles of the outlying systems are given in Table 5.5.
Table 5.5 – Defect Density Outlier Analysis

<table>
<thead>
<tr>
<th>Systems</th>
<th>KLOC</th>
<th>NC</th>
<th>CS</th>
<th>ADC</th>
<th>ASC</th>
<th>APC</th>
<th>TD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cewolf</td>
<td>2.34</td>
<td>108</td>
<td>21.73</td>
<td>4.50</td>
<td>2.50</td>
<td>5.98</td>
<td>20</td>
</tr>
<tr>
<td>StrutsTestCase</td>
<td>4.20</td>
<td>94</td>
<td>45.15</td>
<td>7.21</td>
<td>0.90</td>
<td>15.35</td>
<td>58</td>
</tr>
</tbody>
</table>

The distribution of the Defect Density data is summarized in the box plot in Figure 5.4, showing the outlier. The numbers represent the sequence of the system in the dataset, as shown in Appendix B. The results for univariate analysis of the independent variables with Defect Density are depicted in Table 5.6.

Figure 5.4 – Box Plot for Defect Density

- The Defect Density is not influenced by Average System Complexity (p-value=0.092) or Average Data Complexity (p-value = 0.77).
- Only Average Structural Complexity proves to be highly significant (p-value = 0.001). This means that Defect Density is influenced more by Average Structural Complexity than Average Data Complexity. The negative regression coefficient indicates that the systems with higher Average Structural Complexity are more likely to have lower Defect Density.
- Class Size does not influence Defect Density as shown by p-value = 0.864. The relationship of Class Size and Defect Density is further discussed in Section 6.2.1.
5.4 Analysis Results

Figure 5.5 – Defect Density vs Average Data Complexity

Figure 5.6 – Defect Density vs Average Structural Complexity
Chapter 5: Data Analysis

**Figure 5.7** – Defect Density vs Average Procedural Complexity

**Figure 5.8** – Defect Density vs log10Version Download
5.4 Analysis Results

Table 5.6 – Univariate Analysis for Defect Density

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Coeff.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Data Complexity (ADC)</td>
<td>0.03</td>
<td>0.774</td>
</tr>
<tr>
<td>Average Structural Complexity (ASC)</td>
<td>-0.33</td>
<td>0.001</td>
</tr>
<tr>
<td>Average System Complexity (ASysC)</td>
<td>-0.17</td>
<td>0.092</td>
</tr>
<tr>
<td>Average Procedural Complexity (APC)</td>
<td>0.02</td>
<td>0.864</td>
</tr>
<tr>
<td>log10Version Downloads (VD)</td>
<td>-0.02</td>
<td>0.274</td>
</tr>
<tr>
<td>Number of Developers (Dev)</td>
<td>-0.07</td>
<td>0.465</td>
</tr>
<tr>
<td>Class Size (CS)</td>
<td>-0.02</td>
<td>0.861</td>
</tr>
</tbody>
</table>

- Average Procedural Complexity, log10Version Downloads (VD), and Number of Developers are not significant for estimating Defect Density.

The scatter plots for Defect Density and the independent variables are illustrated in Figure 5.5 to 5.8. The relationships between Defect Density and the independent variables will be discussed in detail in Chapter 6.

5.4.3.3 Hypothesis Tests

This section discusses the results of the hypothesis tests conducted to validate the research hypotheses stated in Chapter 1 and 4.

1. **H1**: Increasing values of the Average System Complexity (Average Structural Complexity plus Average Data Complexity) correlate with increasing post-delivery defect density.

   This hypothesis is tested using the probability test in Spearman’s rank correlation coefficient. The results of the test show that $r_s = -0.17$ and p-value = 0.092, which means it is not correlated with post-delivery defect-density. Hence, H1 is not supported by the result.

2. **H2**: Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Average Procedural Complexity (Cyclomatic...
Chapter 5: Data Analysis

Complexity), as in some non-object-oriented systems.

The result from the univariate analysis of the correlation between Average Data Complexity and Average Procedural Complexity shows that Average Data Complexity is very highly significant as an estimator of Average Procedural Complexity ($r_s = 0.77$, p-value < 0.001). This supports hypothesis H2 that the increase in Average Data Complexity will also increase the Average Procedural Complexity, or number of decisions, in the classes for object-oriented systems as in non-object-oriented systems. The scatter plot of the relationship between these variables is shown in Figure 5.9 and the further discussion related to this relationship is deferred to Section 6.2.4.

3. **H3**: Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Class Size, as in some non-object-oriented systems.

Similarly, the univariate analysis result of Average Data Complexity and Class Size shows that Average Data Complexity is very highly significant as an estimator of Class Size ($r_s = 0.71$, p-value < 0.001). This supports hypothesis H3 that the increase in Average Data Complexity will also increase the Class Size of the systems. The relationship between these variables is depicted in Figure 5.10 and further discussion will be covered in Section 6.2.4.

4. **H4**: There is a relationship between System Complexity and Procedural Complexity that will help minimize defects in the final system.

Another test is conducted to see the correlation between the ratio of System Complexity and Procedural Complexity. The Spearman correlation between the variables supports hypothesis H4 ($r_s = -0.19$, p-value = 0.05). The scatter plot of the relationship between these variables is illustrated in Figure 5.11 and further discussed in Section 6.2.6.

5.4.4 Regression Analysis

In this section, the results for regression or multivariate analysis of the dependent and independent variables are presented. These results are obtained by performing a stepwise Generalized Linear Models using Poisson distribution. It is essential to note that the design metrics, i.e., Average Data Complexity and Average Structural Complexity and usage metrics i.e., Version Downloads are not expected to account for all of the variation of post-delivery defects, since other factors are
possibly important too, for example, developers' experience. Nevertheless, the goal of performing the regression analysis is to determine whether the measures appearing significant in the univariate analysis are complementary and useful for prediction.

For the purpose of achieving that goal, it is important to show that when these measures are used together in a multivariate model, they are significantly
related to post-delivery defects. In other words, when measures remain significant covariates when included in the multivariate model, this means that they are complementary in explaining post-delivery defects.

The regression model is only built for Defect Density as the dependent variable because it is going to be used for the analysis with other independent variables throughout this thesis.

5.4.4.1 Model for Defect Density

In order to build the regression model for Defect Density, all variables were transformed using a log (base e) transformation to normalize the variables. After the transformation, all variables exhibit the normal distribution.

The stepwise backward elimination selection process is applied on independent variables to obtain the prediction model for Defect Density. In regression analysis, coefficients have a tendency to adjust, statistically, for other covariates [Harrell, 2001], [Briand et al., 2000]. Sometimes, covariates are weak predictors of the dependent variable when taken individually, and become more significant when integrated in a multivariate model. The multivariate model for Defect Density is shown in Table 5.7.

The multivariate model for Defect Density (DD) is:
5.4 Analysis Results

Table 5.7 - Multivariate Model for Defect Density

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Coeff.</th>
<th>Std Err</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Structural Complexity (ASC)</td>
<td>-1.14</td>
<td>0.15</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Average Data Complexity (ADC)</td>
<td>1.08</td>
<td>0.45</td>
<td>0.016</td>
</tr>
<tr>
<td>Average Procedural Complexity (APC)</td>
<td>-0.94</td>
<td>0.36</td>
<td>0.010</td>
</tr>
<tr>
<td>Intercept</td>
<td>2.27</td>
<td>0.66</td>
<td>&lt;0.001</td>
</tr>
</tbody>
</table>

\[
\text{Defect Density} = 2.27 \times ASC^{-1.14} ADC^{1.08} APC^{-0.94} \quad (5.1)
\]

**Equation 5.1** indicates that any increase in ASC or APC should exponentially decrease post-delivery defects, whilst any increase in ADC should exponentially increase post-delivery defects. Assuming similar magnitude of the 3 variables, then increases in ASC have the greatest affect on post-delivery defects. However, given that ASC is also likely to be the greatest magnitude then ASC will be dominant in determining post-delivery defects.

To verify the findings, one must control for the effect of program size, using lines of code (LOC) in order to conclude that the relationships between the dependent and independent variables are genuine. Thus, LOC was entered into the model as one of the independent variables (control variable). When LOC was included, Average Structural Complexity is still significant to correlate with Defects Density, while Average Data Complexity and Average Procedural Complexity are not significant anymore. The multivariate model with size control is depicted in **Table 5.8**. Thus, the multivariate model for Defect Density after controlling for size is:

\[
\text{Defect Density} = 4.38 \times ASC^{-0.74} LOC^{-0.34} \quad (5.2)
\]

Table 5.8 - Multivariate Model for Defect Density with Size Control

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Coeff.</th>
<th>Std Err</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Structural Complexity (ASC)</td>
<td>-0.74</td>
<td>0.19</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Lines of code (LOC)</td>
<td>-0.34</td>
<td>0.10</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Intercept</td>
<td>4.38</td>
<td>0.82</td>
<td>&lt;0.001</td>
</tr>
</tbody>
</table>
5.4.5 Analysis of the Number of Developers

Common belief in software engineering states that too many developers in code development and maintenance will “spoil” the quality of the end product [Weyuker et al., 2008]. This means, code units will be less fault-prone if they are written and maintained by only a few, or even just one developer.

The number of developers in a project are also tested in this research to investigate whether this believe holds for the open-source projects. The results in the univariate analysis for the number of developers shows that it is not significant as a predictor for Defect Density (p-value = 0.47) and the relationship of the number of developers and Defect Density is depicted in Figure 5.12. That figure shows that the majority of the systems have < 40 developers. Interestingly, 3 systems which have > 40 developers have Defect Density < 1.

![Figure 5.12 – Defect Density vs Number of Developers](image)

Whilst the number of developers in each project might not be a good indicator of the number of defects reported by users, there is a possibility that developers’ experience does have an effect on defects [Fenton and Pfleeger, 1997]. However, it is not possible to obtain this information in this work due to the nature of the systems being investigated, i.e. open-source systems, where it is difficult to collect the data concerning the experience of all developers in the systems.
5.4 Analysis Results

Several prior works [Mockus and Weiss, 2000], [Lakhani and Wolf, 2005], [Weyuker et al., 2008] have attempted to study the effects of the number of developers and developers’ experience on defects. Particularly, in a study by Mockus and Weiss [Mockus and Weiss, 2000], they investigated four developer variables, such as, the number of developers, a measure of recent experience, a measure of experience for the specific subsystem and an overall measure of experience. Their results show that only the overall experience measure was statistically significant, while the other three variables were not significant, including the number of developers. Another study was conducted by Weyuker et al. [Weyuker et al., 2008] to investigate the impact of using data about the number of developers who access individual code units. They reported that the prediction results only have a slight improvement when the number of developers were included in the model, and concluded that the number of developers is not a major influence.

Most developers who are involved in the development of open-source projects are experienced and skilled people who are professionals in the technology industry, as stated by O’Neil [O’Neil, 2009]:

“A statistical study by Lakhani and Wolf [Lakhani and Wolf, 2005], which surveyed developers from a random sample of open-source projects on Sourceforge.net, found that a solid majority of contributors were experienced, skilled individuals with jobs in the technology industry. The average contributor had more than a decade of programming experience; 55 per cent worked on open-source projects as part of their job.”

5.4.6 Analysis by System Categories

Besides doing general analysis on the overall systems, this section discusses the analysis based on different system categories. As mentioned in Chapter 4, the systems collected are divided into four categories, based on system type. The categories are:

- Games/Multimedia (G/M)
- Internet/Communication (I/C)
- Office/Programming/Database (O/P/D)
- Scientific/Engineering/Operating Systems (S/E/OS)

The purpose of doing the analysis by categories is to see the effects of the independent variables on the dependent variables for different system types.
The analysis for the effects of Average Data Complexity, Average Structural Complexity, Average Procedural Complexity and Average System Complexity on Defect Density are performed using the Kruskal-Wallis one-way Analysis of Variance (ANOVA) test.

The Kruskal-Wallis one-way analysis of variance by ranks is a non-parametric method for testing equality of population medians among groups. It is identical to a one-way analysis of variance with the data replaced by their ranks. It is an extension of the Mann-Whitney U test to 3 or more groups. Since it is a non-parametric method, the Kruskal-Wallis test does not assume a normal population, unlike the analogous one-way analysis of variance. However, the test does assume an identically-shaped and scaled distribution for each group, except for any difference in medians.

The test statistic for the Kruskal-Wallis test is $H$. This value is compared to a table of critical values for $\chi^2$ based on the sample size of each group. If $H$ exceeds the critical value for $\chi^2$ at some significance level (usually 0.05) it means that there is evidence to reject the null hypothesis in favor of the alternative hypothesis. The results show that there is no statistically significant difference in the variations based on system categories as depicted in Table 5.9.

**Table 5.9 – Results of Kruskal-Wallis Test**

<table>
<thead>
<tr>
<th>Metrics</th>
<th>$H$</th>
<th>$\chi^2_{0.05,3}$</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defect Density</td>
<td>4.586</td>
<td>7.815</td>
<td>0.205</td>
</tr>
<tr>
<td>Average Data Complexity</td>
<td>2.977</td>
<td>7.815</td>
<td>0.395</td>
</tr>
<tr>
<td>Average Structural Complexity</td>
<td>2.715</td>
<td>7.815</td>
<td>0.438</td>
</tr>
<tr>
<td>Average System Complexity</td>
<td>3.980</td>
<td>7.185</td>
<td>0.264</td>
</tr>
<tr>
<td>Average Procedural Complexity</td>
<td>5.600</td>
<td>7.185</td>
<td>0.133</td>
</tr>
</tbody>
</table>

**Figure 5.13** illustrates the box plot of Defect Density distribution for each system category. It shows that the variances of Office/Programming/Database and Scientific/Engineering/Operating Systems category are higher than the Games/Multimedia and Internet/Communication category. In other words, the Defect Density in Office/Programming/Database has a wide range (0 to 14) while Games/Multimedia has a range of (0,4). The median of Office/Programming/Database is the highest which means that the Defect Density value of Office/Programming/Database is higher than other categories.
5.4 Analysis Results

Figure 5.13 – Defect Density for System Categories

5.4.7 Defect Analysis

This section will mainly discuss the analysis of two surveys conducted during data collection phase, which have been explained in Section 4.5.3, as well as the categorization of defects using the Orthogonal Defect Classification (ODC), discussed in Section 2.7.

5.4.7.1 Defect Information Survey

This survey was conducted during the early stage of data collection with the main objective of verifying the defects reported by users in SourceForge [www.sourceforge.net]. The developers were contacted through the email address as given in SourceForge and associated project websites. The questions in this initial survey are considerably brief (shown in Appendix C), with the intention of investigating how the developers publish defect reports from users and the number of defects reported for the release under investigation. The response rate is good, with 50% of the contacted developers responding to this survey.

The information given for Question 1 is used as the defect report for the corresponding projects in this study. For Question 2, the majority of the projects (70%) record both pre-release and post-release defects, while another 30% only keep post-release defects. Most answers for Question 3 indicate that the developers only record unique occurrences of defects. For Question 4, the majority of projects
Chapter 5: Data Analysis

publish “known problems” in SourceForge bug tracker.

5.4.7.2 Defect Handling Survey

The second survey was carried out to investigate the general defect handling practices in the open-source community. Prior work has been conducted by Koru and Tian, where they surveyed as many as 75 open-source projects to learn the typical defect handling practices employed [Koru and Tian, 2004]. They reported that in general, there are many similarities across all projects in the ways defects are handled. However, there are some significant differences in the contents of defect records and the amount of discipline applied.

The survey was conducted using online survey software, StatPac [www.statpac.com]. The list of questions for this survey are provided in Appendix D. The response rate for this survey is quite poor, with only about 20 percent projects providing a response. Nonetheless, the responses given were useful for understanding how project teams might handle defect reports.

Question 1 inquires about the defect reporting tools/databases being used. All of the respondents informed that they use SourceForge bug tracker to maintain their defect records.

Question 2 is used to determine the roles assumed by the team member(s) who responded to this survey. The result is depicted in the pie chart in Figure 5.16. Some team members assume multiple roles in the project, with 24 percent of them occupying the role of project manager while 22 percent assuming the role of software developer or engineer.

From the results displayed in Table 5.10, Questions 3 to 6 allow multiple answers, while Questions 7 and 8 only accept an answer to a choice. For Question 3, the majority of the respondents (63.6%) choose source code as the highest source of defects, while the rest chose the manuals as source of defects. None of them choose requirement documents, design documents and test documents as the source of defects in their projects.

The answers for Question 4 shows that about half of the respondents claim that they record both pre-release and post-release defects, with no distinction between the defect types, while 20% of the respondents stated that they record both types of defects and categorize them. The rest only record post-release defects (30%)

Question 5 aims to identify reasons that projects report defects. The results show that most of the projects (60%) stated that testing problems are the main reason for defect reporting, followed by incorrect work (20%), inspection (12%)
Table 5.10 – Defect Handling Survey Results

<table>
<thead>
<tr>
<th>Choice order</th>
<th>Question 3: Defect sources</th>
<th>Question 4: Pre/post release</th>
<th>Question 5: Reasons for defect reports</th>
<th>Question 6: Defect reporting style</th>
<th>Question 7: Initial employment</th>
<th>Question 8: Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Requirement documents (0%)</td>
<td>Pre-release (0%)</td>
<td>Requirement problems (0%)</td>
<td>Accumulated (25%)</td>
<td>Inherited (20%)</td>
<td>Very (10%)</td>
</tr>
<tr>
<td>2</td>
<td>Design documents (0%)</td>
<td>Post-release (30%)</td>
<td>Design problems (0%)</td>
<td>Corrected (37.5%)</td>
<td>Project start (30%)</td>
<td>Almost, great majority (20%)</td>
</tr>
<tr>
<td>3</td>
<td>Source code (63.6%)</td>
<td>Both, no distinction (50%)</td>
<td>Testing problems (60%)</td>
<td>Pending (31.25%)</td>
<td>After design (10%)</td>
<td>Almost, major only (30%)</td>
</tr>
<tr>
<td>4</td>
<td>Test documents (0%)</td>
<td>Both, categorized (20%)</td>
<td>Inspection (12%)</td>
<td>Other (6.25%)</td>
<td>After coding (10%)</td>
<td>Not very (10%)</td>
</tr>
<tr>
<td>5</td>
<td>Manuals (36.4%)</td>
<td>Other (0%)</td>
<td>Incorrect work (20%)</td>
<td>-</td>
<td>Testing (10%)</td>
<td>Not (20%)</td>
</tr>
<tr>
<td>6</td>
<td>Other (0%)</td>
<td>-</td>
<td>Other (8%)</td>
<td>-</td>
<td>Post-release (10%)</td>
<td>Other (0%)</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Other (0%)</td>
<td>-</td>
</tr>
</tbody>
</table>
and other reasons (8%). None of the respondents chose requirement problems and design problems as the reasons for defect reports in their project.

Question 6 aims to identify the defect reporting style or the ways defects are reported. Some projects chose combination of answers, which means they report all defects in their projects. From the results, corrected defects are the highest choice (37.5%), followed by pending defects (31.25%), accumulated defects (25%) and other defects such as priority defects (6.25%).

Responses for Question 7 show that most projects record defects from project start (30%) and some inherit the defect reports from previous releases (20%), others report defects after design starts (10%), after coding starts (10%), after testing (10%), and after release (10%).

The purpose of Question 8 is to investigate the consistency of defect reporting in the projects. Quite a number of projects (30%) report defect consistently, but only for major defects. Several projects report defects consistently and report the majority of the defects in their projects (20%), 10% report defects consistently and report all defects, while another 10% are not consistent at all.

5.4.7.3 Defect Classification

The defects reported in SourceForge are not categorised, except for their status and priority level. Therefore, based on the Orthogonal Defect Classification (ODC)
5.4 Analysis Results

Concept introduced by Chillerage et al. [Chillerage et al., 1991], [Chillerage et al., 1992], the reported defects were categorised. The description of each category/type has been discussed in Section 2.7.

![Defect Classification](image)

**Figure 5.15 – Defect Classification**

The classification of defects in the systems under investigation is illustrated in Figure 5.17. The figure shows that the Interface category has the highest percentage of defects (21.98%), followed closely by Function (19.23%). The category with the lowest percentage is Build/Package/Merge (6.59%).

The defects/KLOC and defects/class measures for each system are depicted in Figure 5.18. From the figure, we can see that Cewolf has the highest number of defects/KLOC, followed by HSQL Database Engine, Jabref and Sweet Home 3D. HSQL Database Engine also has the highest number of defects/class, followed by jTDS and SchemaSpy.

5.4.7.4 Systems Profiles

This section discusses the systems profiles based on the Number of Classes, Class Size and system size measured in thousand lines of code (KLOC). The systems selected in this research vary in size as shown in Figure 5.17. The smallest system is JUnitEE with 1.68 KLOC and the biggest is Jboss with 274.17 KLOC.

The Number of Classes for each system also varies from 32 classes (opencsv) to 7453 classes (Jboss) as illustrated in Figure 5.18 while Figure 5.19 summarizes the Class Size for each system. The Class Size varies from 22 (Cewolf) to 169 (jTDS) lines of code per class. From Figure 5.18, we can see that systems with small
Figure 5.16 - Defects/Class and Defects/KLOC for Systems
Figure 5.17 - System Size in KLOC
Figure 5.18 - Number of Classes for System
Figure 5.19 - Class Size for Systems
number of classes, in particular, SchemaSpy, Xinco, Med’s Movie Manager, jTDS, LaTeXDraw and HSQL Database Engine tend to have big class sizes, as shown in Figure 5.19. However, it is quite the opposite for systems with large number of classes, such as Joone, Squirrel SQL Client and Jboss tend to have small class sizes.

5.5 Results Comparison with using Card and Glass Formula

This section will discuss the comparison of the system complexity model of this work with that of Card and Glass [Card and Glass, 1990]. Whilst still keeping the idea of Average System Complexity as the combination of Average Structural Complexity and Average Data Complexity, some modifications was introduced to suit the object-oriented design environment.

Initially, the Average Data Complexity (ADC), Average Structural Complexity (ASC) and Average System Complexity (ASysC) are calculated using the formula introduced by Card and Glass (CG) as shown by Equation 2.5 (ASysC), 2.6 (ASC) and 2.7 (ADC) in Section 2.6.1.3.

The results of the statistical analysis for CG model are summarized in Table 5.11, which shows the Spearman correlation coefficient or ‘Coeff.’ and p-value for the variables.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Coeff.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Data Complexity (ADC)</td>
<td>0.27</td>
<td>0.006</td>
</tr>
<tr>
<td>Average Structural Complexity (ASC)</td>
<td>-0.42</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Average System Complexity (ASysC)</td>
<td>-0.42</td>
<td>&lt;0.001</td>
</tr>
</tbody>
</table>

Referring to the results in Table 5.11, CG Average Data Complexity is highly significant as a predictor for Defect Density (p-value = 0.006). CG Average Structural Complexity (p-value < 0.001) and CG Average System Complexity (p-value < 0.001) are both very highly significant as an estimator for Defect Density.

However, statistical significance alone does not determine the correctness of the model. A closer look at the CG Average System Complexity shows that it
5.5 Results Comparison with using Card and Glass Formula

Table 5.12 – Results of the Proposed New Model

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Coeff.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Data Complexity (ADC)</td>
<td>0.03</td>
<td>0.77</td>
</tr>
<tr>
<td>Average Structural Complexity (ASC)</td>
<td>-0.33</td>
<td>0.001</td>
</tr>
<tr>
<td>Average System Complexity (ASysC)</td>
<td>-0.17</td>
<td>0.09</td>
</tr>
</tbody>
</table>

is heavily influenced by the CG Average Structural Complexity value as shown in Table 5.14. The subsequent sections will discuss each variable in detail and explain several adjustments made to the CG model to suit the OO programming environment.

5.5.1 Structural Complexity

In essence, CG Average Structural Complexity only considers one aspect of dependencies in OO systems, Fan-out, and does not include inheritance. Therefore, it needs to be adjusted to accommodate both aspects of dependencies in OO systems.

The details of the adjustment have been discussed in Section 3.3. A coupling metric, CBO is chosen as the surrogate measure for Structural Complexity, to replace Fan-out because it captures both dependencies in OO systems, the “accesses”/“uses” (Fan-out) and “inheritance” (Fan-in). Hence, a new formula is introduced to represent the Structural Complexity of OO systems, as given by Equation 3.1 in Section 3.3 and re-iterated here for easy reference.

\[
S = \frac{\sum CBO(i)}{n} \tag{5.3}
\]

where

- \( S \) = Average Structural Complexity
- \( CBO(i) \) = CBO of class i
- \( n \) = Number of classes in system

The comparison of the relationship between Defect Density and Average Structural Complexity using Card and Glass formula and the adjusted formula is depicted by the scatter plot in Figure 5.20.
Chapter 5: Data Analysis

The newly proposed Average Structural Complexity was analyzed using the same statistical techniques as CG Average Structural Complexity. The results are summarized in Table 5.12. The proposed formula is referred to as the Average Structural Complexity (ASC) in this thesis. ASC is highly significant as a predictor for Defect Density (p-value = 0.001), while CG Average Structural Complexity is very highly significant (p-value < 0.001)

5.5.2 Data Complexity

Card and Glass suggest that Average Data Complexity is dependent on its own input/output (I/O) complexity and inversely dependent on the Fan-out of the module. This argument is based on dispersion of the work brought to a module by variables. Their formula for Average Data Complexity was defined by Equation 2.7 in Section 2.6.1.3.

However, in OO systems this does not necessarily happen because of the concept of “encapsulation” or “information hiding”. Based on the arguments brought forward in Section 3.3, a new Average Data Complexity formula is proposed, as shown by Equation 3.2. The values calculated using CG Average
5.5 Results Comparison with using Card and Glass Formula

Data Complexity and the proposed Average Data Complexity are summarized in Table 5.14 and depicted in Figure 5.21.

![Figure 5.21 - Data Complexity Comparison](image)

The results from the statistical analysis for this variable are summarized in Table 5.12. The proposed Average Data Complexity (ADC) is not significant as a predictor for Defect Density with p-value = 0.77 compared to CG Average Data Complexity (p-value = 0.006).

Even though the proposed Average Data Complexity is not significant as a predictor for Defect Density, it is very highly significant as a predictor for Average Procedural Complexity ($r_s = 0.77$, p-value < 0.001) and also can effectively estimate Class Size ($r_s = 0.71$, p-value < 0.001). The correlation between CG Average Data Complexity and Average Procedural Complexity is not as good as the proposed ADC ($r_s = 0.53$, p-value < 0.001). The correlation between CG Average Data Complexity and Class Size is ($r_s = 0.44$, p-value = 0.002).

Further use of these metrics will be discussed in Section 6.2.6.
5.5.3 System Complexity

The Card and Glass concept of Average System Complexity (ASysC) as the combination of Average Structural Complexity (ASC) and Average Data Complexity (ADC) is maintained in this work. The CG Average System Complexity is shown by the Equation 2.5/5.4 and consists of:

$$\text{CG ASysC} = \frac{\sum f^2(i)}{n} + \sum \frac{v(i)}{[f(i) + 1]}$$ (5.4)

Based on the arguments given in Section 3.3, CG Average Structural Complexity is substituted with the Equation 3.1 and CG Average Data Complexity with the Equation 3.2 which makes the proposed Average System Complexity as:

$$\text{ASysC} = \frac{\sum CBO(i)}{n} + \sum \frac{v(i)}{n}$$ (5.5)

After doing the statistical analysis on the proposed Average System Complexity (Equation 5.5), I found that it is not significant in predicting Defect Density with p-value = -0.17 as shown in Table 5.12 while CG Average System Complexity (Equation 5.4) is significant with p-value = <.001. The comparison of results obtained using the two formulas is illustrated in Figure 5.24.

However, the CG Average System Complexity value is dominated by CG Average Structural Complexity as shown in Table 5.14. Hence, the Spearman correlation coefficient and p-value of CG Average System Complexity in Table 5.11 is mainly driven by CG Average Structural Complexity on OO systems.

Considering the comparison between the two models, and particularly the form of the CG model for ASC, another alternative model could be considered as given by Equation 5.6.

$$\text{Alternative model} = \frac{\sum CBO^2(i)}{n} + \sum \frac{v(i)}{[CBO(i) + 1]}$$ (5.6)

This alternative model relates more closely to the CG model and the correlation analysis for this model is shown in Table 5.13. The results show that Average Data Complexity, Average Structural Complexity and Average System Complexity are highly significant in correlation with post-delivery defects. The values for this model are depicted in Table 5.13. Although there are some differences in the results of the three models, the Average Structural Complexity in all models show similar negative correlation. This result is counter-intuitive compared to the usual notion that the increase in Structural Complexity should increase Defect Density in a system. This issue will be further discussed in Section 6.2.2.
There are 3 basic observations that can be had from the results:

1. Average Structural Complexity stands out as having a highly significant relationship with Defect Density in all models.

2. In both the CG model and alternative model, the Average Structural Complexity component has greater contribution than Average Data Complexity in the formulation of Average System Complexity, whereas, in the proposed model, both components are comparatively similar in size.

3. Average Data Complexity, Average Structural Complexity and Average Sys-
tem Complexity are all significant in both CG and the alternative models but only Average Structural Complexity is significant in the proposed model.

Although the proposed Average System Complexity is not significant in predicting Defect Density, the ratio of the Average System Complexity and Average Procedural Complexity is significant in predicting Defect Density with p-value = 0.052. This ratio will be discussed in detail in **Section 6.2.6**.

The results of the calculation are summarized in Table 5.14 which shows that the Average Structural Complexity (ASC), Average Data Complexity (ADC), Average System Complexity (ASysC) and the ratio of System Complexity and Procedural Complexity (SP) for Card and Glass (CG) model are compared with the proposed model (PM) and the alternative model (AM) proposed in this research. Average Procedural Complexity (APC) and Number of Classes (NC) are also included in the table.
5.5 Results Comparison with using Card and Glass Formula
Table 5.14 - Comparison of Results using Two Models

Rystein N a m p
opencsv
Krut
JPOX
struts-menu
SchemaSpy
EZMorph
JOpenChart
JUnitEE
JWebUnit
Xinco
SweetHomeSD
FlickrBackup
JUpload
Davmail
StrutsTestCase
Jimm
MovieManager
Jipe
Barbeque
Jacob-project
Cewolf
ehcache
Jtidy
JavaEmailServer
Eclipse
jTDS
aTunes
OpenCards
EuroBudget
PJIRC
Mars
ControlRemote
LaTeXDraw
CRONOMETER
j Memorize
Barcode4J
JavaHMO
JPodder
FreeTTS
XUI
HTMLParser
A]lianceP2P
Green
Paros
MicroEmulator
Dozer

Nr.
32
33
40
43
45
45
49
50
69
71
73
73
84
94
94
96
98
100
101
105
108
111
123
128
141
141
142
160
176
182
183
188
189
189
203
207
243
250
254
255
257
259
259
260
275
285

continued

CG

PM

AM

2.41
2.39

2.06

4.25

1.12

1.26

1.03
2.14
4.13
4.42
II.22 3.84
1.22
I.46
27.72 4.29
14.23 3.41
7.78
3.34
II.93 2.86
10.65 3.69
11.77 3.96
0.90
1.49
49.29 8.72
24.99 4.91

I.43
2.51
16.42
14.93

19.36
9.68
7.89
7.00
10.71
14.90
10.70
18.03
15.59
22.26
23.66
19.02
17.44
19.39
20.34
59.34
17.37
20.49
8.28
9.26
4.06
9.09
12.74
29.17
17.52
15.08

2.61

4.26
4.45
2.50
3.68
4.72
2.90
5.05
4.30
5.58
6.48
4.38
3.29
5.28
4.50
7.66
4.71
4.72
3.79
3.53
1.12

3.54
3.69
6.46
3.63
4.15
13.35 4.11
15.50 4.00
33.42 2.98

on next

page

CG

6.74
3.81
1.05
6.90
4.58
3.56
17.08 2.82
19.56 3.22
14.72 3.21
2.80
I.49
18.40 2.99
II.62 3.99
11.17 1.78
3.18
8.20
13.62 3.60
15.66 1.78
0.82

76.02
24.09
6.81
18.13
19.78
6.25
13.51

4.84
1.92
3.01
3.17
1.62
1.82

2.01
2.49
22.24 1.70
1.99
8.40
25.50 1.22
18.47 4.02
31.11 2.07
42.01 1.51
19.19 2.60
10.83 2.57
27.92 1.76
20.25 2.51
58.70 2.49
22.22 2.45
22.32 1.30
14.38 1.79
12.50 2.26
1.25
3.50
12.50 2.47
13.65 3.39
41.72 1.90
13.20 3.03
17.26 1.63
16.90 2.43
16.03 2.42
2.35
8.85

SP

ASysC

ADC

ASC

PM

AM

CG

PM

AM

13.69
5.55
10.53
7.21
8.51
9.24
9.24
4.42
8.84
10.07
4.70

4.47
5.20
2.30

9.14
6.21
8.33
6.07

1.66

19.24

1.70
1.91
1.99
1.67
2.28

14.44
4.26
30.71

8.72
3.87
6.25
6.87
18.74
21.26
16.63
3.48

7.85
10.40
5.27
7.21
10.89
10.91
7.36
5.05
5.77

2.03

15.75
6.67
11.55
9.35
12.64
13.67
13.08
5.64
13.13
13.48
8.04
10.71
14.10
9.22
8.12
19.60
15.82
9.97
9.31
10.22
7.00
10.58
10.53
7.76

2.61

1.08

2.22
1.06

3.79
1.12

1.85
2.04
0.96
1.06

1.29
4.50
1.48
6.90
1.02
5.81
1.25
4.86
0.70
4.25
12.41 2.34
1.17
7.73
0.85
6.33
1.52
8.19
1.59
6.84
1.04
6.51
1.51
8.33
12.35 1.43
8.24
1.44
0.76
4.37
1.08
5.17
1.39
6.30
2.55
5.40
1.48
6.72
2.03
9.52
1.07
7.96
1.83
8.50
1.00
5.15
1.44
7.35
1.46
7.32
1.46
5.79

18.16

18.22

9.56
15.11
14.25
13.55
6.33
51.21
28.00
22.53
11.30
9.71
9.01
13.20
16.60

12.70
19.25 9.30
19.61 16.71
24.33 13.30
25.17 12.81
21.63 12.57
20.01 10.13
21.16 11.80
22.84 12.83
61.84 20.02
19.82 12.95
21.79 9.10
10.07
11.52
7.55
11.56
16.13
31.08
20.55
16.71
15.78
17.92
35.77

20.07
13.90
12.25
10.23
15.84
16.72
4.61
77.14
25.94
8.85
19.09

APC
23.97
22.09
38.05
14.67
21.53
24.98
15.41
10.16
23.62
28.75
13.08
19.26
23.89
24.55
15.35
38.25
40.67
12.67

18.54
20.84 14.49
5.98
7.54
14.99 25.70
23.25 15.55
9.65
21.46
26.20 12.18
20.81 40.06
32.28 17.35
42.85 12.20
20.71 19.97
12.42 18.13
28.96 15.07
21.76 19.23
60.12 42.46
23.67 19.80
23.08 12.52
15.46 11.72

8.96
9.84
13.89 18.51
6.52
3.80
13.16
10.25 13.98 16.02
13.21 15.67 20.20
14.42 42.79 20.00
12.13 15.03 19.18
9.31
18.26 13.64

CG

PM

AM

0.38

0.66
0.30
0.30
0.64
0.59
0.55
0.85
0.56
0.56
0.47
0.61
0.56
0.59
0.38
0.53
0.51
0.39
0.42
0.73
0.71
1.17
0.67
0.41
0.36
0.76
0.42
0.77
1.05
0.63
0.56
0.78

0.36
0.18
0.16
0.47
0.87
0.85
1.08
0.34
0.85
0.48
0.94
0.53
0.66
0.68
0.30
2.02
0.64
0.37
1.51
1.44

0.28
0.22

0.41
0.89
0.73
0.94
0.42
1.30
0.63
0.73
0.78
0.60
0.55
0.41
1.34
0.69
0.95
0.89
0.67
1.51
0.83
0.65
0.59
1.58
0.49
1.40
2.06

1.08
1.10

1.40
1.19
1.46
1.00

1.74
0.86
0.62

0.57
0.72
0.80
1.55
1.07
1.23
0.85

11.46 18.34 18.52
11.32 17.49 13.31 1.35
8.76
10.31 8.63
4.15

1.26

0.95
0.90
0.45
2.15
0.52
1.86
3.51
1.04
0.69
1.92

0.67
0.47
0.65
0.73
0.76
0.53
0.50
0.64
0.65

1.13
1.42
1.84
1.32
0.75
0.29
0.87
0.78

0.72
0.63
0.68
0.62
0.85
1.02

2.14
0.78
1.34
0.99
1.31
1.19

1.20


### Table 5.14 – continued from previous page

<table>
<thead>
<tr>
<th>System Name</th>
<th>NC</th>
<th>ASC</th>
<th>ADC</th>
<th>ASysC</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAPI</td>
<td>285</td>
<td>16.68</td>
<td>4.35</td>
<td>18.96</td>
<td>1.15</td>
</tr>
<tr>
<td>Jmol</td>
<td>295</td>
<td>7.08</td>
<td>2.50</td>
<td>6.26</td>
<td>4.59</td>
</tr>
<tr>
<td>AzSMRC</td>
<td>309</td>
<td>15.58</td>
<td>3.74</td>
<td>14.00</td>
<td>2.04</td>
</tr>
<tr>
<td>DrawSWF</td>
<td>310</td>
<td>11.75</td>
<td>3.76</td>
<td>14.15</td>
<td>2.93</td>
</tr>
<tr>
<td>JGraphT</td>
<td>318</td>
<td>9.02</td>
<td>3.03</td>
<td>9.17</td>
<td>1.80</td>
</tr>
<tr>
<td>HSQDatabase</td>
<td>322</td>
<td>33.72</td>
<td>5.99</td>
<td>35.93</td>
<td>3.15</td>
</tr>
<tr>
<td>HmlUnit</td>
<td>325</td>
<td>25.99</td>
<td>5.23</td>
<td>27.34</td>
<td>2.08</td>
</tr>
<tr>
<td>XBrowser</td>
<td>336</td>
<td>6.19</td>
<td>2.82</td>
<td>7.96</td>
<td>2.12</td>
</tr>
<tr>
<td>JStock</td>
<td>351</td>
<td>23.51</td>
<td>3.90</td>
<td>15.19</td>
<td>2.31</td>
</tr>
<tr>
<td>iFreeBudget</td>
<td>381</td>
<td>36.00</td>
<td>6.32</td>
<td>36.73</td>
<td>1.67</td>
</tr>
<tr>
<td>JIn</td>
<td>384</td>
<td>22.72</td>
<td>4.12</td>
<td>16.97</td>
<td>2.28</td>
</tr>
<tr>
<td>Art of Illusion</td>
<td>404</td>
<td>92.54</td>
<td>10.97</td>
<td>120.35</td>
<td>2.01</td>
</tr>
<tr>
<td>OmegaT</td>
<td>426</td>
<td>20.31</td>
<td>4.05</td>
<td>16.38</td>
<td>1.88</td>
</tr>
<tr>
<td>GanttProject</td>
<td>429</td>
<td>22.26</td>
<td>3.21</td>
<td>10.33</td>
<td>1.89</td>
</tr>
<tr>
<td>JPrvOT</td>
<td>467</td>
<td>23.03</td>
<td>4.93</td>
<td>24.32</td>
<td>1.94</td>
</tr>
<tr>
<td>Jajuk</td>
<td>494</td>
<td>96.82</td>
<td>11.81</td>
<td>139.47</td>
<td>0.92</td>
</tr>
<tr>
<td>Galleon</td>
<td>510</td>
<td>18.07</td>
<td>5.16</td>
<td>26.63</td>
<td>1.66</td>
</tr>
<tr>
<td>wmOpen</td>
<td>539</td>
<td>42.44</td>
<td>6.25</td>
<td>39.05</td>
<td>1.76</td>
</tr>
<tr>
<td>iText</td>
<td>541</td>
<td>52.93</td>
<td>7.93</td>
<td>62.94</td>
<td>2.57</td>
</tr>
<tr>
<td>ZK</td>
<td>544</td>
<td>23.30</td>
<td>4.19</td>
<td>17.52</td>
<td>2.05</td>
</tr>
<tr>
<td>dcm4che</td>
<td>546</td>
<td>18.60</td>
<td>4.64</td>
<td>21.52</td>
<td>2.80</td>
</tr>
<tr>
<td>JabRef</td>
<td>574</td>
<td>18.47</td>
<td>4.07</td>
<td>16.55</td>
<td>1.75</td>
</tr>
<tr>
<td>Freemind</td>
<td>580</td>
<td>13.27</td>
<td>2.62</td>
<td>6.89</td>
<td>3.64</td>
</tr>
<tr>
<td>DataCrow</td>
<td>584</td>
<td>45.67</td>
<td>7.08</td>
<td>50.11</td>
<td>0.63</td>
</tr>
<tr>
<td>jPicEdit</td>
<td>600</td>
<td>15.26</td>
<td>4.30</td>
<td>18.48</td>
<td>1.75</td>
</tr>
<tr>
<td>RunaWFE</td>
<td>609</td>
<td>45.75</td>
<td>9.35</td>
<td>87.48</td>
<td>0.80</td>
</tr>
<tr>
<td>JUNG</td>
<td>652</td>
<td>24.01</td>
<td>4.22</td>
<td>17.79</td>
<td>1.38</td>
</tr>
<tr>
<td>Robocode</td>
<td>692</td>
<td>13.52</td>
<td>3.30</td>
<td>10.87</td>
<td>1.97</td>
</tr>
<tr>
<td>TVBrowser</td>
<td>734</td>
<td>26.10</td>
<td>5.45</td>
<td>29.74</td>
<td>1.89</td>
</tr>
<tr>
<td>jFreeChart</td>
<td>750</td>
<td>30.42</td>
<td>6.09</td>
<td>37.03</td>
<td>2.63</td>
</tr>
<tr>
<td>JGAP</td>
<td>769</td>
<td>32.43</td>
<td>6.12</td>
<td>37.42</td>
<td>1.87</td>
</tr>
<tr>
<td>soapUI</td>
<td>781</td>
<td>16.89</td>
<td>3.97</td>
<td>15.76</td>
<td>1.50</td>
</tr>
<tr>
<td>mvmForum</td>
<td>781</td>
<td>57.39</td>
<td>6.77</td>
<td>45.86</td>
<td>2.47</td>
</tr>
<tr>
<td>DrJava</td>
<td>809</td>
<td>21.05</td>
<td>3.61</td>
<td>13.06</td>
<td>1.77</td>
</tr>
<tr>
<td>jEdit</td>
<td>837</td>
<td>24.29</td>
<td>4.37</td>
<td>19.07</td>
<td>1.96</td>
</tr>
<tr>
<td>TripleA</td>
<td>841</td>
<td>44.82</td>
<td>6.54</td>
<td>42.71</td>
<td>1.76</td>
</tr>
<tr>
<td>Frost</td>
<td>847</td>
<td>13.84</td>
<td>3.66</td>
<td>13.39</td>
<td>2.04</td>
</tr>
<tr>
<td>YALE/RapidMiner</td>
<td>855</td>
<td>22.69</td>
<td>5.62</td>
<td>31.58</td>
<td>1.92</td>
</tr>
<tr>
<td>OpenbravoPOS</td>
<td>858</td>
<td>14.27</td>
<td>3.75</td>
<td>14.06</td>
<td>2.14</td>
</tr>
<tr>
<td>Saxon</td>
<td>876</td>
<td>142.13</td>
<td>14.03</td>
<td>196.96</td>
<td>1.42</td>
</tr>
<tr>
<td>GATE</td>
<td>947</td>
<td>31.27</td>
<td>6.18</td>
<td>38.17</td>
<td>1.40</td>
</tr>
<tr>
<td>Findbugs</td>
<td>1003</td>
<td>32.10</td>
<td>5.24</td>
<td>27.44</td>
<td>1.91</td>
</tr>
<tr>
<td>MX4</td>
<td>1011</td>
<td>28.11</td>
<td>4.33</td>
<td>18.76</td>
<td>1.68</td>
</tr>
<tr>
<td>JasperReports</td>
<td>1021</td>
<td>48.84</td>
<td>5.23</td>
<td>27.33</td>
<td>2.11</td>
</tr>
<tr>
<td>OpenXava</td>
<td>1067</td>
<td>24.60</td>
<td>4.48</td>
<td>20.05</td>
<td>2.78</td>
</tr>
<tr>
<td>Phex</td>
<td>1111</td>
<td>32.05</td>
<td>5.18</td>
<td>26.79</td>
<td>1.60</td>
</tr>
<tr>
<td>Ozone</td>
<td>1148</td>
<td>13.92</td>
<td>3.09</td>
<td>9.52</td>
<td>2.64</td>
</tr>
<tr>
<td>Joone</td>
<td>1195</td>
<td>23.70</td>
<td>3.70</td>
<td>13.69</td>
<td>1.63</td>
</tr>
<tr>
<td>VASSALEngine</td>
<td>1233</td>
<td>32.66</td>
<td>5.60</td>
<td>31.35</td>
<td>1.87</td>
</tr>
<tr>
<td>Openuptes</td>
<td>1254</td>
<td>64.19</td>
<td>9.95</td>
<td>98.95</td>
<td>1.81</td>
</tr>
</tbody>
</table>

**continued on next page**
The analyses in this research are done with the objectives of answering the research questions and testing the hypotheses. Furthermore, another motivation for this research is to establish early prediction of post-delivery defects for object-oriented (OO) systems in a similar way to how Card and Glass (CG) had determined for pre-delivery defects in structured systems.

Initially, the analysis was done based on Card and Glass’s original model. However, upon obtaining the results, it became apparent that the model needed some adjustment because of the underlying differences between structured and OO languages. In the CG model, the main contributing factor to Average System Complexity is Average Structural Complexity, which may be due to particular properties or characteristics of OO systems.

It is obvious that the original CG model given in Equation 5.4 is highly significant in correlation with post-delivery Defect Density, while the proposed model is not significant. This result shows that the form of the CG System Complexity Model is applicable to OO systems, as well as structured systems. The main difference is the imbalance between the Average Structural Complexity and Average Data Complexity which means the construct of Average System Complexity is largely influenced by Average Structural Complexity in OO systems.

Although only Average Structural Complexity is significant in the proposed model, it is more balanced. Moreover, Average Data Complexity is very highly significant in correlation to estimating Average Procedural Complexity and Class Size compared to the CG model.

After comparing the two models, it seems appropriate to suggest an alternative model as shown is Equation 5.6. The results in Table 5.13 show that Average
Structural Complexity, Average Data Complexity and Average System Complexity are highly significant in correlation with Defect Density. However, further investigation/study is needed in determining why this form of model works for OO systems.
Chapter 6

Results Discussion

Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.

CeBASE, December 2002

Contents

6.1 Introduction ........................................ 156
6.2 Results Discussion ................................. 156
   6.2.1 Class Size .................................... 156
   6.2.2 Structural Complexity .......................... 159
   6.2.3 Procedural Complexity ......................... 164
   6.2.4 Data Complexity ............................... 167
   6.2.5 System Complexity ............................. 171
   6.2.6 Ratio of System and Procedural Complexity .. 173
6.3 Findings in Relation to the Research Questions ........ 177
   6.3.1 System Complexity and Defect Correlation ...... 178
   6.3.2 Defect Estimation ............................. 178
   6.3.3 Strategies for Minimizing Defects ................. 179
6.4 Limitations ......................................... 180
6.5 Potential Benefits to Industry ....................... 181
6.6 Challenges and Lessons Learnt ....................... 182
6.7 Threats to Validity .................................. 183
   6.7.1 Construct Validity ............................. 183
   6.7.2 Internal Validity .............................. 183
   6.7.3 External Validity .............................. 184
6.1 Introduction

There is no silver bullet to eliminate defects. This statement is repeated again and again in software engineering literature and Glass [Glass, 2003b] pointed out this fact again in his book. Even after thorough testing processes, extensive inspections and reviews, defects still persist after the delivery of software. Heuristically, one would expect that the more defects found during testing, the more defects will appear after delivery. In fact, Compton and Withrow [Compton and Withrow, 1990] have found that as much as a six times greater post-delivery defect density when analyzing modules with defects discovered before delivery. However, this finding is refuted by Fenton and Ohlsson [Fenton and Ohlsson, 2000], where they discovered that modules which exhibit most defects during pre-release testing, later show no defects after delivery.

This research concentrates on studying various independent variables related to system design that may estimate the volume of post-delivery defects as reported by users. Besides the design metrics, another important metric to consider for open-source systems that might have an estimate of the number of defects discovered after delivery, is the number of downloads for the version being investigated. The results reported in Chapter 5 will be further discussed in this chapter, and the applicability of the findings to the software industry at large will be explained in this chapter.

6.2 Results Discussion

This section is an expansion of the discussion surrounding the results in Chapter 5. While Chapter 5 contains a discussion of the statistical based analysis to test the hypotheses stated in Chapter 1 and 3 and possibly build a prediction model, the discussion in this chapter will delve further into the underlying relationships between the variables.

6.2.1 Class Size

One common heuristic in modular design is that smaller modules will have lower defects compared to bigger modules [Card and Glass, 1990]. Several attempts have been made using various empirical data to investigate whether this heuristic is true [Card and Glass, 1990], [Basili and Perricone, 1984], [Shen et al., 1995], [Withrow, 1990], [Hatton, 1997], [Malaysia and Denton, 2000]. In particular, Card and Glass have shown that there is a decrease in Defect Density with increasing
module size, that contradicts the heuristic that smaller modules have fewer defects [Card and Glass, 1990]. They also found that all small modules (1 to 30 LOC) have either no fault or a high fault density, and conclude that there is no relationship between fault density and module size. Also, a study by Fenton and Ohlsson [Fenton and Ohlsson, 2000] at Ericsson found that there is no obvious relationship between size and fault density.

Similar findings were reported by Basili and Perricone [Basili and Perricone, 1984] when they examined FORTRAN modules with fewer than 200 LOC for the most part and found higher defect density in the smaller modules. Likewise, Shen et al. [Shen et al., 1995] found an inverse relationship for these variables up to 500 LOC. They suggested that there are two possible explanations for this phenomena, one being contribution of interface errors, as they are more or less constant regardless of module size. Another explanation is that smaller modules are subject to higher error density due to smaller denominator (size).

Another study by Withrow [Withrow, 1990] found a curvilinear relationship between Defect Density and Class Size. Defect Density decreases with size and then curves up again at the tail when modules become exceedingly large. Withrow found the lowest Defect Density in modules of 250 LOC. He suggested that when module size becomes exceedingly large, the complexity increases to a level beyond a programmer’s immediate span of control and total comprehension.

Thus, issues of whether there is an optimal size for classes arise, and there are several debates on what is the optimal class size to minimize defects. Several investigations have been conducted to answer this question, in particular Hatton [Hatton, 1997] and Malayia and Denton [Malayia and Denton, 2000] analyzed the relationship between Defect Density and module size and found that, as the module size increases, the Defect Density decreases linearly, until it hits a point somewhere between 200 and 400 LOC, where it flattens out and then starts to increase linearly. Fenton and Neil [Fenton and Neil, 1999a] name the theory of the optimal class size for object-oriented system as the “Goldilocks Conjecture”. The “Goldilocks Conjecture” stipulates that there exists an optimal component size, S. As component size increases or decreases away from S, the relative number of defects, or alternatively the probability of a defect, increases [Fenton and Neil, 1999a].

Likewise, El-Emam et al. [El-Emam et al., 2002] conducted a study to investigate the existence of the “Goldilocks Conjecture” in object-oriented applications. They studied three systems written in C++ and Java and used post-release defects as the independent variable. They reported that there is no size threshold effect on defects in object-oriented systems.
In this research, a similar investigation was undertaken to explore the relationship between these two variables. The results show that most systems have a low Defect Density (DD) independent of Class Size (CS) as shown in Figure 6.1. However, there seems to be a greater probability for small sized classes to contain higher Defect Density than larger classes.

![Figure 6.1 - Scatter Plot of Class Size Clusters](image)

Cluster analysis was conducted to group the data according to similar characteristics. Cluster analysis identifies and classifies variables on the basis of the similarity of the characteristics they possess. It seeks to minimize within-group variance and maximize between-group variance. The result of cluster analysis is a number of heterogeneous groups with homogeneous contents. There are substantial differences between the groups, but the individuals within a single group are similar [Hoaglin et al., 1983].

A nonhierarchical cluster analysis, using the K-means method was applied to the data for Defect Density and Class Size. In essence, the nonhierarchical clustering techniques are designed to group items into a collection of K clusters. The number of clusters, K, may either be specified in advance or determined as part of the clustering procedure. The term K-means describes an algorithm that assigns each item to the cluster having the nearest centroid (means). The groupings are determined by computing the Euclidean distance of each item from the group and
reassign each item to the nearest group [Hoaglin et al., 1983].

**Figure 6.1** shows that there are essentially three clusters which help explain the variations in the data. Three clusters were chosen to represent three class sizes, i.e., small, medium and large. In the figure, the rightmost cluster is referred as Cluster 1, next to it is Cluster 2 and the leftmost cluster is Cluster 3. Cluster 3 includes two outliers for Defect Density, which are considered as exceptions. More than 90% of the systems belong to Cluster 3 and 2 (small and medium respectively), while about 8% belong to Cluster 1 (large). The mean values for each of the three clusters are shown in **Table 6.1**. The class size for systems in Cluster 3 range from 20 to 60 LOC per class, while for Cluster 2, they range from 60 to 100 and Cluster 3 ranges from 110 to 170.

**Table 6.1** – Results of Cluster Analysis for Class Size

<table>
<thead>
<tr>
<th>Cluster</th>
<th>n</th>
<th>Mean CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>7</td>
<td>137</td>
</tr>
<tr>
<td>Cluster 2</td>
<td>36</td>
<td>74</td>
</tr>
<tr>
<td>Cluster 3</td>
<td>61</td>
<td>45</td>
</tr>
</tbody>
</table>

The Spearman correlation between these variables (DD & CS) is \( r_s = -0.02 \), and the univariate analysis shows that Defect Density is not influenced by Class Size (p-value = 0.861). This is consistent with the findings of Card and Glass [Card and Glass, 1990] and Fenton and Ohlsson [Fenton and Ohlsson, 2000] that class/module size does not influence Defect Density.

### 6.2.2 Structural Complexity

Structural Complexity/Average System Complexity represents the relationships/dependencies among the modules/classes of a system, in other words, it involves inter-module complexity [Card and Glass, 1990]. The measure of structural complexity used in this work is Coupling between Object Classes (CBO) because the systems being studied are developed using an object-oriented language, Java. The Spearman correlation and univariate analysis were presented in **Table 5.2** and 5.4 in **Chapter 5**. Further discussion of those results will be given here as they apply to the software development industry.

As shown in **Table 5.3** in **Section 5.4.2**, the Spearman correlation coefficient between Average Structural Complexity and Defect Density is \( r_s = -0.33 \), which means that Defect Density decreases with the increase of Average Structural Complexity (ASC). This relationship is depicted by the scatter plot in **Figure 6.2**.
The univariate analysis in Table 5.6 shows that Average Structural Complexity is highly significant as an estimator of post-delivery Defect Density (p-value = 0.001). Again in the multivariate analysis, it is very highly significant (p-value < 0.001). Similarly, Basili et al. [Basili et al., 1996] conducted a study of eight systems developed by students using the Chidamber and Kemerer metrics, and found that CBO was associated with fault-proneness of classes. Moreover, Briand et al. [Briand et al., 2000] explored the relationships between several design measures in OO systems (including CBO) and fault-proneness of classes. They found that CBO is significant in predicting fault-proneness of classes.

The model to estimate Defect Density is given by Equation 6.1

\[
\text{Defect Density} = 2.60 \times \text{ASC}^{-1.11} \tag{6.1}
\]

The equation above means that for each unit of Structural Complexity, Defect Density will decrease by 0.29. The constant (2.60) means that there is an inherent value of Defect Density due to other causes. This result is counter-intuitive to common belief that the increase in structural complexity will also increase Defect Density. The literature search fails to find any study similar to this, but the closest is a study by Gyimóthy et al. [Gyimóthy et al., 2005], who compared the accuracy
of a large metrics suite, including Chidamber and Kemerer metrics, to predict defective classes in one open-source system, Mozilla. They concluded that CBO is the best predictive metric. They also found that LOC to be useful in the prediction.

Although the findings in this research show negative correlation, it is fairly consistent throughout all models, i.e., CG model, the proposed model and the alternative model. In other words, the Average Structural Complexity in all models shows negative correlation with Defect Density. A possible explanation could be the psychological factor in terms of developers' attention paid to complex classes. They may give significantly more attention to complex classes and perhaps even be dismissive of what are deemed simple classes. At the system level, the correlation between Total Structural Complexity and Total Defects is $r_s = 0.26$, which shows a positive correlation, while at class level, the correlation becomes negative. Figure 6.2 shows that most systems with the highest Average Structural Complexity exhibit Defect Density of less than 1. The univariate analysis shows that Number of Developers is not significant in correlation with Defect Density ($r_s = -0.07$, p-value = 0.465). However, prior research has demonstrated that developers' experience is statistically significant in regard to effects on defects [Mockus and Weiss, 2000]. There is a possibility that the developers' experience factor might enable an estimate of post-delivery defects. However, in this research, such information is not available from most of the projects' websites. Further surveys to collect the relevant information would be required. Hence, further investigations are needed to identify the reasons behind this counter-intuitive result.

Figure 6.2 illustrates three clusters which describe the variations in ASC. Three clusters were chosen to represent three types of complexity, i.e., low, medium and high. Similar to Figure 6.1, the rightmost cluster is referred to as Cluster 1, next to it is Cluster 2 and the leftmost cluster is Cluster 3. Cluster 3 includes two outliers for Defect Density, which are considered as exceptions. More than 90% of the systems belong to Cluster 2 and 3, while about 7% belong to Cluster 1. The cluster mean values are described in Table 6.2. The ASC for systems in Cluster 3 range from 1 to 5, for Cluster 2, the range is from 5 to 8 and Cluster 3 ranges from 9 to 14.

Further analysis was conducted on CBO by looking at the ratio of complex classes and less complex classes within each system. Complex classes were defined as the classes which have CBO > 20. The boundary was chosen because the average CBO value for most systems is 20.

The formula of this ratio is:
Chapter 6: Results Discussion

\[ \text{CBO ratio} = \frac{\sum \text{Complex}_{(CBO>20)}}{\sum \text{Less complex}_{(CBO<20)}} \]

where

- \( \text{CBO ratio} = \text{Ratio of total number of classes with high CBO to total number of classes with low CBO} \)
- \( \text{Complex}_{(CBO>20)} = \text{Classes with CBO} > 20 \)
- \( \text{Less complex}_{(CBO<20)} = \text{Classes with CBO} < 20 \)

**Table 6.2 – Results of Cluster Analysis for Average Structural Complexity**

<table>
<thead>
<tr>
<th>Cluster</th>
<th>n</th>
<th>Mean ASC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>6</td>
<td>10.81</td>
</tr>
<tr>
<td>Cluster 2</td>
<td>34</td>
<td>5.77</td>
</tr>
<tr>
<td>Cluster 3</td>
<td>64</td>
<td>3.45</td>
</tr>
</tbody>
</table>

**Figure 6.3 – CBO Complexity Ratio**

The relationship between the CBO ratio and Defect Density is illustrated in Figure 6.3. The correlation coefficient is \( r_s = -0.29 \) and CBO ratio is significant in
### Table 6.3 – CBO and Defects for JBoss Classes

<table>
<thead>
<tr>
<th>Classes</th>
<th>CBO</th>
<th>Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClasspathServlet2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>WebServer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>WebServiceMBean</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ClasspathServlet</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>HsqldbCreateCommand</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JavaSerializationManager</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SSOBaseCase</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>WebIntegrationUnitTestCase</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JBossJSFConfigureListener</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>JSFIntegrationUnitTestCase</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>JspServletOptions</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PostgreSQLCreateCommand</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>ClusteredSingleSignOnUnitTestcase</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>ObjectInputStreamWithClassLoader</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>SSOBaseCase</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SingleSignOnUnitTestCase</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>TxInflowUnitTestCase</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>WebService</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>CachedConnectionBankStressTestCase</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>TargetModuleIDImpl</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>PartitionRestartUnitTestCase</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>AbstractCreateCommand</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>ClusterPartition</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>RetryInterceptorUnitTestCase</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>DRMTestCase</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>TomcatDeployer</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>JDBCStartCommand</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>JDBCEntityBridge2</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>WebMetaData</td>
<td>33</td>
<td>1</td>
</tr>
</tbody>
</table>

predicting Defect Density (p-value = 0.002). **Figure 6.3** shows that systems with CBO ratio $<$ 0.05 tend to have a greater probability of higher Defect Density. On the other hand, the majority of systems with CBO ratio $>$ 0.05 have Defect Density
2. This suggests that systems with more complex classes tend to have low Defect Density. Based on these findings, there is a possibility that complex classes (CBO > 20) receive more attention by the developers than less complex ones.

To test the aforementioned findings, I take a closer look at a system, JBoss, to examine the classes that contain the defects reported by users. The purpose of this exercise is to investigate the CBO value of the defect-prone classes. In order to identify which classes contain the defects, I checked the change log files of JBoss in the project website\textsuperscript{1}. The summary of the defect-prone classes in JBoss is shown in Figure 6.4. The figure shows that most defect-prone classes have CBO value of < 20, while only 2 classes have CBO > 20. This implies that, in the system being investigated, most defects occur in the less complex classes compared to the more complex ones.

### 6.2.3 Procedural Complexity

Procedural Complexity/ Average Procedural Complexity measures the number of decisions in implemented artifacts such as method or classes. In this research, McCabe's Cyclomatic Complexity (as described in Section 2.5.1.2) is used to measure Procedural Complexity. Procedural Complexity can be measured during the implementation phase and checked against the estimate determined from the Data Complexity measure during the design phase. If during the detailed design phase, the designer is not careful about a reasonable balance between Data Complexity and Structural Complexity, then Procedural Complexity may become high.

The results presented in Chapter 5 show that Average Procedural Complexity is not significant in predicting Defect Density with $r_s = 0.07$ and p-value $= 0.864$. Troster attempted to find the correlation between Defect Density and Cyclomatic Complexity using the Pearson Correlation and found that Cyclomatic Complexity is not significant in predicting Defect Density with $r = 0.002$ and p-value $= 0.94$ [Troster, 1992]. The scatter plot of the relationship between these variables is shown in Figure 6.5.

There are three clusters which explain the variations in APC and they are illustrated in Figure 6.4. The reason to choose three clusters is similar to Class Size and ASC. The figure shows that, more than 90% of the systems belong to Cluster 2 and 3, while about 7% belong to Cluster 1. The cluster mean values are described in Table 6.4. The ASC for systems in Cluster 3 range from 5 to 17, while for Cluster 2, the range is from 17 to 27, while Cluster 3 ranges from 29 to 50.

\textsuperscript{1}https://jira.jboss.org/jira/browse/JBAS/fixforversion/12311408
6.2 Results Discussion

Figure 6.4 – Scatter Plot of Average Procedural Complexity Clusters

Figure 6.5 – Cyclomatic Complexity Ratio
Chapter 6: Results Discussion

Table 6.4 – Results of Cluster Analysis for Average Procedural Complexity

<table>
<thead>
<tr>
<th>Cluster</th>
<th>n</th>
<th>Mean APC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>9</td>
<td>35.89</td>
</tr>
<tr>
<td>Cluster 2</td>
<td>35</td>
<td>20.31</td>
</tr>
<tr>
<td>Cluster 3</td>
<td>34</td>
<td>12.78</td>
</tr>
</tbody>
</table>

Further analysis is carried out to investigate the ratio of complex classes in terms of Cyclomatic Complexity (CC) and less complex classes. Complex classes are defined as the classes which have CC of > 50 and less complex classes are the ones having CC of < 50. The threshold of 50 is chosen because in most systems, average method per class is 10 and average CC per method is 5. The formula to calculate this ratio is:

\[
CC\ ratio = \frac{\sum Complex_{(CC>50)}}{\sum Less\ complex_{(CC<50)}}
\]

where

- \(CC\ ratio\) = Ratio of classes with high CC and classes with low CC
- \(Complex_{(CC>50)}\) = Classes with CC > 50
- \(Less\ complex_{(CC<50)}\) = Classes with CC < 50

The relationship of the Cyclomatic Complexity ratio with Defect Density is illustrated in Figure 6.6. Most of the systems have rather low ratio (< 0.20) as shown in the figure, which means they do not have many classes with CC > 50. However, for those systems, Defect Density ranges from 0 to 13.67. The systems that have the ratio higher than 0.20 belong to mixed categories, with the highest, Xinc, belong to Scientific/Engineering/OS category. There is a possibility that the system requires a lot of decision processing as part of its functionality.

Another investigation was conducted on JBoss to check the CC values of the defect-prone classes as reported in the project website. The result is summarized in Table 6.5. The table shows that only 8 out of 29 defect-prone classes have CC > 50 while the rest have CC < 50. That indicates that most defects occur in the classes that have low Cyclomatic Complexity, at least in the system being investigated. Again, there is a possibility that the developers give extra attention to complex classes. Thus, extra care should also be given to classes that are less complex especially if they are required to perform important tasks.
6.2 Results Discussion

Table 6.5 – Cyclomatic Complexity and Defects for JBoss Classes

<table>
<thead>
<tr>
<th>Classes</th>
<th>CC</th>
<th>Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>HsqldbCreateCommand</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQLCreateCommand</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>TxInflowUnitTestCase</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>ClasspathServlet2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>ObjectInputStreamWithClassLoader</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>WebIntegrationUnitTestCase</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>SingleSignOnUnitTestCase</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>PartitionRestartUnitTestCase</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>JBAS4406UnitTestCase</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>JBossJSFConfigureListener</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>JSFIntegrationUnitTestCase</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>JavaSerializationManager</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>AbstractCreateCommand</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>CachedConnectionBankStressTestCase</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>ClasspathServlet</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>WebServiceMBean</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>TargetModuleIDImpl</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>ClusteredSingleSignOnUnitTestCase</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>SSOBaseCase</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>WebService</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>RetryInterceptorUnitTestCase</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>WebServer</td>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>TomcatDeployer</td>
<td>55</td>
<td>1</td>
</tr>
<tr>
<td>DRMTestCase</td>
<td>59</td>
<td>1</td>
</tr>
<tr>
<td>ClusterPartition</td>
<td>71</td>
<td>1</td>
</tr>
<tr>
<td>JDBCEntityBridge2</td>
<td>72</td>
<td>1</td>
</tr>
<tr>
<td>JspServletOptions</td>
<td>97</td>
<td>1</td>
</tr>
<tr>
<td>JDBCStartCommand</td>
<td>122</td>
<td>1</td>
</tr>
<tr>
<td>WebMetaData</td>
<td>158</td>
<td>1</td>
</tr>
</tbody>
</table>

6.2.4 Data Complexity

Data Complexity/Average Data Complexity of a class indicates the functional work it must perform, and consists of data items that are input to or output from other
classes. In this work, Data Complexity is the Number of Parameters (I/O variables) passed to the class and parameters that the class must return to other classes. From Table 5.4, the univariate analysis of Defect Density shows that Average Data Complexity is not significant as an estimator (p-value = 0.774), with correlation coefficient, $r_s = 0.03$. Prior work by Troster [Troster, 1992] also discovered that Average Data Complexity is not a good predictor for Defect Density ($r = 0.05$, p-value = 0.06).

![Figure 6.6 - Scatter Plot of Average Data Complexity Clusters](image)

Figure 6.6 shows that there are three clusters which explain the variations in ADC. Three clusters were chosen to represent three types of complexity, i.e., low, medium and high. In the figure, the rightmost cluster is referred as Cluster 1, next to it is Cluster 2 and the leftmost cluster is Cluster 3. More than 90% of the systems belong to Cluster 2 and 3, while about 7% belong to Cluster 1. The cluster mean values are described in Table 6.6. The ASC for systems in Cluster 3 is between 1 and 6, while for Cluster 2 is between 6 and 9 while Cluster 3 is between 9 to 14.

Whilst Average Data Complexity is not a good estimator for Defect Density, it proves to be a good predictor for Average Procedural Complexity and Class Size. The Average Data Complexity metric, which is available during detailed design is useful to estimate the number of decisions (Average Procedural Complexity) to
be implemented during the coding phase. Warnier [Warnier, 1976] suggested that the decision structure of a module depends mostly on the quantity and structure of the data, in other words, each data item translates into one or more decisions. Card and Glass [Card and Glass, 1990] have demonstrated that CG Average Data Complexity using the Equation 2.7 in Section 2.6.1.3, is a good estimator of Average Procedure Complexity and module size.

The results obtained in this work discovered that the suggestion that Average Procedural Complexity is strongly influenced by Average Data Complexity also holds in OO systems. The Spearman correlation coefficient for these variables is \( r_s = 0.77 \), and p-value < 0.001. The relationship is illustrated in Figure 6.7. The CG Average Data Complexity is less of a predictor for Average Procedural Complexity (\( r_s = 0.53 \), p-value < 0.001). The linear equation for the effect of Data Complexity on Procedural Complexity is:

\[
\text{Procedural Complexity} = 0.68 + 2.48 \times \text{Data Complexity} \quad (6.2)
\]

Equation 6.2 explains that for each unit of Data Complexity for a class (on average), 2 to 3 decisions must be made in the class, and even when there are no parameters there is (on average) 2 out of 3 classes have 1 decisions to begin with. Card and Glass [Card and Glass, 1990] found the fit for RATFOR systems to be Procedural Complexity = 7.7 + 1.3 Data Complexity. In comparison, the number of decisions per unit of Data Complexity is a bit higher than the Card and Glass result and the base decisions without the Data Complexity also differ, where they found that there are 7.7 decisions not related to data function, while the result of this work is approximately 1 decision. The result of this work makes more intuitive sense in terms of the number of base decisions and also with regard to the decision increase per parameter.

A designed class will eventually be implemented in code. It is extremely useful to be able to estimate the size of the class based on early design metrics. Average Data Complexity proves to be a good estimator of Class Size (CS) (\( r_s = 0.71 \), and very highly significant p-value (p-value < 0.001). The CG Average Data Complexity

<table>
<thead>
<tr>
<th>Cluster</th>
<th>n</th>
<th>Mean ADC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>17</td>
<td>11.31</td>
</tr>
<tr>
<td>Cluster 2</td>
<td>45</td>
<td>7.46</td>
</tr>
<tr>
<td>Cluster 3</td>
<td>42</td>
<td>4.95</td>
</tr>
</tbody>
</table>
Chapter 6: Results Discussion

Figure 6.7 – Average Data Complexity vs Average Procedural Complexity

Figure 6.8 – Average Data Complexity vs Class Size
6.2 Results Discussion

is less of a predictor for Class Size \( (r_s = 0.44, \text{ p-value } < 0.001) \). The scatter plot demonstrating this relationship is shown in Figure 6.8. The linear equation for these variables is:

\[
\text{Class Size} = 0.49 + 8.66 \times \text{Data Complexity} 
\]  \hspace{1cm} (6.3)

From Equation 6.3, it it clear that each unit increase of Data Complexity increases Class Size by approximately 9 lines of code (LOC). Also, even if there are no parameters, every second class will contain about 1 line of code on average. Prior work by Basili et al. [Basili et al., 1983] also exhibit similar findings, where they found lines of code to be highly correlated with the number of I/O variables \( (r = 0.79) \).

When brought together, Equations 6.2 and 6.3 make sense in terms of increasing Data Complexity (DC). For example, if a class receives one parameter, then the Procedural Complexity (PC) will be around 3 and the number of LOC will be around 10, or around 3 LOC per decision. If DC = 2, then PC will be around 6 and CS will be around 19, still about 3 LOC per decision.

6.2.5 System Complexity

Usually, systems that exhibit high complexity are assumed to have higher possibility of having more errors than a simple one. But, high complexity does not always mean higher number of errors. In this work, I attempt to find the correlation between system design complexity with Defect Density after system delivery, i.e. post-delivery defects. Card and Glass [Card and Glass, 1990] did a similar study, but they focused on the pre-delivery defects.

From the analysis of 104 open-source systems, the results show that Average System Complexity as the combination of Average Data Complexity and Average Structural Complexity is not a good predictor for Defect Density with the correlation coefficient, \( r_s = -0.17 \) and p-value = 0.092. This result contrasts with the result of Card and Glass research where they found that CG Average System Complexity effectively predicts Defect Density, with correlation coefficient, \( r = 0.83 \) and p-value = 0.02. They also found that each increase of one unit of complexity increases the pre-delivery Defect Density by 0.4.

The main differences in the results of Card and Glass's work and this research are due to the differences in the type of systems being studied and the different phases at which defects were collected. The similarities centre around the types of measure being used to predict/estimate either defects volume and other
Chapter 6: Results Discussion

Figure 6.9 – Scatter Plot of Average System Complexity Clusters

Figure 6.10 – Average System Complexity vs Defect Density by Categories
6.2 Results Discussion

relevant software attributes such as length source code length and the number of decisions to be coded. The combined differences and similarities required several adjustments to be made on the calculation of Average Structural Complexity (ASC) and Average Data Complexity (ADC) as discussed in Section 3.1.1 and 5.4.8. From the discussion in Section 5.4.8, when ASC and ADC are calculated using the original formula proposed by Card and Glass, Average System Complexity (ASC + ADC) is significant in estimating Defect Density with \( r = -0.42 \) and p-value < 0.001. However, due to several reasons described in Section 3.3 and 5.5, the formula for ASC and ADC needs to be adjusted to suit the OO environment.

In Troster's study [Troster, 1992], he found that Average System Complexity is not significant for predicting pre-delivery Defect Density with correlation coefficient, \( r = -0.05 \) and low significant level, p-value = 0.09.

Figure 6.9 shows that there are three clusters which explain the variations in ASysC. The method of choosing the number of clusters is similar to other variables. From the figure, more than 90% of the systems belong to Cluster 2 and 3, while about 7% belong to Cluster 1. The cluster mean values are described in Table 6.7. The ASC for systems in Cluster 3 range from 1 to 6, while for Cluster 2, it ranges from 6 to 9 while Cluster 3 ranges from 9 to 14.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>n</th>
<th>Mean ASysC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>2</td>
<td>24.73</td>
</tr>
<tr>
<td>Cluster 2</td>
<td>12</td>
<td>17.82</td>
</tr>
<tr>
<td>Cluster 3</td>
<td>90</td>
<td>10.60</td>
</tr>
</tbody>
</table>

6.2.6 Ratio of System and Procedural Complexity

The Structural and Data Complexity can be obtained during detailed design, and later in implementation phase, an algorithm will introduce another complexity known as Procedural Complexity/Average Procedural Complexity (number of decisions). This complexity is measured using McCabe's Cyclomatic Complexity [McCabe, 1976], and Card and Glass have shown in their work that CG Average Data Complexity can effectively predict Procedural Complexity [Card and Glass, 1990].

As mentioned in Section 6.2.4, a similar discovery was found, where Data Complexity is very highly significant as an estimator for Procedural Complexity.
Thus, during detailed design time, Data Complexity can already be identified and can be used to predict Procedural Complexity. The prediction equation is given by Equation 6.2.

![Defect Density vs System/Procedural Complexity](image)

**Figure 6.11** – Defect Density vs System/Procedural Complexity

The System Complexity measure explains the complexity of data and the communication of the module/class with other modules/classes in the whole system, whereas Procedural Complexity explains the number of decisions that need to be implemented in the modules/classes. This brings us to a question: “Is there a correlation between System Complexity and Procedural Complexity that will help minimize defects in the final system?”

An experiment was conducted by a Masters student from the same research group to investigate the same concept of System Complexity based on multiple versions of open-source systems [Shukla and Boughton, 2009]. He studied 5 systems with multiple versions by applying the same measure of Data Complexity and Structural Complexity to arrive at System Complexity, and also looked at Procedural Complexity to understand the effects of these metrics on multiple versions of each system. The systems were thoroughly examined by analyzing complexity versus Defect Density for sequenced versions. When the ratio of System Complexity and Procedural Complexity was applied to the 5 systems, it was found that the ratio was either 1 or tending to 1. Any relationship to this trend with
6.2 Results Discussion

Defect Density was inconclusive.

The ratio of System Complexity and Procedural Complexity was also applied to the dataset used in this research and it was discovered that as the ratio moves towards 1, Defect Density decreases significantly, as illustrated in the plot in Figure 6.11. The ratio of 1 means that there is an equal balance between the System Complexity and Procedural Complexity. The balance represents the ideal mapping of detailed design metrics, System Complexity with Procedural Complexity which is available during the implementation phase. If the developers follow the detailed design closely during the implementation, then the ratio of System Complexity and Procedural Complexity will be 1 or close to 1.

The plot in 6.11 shows a decrease of Defect Density as the ratio of System Complexity and Procedural Complexity approaches 1. The univariate analysis of System Complexity and Procedural Complexity ratio and Defect Density shows that the ratio does influence Defect Density (p-value = 0.052) and $r = -0.19$, thus supporting hypothesis H4.

The relationship between Class Size with the complexity ratio above is depicted in 6.12. It seems that there is a steady decrease of Class Size as the ratio tends to 1, where systems with large Class Size tend to have lower ratio of System/Procedural
Complexity. The ratio is very highly significant in estimating Class Size ($r = -0.56$, p-value $< 0.001$). This relationship means that the Class Size during coding can be estimator earlier during the design phase using this ratio.

The plot in 6.13 suggests that systems with large Number of Classes tend to have a greater probability of smaller Class Size compared to those with lower Numbers of Classes. This, in conjunction with the tendencies shown in 6.12 suggests that systems with too little decomposition have greater propensity for higher Defect Density. However, as the system grows with time both size and defects ratio tend toward an optimum.

For the purpose of checking this suggestion/hypothesis, I selected 10 systems having a large gap between the version I'm currently investigating and the penultimate (latest - 1) version. The shifts in the complexity ratio and Defect Density of earlier versions (Version i) to later versions (Version n) of the selected systems are shown in 6.14. The systems are: DataCrow (1), Sweet Home 3D (2), Med's Movie Manager (3), Saxon (4), RunaWFE (5), HtmlUnit (6), JabRef (7), SchemaSpy (8), JFreeChart (9) and ehcache (10).

Out of ten systems, only four exhibit the anticipated movement of increasing ratio and reducing Defect Density. They are points 7, 2, 8, and 3 representing...
6.3 Findings in Relation to the Research Questions

Figure 6.14 – System/Procedural Ratio and Defect Density Shift for Different Versions

Jabref, Sweet Home 3D (SH3D), Schemaspy and Med’s Movie Manager (Med’s), respectively. Only system 9 shows slight reduction in complexity ratio and big reduction in Defect Density while another one, system 4, shows a small reduction in both complexity ratio and Defect Density. The most interesting trend is shown by systems 1 and 5 both of which exhibit significant decrease in ratio but slight change in Defect Density. The results of this check are inconclusive, and require a more significant research effort to resolve. However, some questions that arise from this check are going to be investigated in future research. The questions are highlighted in Chapter 7.

6.3 Findings in Relation to the Research Questions

This section discusses the findings that are useful in answering the research questions that are presented in Sections 1.3 and 3.5.1.
6.3.1 System Complexity and Defect Correlation

Question 1: Is there a general correlation between System (Data + Structural) design Complexity and various types of defects, and thus, can the level of defects be predicted early enough to undertake strategies to minimize them in the final product?

This research question brings about the formulation of H1: Increasing values of the Average System Complexity (Average Structural Complexity plus Average Data Complexity) correlate with increasing post-delivery defect density.

To test this hypothesis, the univariate analyses of Defect Density with Average Data Complexity, Average Structural Complexity, Average System Complexity, Average Procedural Complexity and several other variables have been conducted and the results are summarized in Table 5.6 in Section 5.4.3.2. The results from the univariate analysis show that only Average Structural Complexity (p-value = 0.001) is highly significant. The Average System Complexity, however, is not significant (p-value = 0.09). Hence, when the Average System Complexity is used as a combination of Average Data Complexity and Average Structural Complexity, it is not significant in predicting Defect Density. On the other hand, when the Average System Complexity is examined separately, the results show that Average Structural Complexity is highly significant (p-value = 0.001) as an estimator for Defect Density while Average Data Complexity is not significant (p-value = 0.774).

The second part of the question regarding early defect prediction will be answered in Section 6.3.3.

6.3.2 Defect Estimation

Question 2: Between structural, procedural and data complexity, which has the most influence and/or is the most appropriate for predicting system defects?

To answer this question, the results of the univariate analysis in Section 5.4.3.2 are referred. The results in Table 5.6 show that Average Structural Complexity is significant as an estimator of Defect Density (p-value = 0.001) whereas Average Data Complexity and Average Procedural Complexity are not significant, with p-value = 0.774 and p-value = 0.864, respectively. Thus, Average Structural Complexity has the most influence on Defect Density compared to the other two metrics. Average Structural Complexity has a negative correlation with Defect Density ($r_s = -0.33$), which means as Average Structural Complexity increases, Defect Density decreases. This result is further analyzed to answer Question 3 and is explained in the next section.
6.3 Findings in Relation to the Research Questions

6.3.3 Strategies for Minimizing Defects

Question 3: Are there particular strategies for minimizing defects in open-source software? The results used to answer Questions 1 and 2 are further analyzed to answer this question. Hypotheses H2, H3 and H4 are useful to answer this question and they are:

H2: Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Average Procedural Complexity (Cyclomatic Complexity), as in some non-object-oriented systems.

H3: Increasing values of the Average Data Complexity in object-oriented systems correlate with increasing Average Class Size, as in some non-object-oriented systems.

H4: There is an optimal ratio of System Complexity and Procedural Complexity that will help minimize defects in the final system.

Univariate analysis between Average Data Complexity and Average Procedural Complexity was conducted to test hypothesis H2. The result of the analysis in Section 5.4.3 shows that Average Data Complexity has a strong correlation with Average Procedural Complexity \( r_s = 0.77 \) and also very highly significant as an estimator for Average Procedural Complexity (p-value < 0.001), thus, supports hypothesis H2.

Similarly, hypothesis H3 was tested using the univariate analysis between Average Data Complexity and Class Size. The results reported in Section 5.4.3 show that there's a strong correlation between these variable \( r_s = 0.71 \) and Average Data Complexity is very highly significant in estimating Class Size (p-value < 0.001). These results support hypothesis H3.

Further analysis was carried out by looking at the ratio of Average System Complexity and Average Procedural Complexity. The Average System Complexity can be obtained during the detailed design phase while Average Procedural Complexity is available during the implementation phase. The ratio of 1 represents the ideal mappings of detailed design of a system to the implementation. The results in Figure 6.12 in Section 6.2.6 shows that as the ratio of the systems move towards 1, Defect Density decreases. The univariate analysis between Defect Density and the ratio of Average System Complexity and Average Procedural Complexity shows that the ratio is significant as a predictor for Defect Density (p-value = 0.052). Hence, by aiming for the ratio Average System Complexity and Average Procedural Complexity closer to 1, Defect Density for the system can be reduced and hypothesis H4 is supported.
From the result for hypothesis H2, Average Data Complexity is available during the detailed design phase, and is useful in estimating Average Procedural Complexity (Cyclomatic Complexity) in the implementation phase. Therefore, during detailed design phase, Average System Complexity is known (Average Data Complexity + Average Structural Complexity) and Average Procedural Complexity can be predicted using Average Data Complexity (the estimation model is shown by Equation 6.2 in Section 6.2.4). By using this ratio, defects can be estimated early in the development process so that necessary actions can be made to minimize it in the final product. This answers the second part of Question 1 and at the same time, answers Question 3.

### 6.4 Limitations

The scope of this research was defined by the research objectives presented in Section 1.2. Its main objective is to investigate whether there is a general correlation between system design complexity and post-delivery defects, by studying object-oriented measures relating to data, structural and procedural complexity, and comparing them with post-delivery defects. Another aim is to determine whether during the detailed design phase, measured Data Complexity can predict measured Procedural Complexity and Class Size for the implemented system.

Based on the abovementioned objectives, there are several limitations to this research:

1. The software/systems included in this research are from a snapshot of each system.

2. Metrics being investigated are limited to the main metrics used in the prior work of Card and Glass, for example, CBO is used to represent Structural Complexity, the Number of Parameters metric is used for Data Complexity and Cyclomatic Complexity represents Procedural Complexity. In addition, Version Downloads is used to represent usage metrics.

3. The systems selected are written completely in Java, the systems that are written in other programming languages are excluded from this research.

4. The open-source systems included in this research are limited to four main groups, as described in Section 4.2.

5. Some parts of this research are conducted manually, for instance, the selection of systems to be included in this research.
6.5 Potential Benefits to Industry

One of the contributions of this research is that organizations constructing systems in Java can apply the results.

More collaboration between research and the software industry is extremely important to achieve the goals of more studies with relevance and better transfer of research results. A more efficient transfer of research knowledge can be obtained by involving industry in the studies and rendering studies more credible by conducting them in contexts similar to those in industry [Sjøberg et al., 2007].

On a similar note, data used in this work is obtained from an industrial setting, e.g., open-source software already in production, therefore the results discovered in this research can be applied at least to that industrial perspective. Furthermore, the results produced by this research are shared with the project teams, with initial results already reported to the developers during the survey to collect defect data as described in Chapter 4.

During the collection of defect data, I have initiated contact with the developers of the systems, and they have expressed interest in my work and would like to see the results of the analysis to improve the quality of their product. Therefore, the next step is to report the findings to the respective project teams, which could also act as a "health check" for their systems. This relationship between research and industry is extremely important to ensure that more high-quality products will enter the market, either of a proprietary or open-source nature.

In the open-source development community, the number of user downloads represents user interest in the software itself, therefore in order to move towards a customer-driven development culture, each project development team should aim to produce software of good quality, in this sense, one with minimal defects. Imagine if every download is transformed into dollars and cents, highly reliable software will entice more users to use it and therefore, can be turned into profit to the developers themselves. From the open-source community point of view, they can still benefit from the user downloads/interest by possible donation to keep the project going. In other words, if users would like to continue using the software, they could contribute through donations to fund the development project so that it will keep on running and continue to benefit the overall user community.
6.6 Challenges and Lessons Learnt

Over the course of this research, I have faced many difficulties and challenges, such as:

- The selection process of systems to be included in this research required significant effort. Based on the selection criteria (given in Section 3.5.3 and 4.2), the most suitable systems have been selected out of hundreds of thousands hosted on SourceForge.

- During the survey to investigate the defect handling mechanism in open-source projects, the number of responses received were lower than the target. However, the feedback from the respondents was useful for the analysis.

- Using a combination of multiple measurement tools is fraught with the danger of providing inconsistent results. Therefore, a result validation process needs to be done and that requires significant time and effort.

- Lack of expertise on several areas like script writing and statistical analysis is also a challenge.

Besides all the challenges described above, several modifications are made on the original formula used by Card and Glass to suit the characteristics of object-oriented systems within current, typical open source environments, such as:

1. Use CBO instead of Fan-out to measure Structural Complexity or coupling in object-oriented systems.

2. For Data Complexity, the Number of Parameters (I/O variables) metric is used, instead of dividing it by Fan-out, to ensure the encapsulation concept for the object-oriented paradigm is protected.

3. Post-delivery defects are used instead of pre-delivery defects since in open-source systems, pre-delivery defect reports are generally not available.

4. Use of open-source systems to apply Card and Glass model as opposed to proprietary systems.

Throughout this research work, I have learned several valuable lessons, such as:

1. Different metric extraction tools (at least the ones used in this research) may calculate metrics differently and thus, need to be checked and validated against the original definitions of the metrics.
6.7 Threats to Validity

2. Conducting surveys requires careful planning since the number of responses might be lower than expected, typically because responders may have other priorities, or even lack of interest.

3. Data analysis requires a sound knowledge in statistics, since it is important to use the most appropriate techniques to analyze the data in order to arrive at the correct conclusion.

4. A replication of other research sometimes requires several modifications to suit a new environment or context.

6.7 Threats to Validity

The degree of credibility of any research depends on the validity of how conclusions are drawn. There are two classes of evaluation criteria, as defined by Campbell and Stanley [Campbell and Stanley, 1963], internal validity and external validity. Internal validity defines the degree of confidence in a cause-effect relationship between factors of interest and the observed results, whereas external validity defines the extent to which the conclusions from the experimental context can be generalized to the context specified in the research hypothesis. Another class of evaluation criteria, construct validity, was added by Judd et al. [Judd et al., 1991], which defines the extent to which the variables successfully measure the theoretical constructs in the hypotheses.

The threats to validity of this work are outlined below:

6.7.1 Construct Validity

The design metrics used are reverse engineered from the completed code for the purpose of research, there are possibilities that the post-implementation designs obtained had changed from the original, pre-implementation design, assuming one ever existed. If measurement is conducted before implementation starts, different measurement data could have been obtained because of the uncertainty inherent to early design information. This could have produced different results in the statistical analyses.

6.7.2 Internal Validity

The analysis results in this research are correlational in nature, and do not provide causal relationships. I have shown the correlations between several
independent variables with Defect Density using statistical methods and can only generate general conclusions based on this empirical evidence. However, the general conclusions lead to many new research questions which are highlighted in Chapter 7.

The metrics investigated are generally limited to the ones proposed by Card and Glass because the objective of this work is mainly to confirm the findings of Card and Glass, by applying their concepts to a different environment.

6.7.3 External Validity

The factors that may restrict the generalizability of the results of this work are:

1. The systems chosen in this study are just based on one version (the latest at the data collection time), out of multiple version systems. Some of the systems evolve rapidly and the size, complexity and defects may change from version to version.

2. The defects used in this study are post-delivery defects, using pre-delivery may yield different results.

3. Defect reports considered in this study are the ones reported within 6 months after the version being released. However, there is a possibility that defects will be continuously reported by users for the version after that period. In that situation, the number of actual defects for the considered version could be higher than reported in this study.

4. The metrics used in this study are extracted using multiple tools. Using other tools may lead to apparently different results.

5. The use of open-source projects may introduce threat to validity.
Chapter 7

Conclusions and Future Work

There are thousands of ways to mess up or damage a software project, and only a few ways to do them well.

Capers Jones, Applied Software Measurement

Contents

7.1 Conclusion ........................................... 186
7.2 Contribution of Results to Software Engineering Industry . 188
7.3 Research Questions from the Results of This Research . . . 190
7.4 Future Work ........................................... 190
  7.4.1 Automate work ................................. 190
  7.4.2 Include multiple versions ...................... 190
  7.4.3 Explore other metrics .......................... 191
  7.4.4 Study systems written in other languages .... 191
  7.4.5 Use function points as size measure ........ 191
  7.4.6 Investigate defects reported during testing .... 191
  7.4.7 Prediction using design documents ........... 191
Chapter 7: Conclusions and Future Work

7.1 Conclusion

This research is based on Card and Glass's work, who introduced a System Complexity Model as a combination of Structural and Data Complexity. The background of their work has been described in Section 2.6.1.3.

In essence, this research is conducted with the objective to answer the research questions and test the hypotheses given in Chapter 1 and 3. Findings in relation to the research questions have been discussed in Section 6.3. The research hypotheses have been tested using the Spearman's rank correlation coefficient and the results show that the hypothesis that increasing values of Average System Complexity correlate with increasing post-delivery defects (H1) is not supported ($r = -0.17$, p-value = 0.092). Likewise, Class Size is not significant as an estimator of Defect Density ($r = -0.02$, p-value = 0.861).

Nevertheless, hypothesis H2 is supported where Average Data Complexity is a good estimator for Average Procedural Complexity ($r = 0.77$, p-value < 0.001). Similarly, Average Data Complexity is a good estimator for Class Size ($r = 0.71$, p-value < 0.001), thus supporting hypothesis H3.

In this research, the Average Structural Complexity and Average Data Complexity are initially calculated using the formula introduced by Card and Glass as given in Section 2.6.1.3. The results show that CG Average System Complexity is very highly significant in estimating Defect Density (p-value < 0.001), as demonstrated in Section 5.5.

However, statistical significance alone does not mean that the construct of the model is correct. When the CG Average Structural Complexity and CG Average Data Complexity are examined separately, it shows that the CG Average System Complexity is dominated by CG Average Structural Complexity as shown by the data in Table 5.11 and 5.12 in Section 5.5. In addition, CG Average Structural Complexity includes only one type of dependency relevant to object-oriented (OO) systems, Fan-out, thus excluding the important dependency of 'inheritance'. Therefore, this research proposes a new formula to calculate Average Structural Complexity as given by Equation 3.1 in Section 3.3.

The formula for CG Average Data Complexity is also adjusted to suit the OO environment, as explained in Section 3.3. The proposed new formula for Average Data Complexity as given by Equation 3.2 in Section 3.3 takes into consideration the concept of "encapsulation" in OO languages.

The results reported in this thesis are based on the proposed new formula/model for Data, Structural, and System Complexity. The proposed Average
7.1 Conclusion

Structural Complexity is highly significant as an estimator for post-delivery Defect Density ($r_s = -0.33$, p-value = 0.001). Whilst the proposed Average Data Complexity is not significant in correlation to post-delivery Defect Density compared to CG Average Data Complexity, the opposite is true for correlation with Average Procedural Complexity ($r_s = 0.77$, p-value < 0.001) and Class Size ($r_s = 0.71$, p-value < 0.001) compared respectively with CG Average Data Complexity. That is, CG Average Data Complexity is less effective as an estimator for Average Procedural Complexity ($r_s = 0.53$, p-value < 0.001) and Class Size ($r_s = 0.44$, p-value = 0.002).

One of the key findings of this research is that Average System Complexity is more useful when used together with Average Procedural Complexity. Average System Complexity is available during the detailed design stage of development whereas Average Procedural Complexity is available after the implementation phase. Under ideal circumstances, the detailed design will be implemented as code without many changes. The detailed design should be inclusive of all data attributes, methods and incorporates state handling [Shlaer and Mellor, 1992] such that the ratio of Average System Complexity to Average Procedural Complexity will perhaps be ideally balanced. This research suggests that the ideal ratio of System Complexity and Procedural Complexity should be 1, in order to maintain the consistency between design and implementation.

The complexity ratio above can be used to estimate aspects of post-delivery Defect Density as shown in Figure 6.13 in Section 6.2.6. The figure shows that Defect Density decreases as the complexity ratio approaches 1. The ratio is significant as an estimator of Defect Density ($r_s = -0.19$, p-value = 0.052), this supporting hypothesis H4 that there is a relationship between System Complexity and Procedural Complexity that will help minimize defects in the final system. Figure 6.13 also shows that for systems where the complexity ratio closer to 1, there is a lower probability of high Defect Density.

From Figure 6.13 and 6.14, it can be seen that the majority of open-source systems (OSS) have a low complexity ratio (< 0.6), this might suggest that some OSS start with the implementation then create the design. Determining the accuracy of this latter statement should be the subject of future research.

Figure 6.14 shows that Class Size decreases as the complexity ratio approaches 1. For systems that have a complexity ratio of < 0.5, it is more difficult to accurately estimate the Class Size since there is a wider range of class sizes compared to the systems which have a complexity ratio near 1. This suggests there is an ideal Class Size as the complexity ratio approaches 1.

Another key finding which has been highlighted in Section 6.2.2 that most
Chapter 7: Conclusions and Future Work

Post-delivery defects occur in less complex classes (CBO < 20) compared to more complex classes (CBO > 20). A similar finding is determined for Cyclomatic Complexity (Procedural Complexity) where defects mostly occur in less complex (CC < 50) classes compared to complex ones (CC > 50) as discussed in Section 6.2.3. These results suggest that developers may pay more attention to complex classes, thus reducing the possibility of defects occurring in them, and pay less attention to complex classes. This finding perhaps suggests that more attention be given to less complex classes, especially those that have important functionality.

The defects studied in Card and Glass work are reported during testing (pre-delivery). Testing the software product prior to delivery is a standard practice in the software industry. As with proprietary projects, open-source projects support various testing regimes used by development teams. Where higher numbers of post-delivery defects are reported by users, testing processes may not be so good. Hence, a usage metric such as the number of downloads for open-source systems maybe useful to indicate product maturity, especially if post-delivery Defects decrease with increasing downloads. If the ratio of defects to increasing downloads falls then it may be a sign that the testing regime is working or that the product has stabilized or is stabilizing.

7.2 Contribution of Results to Software Engineering Industry

Software engineering involves the process of developing, maintaining and managing high-quality software in a cost-effective and systematic way. Software engineering research explores real-world phenomena of software development activities and products and relates to the development of new, or modifications of existing technology [Sjöberg et al., 2007].

It is important for software engineering research to apply empirical methods of doing research in order to successfully evolve to a more mature scientifically based body of knowledge. Empirical science includes gathering information on the basis of systematic observation and experiment, rather than deductive logic or mathematics [Sjöberg et al., 2007].

This research fits well into the empirical software engineering perspective. It actually extends the work of Card and Glass by applying their model to another environment and context, i.e. an open-source and object-oriented paradigm using post-delivery defects compared to testing defects. Kan [Kan, 2003] stated that more research is needed to validate Card and Glass’s model and to yield more insights into the applicability of this model to the industry.
7.2 Contribution of Results to Software Engineering Industry

In the software engineering industry, the ability to predict the number of decisions and class size using detailed design metrics is useful for planning the development process. Developers can estimate and adjust the complexity of the final implementation so that it will not be out of control during the maintenance phase. The ratio of System Complexity and Procedural Complexity can be used as an indicator of how well the implementation flows from the detailed design. For example, if Procedural Complexity become too high, there is a possibility that it deviates from the original design and the developers should consider checking requirements and design documents.

In addition, developers can start measuring the System Complexity during the detailed design phase using suitable tools. They can use the Data Complexity to predict the number of decisions (Procedural Complexity) to be implemented in algorithms and code during the implementation phase, together with the number of executable statements (Class Size) needed to implement the algorithms and code. By applying these measurements, they will have better control over the complexity and size distribution of the system.

Even if a development project starts with coding instead of design, the developers can still check the complexity of their systems by running the source code through metric extraction tools. By doing this, they can identify which parts of the design are complex, and use the information to improve the system.

The results of this research show that defects mostly occur in less complex classes compared to the more complex ones, this could be attributed to a psychological factor that developers will assume that defects are more likely to occur in complex classes compared to simple ones. Hence, more attention is given to complex classes in terms of testing and defect removal, whilst the less complex classes are overlooked.

Moreover, the ability to estimate Class Size during the detailed design phase will give an advantage of estimating the effort in the implementation phase. This is useful in a software engineering perspective since monitoring of effort during system development is hampered by how many decisions and how much code needs to be developed.

More research is needed in order to better understand the relationship between complexity and defects (either pre-delivery or post-delivery). Whilst there is no silver bullet for eliminating defects, at least we can attempt to minimize them. It is hoped that the results of this research will generate more interest in studying the application of Card and Glass's System Complexity model and also the newly proposed model from this research to the software engineering industry at large.
7.3 Research Questions from the Results of This Research

From the results discovered in this research, more questions emerge and will be investigated in future research:

1. What is the common development practice in open-source projects, do they start with design before implementation, or vice versa?

2. Is there any correlation between the proposed system design complexity with defects discovered during testing?

3. As systems mature, will they exhibit greater balance of the ratio of System Complexity and Procedural Complexity?

4. Is using the ratio of big classes and small classes useful in predicting defects?

5. Is there any optimal class decomposition?

7.4 Future Work

The limitations highlighted in Section 6.4 and the questions raised in Section 7.3 can be further explored and extended to improve the existing findings and also to expand this research to another level. This section discusses potential enhancements regarding several issues that have not yet been investigated.

7.4.1 Automate work

Some parts of this research involved manual work. Therefore, to ease future replications, those processes should be automated by using parsers or scripts, or by modifying the existing tools to do the additional tasks. The next step is to develop a specific tool to automatically collect the metrics for the proposed System Complexity model.

7.4.2 Include multiple versions

The next step involves the validation of Card and Glass's model from the multiple of versions angle. This can be achieved by investigating several versions of the selected systems, and apply similar metrics to each version. It is interesting to see
the results for different versions and how the application of these metrics correlate with post-delivery Defect Density for each system.

7.4.3 Explore other metrics

Future research will explore additional metrics that could possibly contribute to the prediction of defect numbers, (either pre-delivery or post-delivery).

7.4.4 Study systems written in other languages

This work can be extended by studying systems written in other programming languages like C++, PHP and so on. It would also be interesting to compare the results between different languages in relation to various complexities and defect types.

7.4.5 Use function points as size measure

Besides, lines of code (LOC), function points is another measure of size for a system, and is independent of language. This work can be extended and perhaps generalized by using function points to replace the LOC.

7.4.6 Investigate defects reported during testing

The next step is to use the proposed model to investigate the relationship of the design complexity with defects reported during testing. It would be interesting to see if the results are different than those for post-delivery defects.

7.4.7 Prediction using design documents

Investigate the efficacy of predicting complexity and class size using detailed design documents instead of generating metrics from source code. This can be achieved by using systems which have complete design documents and make the documents available for various sorts of analysis, including prediction of defect numbers.
Appendix A

Research Data
### Table A.1 – Data for Systems in Games/Multimedia Category

<table>
<thead>
<tr>
<th>System Name</th>
<th>NC</th>
<th>ASC</th>
<th>ADC</th>
<th>ASysC</th>
<th>APC</th>
<th>SP</th>
<th>TD</th>
<th>DD</th>
<th>KLOC</th>
<th>CS</th>
<th>VD</th>
<th>DEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataCrow</td>
<td>584</td>
<td>7.08</td>
<td>2.88</td>
<td>9.96</td>
<td>10.05</td>
<td>0.99</td>
<td>12</td>
<td>0.50</td>
<td>23.83</td>
<td>39.87</td>
<td>4.12</td>
<td>1</td>
</tr>
<tr>
<td>Galleon</td>
<td>510</td>
<td>5.16</td>
<td>5.88</td>
<td>11.04</td>
<td>19.54</td>
<td>0.56</td>
<td>21</td>
<td>0.59</td>
<td>35.35</td>
<td>69.30</td>
<td>4.41</td>
<td>8</td>
</tr>
<tr>
<td>Jin</td>
<td>384</td>
<td>4.12</td>
<td>6.88</td>
<td>10.99</td>
<td>18.54</td>
<td>0.59</td>
<td>11</td>
<td>0.54</td>
<td>20.26</td>
<td>52.76</td>
<td>4.65</td>
<td>1</td>
</tr>
<tr>
<td>TvBrowser</td>
<td>734</td>
<td>5.45</td>
<td>7.10</td>
<td>12.55</td>
<td>17.18</td>
<td>0.73</td>
<td>111</td>
<td>2.44</td>
<td>45.43</td>
<td>61.90</td>
<td>5.19</td>
<td>16</td>
</tr>
<tr>
<td>ZK</td>
<td>544</td>
<td>4.19</td>
<td>6.26</td>
<td>10.44</td>
<td>15.50</td>
<td>0.67</td>
<td>47</td>
<td>2.23</td>
<td>21.08</td>
<td>38.74</td>
<td>3.81</td>
<td>18</td>
</tr>
<tr>
<td>Dozer</td>
<td>285</td>
<td>2.98</td>
<td>5.79</td>
<td>8.76</td>
<td>8.63</td>
<td>1.02</td>
<td>10</td>
<td>1.23</td>
<td>8.15</td>
<td>28.59</td>
<td>3.20</td>
<td>8</td>
</tr>
<tr>
<td>Mars</td>
<td>183</td>
<td>5.28</td>
<td>6.51</td>
<td>11.80</td>
<td>15.07</td>
<td>0.78</td>
<td>5</td>
<td>0.58</td>
<td>8.59</td>
<td>46.92</td>
<td>3.86</td>
<td>18</td>
</tr>
<tr>
<td>Phex</td>
<td>1111</td>
<td>5.18</td>
<td>5.71</td>
<td>10.89</td>
<td>14.03</td>
<td>0.78</td>
<td>27</td>
<td>0.50</td>
<td>54.12</td>
<td>48.72</td>
<td>5.05</td>
<td>9</td>
</tr>
<tr>
<td>Sweet Home 3D</td>
<td>73</td>
<td>3.34</td>
<td>4.70</td>
<td>8.04</td>
<td>13.08</td>
<td>0.61</td>
<td>14</td>
<td>4.25</td>
<td>3.30</td>
<td>45.16</td>
<td>3.59</td>
<td>9</td>
</tr>
<tr>
<td>aTunes</td>
<td>142</td>
<td>5.58</td>
<td>7.33</td>
<td>13.30</td>
<td>17.35</td>
<td>0.77</td>
<td>5</td>
<td>0.51</td>
<td>9.75</td>
<td>68.68</td>
<td>2.88</td>
<td>3</td>
</tr>
<tr>
<td>Art of Illusion</td>
<td>404</td>
<td>10.97</td>
<td>13.00</td>
<td>23.98</td>
<td>34.75</td>
<td>0.69</td>
<td>19</td>
<td>0.41</td>
<td>46.17</td>
<td>114.28</td>
<td>4.62</td>
<td>2</td>
</tr>
<tr>
<td>MeD's Movie Manager</td>
<td>98</td>
<td>4.91</td>
<td>10.91</td>
<td>15.82</td>
<td>40.67</td>
<td>0.39</td>
<td>11</td>
<td>0.73</td>
<td>15.04</td>
<td>153.46</td>
<td>4.13</td>
<td>9</td>
</tr>
<tr>
<td>MicroEmulator</td>
<td>275</td>
<td>4.00</td>
<td>7.32</td>
<td>11.32</td>
<td>13.31</td>
<td>0.85</td>
<td>5</td>
<td>0.44</td>
<td>11.32</td>
<td>41.19</td>
<td>4.16</td>
<td>4</td>
</tr>
<tr>
<td>Krut</td>
<td>33</td>
<td>1.12</td>
<td>5.55</td>
<td>6.67</td>
<td>22.09</td>
<td>0.30</td>
<td>2</td>
<td>0.68</td>
<td>2.95</td>
<td>89.42</td>
<td>5.22</td>
<td>1</td>
</tr>
<tr>
<td>FlickrBackup</td>
<td>73</td>
<td>2.86</td>
<td>7.85</td>
<td>10.71</td>
<td>19.26</td>
<td>0.56</td>
<td>6</td>
<td>1.16</td>
<td>5.16</td>
<td>70.71</td>
<td>4.81</td>
<td>3</td>
</tr>
<tr>
<td>Barbeque</td>
<td>101</td>
<td>4.26</td>
<td>5.05</td>
<td>9.31</td>
<td>18.54</td>
<td>0.73</td>
<td>3</td>
<td>0.58</td>
<td>5.18</td>
<td>51.31</td>
<td>5.01</td>
<td>1</td>
</tr>
<tr>
<td>JavaEmailServer</td>
<td>128</td>
<td>2.90</td>
<td>4.86</td>
<td>7.76</td>
<td>21.46</td>
<td>0.36</td>
<td>1</td>
<td>0.11</td>
<td>9.34</td>
<td>72.98</td>
<td>4.98</td>
<td>1</td>
</tr>
<tr>
<td>OpenCards</td>
<td>160</td>
<td>6.48</td>
<td>6.33</td>
<td>12.81</td>
<td>12.20</td>
<td>1.05</td>
<td>2</td>
<td>0.27</td>
<td>7.31</td>
<td>45.69</td>
<td>4.90</td>
<td>1</td>
</tr>
<tr>
<td>JavaHMO</td>
<td>243</td>
<td>3.53</td>
<td>6.30</td>
<td>9.84</td>
<td>18.51</td>
<td>0.53</td>
<td>3</td>
<td>0.18</td>
<td>16.38</td>
<td>67.42</td>
<td>4.94</td>
<td>11</td>
</tr>
<tr>
<td>AllianceP2P</td>
<td>259</td>
<td>3.63</td>
<td>8.50</td>
<td>12.13</td>
<td>19.18</td>
<td>0.63</td>
<td>8</td>
<td>0.57</td>
<td>14.04</td>
<td>54.22</td>
<td>5.33</td>
<td>4</td>
</tr>
<tr>
<td>DrawSWF</td>
<td>310</td>
<td>3.76</td>
<td>8.35</td>
<td>12.11</td>
<td>17.21</td>
<td>0.70</td>
<td>7</td>
<td>0.43</td>
<td>16.29</td>
<td>52.55</td>
<td>5.34</td>
<td>3</td>
</tr>
<tr>
<td>Robocode</td>
<td>494</td>
<td>11.81</td>
<td>6.39</td>
<td>18.20</td>
<td>17.52</td>
<td>1.04</td>
<td>7</td>
<td>0.21</td>
<td>33.56</td>
<td>67.94</td>
<td>4.52</td>
<td>1</td>
</tr>
<tr>
<td>Robocode</td>
<td>692</td>
<td>3.30</td>
<td>5.13</td>
<td>8.43</td>
<td>14.31</td>
<td>0.59</td>
<td>116</td>
<td>3.61</td>
<td>32.15</td>
<td>46.45</td>
<td>5.71</td>
<td>6</td>
</tr>
<tr>
<td>TripleA</td>
<td>841</td>
<td>6.54</td>
<td>7.45</td>
<td>13.98</td>
<td>18.38</td>
<td>0.76</td>
<td>22</td>
<td>0.43</td>
<td>50.62</td>
<td>60.19</td>
<td>5.70</td>
<td>3</td>
</tr>
<tr>
<td>Ozone</td>
<td>1148</td>
<td>3.09</td>
<td>6.66</td>
<td>9.75</td>
<td>13.10</td>
<td>0.74</td>
<td>2</td>
<td>0.04</td>
<td>46.92</td>
<td>40.87</td>
<td>4.79</td>
<td>9</td>
</tr>
<tr>
<td>VASSALEngine</td>
<td>1233</td>
<td>5.60</td>
<td>7.16</td>
<td>12.76</td>
<td>16.22</td>
<td>0.79</td>
<td>125</td>
<td>1.99</td>
<td>62.93</td>
<td>51.04</td>
<td>4.93</td>
<td>17</td>
</tr>
<tr>
<td>System Name</td>
<td>NC</td>
<td>ASC</td>
<td>ADC</td>
<td>ASysC</td>
<td>APC</td>
<td>SP</td>
<td>TD</td>
<td>DD</td>
<td>KLOC</td>
<td>CS</td>
<td>VD</td>
<td>DEV</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>------</td>
<td>-----</td>
<td>----</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>HTMLParser</td>
<td>257</td>
<td>6.46</td>
<td>7.96</td>
<td>14.42</td>
<td>20.00</td>
<td>0.72</td>
<td>16</td>
<td>0.83</td>
<td>19.20</td>
<td>74.71</td>
<td>5.08</td>
<td>4</td>
</tr>
<tr>
<td>Saxon</td>
<td>876</td>
<td>14.03</td>
<td>11.44</td>
<td>25.47</td>
<td>23.67</td>
<td>1.08</td>
<td>34</td>
<td>0.50</td>
<td>67.67</td>
<td>77.25</td>
<td>4.18</td>
<td>1</td>
</tr>
<tr>
<td>XUI</td>
<td>255</td>
<td>3.69</td>
<td>9.52</td>
<td>13.21</td>
<td>20.20</td>
<td>0.65</td>
<td>18</td>
<td>1.25</td>
<td>14.43</td>
<td>56.60</td>
<td>4.23</td>
<td>21</td>
</tr>
<tr>
<td>Azureus</td>
<td>2759</td>
<td>6.25</td>
<td>3.20</td>
<td>9.45</td>
<td>17.77</td>
<td>0.53</td>
<td>20</td>
<td>0.13</td>
<td>152.80</td>
<td>55.38</td>
<td>5.68</td>
<td>33</td>
</tr>
<tr>
<td>JPodder</td>
<td>250</td>
<td>1.12</td>
<td>5.40</td>
<td>6.52</td>
<td>13.16</td>
<td>0.50</td>
<td>12</td>
<td>0.99</td>
<td>12.10</td>
<td>48.40</td>
<td>4.91</td>
<td>6</td>
</tr>
<tr>
<td>RunaWFE</td>
<td>609</td>
<td>9.35</td>
<td>4.44</td>
<td>13.79</td>
<td>9.92</td>
<td>1.39</td>
<td>5</td>
<td>0.21</td>
<td>23.96</td>
<td>39.34</td>
<td>3.53</td>
<td>31</td>
</tr>
<tr>
<td>Jena</td>
<td>1753</td>
<td>7.18</td>
<td>6.69</td>
<td>13.87</td>
<td>15.69</td>
<td>0.88</td>
<td>19</td>
<td>0.23</td>
<td>81.91</td>
<td>46.73</td>
<td>4.41</td>
<td>14</td>
</tr>
<tr>
<td>HtmlUnit</td>
<td>328</td>
<td>5.23</td>
<td>7.57</td>
<td>12.80</td>
<td>13.32</td>
<td>0.96</td>
<td>8</td>
<td>0.55</td>
<td>14.50</td>
<td>44.22</td>
<td>3.85</td>
<td>8</td>
</tr>
<tr>
<td>Frost</td>
<td>847</td>
<td>3.66</td>
<td>5.79</td>
<td>9.45</td>
<td>15.55</td>
<td>0.61</td>
<td>5</td>
<td>0.11</td>
<td>45.67</td>
<td>53.92</td>
<td>3.62</td>
<td>6</td>
</tr>
<tr>
<td>iText</td>
<td>541</td>
<td>7.93</td>
<td>12.65</td>
<td>20.59</td>
<td>31.62</td>
<td>0.49</td>
<td>10</td>
<td>0.19</td>
<td>53.82</td>
<td>99.49</td>
<td>4.44</td>
<td>7</td>
</tr>
<tr>
<td>OpenXava</td>
<td>1067</td>
<td>4.48</td>
<td>8.80</td>
<td>13.28</td>
<td>19.07</td>
<td>0.70</td>
<td>1</td>
<td>0.02</td>
<td>59.24</td>
<td>55.52</td>
<td>3.59</td>
<td>14</td>
</tr>
<tr>
<td>Opentaps</td>
<td>1254</td>
<td>9.95</td>
<td>10.74</td>
<td>20.69</td>
<td>30.05</td>
<td>0.69</td>
<td>12</td>
<td>0.10</td>
<td>123.47</td>
<td>98.46</td>
<td>5.06</td>
<td>37</td>
</tr>
<tr>
<td>HSQL Database</td>
<td>322</td>
<td>5.99</td>
<td>12.49</td>
<td>18.48</td>
<td>34.20</td>
<td>0.54</td>
<td>177</td>
<td>4.54</td>
<td>39.00</td>
<td>121.10</td>
<td>5.89</td>
<td>35</td>
</tr>
<tr>
<td>struts-menu</td>
<td>43</td>
<td>2.14</td>
<td>7.21</td>
<td>9.35</td>
<td>14.67</td>
<td>0.64</td>
<td>11</td>
<td>5.50</td>
<td>2.00</td>
<td>46.49</td>
<td>4.93</td>
<td>3</td>
</tr>
<tr>
<td>JUnitEE</td>
<td>50</td>
<td>1.22</td>
<td>4.42</td>
<td>5.64</td>
<td>10.16</td>
<td>0.56</td>
<td>5</td>
<td>2.98</td>
<td>1.68</td>
<td>33.54</td>
<td>4.83</td>
<td>5</td>
</tr>
<tr>
<td>JWebpack</td>
<td>69</td>
<td>4.29</td>
<td>8.84</td>
<td>13.13</td>
<td>23.62</td>
<td>0.56</td>
<td>3</td>
<td>0.61</td>
<td>4.90</td>
<td>70.94</td>
<td>4.92</td>
<td>3</td>
</tr>
<tr>
<td>JUpload</td>
<td>84</td>
<td>3.69</td>
<td>10.40</td>
<td>14.10</td>
<td>23.89</td>
<td>0.59</td>
<td>6</td>
<td>0.91</td>
<td>6.56</td>
<td>78.07</td>
<td>4.96</td>
<td>4</td>
</tr>
<tr>
<td>Davmail</td>
<td>94</td>
<td>3.96</td>
<td>5.27</td>
<td>9.22</td>
<td>24.55</td>
<td>0.38</td>
<td>17</td>
<td>2.09</td>
<td>8.15</td>
<td>86.72</td>
<td>4.99</td>
<td>1</td>
</tr>
<tr>
<td>Jimm</td>
<td>96</td>
<td>8.72</td>
<td>10.89</td>
<td>19.60</td>
<td>38.25</td>
<td>0.51</td>
<td>2</td>
<td>0.16</td>
<td>12.44</td>
<td>129.61</td>
<td>6.68</td>
<td>4</td>
</tr>
<tr>
<td>EuroBudget</td>
<td>176</td>
<td>4.38</td>
<td>8.19</td>
<td>12.57</td>
<td>19.97</td>
<td>0.63</td>
<td>2</td>
<td>0.16</td>
<td>12.25</td>
<td>69.59</td>
<td>4.88</td>
<td>1</td>
</tr>
<tr>
<td>Paros</td>
<td>260</td>
<td>4.11</td>
<td>7.35</td>
<td>11.46</td>
<td>18.52</td>
<td>0.62</td>
<td>8</td>
<td>0.46</td>
<td>17.44</td>
<td>67.08</td>
<td>5.70</td>
<td>1</td>
</tr>
<tr>
<td>AzSMRC</td>
<td>309</td>
<td>3.74</td>
<td>5.78</td>
<td>9.52</td>
<td>24.85</td>
<td>0.38</td>
<td>27</td>
<td>1.02</td>
<td>26.35</td>
<td>85.26</td>
<td>4.83</td>
<td>5</td>
</tr>
<tr>
<td>XBrowser</td>
<td>336</td>
<td>2.82</td>
<td>3.86</td>
<td>6.68</td>
<td>11.99</td>
<td>0.56</td>
<td>7</td>
<td>0.59</td>
<td>11.80</td>
<td>35.11</td>
<td>4.96</td>
<td>2</td>
</tr>
<tr>
<td>mvnForum</td>
<td>781</td>
<td>6.77</td>
<td>10.82</td>
<td>17.60</td>
<td>23.86</td>
<td>0.74</td>
<td>1</td>
<td>0.01</td>
<td>67.11</td>
<td>85.93</td>
<td>5.26</td>
<td>33</td>
</tr>
<tr>
<td>Columba</td>
<td>1711</td>
<td>5.17</td>
<td>5.17</td>
<td>10.34</td>
<td>9.72</td>
<td>1.06</td>
<td>1</td>
<td>0.02</td>
<td>64.08</td>
<td>37.45</td>
<td>5.26</td>
<td>15</td>
</tr>
<tr>
<td>PJIRC</td>
<td>182</td>
<td>3.29</td>
<td>6.84</td>
<td>10.13</td>
<td>18.13</td>
<td>0.56</td>
<td>2</td>
<td>0.21</td>
<td>9.56</td>
<td>52.50</td>
<td>5.65</td>
<td>2</td>
</tr>
</tbody>
</table>
Table A.3 – Data for Systems in Office/Programming/Database Category

<table>
<thead>
<tr>
<th>System Name</th>
<th>NC</th>
<th>ASC</th>
<th>ADC</th>
<th>ASysC</th>
<th>APC</th>
<th>SP</th>
<th>TD</th>
<th>DD</th>
<th>KLOC</th>
<th>CS</th>
<th>VD</th>
<th>DEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>JabRef</td>
<td>574</td>
<td>4.07</td>
<td>5.35</td>
<td>9.41</td>
<td>14.02</td>
<td>0.67</td>
<td>117</td>
<td>4.31</td>
<td>27.13</td>
<td>47.26</td>
<td>4.95</td>
<td>36</td>
</tr>
<tr>
<td>jEdit</td>
<td>837</td>
<td>4.37</td>
<td>6.32</td>
<td>10.68</td>
<td>20.45</td>
<td>0.52</td>
<td>18</td>
<td>0.35</td>
<td>51.71</td>
<td>61.78</td>
<td>4.33</td>
<td>158</td>
</tr>
<tr>
<td>LaTeXDraw</td>
<td>189</td>
<td>7.66</td>
<td>12.35</td>
<td>20.02</td>
<td>42.46</td>
<td>0.47</td>
<td>36</td>
<td>1.14</td>
<td>31.54</td>
<td>166.89</td>
<td>2.89</td>
<td>1</td>
</tr>
<tr>
<td>FindBugs</td>
<td>1003</td>
<td>5.24</td>
<td>6.87</td>
<td>12.11</td>
<td>17.76</td>
<td>0.68</td>
<td>120</td>
<td>2.37</td>
<td>50.68</td>
<td>50.53</td>
<td>4.39</td>
<td>10</td>
</tr>
<tr>
<td>DrJava</td>
<td>809</td>
<td>3.61</td>
<td>4.94</td>
<td>8.56</td>
<td>16.17</td>
<td>0.53</td>
<td>15</td>
<td>0.34</td>
<td>44.72</td>
<td>55.27</td>
<td>3.39</td>
<td>35</td>
</tr>
<tr>
<td>SchemaSpy</td>
<td>45</td>
<td>4.13</td>
<td>8.51</td>
<td>12.64</td>
<td>21.53</td>
<td>0.59</td>
<td>11</td>
<td>3.15</td>
<td>3.48</td>
<td>77.49</td>
<td>3.95</td>
<td>1</td>
</tr>
<tr>
<td>WfMOpen</td>
<td>539</td>
<td>6.25</td>
<td>7.08</td>
<td>13.32</td>
<td>17.07</td>
<td>0.78</td>
<td>2</td>
<td>0.06</td>
<td>33.40</td>
<td>61.96</td>
<td>3.73</td>
<td>5</td>
</tr>
<tr>
<td>Eclipse Checkstyle</td>
<td>141</td>
<td>5.05</td>
<td>4.25</td>
<td>9.30</td>
<td>12.18</td>
<td>0.76</td>
<td>10</td>
<td>1.53</td>
<td>6.52</td>
<td>46.26</td>
<td>5.06</td>
<td>5</td>
</tr>
<tr>
<td>GuanttProject</td>
<td>429</td>
<td>3.21</td>
<td>4.97</td>
<td>8.18</td>
<td>11.12</td>
<td>0.74</td>
<td>9</td>
<td>0.56</td>
<td>16.19</td>
<td>37.75</td>
<td>4.68</td>
<td>6</td>
</tr>
<tr>
<td>JasperReports</td>
<td>1021</td>
<td>5.23</td>
<td>7.69</td>
<td>12.92</td>
<td>17.27</td>
<td>0.75</td>
<td>27</td>
<td>0.50</td>
<td>54.17</td>
<td>53.06</td>
<td>4.88</td>
<td>8</td>
</tr>
<tr>
<td>soapUI</td>
<td>781</td>
<td>3.97</td>
<td>4.51</td>
<td>8.48</td>
<td>10.98</td>
<td>0.77</td>
<td>111</td>
<td>3.63</td>
<td>30.56</td>
<td>39.13</td>
<td>5.11</td>
<td>12</td>
</tr>
<tr>
<td>jTDS</td>
<td>141</td>
<td>4.30</td>
<td>12.41</td>
<td>16.71</td>
<td>40.06</td>
<td>0.42</td>
<td>53</td>
<td>2.23</td>
<td>23.79</td>
<td>168.72</td>
<td>5.35</td>
<td>10</td>
</tr>
<tr>
<td>Squirrel SQL</td>
<td>1791</td>
<td>3.77</td>
<td>5.49</td>
<td>9.26</td>
<td>11.05</td>
<td>0.84</td>
<td>11</td>
<td>0.16</td>
<td>68.91</td>
<td>38.48</td>
<td>4.50</td>
<td>61</td>
</tr>
<tr>
<td>opencsv</td>
<td>32</td>
<td>2.06</td>
<td>13.69</td>
<td>15.75</td>
<td>23.97</td>
<td>0.66</td>
<td>9</td>
<td>3.29</td>
<td>2.74</td>
<td>85.53</td>
<td>5.00</td>
<td>3</td>
</tr>
<tr>
<td>JPOX</td>
<td>40</td>
<td>1.03</td>
<td>10.53</td>
<td>11.55</td>
<td>38.05</td>
<td>0.30</td>
<td>13</td>
<td>2.59</td>
<td>5.02</td>
<td>125.58</td>
<td>5.09</td>
<td>1</td>
</tr>
<tr>
<td>JOpenChart</td>
<td>49</td>
<td>3.84</td>
<td>9.24</td>
<td>13.08</td>
<td>15.41</td>
<td>0.85</td>
<td>5</td>
<td>2.14</td>
<td>2.34</td>
<td>47.73</td>
<td>5.00</td>
<td>1</td>
</tr>
<tr>
<td>StrutsTestCase</td>
<td>94</td>
<td>0.90</td>
<td>7.21</td>
<td>8.12</td>
<td>15.35</td>
<td>0.53</td>
<td>58</td>
<td>13.67</td>
<td>4.24</td>
<td>45.15</td>
<td>5.14</td>
<td>4</td>
</tr>
<tr>
<td>Jipe</td>
<td>100</td>
<td>2.61</td>
<td>7.36</td>
<td>9.97</td>
<td>23.69</td>
<td>0.42</td>
<td>1</td>
<td>0.13</td>
<td>7.97</td>
<td>79.72</td>
<td>4.76</td>
<td>6</td>
</tr>
<tr>
<td>Jacob-project</td>
<td>105</td>
<td>4.45</td>
<td>5.77</td>
<td>10.22</td>
<td>14.49</td>
<td>0.71</td>
<td>9</td>
<td>2.00</td>
<td>4.50</td>
<td>42.82</td>
<td>5.51</td>
<td>2</td>
</tr>
<tr>
<td>Jtidy</td>
<td>123</td>
<td>4.72</td>
<td>5.81</td>
<td>10.53</td>
<td>15.55</td>
<td>0.41</td>
<td>28</td>
<td>2.65</td>
<td>10.55</td>
<td>85.80</td>
<td>5.42</td>
<td>7</td>
</tr>
<tr>
<td>Barcode4J</td>
<td>207</td>
<td>3.79</td>
<td>5.17</td>
<td>8.96</td>
<td>11.72</td>
<td>0.76</td>
<td>16</td>
<td>1.79</td>
<td>8.92</td>
<td>43.11</td>
<td>4.81</td>
<td>4</td>
</tr>
<tr>
<td>Green</td>
<td>259</td>
<td>4.15</td>
<td>5.15</td>
<td>9.31</td>
<td>13.64</td>
<td>0.68</td>
<td>4</td>
<td>0.32</td>
<td>12.39</td>
<td>47.82</td>
<td>4.92</td>
<td>3</td>
</tr>
<tr>
<td>Jpivot</td>
<td>467</td>
<td>4.93</td>
<td>6.66</td>
<td>11.59</td>
<td>14.82</td>
<td>0.78</td>
<td>5</td>
<td>0.22</td>
<td>22.68</td>
<td>48.55</td>
<td>5.61</td>
<td>3</td>
</tr>
<tr>
<td>dcm4che</td>
<td>546</td>
<td>4.64</td>
<td>9.06</td>
<td>13.70</td>
<td>19.89</td>
<td>0.69</td>
<td>27</td>
<td>0.87</td>
<td>31.05</td>
<td>56.86</td>
<td>4.79</td>
<td>1</td>
</tr>
<tr>
<td>iReport</td>
<td>1535</td>
<td>4.75</td>
<td>7.86</td>
<td>12.61</td>
<td>18.15</td>
<td>0.69</td>
<td>31</td>
<td>0.29</td>
<td>106.92</td>
<td>69.65</td>
<td>6.37</td>
<td>14</td>
</tr>
<tr>
<td>JRefactory</td>
<td>2682</td>
<td>2.46</td>
<td>4.12</td>
<td>6.58</td>
<td>12.72</td>
<td>0.52</td>
<td>13</td>
<td>0.12</td>
<td>110.02</td>
<td>41.02</td>
<td>4.94</td>
<td>3</td>
</tr>
</tbody>
</table>
## Table A.4 – Data for Systems in Scientific/Engineering/Operating Systems Category

<table>
<thead>
<tr>
<th>System Name</th>
<th>NC</th>
<th>ASC</th>
<th>ADC</th>
<th>ASysC</th>
<th>APC</th>
<th>SP</th>
<th>TD</th>
<th>DD</th>
<th>KLOC</th>
<th>CS</th>
<th>VD</th>
<th>DEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freemind</td>
<td>580</td>
<td>2.62</td>
<td>7.99</td>
<td>10.61</td>
<td>20.42</td>
<td>0.52</td>
<td>96</td>
<td>2.50</td>
<td>38.37</td>
<td>66.16</td>
<td>6.50</td>
<td>8</td>
</tr>
<tr>
<td>Cewolf</td>
<td>108</td>
<td>2.50</td>
<td>4.50</td>
<td>7.00</td>
<td>5.98</td>
<td>1.17</td>
<td>20</td>
<td>8.52</td>
<td>2.35</td>
<td>21.73</td>
<td>4.67</td>
<td>5</td>
</tr>
<tr>
<td>GATE</td>
<td>947</td>
<td>6.18</td>
<td>5.67</td>
<td>11.85</td>
<td>18.37</td>
<td>0.65</td>
<td>3</td>
<td>0.05</td>
<td>55.86</td>
<td>58.98</td>
<td>3.56</td>
<td>26</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>750</td>
<td>6.09</td>
<td>9.89</td>
<td>15.98</td>
<td>20.26</td>
<td>0.79</td>
<td>50</td>
<td>0.86</td>
<td>58.22</td>
<td>77.62</td>
<td>5.13</td>
<td>9</td>
</tr>
<tr>
<td>Jmol</td>
<td>295</td>
<td>2.50</td>
<td>10.38</td>
<td>12.88</td>
<td>25.26</td>
<td>0.51</td>
<td>43</td>
<td>2.03</td>
<td>21.15</td>
<td>71.71</td>
<td>4.47</td>
<td>29</td>
</tr>
<tr>
<td>Xinco</td>
<td>71</td>
<td>3.41</td>
<td>10.07</td>
<td>13.48</td>
<td>28.75</td>
<td>0.47</td>
<td>8</td>
<td>0.97</td>
<td>8.25</td>
<td>116.14</td>
<td>4.21</td>
<td>2</td>
</tr>
<tr>
<td>Joone</td>
<td>1195</td>
<td>3.70</td>
<td>4.57</td>
<td>8.27</td>
<td>13.16</td>
<td>0.63</td>
<td>10</td>
<td>0.20</td>
<td>49.78</td>
<td>41.66</td>
<td>4.41</td>
<td>11</td>
</tr>
<tr>
<td>JUNG</td>
<td>652</td>
<td>4.22</td>
<td>5.25</td>
<td>9.46</td>
<td>10.28</td>
<td>0.92</td>
<td>16</td>
<td>0.66</td>
<td>24.25</td>
<td>37.19</td>
<td>4.24</td>
<td>9</td>
</tr>
<tr>
<td>ControlRemote</td>
<td>188</td>
<td>4.50</td>
<td>8.33</td>
<td>12.83</td>
<td>19.23</td>
<td>0.67</td>
<td>3</td>
<td>0.27</td>
<td>11.26</td>
<td>59.90</td>
<td>3.34</td>
<td>9</td>
</tr>
<tr>
<td>ehcache</td>
<td>111</td>
<td>3.68</td>
<td>6.90</td>
<td>10.58</td>
<td>25.70</td>
<td>0.67</td>
<td>11</td>
<td>2.07</td>
<td>5.33</td>
<td>47.98</td>
<td>4.17</td>
<td>4</td>
</tr>
<tr>
<td>JBoss</td>
<td>7453</td>
<td>4.08</td>
<td>5.07</td>
<td>9.15</td>
<td>11.66</td>
<td>0.78</td>
<td>37</td>
<td>0.13</td>
<td>274.17</td>
<td>36.79</td>
<td>5.24</td>
<td>83</td>
</tr>
<tr>
<td>YALE/RapidMiner</td>
<td>855</td>
<td>5.62</td>
<td>7.45</td>
<td>13.07</td>
<td>15.75</td>
<td>0.83</td>
<td>12</td>
<td>0.26</td>
<td>46.77</td>
<td>54.71</td>
<td>4.40</td>
<td>19</td>
</tr>
<tr>
<td>HAPI</td>
<td>285</td>
<td>4.35</td>
<td>3.61</td>
<td>7.96</td>
<td>12.85</td>
<td>0.62</td>
<td>7</td>
<td>0.59</td>
<td>11.90</td>
<td>41.76</td>
<td>3.56</td>
<td>16</td>
</tr>
<tr>
<td>EZMorph</td>
<td>45</td>
<td>4.42</td>
<td>9.24</td>
<td>13.67</td>
<td>24.98</td>
<td>0.55</td>
<td>1</td>
<td>0.30</td>
<td>3.30</td>
<td>73.29</td>
<td>4.80</td>
<td>1</td>
</tr>
<tr>
<td>CRONOMETER</td>
<td>189</td>
<td>4.71</td>
<td>8.24</td>
<td>12.95</td>
<td>19.80</td>
<td>0.65</td>
<td>6</td>
<td>0.50</td>
<td>11.95</td>
<td>63.23</td>
<td>5.04</td>
<td>4</td>
</tr>
<tr>
<td>jMemorize</td>
<td>203</td>
<td>4.72</td>
<td>4.37</td>
<td>9.10</td>
<td>12.52</td>
<td>0.73</td>
<td>29</td>
<td>3.05</td>
<td>9.52</td>
<td>46.89</td>
<td>5.20</td>
<td>2</td>
</tr>
<tr>
<td>FreeTTS</td>
<td>254</td>
<td>3.54</td>
<td>6.72</td>
<td>10.25</td>
<td>16.02</td>
<td>0.64</td>
<td>9</td>
<td>0.67</td>
<td>13.48</td>
<td>53.07</td>
<td>5.56</td>
<td>2</td>
</tr>
<tr>
<td>JGraphT</td>
<td>318</td>
<td>3.03</td>
<td>4.49</td>
<td>7.52</td>
<td>8.01</td>
<td>0.94</td>
<td>3</td>
<td>0.35</td>
<td>8.53</td>
<td>26.81</td>
<td>4.92</td>
<td>15</td>
</tr>
<tr>
<td>JStock</td>
<td>351</td>
<td>3.90</td>
<td>6.79</td>
<td>10.69</td>
<td>18.49</td>
<td>0.58</td>
<td>2</td>
<td>0.09</td>
<td>22.06</td>
<td>62.85</td>
<td>5.08</td>
<td>1</td>
</tr>
<tr>
<td>iFreeBudget</td>
<td>381</td>
<td>6.06</td>
<td>6.70</td>
<td>12.76</td>
<td>17.23</td>
<td>0.74</td>
<td>4</td>
<td>0.15</td>
<td>27.07</td>
<td>71.06</td>
<td>5.01</td>
<td>1</td>
</tr>
<tr>
<td>OmegaT</td>
<td>426</td>
<td>4.05</td>
<td>5.67</td>
<td>9.72</td>
<td>14.14</td>
<td>0.69</td>
<td>37</td>
<td>1.83</td>
<td>20.26</td>
<td>47.55</td>
<td>5.34</td>
<td>16</td>
</tr>
<tr>
<td>jPicEdt</td>
<td>600</td>
<td>4.30</td>
<td>5.54</td>
<td>9.84</td>
<td>11.66</td>
<td>0.84</td>
<td>8</td>
<td>0.34</td>
<td>23.75</td>
<td>39.59</td>
<td>4.97</td>
<td>5</td>
</tr>
<tr>
<td>jGAP</td>
<td>769</td>
<td>6.12</td>
<td>7.34</td>
<td>13.45</td>
<td>12.38</td>
<td>1.09</td>
<td>1</td>
<td>0.03</td>
<td>32.29</td>
<td>41.99</td>
<td>4.81</td>
<td>2</td>
</tr>
<tr>
<td>OpenbravoPOS</td>
<td>858</td>
<td>3.75</td>
<td>6.10</td>
<td>9.85</td>
<td>11.76</td>
<td>0.84</td>
<td>97</td>
<td>2.57</td>
<td>37.71</td>
<td>43.95</td>
<td>5.61</td>
<td>2</td>
</tr>
<tr>
<td>MX4J</td>
<td>1011</td>
<td>4.33</td>
<td>5.27</td>
<td>9.60</td>
<td>12.22</td>
<td>0.79</td>
<td>21</td>
<td>0.56</td>
<td>37.18</td>
<td>36.78</td>
<td>5.01</td>
<td>27</td>
</tr>
<tr>
<td>Grinder</td>
<td>1286</td>
<td>3.47</td>
<td>3.47</td>
<td>6.94</td>
<td>8.86</td>
<td>0.78</td>
<td>10</td>
<td>0.23</td>
<td>43.06</td>
<td>33.48</td>
<td>5.37</td>
<td>4</td>
</tr>
</tbody>
</table>
Questions for First Survey
Questions in the first survey to get defects information from the open source project developers:

1) How many defects have you discovered for the project release under investigation? (please refer email for which release I'm referring to)

2) When do you collect defect data, is it during system testing and/or after deployment?

3) What do you keep in defect logs for your current system? Do you just simply record all defects as they are reported by testers and users, or do you record defects only by difference (treating the same defects reported multiple times as 1 defect)?

4) Do you keep and publish a 'known problems' list which represents current defects?

5) Is it possible to get defect reports corresponding to different system versions that your team release? Do you have the latest release version list of reported, fixed and pending defects?
Questions for Second Survey
Questions in the second survey to understand the defect handling mechanism in open source projects:

1. Which of the defect tools/databases listed below do you use for your project?

   Apache (http://nagoya.apache.org/bugzilla)
   GNOME (http://bugzilla.gnome.org)
   Horde (http://bugs.horde.org)
   KDE (http://bugs.kde.org/)
   LyX (http://bugzilla.lyx.org)
   Mandrake (https://qa.mandrakesoft.com/cgi-bin/index.cgi)
   Mozilla (http://bugzilla.mozilla.org)
   OpenOffice (www.openoffice.org/issues/query.cgi)
   If another, please give the name of the defect database and its Web address.

2. Which of the roles below best describe your participation in the project? (choose all that apply)

   Technical Lead
   Software Developer/Engineer
   Technical Documenter
   Project Manager
   Tester
   Other (please explain)

3. The defects reported in your project include defects in the ... (choose all that apply)

   Requirement documents
   Design documents
   Source code
   Test documents
   Manuals
   Other (please explain)

4. The defects reported in your projects include ... (choose all that apply)

   Pre release defects
   Post release defects
   Both pre release and post release defects with no distinction
   Both pre release and post release defects but put into categories
   Any other strategy used regarding pre/post release defects (please explain)

5. In your current practice, a defect is usually recorded in the bug tracking system due to .... (choose all that apply)
Appendix C: Questions for Second Survey

Any problems found with the requirements
  Any problems found in the design
  Any problems arising out of testing
  Any problems found in the code through inspection
  Any incorrect part of the software that must be fixed
  Any other types of defect that apply to your project (please explain)

6. How do you categorize the defects being reported in the bug tracking system that you use? (choose all that apply)
   Accumulated for all releases/versions
   Defects that have been corrected
   Defects which have not been corrected
   Other (please explain)

7. When was the defect database first used in your project?
   Inherited from another project or from a previous release
     Right after the project started
     Right after the design started
     Right after coding started
     Right after pre release testing started
     In the post-release stage
     Other (please explain)

8. How consistent was the usage of defect the database in your project?
   Very consistent, all defects are reported before being fixed
     Generally consistent, the greater majority of defects are reported
     Generally consistent, but only for major defects
     Not very consistent, intermittently report
     Not consistent at all, defects reported only as time allows, most are not reported
     Other way in which you can describe the level of consistency of reporting defects

9. How many defects and which type (if applicable) have been reported for the project release under investigation? (please refer email for which release I'm referring to)
Appendix D

Metrics Extraction Tools

Contents

D.1 Logiscope ................................................................. 209
D.2 Resource Standard Metrics (RSM) ............................... 209
D.3 JStyle ................................................................. 210
D.4 Chidamber and Kemerer Java Metrics (CKJM) ........... 211
D.5 JHawk ................................................................. 212
D.1 Logiscope

Logiscope is a metrics extraction tool provided by Telelogic (now owned by IBM [www.telelogic.com]). This tool can extract metrics from the source of systems written in the Ada, C, C++ and Java programming languages. It can compute Comment metrics, Data metrics, Textual metrics, Halstead metrics, Structural metrics and Call graphs metrics. Altogether, it can compute about 68 metrics for all four languages but only 23 metrics for Java.

The problem with Logiscope is, most metrics I require for this research cannot be provided by this tool, for example, Fan-out and CBO. Furthermore, it cannot present the results in Microsoft Excel or OpenOffice.org Spreadsheet format. In addition, the license fee is very expensive to maintain. Those are the main drawbacks of the tool and that makes it unsuitable for my work. Later, I continued my search for another tool that can assist me with my metrics extraction and the next one I found was Resource Standard Metrics (RSM).

D.2 Resource Standard Metrics (RSM)

Resource Standard Metrics [www.msquaredtechnologies.com], is a source code metrics and quality analysis tool for systems written in C, ANSI C++, C and Java source code across operating systems. This tool can provide measurements at several levels of a system, for example, project level, package level, file level, class level, interface level and method level.

The metrics provided by RSM and which are relevant to my work are:

- Total number of classes
- Inheritance Tree
- Number of Base Classes
- Number of Derived Classes
- Maximum and Average Inheritance Depth
- Maximum and Average Number of Child Classes
- Public, private, protected data attributes
- Public, private, protected methods
- Lines of Code (LOC)
Appendix D: Metrics Extraction Tools

- Effective LOC (eLOC)
- Logical Statements LOC (ILOC)
- Number of Input Parameters
- Number of Return Points
- Interface Complexity (Parameters + Returns)
- Cyclomatic Complexity Logical Branching
- Class Complexity (Interface + Cyclomatic)
- Total Quality Profile

This tool performs well in the presentation of information or results and is able to provide many relevant metrics that are useful for my work. The only drawback of this tool is that it cannot provide the main metrics that I would like to study, such as Fan-out and CBO. However, the results given by this tool are very useful for my analysis, hence, to complement the results obtained by this tool, JStyle is identified as the tool to calculate Fan-out and CBO.

D.3 JStyle

JStyle [www.mmsindia.com/jstyle.html] is a tool for collecting various software metrics including the Chidamber and Kemerer object-oriented (OO) metrics (CK metrics). This tool supports the measurement of Java software and has four levels of object-oriented metrics: project level, module level, class level and method level. The metrics provided by JStyle and which are relevant to my work are:

- Depth of Inheritance (DIT)
- Number of Children (NOC)
- Response For Class (RFC)
- Lack of Cohesion in Methods (LCOM)
- Weighted Methods Complexity (WMC)
- Fanin (FI)
- Fan-out (FO)
It seems that JStyle can provide Fan-out but not CBO. This is unfortunate because I intended to use CBO as a surrogate for Fan-out in object-oriented systems. Therefore, I continued to search for a tool that provides CBO results. This led to the discovery of the Chidamber and Kemerer Java Metrics (CKJM) tool.

**D.4 Chidamber and Kemerer Java Metrics (CKJM)**

CKJM provides results for the Chidamber and Kemerer object-oriented metrics by processing the byte code of compiled Java files. The program calculates the following six metrics proposed by Chidamber and Kemerer, for each class:

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling between object classes (CBO)
- Response for a Class (RFC)
- Lack of cohesion in methods (LCOM)
- Afferent couplings (Ca)
- Number of public methods (NPM)

During my search for tools that support the metrics required for this research it became apparent that no one tool could provide all of them. RSM, JStyle and CKJM together were able to provide the required metrics and fortunately, in several instances, they all provided some of the metrics needed, such as Number of Parameters, CBO and Cyclomatic Complexity.

CKJM [www.spinellis.gr/sw/ckjm](www.spinellis.gr/sw/ckjm) is freely available as open-source software. It can compute the value of CBO thus completing my search for the tools that can give me all the required metrics. However, there is a downside to CKJM which shed doubt on its results when compared with other tools. Both RSM and JStyle use source code as input, while CKJM uses byte code (or Java files). The fact that both tools use different types of input (source code as opposed to byte code), posed a problem. Apparently, when byte code is generated by Java compiler, some information, such as type information is discarded. Therefore, the results provided by both tools are not comparable and there is a need to find a tool that can produce...
both CBO and Fan-out values. After an extensive search, JHawk was exposed as the appropriate tool.

## D.5 JHawk

JHawk [www.virtualmachinery.com/jhawkmetrics.htm] is a tool that can calculate object-oriented metrics such as CK OO metrics as well as structured metrics such as Henry and Kafura's metrics and Halstead's metrics. Besides that, it can calculate metrics at system, package, class and method level. One of the most important features of this tool is that it can produce both Fan-out and CBO metrics making analysis easier than when these metrics are produced by two different tools. Among the metrics JHawk provides which are important to this research are:

- Cyclomatic Complexity (CC)
- Calculated Class Value (CCV)
- Number of Statements (NOS)
- Coupling Between Object Classes (CBO)
- Response for a Class (RFC)
- Lack of Cohesion in Methods (LCOM (Henderson-Sellars))
- Total Halstead Effort (THE)
- Number of Methods (NOM)
- Instance Variables (INST)
- Message Passing Coupling (MPC)
- Fanin (FI)
- Fan-out (FO)
- Maintainability Index (MI)
Bibliography

Linux in government, Linux Online.


Awang Abu Bakar, N. S., and C. V. Boughton, Using a combination of measurement tools to extract metrics from open source projects, in Software Engineering and Applications (SEA '08), 2008.


BIBLIOGRAPHY


BIBLIOGRAPHY


Li, P. L., A catalog of techniques that predict information about the count or rate of field defects, CMU-ISRI-06-122, 2006.


Malaysia, Y., and J. Denton, Module size distribution and defect density, in International Symposium on Software Reliability Engineering (ISSRE’00), 2000.


Nagappan, N., and T. Ball, Use of relative code churn measures to predict system defect density, in *Proceedings of International Conference of Software Engineering (ICSE’05)*, 2005.


Nagappan, N., T. Ball, and A. Zeller, Mining metrics to predict component failures, in *Proceedings of International Conference of Software Engineering (ICSE’06)*, 2006b.


BIBLIOGRAPHY


www.openbsd.org.

www.opensource.org.

www.sourceforge.net.

www.spinellis.gr/sw/ckjm.


Glossary

ADC Average Data Complexity. 194
APC Average Procedural Complexity. 194
ASC Average Structural Complexity. 194
ASysC Average System Complexity. 194

**Average Data Complexity** The numbers of parameters/arguments passed to the class and the parameters that the class must return to other classes. 113

**Average Procedural Complexity** Measures the number of decisions in a class, also known as McCabe's Cyclomatic Complexity. 113

**Average Structural Complexity** Measures dependencies of classes, also known as Coupling between Objects Classes. 113

**Average System Complexity** The combination of Average Structural and Average Data Complexity. 113

CG Card and Glass. 142
CS Class Size. 194
DD Defect Density. 194

**Defect Density** Total defects in the system divided by size (KLOC). 118

**Defects** A concrete manifestation of errors within the software. 71

DEV Number of Developers. 194

KLOC Thousand lines of code. 194

NC Number of Classes. 194

**Outliers** Data points which are located in an otherwise empty part of the sample space. 110

**Regression coefficient** When the regression line is linear \((y = ax + b)\) the regression coefficient is the constant \((a)\) that represents the rate of change of one variable \((y)\) as a function of changes in the other \((x)\); it is the slope of the regression line. 115
Glossary

SP  Ratio of System and Procedural Complexity. 194

Standard error  The standard deviations of the sample in a frequency distribution, obtained by dividing the standard deviation by the total number of cases in the frequency distribution. 115

Statistical significance  The probability that the coefficient is different from zero by chance Hoaglin et al. [1983], Harrell [2001] and usually referred to as $\alpha$ level. 116

TD  Total Defects. 194

VD  log10Version Downloads/Version Downloads. 194

Version Downloads  Represents the number of downloads for the version being studied. 70