Patterns of Change: Can modifiable software have high coupling?

A thesis
submitted in fulfilment
of the requirements for the degree
of
Doctor of Philosophy

at
The University of Waikato

by
Craig Taube-Schock

Department of Computer Science
Hamilton, New Zealand
May 2012

Abstract

There are few aspects of modern life that remain unaffected by software, and as our day-to-day challenges change, so too must our software. Software systems are complex, and as they grow larger and more interconnected, they become more difficult to modify due to excessive change propagation. This is known as the *ripple effect*. The primary strategies to mitigate it are modular design, and minimization of *coupling*, or between-module interaction. However, analysis of complex networks has shown that many are *scale-free*, which means that they contain some components that are highly connected. The presence of scale-free structure implies high coupling, which suggests that software systems may be hard to modify because they suffer from the ripple effect.

In this thesis, a large corpus of open-source software systems is analyzed to determine whether software systems are scale-free, whether scale-free structure results in high coupling, and whether high coupling results in ripple effects that propagate change to a large proportion of classes.

The results show that all systems in the corpus are scale-free and that that property results in high coupling. However, analysis of system evolution reveals that existing code is modified infrequently and that there is rarely sufficient evidence to be confident that ripple effects involving a high proportion of classes have actually occurred. This thesis concludes first that while it is desirable to avoid excessive interconnectivity, it is difficult to completely eliminate high coupling; and second, that the presence of high coupling does not necessarily imply poor system design.

Acknowledgements

First and foremost, I would like to thank my chief supervisor, Ian Witten. He is a brilliant researcher and a wonderful person, and it has been both an honour and privilege to have worked with him for the past three and a half years. I have always admired how he is able to craft language so that even complex ideas are easy to understand and enjoyable to read, and I am particularly fortunate to have been privy to his writing processes and victim of his editing pen. Thank you, Ian!

I would also like to thank Rob Walker for introducing me to the field of software evolution, and for his continued direction and support through the thesis. Rob's devotion to the field and to high quality research are unsurpassed, and that has a positive impact on those with whom he works. There were times throughout this PhD when I would complete some work only to go back and improve upon it later because I knew that Rob would look at it. The field of Software Engineering is lucky to have him as a contributor!

Thanks to my other supervisors Dave Nichols and Mark Utting. I asked several questions of each of you throughout this work, and your valuable feedback via progress reports is very much appreciated. As the thesis writing process came to a close, Dave provided heaps of information and good ideas for improving its overall structure and quality. I am very much indebted to each of you.

Many thanks to my fellow PhD students in the Digital Library lab: Rob Akscyn, Kathryn Hempstalk, Olena Medelyan, David Milne, Shaoqun Wu, Veronica Liesaputra, Anna Huang, Akram Darwish, Antti Puurula, and Michael Walmsley. Our ongoing research meetings were a constant source of inspiration, and it was a great a pleasure to share our many dinners and trips to Waiheke Island. From this group, I would particularly

like to thank Rob Akscyn, with whom I have worked very closely over the last three years. His insights into the structure of scientific writing have helped me immensely to develop my writing style, and the use of “Expeditee” has dramatically reduced the time it takes me to identify defects in my own writing.

Many thanks, too, to those in the Laboratory for Software Modification Research at the University of Calgary, with particular thanks to Rylan Cottrell. His expertise in dealing with Eclipse was key to the completion of my toolset, and I shudder to think about how things would have gone had he not been there.

The Symphony high performance computing cluster was used extensively throughout this research. Many thanks to the Department of Computer Science for the use of the cluster, and to Donald Neal, its administrator.

I would also like to thank the National Science and Engineering Research Council (NSERC) of Canada for the PGS-D3 scholarship that I was able to take abroad for my PhD studies. Without the scholarship, this thesis and the research it presents would not have been completed. Thanks to the University of Waikato for providing a six month Doctoral scholarship, which is precisely the amount of time that I needed to complete the thesis.

Finally, I would like to thank my family. To my mom who always instilled in me the importance of education, and provided me with a childhood that was second to none. Words cannot express how thankful I am to you for everything that you’ve given me over my life. To my partner, Ruth, and my daughter, Rachel—I could not have got through this without you two. Whenever I disappeared into the PhD fog, Ruth was always there to ensure that I made it out again. Whenever things seemed to become too big to deal with, Rachel would always help me to remember the importance of the little things in life. I love you all very much and I anxiously await the new arrival to our family.

Contents

1 Introduction 1
1.1 Building complex systems that change 2
1.2 Research questions 3
1.3 Empirical investigation 4
1.4 Thesis organization 5

2 Literature Review 7
2.1 Model of a complex system 8
2.1.1 System growth 9
2.1.2 Connectivity and the ripple effect 9
2.1.3 Regulating change propagation between modules . 12
2.2 Notation used in the thesis 13
2.2.1 System structure and evolvability 14
2.3 Design principles and software evolvability 16
2.3.1 Information hiding 16
2.3.2 Design rules and the design structure matrix 18
2.3.3 Coupling and cohesion 19
2.3.4 Discussion 20
2.4 Measuring cohesion and coupling 20
2.4.1 Measuring coupling 20
2.4.2 Measuring cohesion 23
2.4.3 Empirical validation of cohesion and coupling metrics 25
2.5 Random models of complexity 26
2.5.1 Power-law distributions 27

2.5.2 Generating random power-law networks 28
2.5.3 Small-world networks 28
2.6 The ripple effect in scale-free and small-world networks . 29
2.6.1 The Susceptible, Infective, Removed model 29
2.6.2 Empirical studies 30
2.6.3 Change propagation through inferred links 31
2.7 Network analysis of software 32
2.8 Preferential attachment and source code evolution 35
2.9 The matching problem 37
2.10 Summary and discussion 40
2.10.1 Research questions revisited 42

3 Tools 45
3.1 Conceptual overview 45
3.2 Corpus of software systems 47
3.3 The Eclipse AST parser 48
3.3.1 Parse file format 49
3.3.2 Abstract syntax tree example 51
3.4 CodeNet 53
3.4.1 Constructing system graphs 55
3.4.2 References to external entities 57
3.4.3 Reverse inheritance 57
3.4.4 Semgraph file format 58
3.4.5 Quality control 59
3.5 Analysis 59
3.5.1 Analysis process 59
3.5.2 Semgraph data structure 60
3.5.3 Example analysis 62
3.5.4 Integration with a computing cluster 63
3.6 Discussion 64
3.6.1 Existing tools 64
3.6.2 Toolset evolution 65

4 Scale-free structures and coupling 67
4.1 Perspectives of degree distributions 67
4.1.1 Inlinks, outlinks and combined perspectives 68
4.1.2 Within-module versus between-module links 68
4.1.3 Link aggregation 69
4.1.4 Combining degree distribution perspectives 69
4.1.5 Hierarchical links 70
4.2 Hypotheses 70
4.2.1 Proposed model 71
4.2.2 Hypothesis 1: Scale-free structure in source code networks 73
4.2.3 Hypothesis 2: Scale-free structure and coupling . . 73
4.2.4 Hypothesis 3: Outlink constraints 73
4.2.5 Hypothesis 4: Aggregate measures of coupling . . . 74
4.2.6 Hypothesis 5: Aggregate outlink distributions . . . 75
4.3 Experimental design 76
4.3.1 Computing module boundaries 76
4.3.2 Testing the hypotheses 77
4.4 Results 79
4.4.1 Hypothesis 1 79
4.4.2 Hypothesis 2 81
4.4.3 Hypothesis 3 83
4.4.4 Hypothesis 4 87
4.4.5 Hypothesis 5 90
4.4.6 Post-hoc analysis for Hypothesis 5 91
4.5 Discussion 92
4.5.1 High coupling caused by node-level interaction . . 92
4.5.2 High coupling caused by aggregate interaction . . 93
4.5.3 Internal validity 94
4.5.4 Construct validity 95
4.5.5 External validity 95
4.5.6 Open research question 96

5 Patterns of change 97
5.1 Automated Matching Method 98
5.1.1 First pass—Applying matchers 98
5.1.2 Second pass—Dependency analysis 100
5.2 Automated Matching Method performance evaluation . . . 102
5.2.1 Unmatched classes 103
5.2.2 Unmatch rates for individual pairings 104
5.2.3 Unmatched classes correlated with coupling 105
5.2.4 Discussion 106
5.3 Measuring change 107
5.3.1 Computed measures 107
5.3.2 The lifetime of a class 109
5.4 Analysis of change 111
5.4.1 Confounding factors 111
5.4.2 Change measures that equal zero 112
5.4.3 Correlation of change measures 113
5.4.4 Discussion of global observations 114
5.5 Observed patterns of change 115
5.5.1 Pattern classification 115
5.5.2 Example classifications 116
5.5.3 Pattern frequency 118
5.6 Discussion 119

6 High coupling and software evolution 121
6.1 The evolution of inlink coupling 121
6.1.1 Hypothesis 6: Distribution of changes to M7 124
6.1.2 Testing Hypothesis 6 124
6.1.3 Results 124
6.1.4 Discussion 126
6.2 High coupling and the ripple effect 127
6.2.1 Criteria for the ripple effect in software 128
6.2.2 Hypothesis 7: Identifying ripple effects in software 129
6.2.3 Results 130

6.2.4 Discussion 132
6.2.5 Limitations of this analysis 134

7 Conclusions and Future Work 137
7.1 Research questions 137
7.1.1 Is the structure of software scale-free? 138
7.1.2 Does scale-free structure result in high coupling? . 140
7.1.3 Can the ripple effect be observed in software with high coupling? 141
7.2 Closing remarks 143
7.3 Contributions 144
7.4 Future work 145
7.4.1 Ripple effect detection and impact analysis 145
7.4.2 Research support and reproducibility 146

References 149

A Example parse file 167

B System structural measures 171

C Version pairs 175
C.1 *ant* release pairs 175
C.2 *antlr* release pairs 176
C.3 *argouml* release pairs 177
C.4 *azureus* release pairs 178
C.5 *freecol* release pairs 180
C.6 *hibernate* release pairs 181
C.7 *jgraph* release pairs 183

C.8 *jmeter* release pairs 184
C.9 *jung* release pairs 185
C.10 *junit* release pairs 186
C.11 *lucene* release pairs 187
C.12 *weka* release pairs 188

List of Figures

2.1 Relationship between the number of nodes and the number of possible connections between them. 9
2.2 Notational items used in the thesis. 14
2.3 Notational conventions used in the thesis. 15
2.4 Two different modularizations of the *KWIC* system. 17
2.5 Example of a design structure matrix. 18
2.6 Common aspects of coupling measures. 21

3.1 Toolset architecture. 46
3.2 Code listing 1. 51
3.3 Abstract syntax tree for Java listing 1. 52
3.4 Directed graph meta-model. 54
3.5 Semgraph file structure. 58
3.6 Class model used to encode semgraphs. 61
3.7 Code listing 2. 62

4.1 Inner class definition and coupling. 77
4.2 Distribution of overall connectivity for three sample systems. 80
4.3 Mean degree vs. maximum degree for all systems. 80
4.4 Between-module degree distributions for three sample systems. 81
4.5 Comparison of overall and between-module connectivity distributions for three sample systems. 82
4.6 Comparison of α estimates for all systems. 83
4.7 All inlinks for *derby-10.1.1.0*. 83
4.8 Between-module inlinks for *derby-10.1.1.0*. 84
4.9 All outlinks for *derby-10.1.1.0*. 84
4.10 Between-module outlinks for *derby-10.1.1.0*. 85

4.11 Comparison of all inlinks vs. all outlinks for *derby-10.1.1.0*. . 85
4.12 Comparison of between-module inlinks vs. between-module outlinks for *derby-10.1.1.0*. 86
4.13 Between-module outlinks for *jgraph-5.9.2.1*. 86
4.14 Degree distribution ranges of inlinks and outlinks for all systems. 87
4.15 Aggregate distribution for all links for *springframework-1.2.7*. 88
4.16 Aggregate distribution for *fitjava-1.1*. 88
4.17 Max degree and mean degree for aggregate distributions. . . 88
4.18 Max degree and mean degree for between-module aggregate distributions. 89
4.19 Max inlink and outlink degrees for aggregate distributions. . 89
4.20 Max inlink and outlink degrees for between-module aggregate distributions. 90
4.21 Class size versus aggregate between-module outlink count for *derby-10.1.1.0* (r=*0.89*) 90
4.22 Correlation (r) between class size and aggregate between-module outlink count for all systems. 91
4.23 Correlation (r) between class size and aggregate between-module outlink count for all systems. 91

5.1 Matching classes between versions—pass 2. 101
5.2 Eight parameters of change for a module. 107
5.3 Change measures for eight versions of the same class. 110
5.4 Non-uniformity in the Qualitas corpus. 111

6.1 Frequency of changes in aggregate between-module inlink count. 125
6.2 Increase in inlink count versus current inlink count. 126
6.3 Ratio of changed and unchanged classes for both degree centrality and percentage change. 131

List of Tables

3.1 Parsefile tags 49
3.2 Semgraph file tags 58

4.1 Distributions required to test Hypotheses 1 – 4 78

5.1 Measures to assess quality of the matching process. I=Empty Inlink Set, O=Empty Outlink Set, B=Both sets empty, N=No match found, >1=Too many candidates 103
5.2 List of release pairs with > 15% unmatched classes. 105
5.3 Number of unmatched classes with > 50 between-module links. 105
5.4 Change matrix for example class 110
5.5 Number of change measures that equal 0. CR=Change Row Count, ZR=Count of rows where all entries = 0, MX=Count of rows with measure X = 0 112
5.6 Correlation of change measures across the Qualitas corpus . 113
5.7 Pattern classification based on the percentage of zero-valued entries. 115
5.8 Unchanging class with stable M1, moderately unstable M3, and unstable M7 116
5.9 Moderately stable class with stable M8 and unstable M3 and M7 117
5.10 Moderately unstable class with unstable M7 118
5.11 Frequency of patterns observed for each measure and for classes 118

6.1 Moderately stable class with moderately unstable M1 and M8 and unstable M3 and M7 123

6.2 Number of version pairs where influence of changed classes exceeds that of unchanged classes. 132

Chapter 1

Introduction

> Change is the only constant. Hanging on is the only sin.
>
> —Denise McCluggage

Complexity abounds! No matter where we look, from our immediate surroundings to the very frontiers of our exploration, we find bewildering complexity. We find it in the simplest of life forms, which rely on an intricate set of metabolic interactions, delicately balanced, to support the processes of life. We find it in the oceans and the air, which give us our stable and livable climate, but unpredictable weather systems. We even find it when we look to the heavens, beyond our own small planet. Complexity awaits everywhere, for us to discover.

Complexity is not limited to the natural world. Humankind is capable of creating instruments of immense complexity, and it is difficult to find any aspect of modern day life that remains unaffected by complex artificial systems. From the machines that transport us and our goods around the world, to the network of computers that enable us to communicate, almost instantaneously, between any two points on earth, we have added artificial complexities to our natural world.

The problems we solve, however, do not remain fixed. As magnificent as our tools are, they are only useful if they solve problems that are relevant. How many times in history have creations become obsolete because they address problems that we no longer face? No system, no matter how complex, is immune to obsolescence.

1.1 Building complex systems that change

This thesis is about software, and how its structure can affect its ability to be adapted in response to the pressure of change. It begins with an exploration of design theory, which reveals that the evolvability of a complex system is closely related to the interconnectivity between its components. Too much interaction causes change to propagate from component to component, and small changes can ripple to seemingly unrelated sections of the system. This leads to the avoidance of even small changes because of the effort they require, rendering the system inflexible.

Design theory teaches methods of mitigating unconstrained propagation of change. Here, modularization is the primary tool. It enables us to group components to restrict their mutual changes to specific parts of the system, thereby making it more flexible in its response to change pressures. Modularization, however, is no panacea. If change is allowed to propagate from module to module, the benefit is lost. Constraint is maintained by the regulation of between-module interaction.

Design theory is applied to software using design principles. Many of these involve the concept of *coupling*. Coupling is the strength of association that is caused by the relationship between two modules, and it relates directly to the probability that change will propagate from one module to the next. Design principles limit change propagation by reducing coupling. However, an exploration of measures of coupling show that those measures are largely unvalidated against external properties of software systems.

The analysis of complex networks has shown that many have a property called *scale-free*. A scale-free network is organized in such a manner that most of its components have limited connectivity, but some have very high connectivity. Networks in numerous fields of study, including software networks, have been found to be scale-free.

Scale-free structure, however, appears incongruous with the principles of minimizing connectivity and coupling. It seems that a network that is scale-free would have areas of high coupling, and this suggests that they would, therefore, be less modifiable. How scale-free structure affects a system's modifiability is the main theme of this work.

1.2 Research questions

This thesis addresses three research questions:

1. Is the structure of software scale-free?
2. Does the high level of connectivity that is associated with scale-free structure result in high coupling?
3. Can the ripple effect be observed in the version history of software that has high coupling?

Research that views software systems as complex networks has revealed that several properties of software are scale-free. However, these investigations are limited to examining the modular structure of systems, and avoid piercing module interfaces. Because it is the internal components of modules that determine between-module connectivity, it is prudent to go beyond module boundaries to inspect what is happening inside. This thesis aims to determine whether the source-code network—the part of the system that programmers modify directly—is also scale-free. This is the first research question.

If source-code networks are scale-free, some components will inevitably have very high connectivity. However, high connectivity does not necessarily translate to high between-module interaction. Highly connected components may only connect with other components that are contained within the same module, thereby resulting in low coupling. This thesis aims to determine whether scale-free structure at the source-code level consistently results in high between-module interaction. This is the second research question.

Systems that suffer from ripple effects are considered to be of low modifiability, but there has been no empirical validation showing that high coupling necessarily causes ripple effects. Related research considers system modifiability in a prospective manner, and asks questions such as "what is the impact of changing this variable?" This research, however, seeks to examine software retrospectively to determine if a relationship between high coupling and ripple effects can be supported empirically. This is the third research question.

1.3 Empirical investigation

The questions above are investigated through an empirical analysis that uses a large, curated corpus of open-source software systems. Since there are significant barriers to performing analyses of such large scope, appropriate tools have been constructed. A tool called *CodeNet* translates software systems into large networks, which are then analysed using the methods provided by network theory. Since network analysis algorithms tend to have high time complexity and the corpus represents a large collection, automated methods for distributing the required analyses onto a high performance computing cluster have also been constructed. Finally, in order to facilitate an analysis of how software evolves, a tool called *Automated Matching Method* (AMM) was constructed to match classes between different versions of software systems. AMM is based on existing methods, but is tailored to this particular investigation.

The empirical investigation comprises seven hypotheses. Research questions 1 and 2 are addressed by Hypotheses 1 through 6, and research question 3 is addressed by Hypothesis 7. The results of Hypothesis 7 are supported by an exploratory study that examines the patterns of change that occur in the corpus.

The structure of the software systems and whether that structure results in high coupling is addressed by comparing several different degree distributions. Overall scale-free structure is determined by computing a degree distribution for the whole software network, and between-module interaction is determined by computing a degree distribution that only considers between-module links. To analyse coupling at the class level, aggregate degree distributions are used.

Detecting ripple effects is the most problematic aspect of this work because there is no established method to quantify these. A method based on a model of disease propagation through human populations is devised and presented in conjunction with an exploratory study of observed patterns of change.

1.4 Thesis organization

This thesis is organized as follows. Chapter 2 provides background from three distinct areas of study: design theory, software engineering, and network theory. It begins by presenting a model for complex systems, and describes methods for constraining systems to improve their evolvability. Once the model is established, existing applications of design principles and metrics are reviewed. Then, the field of network theory is introduced with particular attention paid to scale-free structure, since it this that clashes with the existing literature on design theory and software design principles. Finally, research that applies network theory to software is reviewed. This is found to be exploratory, focusing on demonstrating the existence of scale-free structure, but does not relate such structure to existing design principles.

Chapter 3 presents the tools used to support this work. It begins with a conceptual overview of the analytic framework, and then provides a detailed description of its four components: the corpus, system parsing, graph construction, and analysis. This thesis uses the Qualitas corpus (Tempero, Anslow, Dietrich, Han, Li, Lumpe, Melton and Noble, 2010), a large curated collection of open-source systems written in Java (Gosling, Joy, Steele and Bracha, 2005). System parsing is performed using the AST parser in the Eclipse system (Eclipse Foundation, 2011). Graph construction is accomplished using the custom-built *CodeNet* tool. Analysis is completed using analysis modules that are integrated with CodeNet. The chapter ends with a description of how the framework is distributed onto a high performance computing cluster.

Chapter 4 presents a study designed to answer research questions 1 and 2. Five hypotheses about the structure of software are proposed, and tested against 97 open-source systems written in the Java programming language. As a justification for the hypotheses, a method of software evolution based on a well-known network evolution model is proposed.

Chapter 5 establishes the two methods used to investigate the evolution of software. The first is an automated technique for matching classes, based on existing, non-automated techniques. The second is a framework for measuring how structural properties of software change

over successive system versions. Both techniques are used as part of an exploratory study that examines three aspects of software evolution: the frequency of change events, the correlation of change measures, and the patterns of change exhibited by classes.

Chapter 6 examines the relationship between coupling and software evolution, and is presented in two parts. The first examines the distribution of changes to inlink coupling to determine whether there is support for the model of software evolution presented in Chapter 4. The second builds on the exploratory study performed in Chapter 5 to determine whether ripple effects can be observed in the corpus. As part of this investigation, an observational method based on a model of disease propagation in human populations is used.

Chapter 7 presents the conclusions of this thesis, along with a discussion of future work.

Chapter 2

Literature Review

> Forschen: Sehen, was jeder sieht, und denken, was keiner denkt
>
> — Albert von Szent-Györgyi Nagyrápolt

This thesis draws on a large body of literature that is spread over three distinct fields of study—design theory, network analysis, and software evolution. This chapter begins by introducing the model that is used to represent software systems. It is based on a mathematical construct, the directed graph, and is constrained by rules that are expressed in design theory. Then the notation used in this thesis to represent complex systems is described.

Once the model has been established, three examples of how constraints are applied to software design are examined. The third introduces the concepts of *coupling* and *cohesion*, which are described in detail. A review of the literature concerning cohesion and coupling measures is presented, and it is shown that these measures are seldom validated against external properties of software, such as evolvability.

An overview of network theory is presented, which introduces the notion of *scale-free* structure. A network with scale-free structure is one that has many nodes with low connectivity, but some with very high connectivity, and it is this characteristic that is inconsistent with the previously presented literature. Studies that use network theory to analyze software systems are examined; However, these studies are exploratory in nature since they do not attempt to relate software structure to software design principles and software evolvability. Because this thesis examines the evolvability of software, literature relevant to the *matching*

problem—the problem of matching classes between different software versions so that they can be compared—is presented.

The chapter ends with a discussion that expands upon the research questions introduced in Chapter 1.

2.1 Model of a complex system

This thesis investigates the relationship between software system structure and its evolvability. To study this relationship, we first need a definition of what a system is and how it is represented.

A *system* is defined as a group of interacting components that collectively provide utility. Not only is a system composed of things, but there is a reason to put those things together. The minimal system is one with two components that are related through a single interaction. An example is the composition of a nut and a bolt. Alone, each provides limited functionality, but when used in concert, they fasten together. This functionality can be used in a larger system to fasten other components together.

Systems are represented as *directed pseudographs*, which are graphs that allow loops and multiple links between nodes (Zwillinger, 2003). A graph is defined as $G = (V, E)$ where V is a set of nodes (vertices) and E is a set of links (edges) that connect them. Each node in the graph represents a component and each link represents an interaction. In this investigation, a system requires that all of its nodes are *weakly connected*. That is, there must exist a path between each pair of nodes using the links in E. If G is not weakly connected, it represents more than one system. Graphs are commonly used to model complex systems (Ashby, 1952; Alexander, 1964; Simon, 1996), including software systems (Valverde and Sole, 2003; Marchesi, Pinna, Serra and Tuveri, 2004; Zhou, Lu and Xu, 2004; Chatzigeorgiou, Tsantalis and Stephanides, 2006; Jenkins and Kirk, 2007; Ichii, Matsushita and Inoue, 2008; Liu, Lü, He, Li and Tse, 2008; Pan, Li, Ma and Liu, 2011).

Because the term *component* has many different meanings, its use can be ambiguous in different contexts. Instead, we use the term *node* to

mean “a piece of a system.”

2.1.1 System growth

Systems are built to satisfy requirements, so as requirements grow, so too do systems. Growth manifests itself as the addition of nodes and links, and there is a mathematical relationship between the number of nodes and the maximum number of links.[1] This relationship is shown in Figure 2.1 and is represented by the formula $|E| <= \frac{|V|(|V|-1)}{2}$.

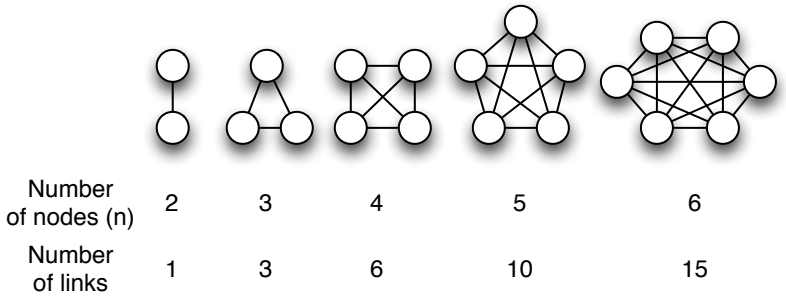

Figure 2.1: Relationship between the number of nodes and the number of possible connections between them.

The management of complexity focuses on minimizing the number of links, because they have the potential to increase faster than nodes. The number of interdependencies between nodes is widely believed to be the primary factor affecting the evolvability of systems due to the *ripple effect* (Simon, 1962; Alexander, 1964; Haney, 1972; Parnas, 1972; Stevens, Myers and Constantine, 1974; Chidamber and Kemerer, 1994; Baldwin and Clark, 2000).

2.1.2 Connectivity and the ripple effect

The ripple effect occurs when a change in one node necessitates changes in other nodes and these changes propagate through the system (Alexander, 1964; Haney, 1972). Complex systems rely on interaction between nodes, but those interactions represent dependencies that may facilitate change propagation. It is this tradeoff that needs to be resolved by sys-

[1] Excluding self links and multiple links between nodes.

tem designers. Ultimately, any new nodes must be connected to the system, but the designer must determine *how* they are connected. Design principles stress the minimization of interconnections (Simon, 1962; Alexander, 1964; Stevens et al., 1974; Baldwin and Clark, 2000).

To illustrate this point, Simon and Alexander both describe examples of the ripple effect in complex systems. Simon's example takes the form of a parable of two watchmakers (Simon, 1962). Each makes a watch whose design uses one thousand pieces. In the first design, the pieces depend on each other in such a way that if construction is interrupted, it must be restarted from the beginning. In the second, the watch is composed of sub-modules, which can be constructed independently. While the second design takes longer to build because of the added complexity of the sub-modules, the watchmaker need not have to restart construction from the beginning should the process be interrupted.

Alexander's example is a simulation of change propagation through a graph (Alexander, 1964). In this simulation, nodes are represented by light bulbs, each of which can be in one of two states (*on* or *off*). The system is in equilibrium when all bulbs are *off*, and that state is the desired goal. A bulb that is *on* represents a node that has changed, and during the simulation, any bulb that is *on* has a 50% chance of turning *off* in the following second. Any bulb that is *off* has a 50% chance of turning *on* in the following second provided that one of the bulbs to which it is connected is also *on*. The simulation begins from the state of equilibrium, and one randomly chosen bulb is turned *on*.

How the simulation progresses depends on how the bulbs are interconnected. Alexander's first simulation uses 100 bulbs with no interconnections. This takes, on average, two seconds to reacquire equilibrium. His second simulation adds a connection between each pair of bulbs. This takes, on average, 2^{100} seconds (10^{22} years) to reacquire equilibrium.

Alexander contrasts this with a third simulation, where bulbs are placed in ten groups of ten bulbs each. In each group, there is a connection between each pair of bulbs, but there are no connections between bulbs in different groups. The third simulation reacquires equilibrium in 2^{10} seconds, on average (about 15 minutes).

Both examples show the role that dependency plays in the propagation of change. With more connections, there is a larger number of pathways for change to propagate, which increases the probability that propagation will occur. The ripple effect, therefore, is more likely to occur in the presence of high connectivity. Both examples also propose that the solution to minimizing the total number of connections is to place nodes into formalized groups. These groups are called *modules*.

When relating Alexander's simulations to complex systems, however, his third simulation is problematic because it allows for no interconnections between bulbs in different modules. It stabilizes quickly because it is a simulation of ten independent systems, where a change in one system cannot possibly necessitate changes in any of the others because no pathways exist. For this simulation to represent a single system, it requires interconnections between modules. Simon calls this property *near decomposability* (Simon, 1962). When a system is modularized, the overall number of connections between nodes is decreased, but the interconnections between nodes in separate modules cannot be fully eliminated. The result is that in complex systems, there will always be interconnections between modules, and it is these connections that become the focus of system design.

One final note is that Simon's parable highlights the role of context in design. Each design produces perfectly good watches, but the judgment as to which design is better depends on the context within which the watches are made. In an environment where there are few interruptions, the first design is arguably better because construction is less complex. However, in an environment where there are many interruptions, the second is better because even with the more complex design, the overall average time to build a watch will be less. Design attempts to satisfy multiple contexts, which are often conflicting. This is why the design of complex systems is so difficult; addressing the demands of one context often limits the ability to address the demands of others (Alexander, 1964). For a given complex problem, there is no single best solution, and designers are required to exercise judgment in their decisions.

2.1.3 Regulating change propagation between modules

Alexander suggests that nodes be grouped into modules based on their degree of interaction (Alexander, 1964). If two nodes are highly interactive, they are likely to have a high probability of change propagation, so placing them in the same module provides a mechanism for limiting the effects of change. However, nearly decomposable systems will have interactions that cross module boundaries. If change propagation between modules is high, the benefit achieved through modularization is negated, because change may propagate freely from module to module. While modularization reduces the overall number of connections, the probability of propagation between modules must also be reduced, and this is accomplished by regulating between-module interaction.

In software systems, regulation can be achieved using abstraction (Parnas and Siewiorek, 1975; Verelst, 2005). From the definition of abstraction, Kramer (2007) focuses on two pertinent aspects. Abstraction is:

- generalization of concept by factoring commonality; and
- simplification by the removal of detail.

When two nodes interact directly, the probability of change propagation is based on their dependency. Abstraction provides a mechanism through which dependency can be reduced by introducing an intermediary proxy node whose definition is a generalized concept of the interaction. For example, if node A uses node C, and C provides specific utility, then changing C can necessitate a change in A. However, if an intermediary node B is introduced that provides a generalized concept of C, C can change without necessitating a change in A so long as its new utility remains faithful to the concept defined by B.

An example of this is seen in *information hiding*, which was first introduced by Parnas (1972) as a criterion for decomposing a system into modules. Parnas noted that some relationships have higher probability of change propagation than others. Dependency relationships to variables, for example, have a high probability of change propagation because variables represent a specific implementation of functionality. Should the

type of a variable change, for example, then any nodes dependent on it will have to be changed to become consistent with its new type. However, dependency on function definitions has a lower probability of propagation because they represent a generalized concept of utility. The function definition provides a proxy for nodes that provide the specific implementation, and should those nodes change, there will be no propagation so long as the new implementation remains consistent with the general concept of utility provided by the function definition.

With information hiding, the designer determines which nodes of a system are likely to change and hides them within modules using abstraction. Access to hidden nodes is through abstracted *interfaces*, which regulate the propagation of change across module boundaries. (Liskov, 1987) The degree to which the specific implementation is exposed by its interface is called its *transparency*. Transparent modules offer little protection from change propagation (Parnas and Siewiorek, 1975).

Simon notes that most complex systems are composed of interrelated submodules, which are hierarchical until some level of elementary submodule is reached (Simon, 1962). This view is consistent with the use of abstraction, where higher levels in the hierarchy represent higher levels of abstraction.

2.2 Notation used in the thesis

We represent complex systems as a directed graph, constrained by the use of modularization and abstraction. Figure 2.2 shows notational elements used in this thesis. Nodes and proxy nodes[2] are shown as circles or ovals, and can optionally contain descriptive text. Interactions based on usage are shown as solid lines, and may optionally show dependency. Hierarchical relationships are shown as solid lines with unfilled arrowheads in the direction of the parent node. Transitive relationships are used to show when two nodes cannot be considered independently. They are shown using a dashed line, and may also be hierarchical. Modules are shown as rectangles with the root node at the top and all contained

[2] In this notation, the role of a node is represented by its position on the diagram. If a single node has multiple roles, proxy nodes are used to mark each role.

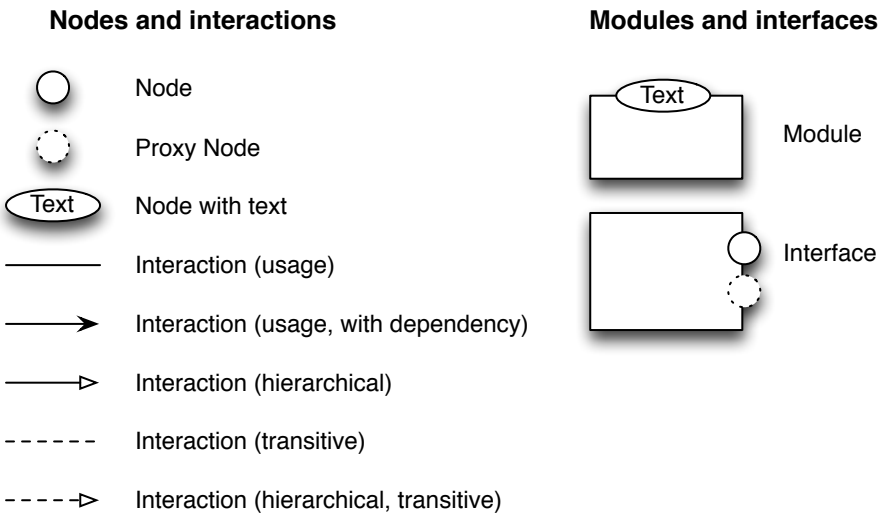

Figure 2.2: Notational items used in the thesis.

nodes within. Interfaces to modules are nodes that are placed on the border of the three remaining sides of the module rectangle.

Figure 2.3 illustrates the conventions used to represent object-oriented system structure. Nodes are class declarations, method declarations, variable declarations, function declarations, blocks, generic modules, and statements, and are differentiated by numbering, when necessary. Different versions of the same node are denoted with the same number, but with distinguishing letters. Module interfaces are denoted using proxy nodes and transitive connections, which illustrate the nodes that make up the interface.

2.2.1 System structure and evolvability

Researchers have noted that there is a relationship between system structure and its *evolvability*. Simon (1962) concluded that complex systems evolve faster when they are built from modules because modules represent a "stable intermediary form" upon which further evolution can take place. Alexander (1964) argued that high levels of interconnectivity could make even simple changes difficult, thereby limiting the extent to which a system can change. Haney (1972) argued that module interconnections can result in "rippling changes," thereby increasing the time

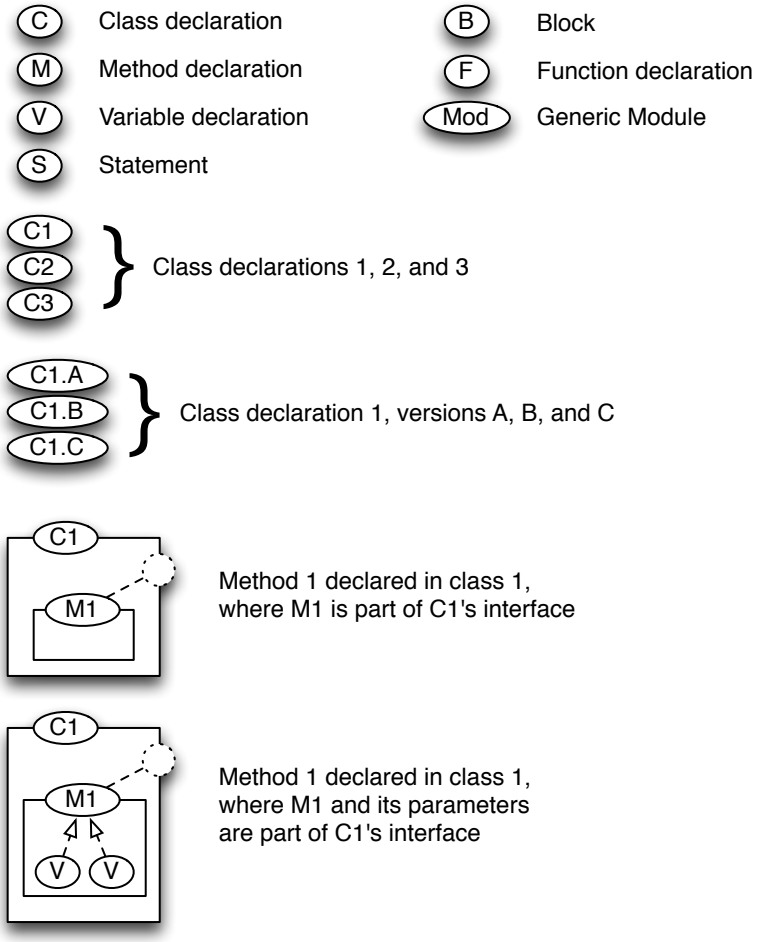

Figure 2.3: Notational conventions used in the thesis.

it takes for a system to stabilize after changes are introduced. Parnas (1972, 1994) argued that the encapsulation of volatile design decisions creates structures that are more robust to change propagation. He also argued that systems with fewer between-module interactions are easier to understand because the modules can be understood independently of each other, and understanding, in part, contributes to the amount of effort needed to modify a system. Baldwin and Clark (2000) argued that modularization promotes the decentralization of "valuable design options" so that they may change independently, thereby enabling evolution.

However, none of these researchers have offered a precise definition

of what it means for a system to be *evolvable*.[3] Rather, they rely on the logical implication that making more changes requires more effort, and since the supply of effort is finite, any system that suffers from increased change as a result of change propagation is less evolvable.

The primary reason that evolvability is not precisely defined is because change pressures typically originate from external sources at a future time, so they are unavailable when the system is created. Because dissimilar structures respond differently to change pressures, a structure that is ideal for a particular sequence of change pressures may not necessarily be ideal for a different sequence. To claim that a particular structure has "high evolvability" requires a corollary definition of the sequence of change pressures. Because the actual sequence is unknown, programmers must make design decisions based on anticipating the kind of pressures that will transpire.

2.3 Design principles and software evolvability

Principles have been formulated to help programmers make "good" design decisions when faced with limited information about future change pressures. They are rules of thumb that are generally believed to produce better system design. This section shows how software design principles attempt to improve modifiability by reducing change propagation between modules.

2.3.1 Information hiding

Parnas (1972) examined two different implementations of the *KWIC* system, which differed in the criteria used for modularization. In the first version, modularization was based on an analysis of the subprocesses that make up the system, while in the second it was based on information hiding. Parnas then described several change pressures that were likely to occur, and noted that to address one of them the first modularization would require changes to every module. Figure 2.4 shows the two

[3] In the literature, the terms *evolvable*, *modifiable*, *changeable*, and their variants, are used interchangeably.

structures.

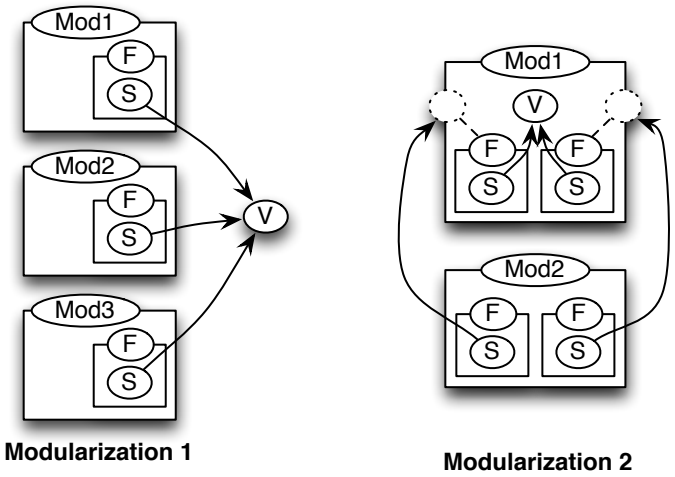

Figure 2.4: Two different modularizations of the *KWIC* system.

Because the first modularization was based on process flow, primary data storage for the program is shared between modules. Since each of the modules contains statements that use the data store directly, changes to the store have a high probability of propagating into the accessing modules, resulting in their destabilization. Even though this system is modularized, its structure provides little protection to the propagation of change. Parnas judged it to be of poorer design because the implementation of the data store is a design decision that has a high probability of change (Parnas, 1972).

In the second modularization, only functions defined within Module 1 have direct access to the data store. All accesses to the store from outside Module 1 are through function declarations, which only provide indirect access to the implementation of the store. The pathways of change propagation from V to external modules include accessing statements and function declarations, and since the latter are an abstraction of the implementation, they have a lower probability of propagating change. In this version, the structure itself acts as an impediment to the propagation of changes made to the data store.

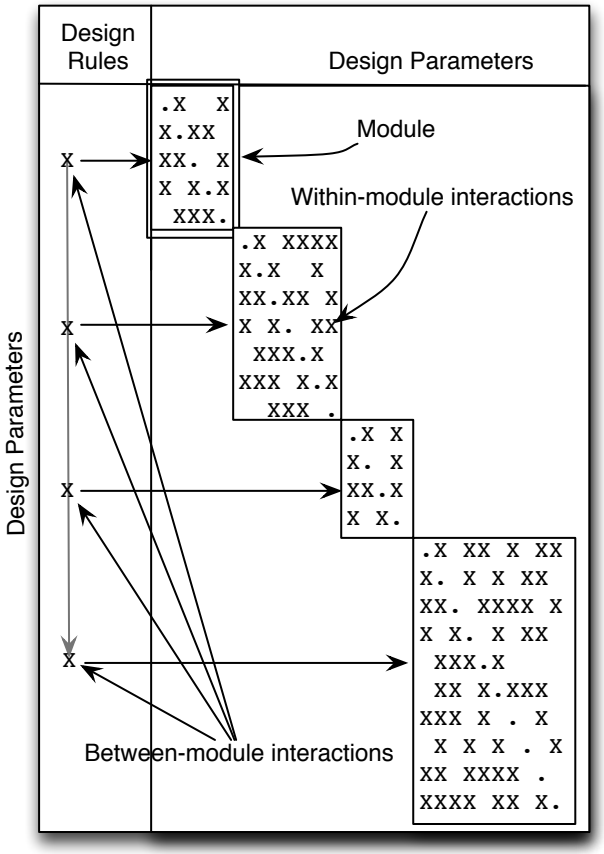

Figure 2.5: Example of a design structure matrix.

2.3.2 Design rules and the design structure matrix

Baldwin and Clark (2000) and Sullivan, Griswold, Cai and Hallen (2001) describe a design tool called the *design structure matrix* (DSM), shown in simplified form in Figure 2.5, which implements a form of information hiding. In a DSM, the interaction of design parameters are shown using an "X", and modules are shown by a rectangle that groups interacting components. Interaction markers within the designated boxes signify interactions between design parameters that are contained within the same module. However, when interactions are required between parameters in different modules, a *design rule* is created. Design rules are shown on the left hand side of the matrix, and provide an indication of those parameters that are visible and cannot be changed without impact on other parts of the system. Parameters that do not appear in design

rules are hidden from the rest of the system and are available for designers to modify freely. DSMs can be viewed as adjacency matrices where modules represent strongly connected design parameters.

When using a DSM, the regulation of between-module interaction is accomplished through the declaration of design rules, which specify the parameters that are not available for modification by designers. This provides a framework within which designers may explore alternative designs without fear that their modifications may have a system-wide impact. Module interfaces have been used as a basis for design metrics as a method for computing *design stability* (Yau and Collofello, 1985; Kelly, 2006), which is primarily used to predict the likely occurrence of the ripple effect (Yau and Collofello, 1985).

2.3.3 Coupling and cohesion

Stevens et al. (1974) introduced the principles of *coupling* and *cohesion* to help system designers minimize the likelihood of the ripple effect caused by between-module interaction. Coupling is defined as "the measure of the strength of association established by a connection from one module to another," where *strength of association* relates to the probability of propagating change. Stevens et al. (1974) cite three factors which effect the strength of association. The first is the complexity of the connection. The second is whether the connection is being made to the module itself, or to something contained inside it. The third is what is being sent or received.

Stevens et al. (1974) propose two methods for reducing the amount of coupling in a system. The first is to minimize the number of between-module connections, and the second is to maximize the number of within-module connections. They believe that maximizing within-module links causes highly interactive nodes to be placed within the same module, which accords with Alexander's approach to modularization (Alexander, 1964). If highly interactive nodes span module boundaries, they are likely to provide a conduit through which change freely propagates between modules. Stevens et al. (1974) call the binding between nodes within a module a measure of the module's *cohesion*.

2.3.4 Discussion

While these principles use different mechanisms, there is commonality between their goals. Each principle proposes methods that try to avoid the ripple effect by minimizing the probability of change propagation between modules, and stresses the use of interfaces as an aid to achieving that goal.

Where the principles differ is that neither information hiding nor DSM are concerned with interactions that occur within the modules that are defined. Only the principles of coupling and cohesion state that there is a relationship between the internal structure of a module and how it relates to other modules. The maxim of "high cohesion and low coupling" is borne out of these principles.

2.4 Measuring cohesion and coupling

While Stevens et al. (1974) provide examples to illustrate cohesion and coupling in source code, their descriptions are qualitative. There are several different ways of measuring both coupling (Selby and Basili, 1991; Briand, Daly and Wüst, 1999b) and cohesion (Briand, Daly and Wüst, 1998). This section provides a brief overview of them.

2.4.1 Measuring coupling

There are two categories of coupling measurement. The first is coupling *frameworks* (Eder, Kappel and Schrefl, 1992, 1994; Hitz and Montazeri, 1995; Briand, Devanbu and Melo, 1997), and the second is coupling *metrics* (Chidamber and Kemerer, 1991; Li and Henry, 1993; Chidamber and Kemerer, 1994; Martin, 1994; Abreu, Goulão and Esteves, 1995; Lee, Liang, Wu, and Wang, 1995). They all use the class as the primary module definition and they all use the interactions between classes as the basis for measuring the strength of association. The differences between the various metrics and metric frameworks are based on interaction categorization, and by the weightings assigned to each category. Where categorization is used, the rationalization is based on the belief that different categories have different probabilities of propagating change.

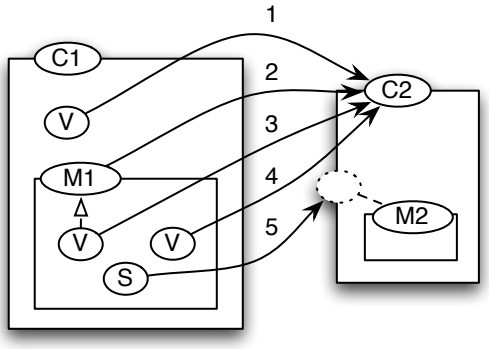

Figure 2.6: Common aspects of coupling measures.

Figure 2.6 shows five different ways that interactions between classes can be categorized. With 1, a variable is defined to be of type C2. With 2, M1 defines C2 as its return type, and with 3, a parameter to M1 is of the type C2. With 4, a local variable in M1 is defined to be of type C2. Finally, with 5, a statement invokes the method M2. In the *Coupling between objects* (CBO) metric, for example, there is no categorization of links, so all links are treated equally (Chidamber and Kemerer, 1991, 1994). However, Eder et al. (1992, 1994) differentiate between method to method, class to class, and class to class interactions through inheritance relationships, so interactions 1 through 4 are considered separately from interaction 5.

Amongst the different metrics and metric frameworks, link categories include:

- Class–attribute, class–method, method–method, inheritance, friend class, and other (Briand et al., 1997).

- Method–method, class–class, class–class through inheritance, which are further differentiated as types of “interaction coupling”, “component relationships,” and “inheritance relationships” (Eder et al., 1992, 1994).

- Class–class and object–object (Hitz and Montazeri, 1995).

- Class–class as client-server relationships (Abreu et al., 1995).

- Method invocations weighted by the number of parameters (Lee et al., 1995).
- Method invocations and variable types (Li and Henry, 1993).
- No categorization (Chidamber and Kemerer, 1991, 1994; Martin, 1994).

Two of the coupling frameworks differentiate between interactions that used interfaces and those that access internal nodes of other classes. Briand et al. (1997) defines subcategories of usage based on interface, "friend" and "other" relationships. Hitz and Montazeri (1995) subcategorize interactions based on "access to interface" and "access to implementation".

Some of the measures consider direction of coupling (Martin, 1994; Abreu et al., 1995; Hitz and Montazeri, 1995; Briand et al., 1997). This is a potentially contentious issue because the direction of coupling implies causality of change propagation. This is best demonstrated by the following statement made by Martin (1994): "Consider again the forces that could make [independent classes] change. They depend upon nothing at all, so a change from a dependee cannot ripple up to them and cause them to change." Martin defines *independent classes* as those that do not depend on other classes, and since they do not depend on other classes, there are no reasons for them to change. However, he does not consider the case where change may propagate from dependent classes. Consider a situation where class A depends on class B, which, in turn, depends on class C. Changes to class B can propagate to class A because A depends on B. However, it is also possible to propagate changes from B to C when, for example, changes to B highlight errors that exist in C.

This scenario is possible when a system is being refactored (Fowler, 1999). Refactoring occurs when programmers want to change a system for the purpose of improving its structure, as opposed to changing its functionality. It is common for programmers to make design decisions with incomplete information about future change pressures, thereby leaving the system's structure deficient in some way (Fowler, 1999). In the above scenario, changes to B may be difficult because of

the complexity of class C's interface. With refactoring, the programmer is encouraged to simplify the relationship between B and C by changing C.

Simon concludes that a necessary criterion for demonstrating causality between two entities is the identification of an asymmetry in their relationship (Simon, 1953). To illustrate this asymmetry, he uses the following example from economics:

Poor weather $\rightarrow$ *Reduced wheat harvest* $\rightarrow$ *Higher wheat prices*

The causal relationship is clear: poor weather can cause a reduction in the amount of wheat harvested, which can result in high wheat prices. The asymmetry is demonstrated when the relationship is written in the reverse order. An increase in wheat prices cannot cause poor weather.

It is not clear that dependency between elements of software (such as classes and methods) is asymmetric in the manner defined by Simon. While there are some cases where a programmer may only have control over one side of a relationship, for example, when he uses libraries, change propagation may occur in either direction, regardless of the direction of dependency.

2.4.2 Measuring cohesion

There are several different ways of measuring cohesion (Briand et al., 1998). There are four categories of cohesion metric: structural (Chidamber and Kemerer, 1991, 1994; Eder et al., 1994; Bieman and Kang, 1995; Hitz and Montazeri, 1995; Lee et al., 1995; Henderson-Sellers, 1996; Briand et al., 1998; Zhou, Xu, Zhao and Yang, 2002), semantic (Etzkorn and Delugach, 2000; Marcus and Poshyvanyk, 2005), program slice (Meyers and Binkley, 2004), and information theory-based (Allen, Khoshgoftaar and Chen, 2001). Since the scope of this thesis is limited to software structure, only structural metrics will be described in detail.

While coupling measures between-module connectivity, cohesion measures within-module connectivity. Alexander's principle for modularization is to group nodes that have high interaction (Alexander, 1964), How-

ever, he did not consider different node categories. Cohesion metrics consider two categories, methods and attributes,[4] and it is their interaction that is measured.

Most of the structural metrics are based on that of Chidamber and Kemerer (1991, 1994), which derive class cohesion from the similarity of its methods. *Lack of cohesion in methods* (LCOM) is an inverse measure: the greater its value the lower the cohesion. Cohesion is computed by constructing the set of accessing methods for each attribute. For n attributes, there will be n sets, and LCOM is the number of disjoint sets that are formed by the intersection of those sets (Chidamber and Kemerer, 1991). This definition was later changed to be the number of pairs of methods that share no common attribute references minus the number of pairs that do (Chidamber and Kemerer, 1994). In the second definition, if the computed value for LCOM < 0, then LCOM = 0.

Hitz and Montazeri (1995) provide a different interpretation of cohesion, and define a measure using directed graphs. A class is a graph of methods that are linked when both share an attribute. The cohesion of the class is its number of connected methods. Hitz and Montazeri (1995) found, however, that accessor methods[5] confounded the metric. To compensate, they added a link between methods wherever one invoked the other.

Bieman and Kang (1995) also recognized the problems with accessor methods, but expanded their solution to include all cases where attributes were accessed transitively. For example, if method A invoked method B, which invoked method C, and C accessed the attribute X, then methods A, B, and C all share the common attribute X. Methods are connected if they share any attribute either directly or indirectly. Bieman and Kang's metric, *tight class cohesion* (TCC), is the percentage of pairs of public methods of a class that share a common attribute.

Henderson-Sellers (1996) defines perfect cohesion when each method accesses every attribute, for which the measure yields the value 0. If each method only accesses a single attribute, then the measure yields 1.

[4] also called "fields," "members," "data members," "variables," and "instance variables".

[5] also called "getters" and "setters".

To compute the cohesion of a class, the percentage of attributes that are access by each method is computed, and averaged over the methods.

Lee et al. (1995) have a slightly different approach to measuring cohesion. The cohesion of a method is defined as number of within-module method invocations, weighted by the number of parameters, and the cohesion for a class is the sum of the cohesion of each of its methods.

While each of these metrics maintain a slightly different interpretation of cohesion, they are all based on on the notion that classes with more internal interaction are more cohesive. Marcus and Poshyvanyk (2005) note that none of the measures is accepted as the standard measure of cohesion.

2.4.3 Empirical validation of cohesion and coupling metrics

There has been little empirical validation of coupling and cohesion metrics against external properties of software (Briand et al., 1998, 1999b; Brito e Abreu and Goulao, 2001), and what has been done is limited in its scope and applicability (Brito e Abreu and Goulao, 2001). Basili, Briand and Melo (1996) and Briand, Morasca and Basili (1999c) investigated the relationship between object-oriented metrics against fault proneness. The study by Basili et al. (1996) only used student subjects, which are not likely to respond as expert programmers would, and that by Briand et al. (1999c) only examined three systems. Chidamber and Kemerer (1994) claim the first validated object-oriented metrics suite, but their validations were not against external properties of software systems, and their investigation only considered two systems. Chidamber, Darcy and Kemerer (1998) concluded that the C&K metrics (Chidamber and Kemerer, 1991, 1994) give significant insight into the impact of object-oriented design decisions, but their validations were against three small systems (<45 classes), one of which comprised only design documentation.

Other investigations include El-Emam, Benlarbi, Goel and Rai (1999), who examined one commercial Java application, Briand, Wüst, Daly and Porter (2000), who examined six systems of limited complexity using non-expert subjects, Bansiya and Davis (2002), who examined a limited set of small systems (< 29 classes), Gyimothy, Ferenc and Siket (2005), who

examined one system, and Subramanyam and Krishnan (2003), who examined one system. Some studies provide only a theoretical validation (Harrison, Counsell and Nithi, 1998; Zhou et al., 2004). Succi, Pedrycz, Djokic, Zuliani and Russo (2005) performed an analysis of colinearity between several metrics and concluded that some measures exhibit a high degree of colinearity.

There have been many calls for more empirical studies in software engineering (Briand et al., 1998, 1999b; Briand, Arisholm, Counsell, Houdek and Thévenod–Fosse, 1999a; Mens and Demeyer, 2001; Kitchenham, Pfleeger, Pickard, Jones, Hoaglin, El Emam and Rosenberg, 2002) and for metrics specifically (Briand et al., 1998, 1999b; Brito e Abreu and Goulao, 2001). Briand et al. (1999a) and Kitchenham et al. (2002) note that when collecting data for software engineering studies it is difficult to control for variables, and Briand et al. (1998) note that coupling and cohesion measures are not defined in terms of explicit empirical models. These factors contribute to the lack of empirical validation of coupling and cohesion measures. More empirical investigations that consider a larger number of systems of greater size are required.

2.5 Random models of complexity

Scientific investigations into random networks began with Solomonoff and Rapoport (1951), who discovered that when the ratio between links and nodes exceeds one, groups of randomly connected nodes abruptly switch from being collections of small disjoint networks to being a single, fully-connected network. Their findings were followed by Erdős and Rényi (1959, 1960), who laid the groundwork for the use of network analysis in scientific investigations. Among their findings, Erdős and Rényi (1959) discovered that many of the properties found in randomly generated graphs appear suddenly, and this transition is related to the ratio between the number of nodes and links (Erdős and Rényi, 1960).

Erdős and Rényi (1960) defined a general method for generating random networks so that they could be used to model complex phenomena. Their method requires two parameters: the number of nodes n and the

number of outgoing links *k*. Construction of the graph begins with the creation of *n* nodes. Then, for each node, *k* links are associated with randomly chosen nodes.

When using random networks for empirical analysis, Barabási and Albert (1999) discovered that networks that were generated using Erdős and Rényi's process do not accurately reflect the structure of networks that they observed in real networks. The main difference is that the distribution of frequency of occurrence against node degree for Erdős and Rényi networks is a Poisson distribution (Watts and Strogatz, 1998; Barabási and Albert, 1999; Newman, Strogatz and Watts, 2001), whereas for real networks it is a power-law distribution (Watts and Strogatz, 1998; Barabási and Albert, 1999; Newman, 2005).

2.5.1 Power-law distributions

In networks that possess a power-law degree distribution, the probability that a node x has the degree $\deg(x)$ is proportional to $\deg(x)^{-\alpha}$ where $\alpha > 1$:

$$p(\deg(x)) = C\ \deg(x)^{-\alpha}, \tag{2.1}$$

for some normalization constant C. In most power-law distributions encountered in practice, $2 \leq \alpha \leq 3$, but this is not always the case (Clauset, Shalizi and Newman., 2009). From such distributions, two key connectivity characteristics emerge:

1. The mean connectivity is low relative to the range because the distribution is *right-skewed*.

2. The range of connectivity has the potential to be several orders of magnitude greater than the mean, depending on the size of the network.

These properties suggest that most nodes in the system have low connectivity, but there will be nodes that exhibit high degrees of connectivity with respect to the mean; these nodes reside in the *long tail*[6] of the distribution. These highly connected nodes are called *hubs*. Networks

[6] sometimes called *heavy tail*.

with a power-law degree distribution are called *scale-free* (Albert, Jeong and Barabási, 1999; Barabási and Albert, 1999), due to the fact that they are self-similar at all scales.

2.5.2 Generating random power-law networks

While several network generation processes result in power-law degree distributions (Simon, 1955; Keller, 2005), the most commonly used in network analysis is the *preferential attachment* model (Newman, Barabási and Watts, 2006). This differs from Erdős and Rényi's method in two ways. First, the network evolves through the addition of nodes and links. Second, links between nodes are not random, but are instead based on a preferential attachment function. It is also called the BA model, named for those who first proposed it (Barabási and Albert, 1999).

When a new node is added to the network, the probability that it will attach to i is:

$$p_i = \frac{k_i}{\sum_j k_j}, \tag{2.2}$$

where k_i is the degree of node i and $\sum_j k_j$ is the sum of degrees of each node in the network.

The power-law degree distribution emerges for two reasons. First, the preferential attachment function defines the probability of attachment as directly proportional to the number of links that a node already has. Second, unlike the Erdős and Rényi model, the size of the network evolves rather than being fully defined at the start. Both factors contribute to a probability function that is non-uniform across the nodes in the network. The resulting network is one where nodes that have been in the network the longest are likely to have the greatest number of links (Barabási and Albert, 1999; Albert et al., 1999). This is often summarized as "the rich get richer" (Newman et al., 2006).

2.5.3 Small-world networks

Another network property that often accompanies scale-free structure is called *small-world* (Watts and Strogatz, 1998). A network has this property if its nodes are highly clustered and the average path length between

nodes is small. The small-world phenomenon is best known through the work of Milgram (1967) and Travers and Milgram (1969), who sent letters to random people and asked them to forward them to a specific person. If the subject did not know the intended target, they were instructed to forward the letter to someone whom they thought might know the target. Although there was a small response rate, Travers and Milgram found the number of steps between a randomly selected subject and a specific target to be 6. Since the path chosen was not necessarily optimal, they concluded that the actual average distance was less.

In a small-world network, the small average path length enables information to be shared between all nodes in a manner that minimizes direct connectivity. Two nodes need not have a direct connection if they share a neighbour through which they can pass information.

2.6 The ripple effect in scale-free and small-world networks

The scale-free and small-world properties have implications for the ripple effect.[7] A complex network that is scale-free has hubs, which, if changed, have the ability to propagate change to a large number of nodes. In a small-world network, the propagation of change has a greater chance of being system-wide because of the short average path length (Monasson, 1999; Pandit and Amritkar, 2001). This has lead to the rise of the *SIR* model in epidemiology (Ball, Mollison and Scalia-Tomba, 1997a).

2.6.1 The Susceptible, Infective, Removed model

While the focus of the SIR model is disease spreading through a population, it can be applied to any network propagation problem. In it, nodes can be in one of three states: *susceptible*, *infective*, or *removed*. Susceptible nodes are those which have not yet caught the disease; infective nodes are those that have caught the disease and are able to pass it along to others; and removed nodes are those that have either died or recovered, and cannot further propagate the disease.

[7] Also called *Percolation* and *Diffusive propagation* in network analysis.

When researchers used the Erdős and Rényi (1960) model to simulate disease propagation, they discovered that their models did not behave like actual epidemics (Ball et al., 1997a). They attributed the observed differences to the fact that Erdős and Rényi networks are neither scale-free nor small-world. In such networks, the Poisson degree distribution means that the degree is similar for all nodes, so the probability of infection propagation is closely related to the number of infective nodes. However, in a scale-free network, the probability of propagation from infective to susceptible nodes is based on the number of links between those node types, and due to the presence of hubs, this can be considerably larger than the number of infective nodes. Because it is generally easier to estimate the number of infective nodes than the number of links between infective and susceptible, this can produce unexpected levels of propagation within the network (Bailey, 1975). This effect is also called *the tipping point* (Gladwell, 2002).

If software systems are scale-free and small-world networks, the SIR model has implications for the propagation of change and the ripple effect. While most nodes would have low connectivity, hubs would be present and could facilitate propagation to large numbers of nodes. The small average path length of the network guarantees that each node is only a few steps from a hub, which means that changes to any node could conceivably have system-wide effects. It must be noted that in the SIR model, nodes gain immunity, which prevents further propagation of the disease. In a software system, nodes may be "re-infected", and cause further change propagation.

2.6.2 Empirical studies

Investigations into the ripple effect in software are limited to simulations (Yau and Collofello, 1985; Tsantalis, Chatzigeorgiou and Stephanides, 2005; Sharafat and Tahvildari, 2007; Li, Qian and Zhang, 2009). Yau and Collofello (1985) derive stability measures based on assumptions about existing software systems, and the computation of a "potential ripple effect" that occurs as a consequence of modifying a module. Tsantalis et al. (2005) propose to measure change proneness by estimating which mod-

ules will be affected when new functionality is added, and validate this against two systems. Sharafat and Tahvildari (2007) compute change proneness based on dependencies extracted from UML diagrams and source code, and validate their approach against one system. Li et al. (2009) propose a design evaluation metric that constructs a simulation of change propagation from existing software, which is evaluated using small samples.

Other simulations include *impact analysis*, which computes the impact of changing some aspect of a software system (Black, 2001, 2006, 2008; Abdi, Lounis and Sahraoui, 2009). Black (2001) describes how impact analysis can be used to predict which modules change when a variable is modified. She demonstrates that her technique correlates with Yau and Collofello's (1985) measure (Black, 2006), and also correlates her measure against the performance of a human subject (Black, 2008). Abdi et al. (2009) propose a probabilistic method of impact analysis that relies on Bayesian networks, and validate their approach against a single system.

All these techniques focus on predicting the effects of software modification. They are designed to help software maintainers understand the impact of proposed changes before those changes are made. None of these studies examine the history of software systems to determine if, indeed, a ripple effect has occurred.

2.6.3 Change propagation through inferred links

Typically, change propagation occurs through links between modules that have been established formally. However, there is also interest in considering the propagation of change that occurs through inferred (Cubranic and Murphy, 2003) or hidden (Gîrba, Ducasse, Marinescu and Raţiu, 2004) dependencies. This form of change propagation is called *co-change*, and links are inferred using a variety of techniques that identify common records of change called *modification records* (German, 2006). Using source code repositories Zimmermann, Weißgerber, Diehl and Zeller (2004) and Ying, Murphy, Ng and Chu-Carroll (2004) identify hidden links by considering the temporal aspects of code *commits*.

Changed files that are committed to the repository at the same time are deemed to be linked. Cubranic and Murphy (2003) identify hidden links by examining Bugzilla[8] reports, online developer forums and usenet newsgroups, developer emails, and design documents. Modules that are identified in the same container (e.g. email) are deemed to be related. Hassan and Holt (2004) devised seven heuristic and pruning techniques based on entity, developer and process data to infer links. Gîrba, Ducasse, Kuhn, Marinescu and Daniel (2007) used *concept analysis*, which is grouping entities based on common properties, as their basis for identifying hidden links.

With all of these techniques, the presence of a link between nodes may be coincidental. For example, some developers do not commit code after each small change, but rather commit code changes in larger groups. In the latter case, links would be inferred between classes that did not change in tandem. Contrast this with networks generated from source code where all links are formally declared and verified to satisfy a programming language grammar. Although ripple effects may propagate through inferred links, the analyses performed in this thesis are based on formally declared links. Instances of change propagation due to informal association are considered to be out of the scope of this thesis.

2.7 Network analysis of software

Several investigations into the structure of software systems have revealed the presence of power laws and other long-tailed distributions. Wheeldon and Counsell (2003) examined power laws in the class coupling relationships in three industrial systems for the purpose of using power-law distributions to predict coupling patterns. They examined five different class-coupling relationships (inheritance, interface, aggregation, parameter type, and return type) and concluded that not only does each have a power-law distribution but that the relationships are independent. Wheeldon and Counsell do not consider coupling as a result of method invocation, and perform no analysis beyond the class level.

[8] www.bugzilla.org

Myers (2003), Marchesi et al. (2004), Potanin, Noble, Frean and Biddle (2005), and Gao, Xu, Yang, Niu and Guo (2010) observed power-laws in both the in-degree and out-degree distributions of modules in 26 different software systems. Baxter, Frean, Noble, Rickerby, Smith, Visser, Melton and Tempero (2006) examined 56 systems for a large set of measures including some coupling measures, but considered them independently from one another. They observed log-normal out-degree distributions, and some specific coupling measures did not match a long-tailed distribution in some instances, hinting at a lack of universality. Jing, Keqing, Yutao and Rong (2006) found power laws in two measures, weighted methods per class (WMC) and coupling between objects (CBO), for four open-source software systems. Concas, Marchesi, Pinna and Serra (2007) examined ten properties of three software systems and found them to have both Pareto (1897) and log-normal distributions. Ichii et al. (2008) examined four measures (including two variants of WMC) on six systems, finding that in-degree follows a power law while out-degree follows some other heavy-tailed distribution. Louridas, Spinellis and Vlachos (2008) found power laws present in the dependencies of software libraries, applications, and system calls in the Linux and FreeBSD operating systems, and concluded that they are ubiquitous in software systems.

None of the aforementioned investigations considered software systems at the level of statements and variables, limiting the generality of the findings. Some of the investigations did not explicitly plan to investigate coupling. Myers (2003) considered only inheritance and aggregation relationships. Concas et al. (2007) focused mostly on size measures, but did include a count of method invocations between classes, which they found to conform to a power law; however, they did not examine other forms of coupling. Gao et al. (2010) considered method–method interaction, excluding other class-level coupling measures.

Hyland-Wood, Carrington and Kaplan (2006) examined coupling relationships at differing levels of granularity (package, class, and method level, but not statement level) for two separate open source projects over a 15-month period and concluded that scale-free properties were present at all levels of analysis for each snapshot, although they note that these

properties were approximate in most cases. While demonstrating the relationship of scale-free structure between differing levels of granularity, this study's lowest level of analysis was that of methods.

Vasa, Lumpe, Branch and Nierstrasz (2009) noted that many software metrics have a skewed distribution, which makes the reporting of data using central tendency statistics unreliable. To address this, they recommend using the *Gini coefficient*, which has been used in the field of economics to characterize the relative equality of distributions. They examined 46 systems on a variety of measures, two of which (in-degree count and out-degree count) are related to coupling. Their findings appear to support those of Myers (2003) and Gao et al. (2010) that in-degrees and out-degrees have differing distributions. However, they do not address the structure of software at the source code level.

Some of the investigations had confounding factors, which makes them difficult to directly compare with this work. Marchesi et al. (2004) examined classes in Smalltalk systems, but issues of dynamic binding prevented precise resolution of between-module interactions. To circumvent this, dependency relationships that could only be resolved at runtime were approximated using a weighting function, but it is not clear what effect this transformation may have had. Potanin et al. (2005) investigated object graphs, which are not directly comparable to class graphs. For example, collection objects may have many runtime associations that are undetectable through static analysis. Similarly, the number of instances of each class could skew the total degree distribution, because classes with higher numbers of instances would have greater weight in the analysis. It is not clear that scale-free structure in an object graph translates to scale-free structure in its corresponding class graph.

Valverde, Cancho and Sole (2002) and Jenkins and Kirk (2007) note that hubs fall into "the set of bad design practices known as antipatterns" (Koenig, 1995). However, they fail to observe that the ubiquitous presence of heavy-tailed distributions implies the presence of hubs.

2.8 Preferential attachment and source code evolution

While it has been observed in the literature that various levels of software structure are scale-free, none of the investigations has examined software at the statement level (Wheeldon and Counsell, 2003; Myers, 2003; Marchesi et al., 2004; Potanin et al., 2005; Baxter et al., 2006; Hyland-Wood et al., 2006; Concas et al., 2007; Ichii et al., 2008; Louridas et al., 2008; Gao et al., 2010). Analyses are typically performed at the class level (Wheeldon and Counsell, 2003; Myers, 2003; Marchesi et al., 2004; Potanin et al., 2005; Valverde and Sole, 2005; Ichii et al., 2008; Gao et al., 2010) and, in some cases, as low as the method level (Hyland-Wood et al., 2006). However, it seems reasonable that if scale-free structure is observed at the class and method levels, then it is derived from the underlying interactions between variables, statements and methods. In order to formulate a hypothesis about scale-free structure at those levels, there needs to be reasonable justification as to why one would expect such a structure to evolve. As a starting point, we consider the preferential attachment model.

Several criticisms have been brought forward regarding the BA model, especially as it applies to software systems. As a general criticism, Keller notes that the presence of a particular degree distribution does not identify the process that was actually used to form the network Keller (2005). Simply stated, knowing the result does not explain how the result was achieved. She provides several examples of different network evolution processes, all of which result in power-law degree distributions. The point of her criticism is that one should not accept a network evolution model simply because it generates the expected distribution. The process described by the model must be relevant to the kinds of processes that are at work as the system evolves.

Valverde et al. (2002) suggest that the BA model is not relevant to software development because “no design principle explicitly introduces preferential attachment.” They offer an alternative model based on design principles that minimize path length between nodes. However, the

distance between source-code nodes remains largely hidden from the programmer, so it is unlikely that such a criteria would be used by programmers because they would have to perform additional analysis in order to optimize this property. Design principles themselves are generally considered to be rules of thumb, and are most often incomplete, so the lack of a specific design principle espousing preferential attachment does not mean that it does not occur in some form.

Myers (2003) dismisses the BA model because it does not generate hierarchical structures like those found in software systems. Alternatively, he suggests that scale-free structure arises from continuous refactoring. However, not all software undergoes non-trivial refactoring, so we would expect to find systems that do not exhibit scale-free structure, which is not congruent with findings in the literature (Wheeldon and Counsell, 2003; Marchesi et al., 2004; Potanin et al., 2005; Baxter et al., 2006; Hyland-Wood et al., 2006; Concas et al., 2007; Ichii et al., 2008; Louridas et al., 2008; Gao et al., 2010). What Myers does not consider is a model based on preferential attachment where hierarchical structure is imposed upon the generation process from an external source, such as a programming language grammar. When a programmer edits source code, he adds new code into a program file. The associated network nodes are inserted into the system hierarchy through a translation process that is based on the rules of a program language grammar, which define a hierarchical structure. If the newly added source code does not comply with the syntactic requirements of the language, it is rejected and any associated nodes are not added to the system until those requirements have been met. In short, new nodes cannot be added to a system without explicit placement within the system's hierarchy. This constraint enforces preferential attachment, based on grammar, between parent and children nodes, and does not preclude some form of preferential attachment between nodes based on usage.

Jenkins and Kirk (2007) state that "preferential attachment relies on newly added nodes having prior knowledge of the rest of the network, which seems implausible since software is build in pieces from a series of sources using various rules for design patterns which do not apply to

the finished software graph." It is difficult to imagine how a programmer could add any new nodes to a software system without having some prior knowledge of at least part of that system. Jenkins and Kirk imply that the entire network must be known in order for a programmer to add new nodes, but this position is clearly false. To add new nodes, programmers only require knowledge about the location in the network where they are to be attached. More knowledge about the structure of the network might enable programmers to make better decisions, but programmers can and do make mistakes.

Chen, Gu, Wang, Chen and Chen (2008) described a modified version of the BA model with a parameter that made it less likely that new nodes will attach to existing nodes that were contained in other modules. In this model, they fail to consider how modules themselves are added, deleted or modified. Furthermore, they validate their model against a single system, and that limits its generalizability.

Strangely, Li, Han and Hu (2008) accept the BA model wholeheartedly as a model of software system evolution without considering the criticisms raised in the literature. It seems unlikely that nodes newly added to a software system network are preferentially attached to existing nodes based solely on the number of links that that the existing nodes already have.

Research addressing the question of how scale-free structures evolve in software is focused on class-level structures, and ignores the fact that scale-free structure may emerge at the class level because the underlying source-code network is also scale-free. Classes are modules made up of class declarations, variable declarations, method declarations, blocks and statements. It is these underlying nodes that programmers modify directly, so it seems reasonable to examine the evolution of those nodes to determine how scale-free structure might emerge.

2.9 The matching problem

To address questions about how software evolves, it is necessary to have a method by which software systems, in their various states of evolu-

tion, can be observed and compared. To illustrate the problems associated with this process, consider the following scenario. A programmer is asked to compare two versions of the same software system in an attempt to ascertain what changes have been made. Because of the size and complexity of the source code, direct comparison is not feasible. Conveniently, however, the system is broken down into modules called classes, which are smaller and simpler structures upon which comparisons can be made. The programmer thinks that matching classes between releases will be fairly easy, because each can be uniquely identified by name.

He soon recognizes, however, that the names of some of the classes change from one version to the next, which makes matching classes based solely on their names unreliable. To address this problem, he adds class structure as a matching criteria, only to discover that many classes have similar structure because of inheritance or interface obligations, or from having been created from similar existing code structures. Upon further analysis, he also recognizes that methods and variables that were previously in one class have been moved to another class. In some cases, classes have been broken into three or four classes, of which each contains parts of the original class along with newly added and unrecognizable code. The resulting classes are so different from the original that it's difficult to decide whether the original class continues to exist, even though identifiable pieces remain in the code base.

This scenario illustrates some of the difficulties associated with comparing different versions of software. A process that appears simple is complicated by the fact that the properties that are used to match classes between versions are themselves subject to change. The process is further confused when parts of classes that have been removed remain in the system as parts of other classes, making it difficult to ascertain that the removed class was actually removed. This process relies on the judgment of the person making the comparison, and if performed by different people, the results are likely be different for the difficult cases.

The matching problem is common to research in software evolution (Tu and Godfrey, 2002) and there are several different approaches. *Origin Analysis* is an approach that has been used to study the evolution of both

software architecture and the evolution of individual system components (Tu and Godfrey, 2002), and to track the merging and splitting of source entities across software releases (Godfrey and Zou, 2005). Origin Analysis is based on *Bertillonage analysis*, which is a process that was used by the French police prior to the time when fingerprinting came into common usage. In Bertillonage analysis, measurements of various body parts were used to reduce a large number of possible matching people to a small set of likely candidates, and Origin Analysis applies the same technique to software artefacts.

Origin Analysis makes use of *matchers* to measure individual characteristics of software. Tu and Godfrey (2002) use the measures described by Kontogiannis (1997), and Godfrey and Zou (2005) define their own matchers—name, declaration, metrics, call relation, and expression—to detect code merging and splitting. These measures are used to compare classes between different system versions so that the number of possible matching classes can be reduced.

Origin Analysis also uses *dependency analysis*. For entities that cannot be matched using the measured properties of Bertillonage analysis, an analysis of dependencies—both incoming and outgoing—is used. Specifically, if an entity A_{old} does not match its new counterpart A_{new} using matchers, the *callers* and *callees* of A_{old} are cross referenced against classes in the new version, which is likely to identify A_{new} as the most suitable candidate.

Origin Analysis ultimately relies on human experience to match difficult cases, thereby limiting it to be a semiautomatic process. Its semiautomatic nature makes it suitable for software developers who evaluate their own software systems, and for researchers in software evolution who analyse a small number of systems, but does not scale well to a large corpus like Qualitas.

Another approach, used by Demeyer, Ducasse and Nierstrasz (2000) uses change metrics to identify code refactorings between versions. A series of assumptions about refactoring are used with a set of class measures to classify splitting and merging within superclass/subclass structures, or splitting and merging between sibling classes under common

superclasses. Demeyer et al. admit that their technique is unreliable when classes are renamed, and when different kinds of change are applied to a single entity. Their approach is also semiautomatic, because a human must inspect the results to ensure that the identified refactorings actually occurred.

Clone detection offers a valuable source of techniques for matching classes: a version of a class in one release can be viewed as a clone of the class from a previous release. Baxter, Yahin, Moura, Sant'Anna and Bier (1998) present a method of detecting code clones by subjecting program ASTs to a combination of algorithms that detect subtree clones, sequences of subtree clones, and near-miss clones—those that are nearly identical. Baker's (1995) *dup* and *CCFinder* by Kamiya, Kusumoto and Inoue (2002) are methods that match code clones through lexical analysis. Finally, Komondoor and Horwitz (2001) identify code clones by identifying similar program dependency subgraphs.

A weakness of these solutions is their use of a similarity measure. When something is changed, its similarity to the original decreases, rendering it less likely to be matched. The likelihood that there will be a failure to match is proportional to the amount of change that has occurred.

To compensate for this weakness, multiple similarity measures are used, each measuring an independent aspect of the entities that are being compared. This approach relies on the assumption that no matter how much an entity changes, it will only be subjected to one or two types of change at one time. If a class has five properties, for example, then changes to two of them will not likely result in match failure because three of the measures will be unaffected. However, if all five properties change at the same time, then it is likely that the class will fail to match.

2.10 Summary and discussion

This chapter began by introducing of a model for complex systems, which is based on graph theory. Design theory research states that the evolvability of a complex system is based, in part, on the level of intercon-

nectivity between its nodes. Greater interconnectivity results in higher levels of change propagation, which makes the system less stable when modified. In the worst case, small changes can result in large-scale change propagation, which is known as *the ripple effect*. To reduce the overall level of node interconnectivity, modularization is used.

For modularization to be effective, it must not only reduce the amount of node interconnectivity, but also the probability of change propagation between modules. In software, regulation of between-module interaction is accomplish using abstract proxy nodes, which present a generalized concept of the utility of the module. Changes that occur within the module can be contained provided that their new configuration remains faithful to the concept presented by their proxy.

The use of modularization and abstraction are observed in software design principles. Three different design principles were presented. From one is derived the maxim of "high cohesion/low coupling," which states that minimizing between-module interaction is accomplished by maximizing within-module interaction. There are several metrics that purport to measure coupling and cohesion, but they remain largely unvalidated against external properties of software, such as evolvability.

Analysis of networks reveals two properties that are common to complex systems: scale-free and small-world. Scale-free structure dictates that high connectivity is present, and small-world structure ensures that the average distance between nodes is small. The application of network analysis to software systems has revealed that scale-free structure is ubiquitous.

This presentation reveals a clear gap in the literature. The literature on coupling metrics and its validation make no mention of scale-free structure and its attendant high connectivity, and the literature that applies network analysis to software makes no connection between the observed scale-free structure and high coupling. Furthermore, the application of network analysis techniques to software have determined that scale-free structures are ubiquitous. If this is indeed the case, then all software must contain high coupling, which makes it susceptible to the ripple effect and limits evolvability.

2.10.1 Research questions revisited

Chapter 1 presented three broad research questions. Based on the literature review presented in this chapter, these questions can be expanded.

The first question asks whether software is scale-free. The literature presented in Section 2.7 observes that many aspects of software systems are scale-free, but does not attempt to ascertain how these structures evolve. Section 2.8 presents models of how scale-free structure might evolve in software, but those investigations only consider scale-free structure in the relationships between classes, and they present no empirical validation. Classes, however, are an artefact of modularization, and their interdependence is based on the interactions of the nodes they contain. To understand how scale-free structure evolves in class dependencies, I believe it is necessary to study interactions below the level of the class. This research considers the interactions of variables, statements and methods.

The second question asks whether the presence of scale-free structure results in high coupling. Section 2.3.3 introduced the concepts of coupling and cohesion, which differentiate between within-module and between-module links. The presence of scale-free structure implies high connectivity, but it is possible that nodes with high connectivity may resolve their links within their same module. If this is the case, then areas of high connectivity may not necessarily equate to high coupling.

The third question asks whether the ripple effect can be observed in the change history of systems that contain high coupling. Section 2.6.2 shows that the literature concerned with studying the ripple effect in software focuses on predicting the effects of changes to software. None of the literature presented defines a mechanism for ripple identification in software version histories, so a technique for identifying the effect must be developed.

While a network with scale-free structure is strictly defined as having a power-law degree distribution (Barabási and Albert, 1999), many researchers note that the power-law isn't the only distribution that exhibits a long tail (Simon, 1955; Keller, 2005; Clauset et al., 2009). The goal of this research is not to replicate the findings of Potanin et al. (2005);

Baxter et al. (2006); Concas et al. (2007); Louridas et al. (2008); Hatton (2009), which showed the presence—and in some cases, absence—of power-laws in software systems, but rather to investigate the effects of long tailed distributions on coupling and overall software system design. In this thesis, the term *scale-free* is used in a manner that is inclusive of other distributions that exhibit similar connectivity properties, which includes power-law with cuttoff, exponential, and log-normal distributions.

Research questions 1 and 2 are addressed in Chapter 4. Question 3 is addressed in Chapters 5 and 6.

Chapter 3

Tools

> I think it's the tragedy of our time that we're not aware of the effect of the manner in which we've adopted tools. Those tools have become who we are.
>
> —Godfrey Reggio

This chapter describes the tools created to address the research questions raised in Chapter 1. It begins with a conceptual overview of the four key tools that are used, followed by a detailed description of each one. In essence, the tools provide the ability to convert a wide variety of software systems from source code form into a directed graph form. This form is then used to address research questions regarding the structure and evolution of the software systems. The chapter closes with a discussion of the key challenges that had to be overcome in order to complete the research described in this thesis.

3.1 Conceptual overview

Figure 3.1 illustrates the tools used in this research: The corpus of software systems, the parsing subsystem, the CodeNet subsystem and the analysis subsystem. The software systems that provide the empirical data are supplied by the Qualitas corpus. These systems are parsed into *Abstract Syntax Trees* (ASTs) using the Eclipse AST Parser (Eclipse Foundation, 2011) and stored in intermediary files called *parsefiles*. Parsefiles are translated into graph form by the CodeNet subsystem and the resulting graphs are used by the analysis subsystem. Communication between

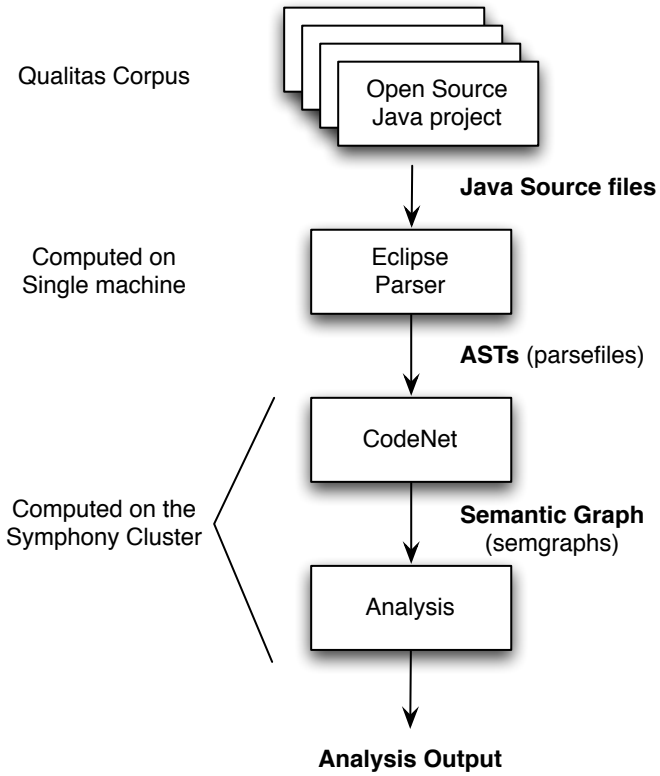

Figure 3.1: Toolset architecture.

the CodeNet and analysis subsystems is facilitated using an intermediary data structure called a *semgraph*. The name is chosen because a semgraph is a directed graph structure that retains semantic information obtained using the AST parser. For each system in the corpus there is single parsefile and a single semgraph file, which are maintained in a human readable form.

The use of these files provides several advantages over a streamed process. First, they can be examined and tested for correctness, which allows the quality of the data to be assessed through every step of the process. Second, separating the intermediary stages of processing allows easier deployment within the Symphony High-Performance Computing cluster (University of Waikato, 2011). Finally, the files can be shared with other researchers, making the results more reproducible.

3.2 Corpus of software systems

Qualitas is a corpus of open-source software systems written in Java that has been made available to researchers for analysis (Tempero et al., 2010). It was selected as a basis for this research for two reasons:

1. It provides for the investigation of both general properties of software and how systems evolve over time.

2. It is beginning to be widely used in research, which facilitates reproducibility.

The corpus comprises two releases labelled “r” and “e”. The “r” release contains a single version of many systems, and is intended for research that investigates general characteristics of software systems. The “e” release contains numerous versions of a smaller number of systems, and is intended for investigations that consider the evolution of software systems over time. This research investigates both general software properties and how systems evolve over time, which makes the corpus well suited.

The authors of the Qualitas corpus cite six criteria for inclusion (Tempero et al., 2010):

- Systems that were present in previous releases of the corpus are present in future releases so that researchers need not maintain multiple versions of the corpus as it evolves.

- All systems are written in the Java programming language (Gosling et al., 2005).

- The systems are distributed in both source and binary form. This is because of the difficulty of building each of the systems individually. The assumption is made that the binary form distributed by system authors is derived from the source code that is provided.

- A given system’s binary form is distributed as a set of jar files. This limitation is imposed by the set of management tools used to create the corpus.

- The system must be available to anyone, independent of the existence of the corpus. This criterion allows external researchers the ability to test the corpus for issues of quality.
- The contents of the system must be easily identifiable. This criterion serves to exclude systems for which the distribution contains significant amounts of irrelevant content.

These criteria focus on issues of managing a large research data set and do not involve properties of the systems themselves. Systems were not chosen because they address particular kinds of problems or because their implementation was particularly good (or bad). For this reason, one must be cautious about drawing conclusions based solely on their inclusion in the corpus. For example, one should not conclude that the systems are well designed simply because they are included in the corpus.

3.3 The Eclipse AST parser

Eclipse (Eclipse Foundation, 2011) is an integrated development environment that is commonly used for Java development in industrial settings. To facilitate the creation of plugins that can aid the development process, it contains sophisticated source code analysis tools, which have also been used by researchers to perform analyses on source code (eg. Holmes, Walker and Murphy, 2005). Because these tools are used by a broad base of users for industrial strength software development and for research into software systems, they are reasonably robust and free from error. This research utilizes the AST parser contained within the Eclipse system.

To parse the systems in the Qualitas corpus, they are first loaded as individual projects into Eclipse under its main *workspace*, and each system is configured to be fully recognized. Minimally, this requires that the CLASSPATH variable is set to include all necessary jar files and that the correct Java compiler version number is set. After configuration, the systems are parsed using a small initiator program that is embedded into Eclipse's plugin framework. This program iterates through all Java

projects stored in the Eclipse workspace, and for each one, retrieves individual Java source-code files and parses them using the AST parser. Once an AST is obtained, its structure and semantic information is written to the appropriate parsefile.

3.3.1 Parse file format

Parsefiles are tag-based text files that contain an AST for each *compilable unit*[1] in the corresponding software system. The tags used are shown in Table 3.1. The hierarchical structure of the AST is encoded using *begin* and *end* tags in a manner that is similar to the XML specification (Bosak, Bray, Connolly, Maler, Nicol, Sperberg-McQueen, Wood and Clark, 1998). Data between these two tags are attributes of the corresponding element. Similarly, pairs of *begin*/*end* tags that are enclosed within the *begin* and *end* tags of another element are children of the enclosing element. For tags that are specified with parameters, the tag name and parameters are separated by a *space* character.

The entities that are extracted from the source code include class declarations, variable declarations, method declarations, statements, enumeration declarations, enumeration constant declarations and expressions. Semantic information, like an entity's name and type, or the name of a method that it invokes, is encoded using attribute tags. To reduce space requirements, semantic information is limited to whatever is deemed relevant to this research, but discarded information could be reintroduced into the structure, if required, for future research purposes.

Tag Name	Parameters
project	name
module	path
entity	category, definition
attribute	name, data
end	category

Table 3.1: Parsefile tags

[1] This includes public classes, non-public classes and interfaces. These are abstractly represented by the *CompilationUnit* interface in the Eclipse Java Development Kit.

The *project* tag must be the first tag in a parsefile and may only be used once. It signifies to the CodeNet system that the parsefile does not contain extraneous information at the start of the file. This tag requires a name parameter, which is used internally by CodeNet to differentiate multiple systems that are loaded for comparison purposes. CodeNet returns an error when an attempt is made to define two projects with the same name.

The *module* tag is used to specify the namespace for an AST. The decision to use this name was made early in the development of CodeNet. However, recall from Section 2.1.2, a *module* is any formalized grouping of nodes, and is not limited to refer only to namespaces. Ideally, this tag will be renamed to *namespace* in future versions of this research. The namespace is written in the same way as for the *package* keyword in the Java programming language.

The *entity* tag represents the start of an entity definition. It requires one parameter: the declaration of the node's category. Acceptable categories include block, class, enum, enumConstant, expression, methodDeclaration, statement, variable and expression. To aid debugging, the tag provides an optional parameter called *definition*, which allows the programmer to trace an *entity* tag to its point of origin should the entity be found to contain an error.

The *attribute* tag requires two parameters, which specify the name of the attribute and its associated data. Attributes are name/value-list pairs: each name maps to a list of values. Multiple values are defined by specifying several attribute tags with the same name. For example, a method definition that has both "public" and "abstract" modifiers would have each assigned as an attribute of the same name. Because each tag within a parsefile must be defined on a single line, attributes that contain newline characters in their data must have those characters escaped as "\n".

The definition of an entity ends when its corresponding *end* tag is reached. Any attributes contained between an entity and its corresponding end tag are assigned to that entity. Entities defined within another entity/end pair are children of the containing pair. To facilitate debug-

ging, the end tag has a single parameter, which is the category of the node to which the end tag is applied. The category defined by each corresponding *entity* and *end* tags must match, or the CodeNet system will signal an error. This ensures that the tags are correctly balanced and that no errors have been introduced into the parsefile.

3.3.2 Abstract syntax tree example

To illustrate how an AST is translated into a parsefile, this section shows an example Java program (Figure 3.2), its AST (Figure 3.3), and the resulting parsefile (see Appendix A). To simplify the diagram, some of the information in the AST is not shown. In Figure 3.3, entities that are identified by the AST are represented as ovals, and the text in each oval specifies the entity's category and starting location within the source code listing. To aid readability, attributes are written beside each oval.

```
1  package org.afox.codenet;
2
3  import java.util.HashMap;
4
5  public class Entity {
6          private String name;
7          private HashMap attributes;
8
9          public Entity(String name) {
10                 this.name = name;
11                 attributes = new HashMap();
12         }
13 }
```

Figure 3.2: Code listing 1.

In this example, line 1 provides the hierarchical package definition (namespace) within which the remaining code resides. The code defines a single class (line 5, called *Entity*) that contains two instance variables (lines 6 and 7, called *name* and *attributes*). The class also contains a

single constructor declaration. It (line 9, called *Entity*) has a single parameter (called *name*) and contains a block with two statements (lines 10 and 11). The import statement (line 3) does not contribute to the structure of the AST, but allows the programmer to refer to the class "java.util.HashMap" by its simple name "HashMap".

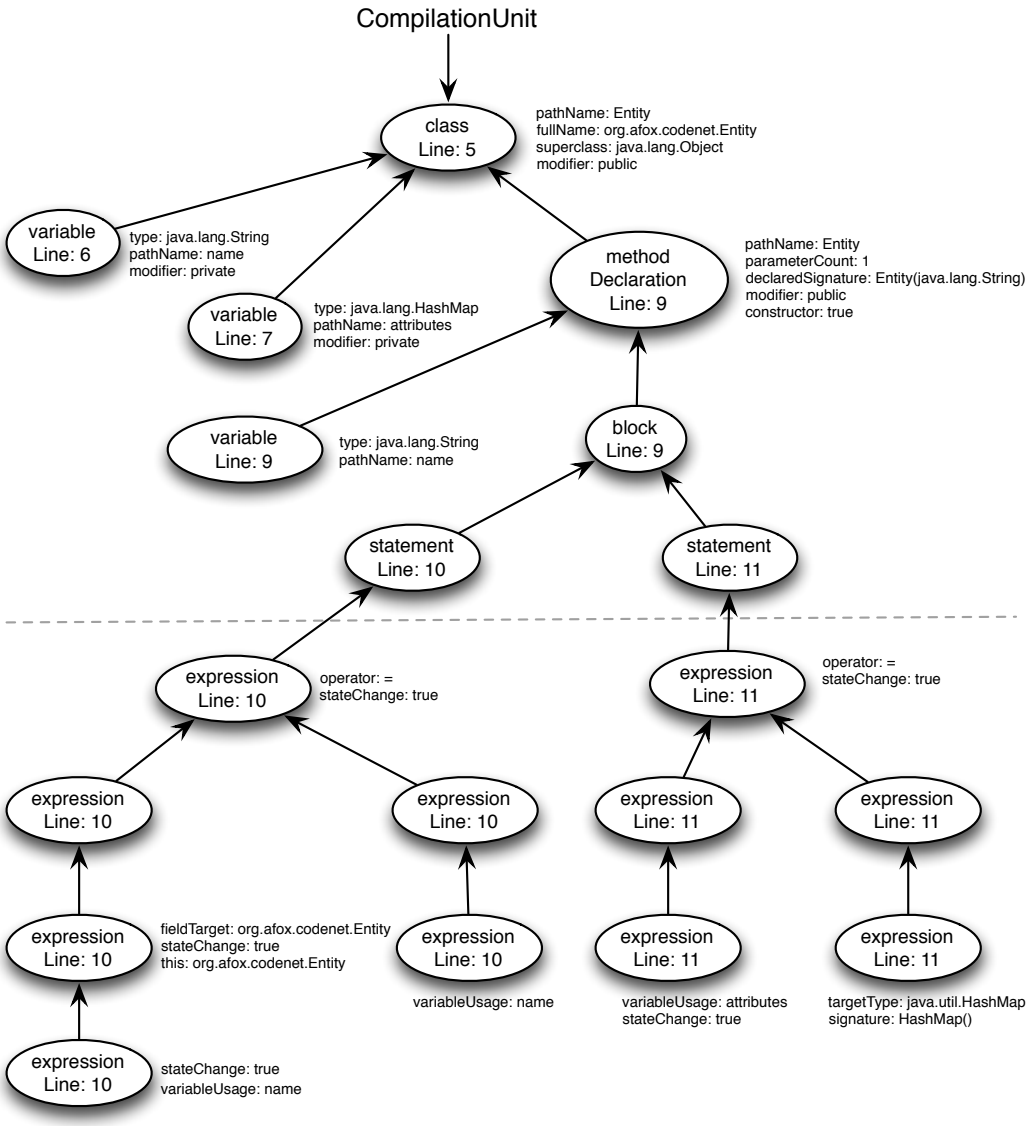

Figure 3.3: Abstract syntax tree for Java listing 1.

When the source code in Figure 3.2 is parsed, the parser returns a reference to the root node of the resulting AST. This reference is of type *CompilationUnit,* which is an interface in the Eclipse JDK. The JDK im-

plements a *Visitor* design pattern (Gamma, Helm, Johnson and Vlissides, 1994), which is used to execute a depth-first traversal of the AST. As each entity is visited, its start tag is written to the corresponding parsefile along with its semantic information in the form of attributes. If the entity has children, those entities are recursively visited; once all children have been addressed, the end tag is written.

In the example AST (Figure 3.3), the class declaration node is visited first. Its *module* tag is written to the parsefile, followed by the *entity* tag for the class declaration. After the class declaration's attributes have been written, each child entity is visited. Since the method declaration in this class has children, those children are visited recursively, and so on. Once all of the entities in the AST have been visited, the end tag for the root node is written to the parsefile.

3.4 CodeNet

The purpose of CodeNet is to translate parsefiles into semgraphs, which are the basis of analysis in this research. To accomplish this, it performs two tasks:

- It unifies the ASTs in the parsefile into a single hierarchical structure.
- It translates relationships that are encoded as attributes in the AST into links between nodes.

Each parsefile is a collection of ASTs, each of which resides in a defined namespace. A semgraph contains a root node whose child nodes define namespaces for the system. The root node for each AST is inserted into the semgraph as a child node of the corresponding namespace. Once the full hierarchical structure for the semgraph is complete, relationships that are defined as attributes are translated to links between nodes in the semgraph structure. Attributes in the ASTs that do not represent relationships remain as attributes in the corresponding semgraph.

The process of translating fully qualified names to links is guided by a meta-model, shown in Figure 3.4. Hierarchical structure is represented

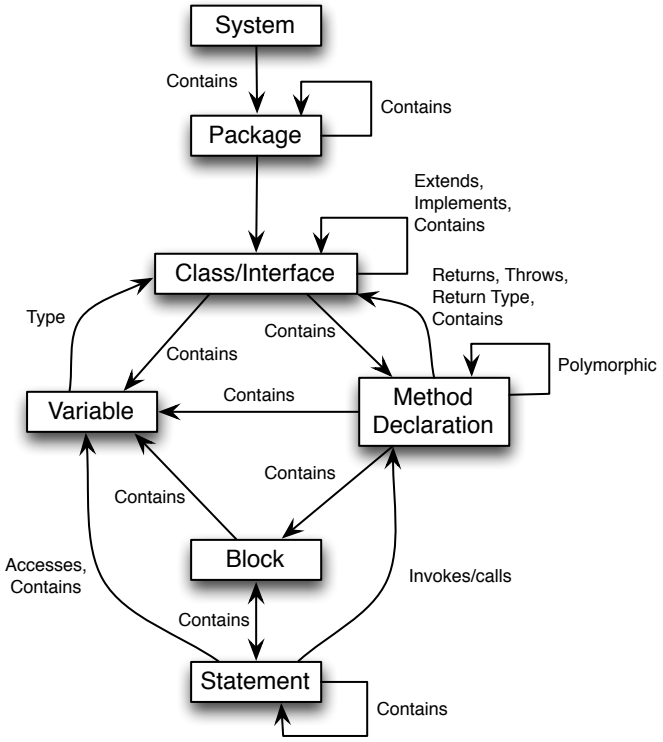

Figure 3.4: Directed graph meta-model.

as links between a parent node and its children (shown as "contains" in Figure 3.4). Nodes for classes may have relationships with nodes for other classes and interfaces because of *extension* and *implementation*. In Java, a subclass *extends* a superclass and *implements* interfaces. A class has one superclass but may implement many interfaces. Nodes for variables have an association with the node for the class that represents the variable's type. In the case of generic types, a variable's node may have relationships with multiple class nodes. Method declarations have a return type and may throw exceptions, which are represented as a relationship between the declaration nodes and the nodes for the corresponding classes. Nodes for method declarations have relationships with the nodes of the methods that are overridden, and nodes for statements have relationships with the nodes for the methods that the statements invoke. Constructors are modeled as method nodes with an attribute that identifies them are constructors. Finally, statements may use variables and this usage is represented as a relationship between the nodes for

statements and the nodes for the variables those statements use.

In semgraphs, nodes and links are categorized according to their function in the software system. Node categorization provided by the AST parser is carried forward into the semgraph. Link categorization is determined based on the meta-model. For example, Figure 3.4 shows that a link from a variable node to a class can only be of one category: “type”. However, a link from a method to a class can be one of three categories: “return type”, “throws” (in the case of exceptions), or “contains” (in the case of inner classes). Semantic information about the nature of the relationship between source code entities is provided by the AST and maintained in the resulting parsefiles. This information is used by CodeNet to assign the correct category to links in the semgraph.

The intent of node link categorization is to help identify and analyze particular structures. For example, a researcher may be interested in the inheritance hierarchy of classes. In this case, she would formulate an analysis that traversed semgraphs to identify nodes of the *class* category and links of the *superclass* category. Similarly, a researcher interested in obtaining a list of all invocations of a particular method (including polymorphic invocations) would formulate an analysis that considered nodes of *methodDeclaration* category and links of the *methodInvocation* and *polymorphic* categories.

3.4.1 Constructing system graphs

Semgraphs are constructed using a four phase process. First, a single root node for the system is created. Second, a hierarchical tree of nodes is built from each AST in the parsefile and inserted under the root based on the AST’s namespace. Third, each node in the semgraph is examined for entity relationships, which are resolved as links between nodes. These relationships include the method return type, exceptions thrown by methods, variable type, superclass and superinterface relationships, method invocations, variable usage, and method overriding (polymorphism). Finally, expression nodes are removed from the semgraph through a process called *reverse inheritance*.

In the Java programming language, source code entities are referenced

by name, which may be simple or qualified. Simple names are those that do not include the node's namespace, and the name assigned to a node upon declaration must be simple. For example, a class may be called `Person`, or a variable may be called `name`. The class called `Person` may be placed in the package called `org.apache.users`. In this case, the fully qualified name for the class is `org.apache.users.Person`, where the notation uses a "." (dot) to separate levels in the namespace hierarchy.

Because classes and methods are named, they contribute to the fully qualified name of the entities they contain. For example, if the class `org.apache.users.Person` contained a variable called `name`, the fully qualified name of the variable is `org.apache.users.Person.name`. Similarly, if the same `Person` class contained a method called `getLastName`, which in turn contained a looping variable called `index`, the fully qualified name is `org.apache.users.Person.getLastName.index`.[2] Because nodes are placed into the semgraph relative to the root node by namespace, any node in the system that has a fully qualified name can be found by starting at the root node and traversing nodes at each level of the name. All nodes with a fully qualified name can be found in the same way so creating links to nodes with fully qualified names is a straightforward process of finding the node and then inserting the link.

Some nodes, however, do not have a fully qualified name, because they are contained in entities that are *anonymous*. Block and statement nodes do not have a name, so any variables contained within a block or statement cannot be referenced using a fully qualified name. Similarly, Java allows for the definition of *anonymous inner classes*, which are class definitions that are utilized *in situ* and are not given a name. Those classes and any entities contained within them cannot be referenced using a fully qualified name. In these cases, the nodes can only be referenced relative to the position of another node within the semgraph.

Referencing nodes that do not have a fully qualified name is equivalent to computing the scope of a program. In the case where a source node

[2] Since local variables cannot be accessed from outside their scope, some would argue that they don't strictly have a fully qualified name. Because the structural analysis defined here is not limited by scope, local variables can be referenced using a fully qualified name.

must link to a target node that does not have a fully qualified name, the scope of the source node is computed. The scope for any given node in the semgraph is the ordered list of all nodes from that node to the root node. Once the scope is obtained for the source, the target node can be identified within the scope using the node's simple name.

3.4.2 References to external entities

Virtually all systems utilize external libraries, which contain nodes for which the declaration is unknown to the AST parser. To address this situation, CodeNet creates a proxy node for each external entity that is identified by the AST parser. Proxy nodes are assigned the category *external*. Their use allows researchers to formulate analyses that address questions about the external relationships of software systems, and at the same time provides a simple mechanism through which externally defined nodes can be excluded from the analysis.

3.4.3 Reverse inheritance

One of the primary goals of any research is to utilize methods that are reproducible. The hierarchical structures extracted from source code using the parser resemble the structures obtained from any Java parser, except in the case of expressions. Different parsers may produce different expression hierarchies because of the application of compiler optimizations. For example, one parser may optimize subexpressions using directed acyclic graphs (Aho, Johnson and Ullman, 1976), while another may not. The effect that such optimizations might have on any subsequent analysis is not clear. For this reason, CodeNet eliminates expression nodes from semgraphs in the final stage of computation.

The removal of expression nodes is called *reverse inheritance*. In this process, all links associated with an expression are inherited by its nearest non-expression parent. For example, consider a statement node that contains five subexpressions, each of which contain a *methodInvocation* link. The *methodInvocation* link associated with each subexpression node is rewritten as a link between the original target method—the

method being invoked—and the statement node. Once this link rewriting process is complete, the statement node will posses five *methodInvocation* links, and the subexpression nodes will be removed from the semgraph. On Figure 3.3, the expression nodes below the dashed line will be eliminated from the semgraph.

3.4.4 Semgraph file format

Once a semgraph has been computed, it is stored in an intermediary file called a *semgraph File*. Semgraph files are text files that have two sections, shown in Figure 3.5. The first contains node and attribute definitions, while the second contains link definitions. Nodes are assigned an index (starting from zero) based on their position within the file, and node declarations include a specification of the node's category, which is followed by a declaration of the node's attributes. Link definitions include the category of the link as well as the source and target node indexes. The format is shown in Table 3.2.

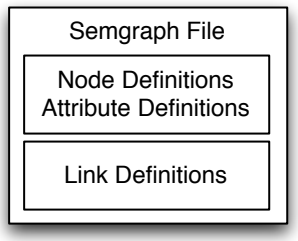

Figure 3.5: Semgraph file structure.

Tag Name	Parameters
entity	category
attribute	name, value
link	category, source node index, target node index

Table 3.2: Semgraph file tags

3.4.5 Quality control

Of primary concern for any analytic tool is the integrity of the data it produces with respect to the phenomena that it models. For this reason, a quality control process was initiated before building the tool and applied throughout its construction. Quality was maintained using two key processes. The first involved the creation of representative code snippets that model key components of software structure. Examples include class declaration, method declaration, variable declaration, and various code statement structures such as loops and conditionals. Parsefile and semgraph files were generated for each of these structures, and hand-inspected for correctness. When major changes were introduced into the system during its development, new parsefiles and semgraphs were constructed from the representative code snippets and compared to the hand-inspected versions.

The second process involved sampling random source-code components from the Qualitas Corpus. The source code of the whole corpus was concatenated into a single text file (with line numbers). Random numbers were generated and the source code at those line numbers was hand inspected for correctness. This process was performed for 150 locations throughout the corpus.

3.5 Analysis

3.5.1 Analysis process

To simply the process of semgraph analysis, a generalized analysis framework has been created. This provides a standard mechanism for defining analyses using Python code, and a standard mechanism for executing specific analyses. All analyses are initiated from a terminal shell using the following command:

```
$ analysis.py analysis_name input=path outputDir=path ...
```

Each analysis is assigned a name; the first parameter on the command line. All analyses require a minimum of two further parameters, *input*

and *outputDir*. The first may be either the path of a semgraph file or the path of a directory that contains semgraph files. The second parameter specifies where the results of the analysis are to be placed (*outputDir* must specify a directory to which the user has write access). If either of these parameters is missing, an error is reported. Further, analysis-specific, parameters are specified in the form *name=value*; their order is immaterial. They are parsed by the Python framework and are placed in a dictionary so that they may be forwarded to the analysis implementation.

An analysis implementation is a Python function that has been decorated[3] with the following:

```
@startup.EntryPoint
```

Upon startup, the program inspects the *analysis_name* parameter and attempts to match it with the name of a decorated function. If no such function can be found, an error is returned to the user. If the method is found, the program loads the appropriate semgraph file(s) and invokes the specified analysis method with the following parameters:

```
analysis_method(project, project_name, parameters)
```

The first parameter is a reference to the semgraph, the second the name of the project, and the third a dictionary of parameters that were parsed from the initial command line arguments. If the *input* parameter is a directory, the system loads each semgraph in it and invokes the analysis method for each.

3.5.2 Semgraph data structure

Figure 3.6 illustrates the UML (Booch, Rumbaugh and Jacobson, 2005) class model for the CodeNet representation of semgraphs. Nodes in the graph are represented by the *Entity* class, and edges between nodes are represented by the *Link* class. Each Entity and Link maintain a reference

[3] The Python programming language provides a mechanism for *wrapping* function invocations. This mechanism is called a *decorator*. The decorator in this case places functions that have been duly decorated into a dictionary so that they can be invoked based on the value of a specified command-line parameter.

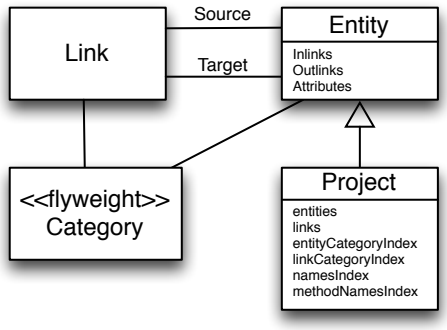

Figure 3.6: Class model used to encode semgraphs.

to their associated *Category*, where Category objects are implemented using the FlyWeight design pattern (Gamma et al., 1994). In this design pattern, there is only one instance for each category, which is shared by all objects of that category. This is particularly useful when determining whether two objects have the same category, because object references can be compared directly.

The system root node is an instance of the *Project* class, and there is only one Project instance for each system. The Project maintains a list of all entities and links, as well as specialized indexes that are used to optimize performance. Lists of nodes and links that are part of the semgraph can be obtained using a method call on the root node. Lists of all nodes or links of a specific category can also be obtained. This is useful for analyses that focus on nodes of specific categories.

Each Entity object maintains a list of both the incoming and the outgoing links. Even though semgraphs are directed, maintaining a list of all links for each node simplifies its traversal, because links can be followed in either direction. Entities also maintain a dictionary of attributes, which encode semantic information that is not encoded as links between nodes. For example, a particular node may represent a source code component whose visibility "modifier" is *public*. This information can be obtained by querying the node for its "modifier" attribute. Attributes are stored as name-value pairs whose value is a list of strings. In this way, multiple values can be stored for each attribute. For example, a source code component may be both *public* and *abstract*.

3.5.3 Example analysis

The listing shown on Figure 3.7 shows the Python code for computing the degree distribution for the project named in the first parameter (line 1). This analysis computes a combination of all incoming and outgoing links, or between-module links only, as determined by the *module* parameter, which defaults to *False* if not specified by the calling routine (line 1). The variable *bins* (line 2) holds a reference to an object that accumulates data for the degree distribution. The variable *total* (line 3) accumulates the total number of links in the distribution, and the *count* variable (line 6) computes the number of links for a given node. Because some types of link may be excluded from analysis (such as links representing structural hierarchy), computing the degree distribution must consider each link individually rather than simply summing the number of incoming and outgoing links.

```
1  def degreeDistributionImpl(project, module = False):
2        bins = stats.Bins()
3        total = 0
4        for entity in project.entities:
5              if entity.isValidForAnalysis():
6                    count = 0
7                    for link in entity.inlinks + entity.outlinks:
8                          if link.addsToDegree(module = module,
                               parent=False):
9                                count += 1
10                   entity.degree = count
11                   bins.add(count)
12                   total +=count
13       return bins, total
```

Figure 3.7: Code listing 2.

The *for loop* (line 4) ensures that all entities within the project are considered for evaluation. For each one, a test is made to ensure that the entity is valid for analysis (line 5). For example, proxy objects that are

created to represent externally defined entities are excluded from analysis. For each entity, both its inlinks and outlinks (line 7) are considered. For each link, a test is made to determine whether the link is relevant to the analysis (line 8). Links to parent nodes, for example, are often not considered for analysis. Similarly, one may wish to only consider links that cross module boundaries. These cases are controlled by setting the *module* and *parent* parameters accordingly.[4] When a link is deemed to be valid for the analysis, it is added to the count (line 9). Once the degree of a node has been computed, it is added to the distribution (line 11) and the total count of links is updated (line 12).

As a convenience, each node's *degree* instance variable is initialized to the number of valid links (line 10), so that this value need not be recomputed. Finally, the distribution and the total number of links are returned to the calling routine (line 13).[5]

3.5.4 Integration with a computing cluster

The University of Waikato maintains a high-performance cluster called *Symphony*.[6] It is a cluster of 180 cores that are available for general purpose computing. Interaction with the cluster occurs through the head node, and jobs are distributed through the cluster using the *Torque* scheduler.

The scheduler provides a series of command-line tools for submitting and managing jobs. Jobs are submitted using the *qsub* program, for which the user defines two sets of parameters:

1. Ones specific to the job being submitted.

2. Ones that describe the job, which are used by the Torque scheduler—such as memory or CPU time limits—to allocate resources.

[4] The Python programming language allows for both named and positional parameters. In this example, the parameters are passed as named parameters. Positionally, the *module* parameter is defined first and the *parent* parameter is defined second.

[5] The Python programming language allows functions and methods to have multiple return values: each of the parameters is returned in a tuple.

[6] http://symphony.waikato.ac.nz

The full command line required to execute a specific job (including parameters from point 1 above) is placed in a script file, which is submitted along with parameters from point 2 above to the *qsub* program.

As noted in Section 3.5.1, the input parameter for an analysis can be a single semgraph file or a directory that contains multiple files. When distributed on the Symphony Cluster, analyses are invoked using the single file form. Because of this, a script must be created for each semgraph file, and each script must be individually submitted to Torque using the command line program *qsub*. To minimize errors, this process has been automated. A script generation program has been integrated into CodeNet that accepts the path to a directory containing semgraph files, a series of job parameters, and a series of *qsub* parameters. This program generates a command line script for each semgraph file and a master shell script, which contains code that handles the submission of each individual job. To use this system, a user generates the scripts for a given directory of semgraph files and then execute the master shell script to submit the jobs.

3.6 Discussion

3.6.1 Existing tools

Before beginning to construct the toolset, I performed a review of commercial and open-source products that could supply the same functionality. While there are many existing tools available that perform specific kinds of analyses (e.g. Burn, 2011; Hovemeyer and Pugh, 2004; Dixon-Peugh, 2011; Vallée-Rai, Hendren, Sundaresan, Lam, Gagnon and Co, 1999; Parasoft, 2011; Semmle Limited, 2011; Software-Tomography GmbH, 2011), they do not provide the general framework for analysis that is required for this research. The closest contender was the Moose platform (Ducasse, Lanza and Tichelaar, 2000), which is based on a metamodel defined by Mens and Lanza (2002). While CodeNet is similar to this framework, there are three main differences.

1. CodeNet explicitly defines nodes that represent packages and blocks,

which are implicit in Mens and Lanza's model. It is not clear what effect these structural entities have on the analysis of software structure. Since the Mens and Lanza model makes these entities implicit, they are difficult to consider as part of analysis.

2. CodeNet is more explicit about containment and hierarchical structure, which allows more precise representation of inner classes and hierarchical relationships between statements. Also, it represents different kinds of variables explicitly whereas the Mens and Lanza model focuses exclusively on instance variables.

3. CodeNet supports relationships that do not exist in the Mens and Lanza model. For example, variables have a type, which CodeNet represents as a relationship between the variable declaration node, and the associated type declaration node. Similarly, it represents method invocation as a relationship between the invoking statement and the associated method declaration node whereas the Mens and Lanza model represents invocation as an entity contained within a method.

Because of the above differences, I decided to construct my own tools to support the research.

3.6.2 Toolset evolution

Over the lifetime of this research, the structure of the toolset has evolved considerably. The original design consisted of a corpus of open-source systems written in Java and a single program that parsed it, generated semgraphs, and performed analysis in a single pass. However, a series of problems unfolded which made this structure unfeasible. First, parsing medium and large-scale software systems and the subsequent generation of semgraphs is a computationally expensive process, and recomputing these structures for each analysis soon became cumbersome. To compensate, graph generation was separated from graph analysis.

A second problem that arose was associated with the use of parsers. The original development used the Eclipse AST parser, but I could not

get it to work “headless”, that is, outside the graphical environment, which is required for execution on the Symphony Cluster. In fact, many users in online Eclipse forums claimed that it was possible to run the AST parser in a headless fashion, and considerable effort was wasted trying to accomplish this goal. While I was able to get the parser to run without the supporting GUI, I could not get it to produce bindings, which are necessary for CodeNet to create links between semgraph nodes. Eventually, I switched to another Java parser (Gesser, 2008), but that parser did not work with Java generics. After a year of using the alternate parser, it became clear that more and more systems were using Java generics, and excluding those from a scientific investigation was undesirable. Thus development reverted to using the Eclipse AST parser.

One of the drawbacks of reverting to Eclipse was that it would only produce bindings while running within the Eclipse GUI framework. This posed problems with Symphony because it does not allow direct interaction with cluster nodes, so execution of Eclipse is not possible. To address this, parsing and semgraph generation were separated into two processes; one remaining on a single machine capable of running the Eclipse GUI and the other being executed on the cluster. Parsefiles were introduced to bridge the gap between these now separated processes.

Another shift in the development of the tools was the move from Java to Python. The original CodeNet system was written in Java. As issues with the various Java parsers arose, it was easiest to compensate by using Java reflection. While reflection offers a simple solution to some of the problems, the result was that CodeNet started to become overly dependent on the systems that were being analysed, and a considerable amount of code was being written just to maintain its generality. A decision was made to reimplement CodeNet in a language other than Java. This proved to be an excellent move because removing dependencies between CodeNet and the systems being analysed simplified its design.

Chapter 4

Scale-free structures and coupling

> The beauty of a living thing is not the atoms that go into it, but the way those atoms are put together.
>
> —Carl Sagan

This chapter uses the tools presented in Chapter 3 to address the research questions raised in Chapter 1. To do this, we examine a series of degree distributions that are extracted from semgraph representations of the systems in the Qualitas corpus. To provide a complete treatment of the research questions, twelve different types of degree distributions are considered. To avoid confusion about the differences between them, we begin with a definition of each type and a description of how it is constructed. We then state five hypotheses along with a justification for each, followed by a description of experiments that are designed to test them. The chapter ends by presenting the results of the experiments, along with statements of validity and a discussion of the implications of these results for software systems.

4.1 Perspectives of degree distributions

As noted in Chapter 2, a common technique for analysing the general level of connectivity in a network is to compute its degree distribution, which plots node degree against frequency of observation. The most general method of computing a degree distribution is to examine each node in the network, count its connections, and accumulate the counts in the form of a histogram. This method produces a general degree distribution for the network, and assumes that all links are equivalent. However,

networks can have connectivity properties that can only be found by analyzing certain types of link. This section discusses three factors that, when combined, produce twelve ($3 \times 2 \times 2$) different types of degree distribution that are used to test hypotheses about the structure of software systems.

4.1.1 Inlinks, outlinks and combined perspectives

The literature in network theory (Albert et al., 1999) and software systems (Myers, 2003) demonstrates that distributions for inlinks and outlinks generally assume different shapes. This occurs because the reasons for creating a link *to* a particular node are different than those for creating a link *from* a particular node. While a combined degree distribution—one that counts both inlinks and outlinks—shows the overall pattern of connectivity, it conceals the differences between inlinks and outlinks.

To reveal these differences, two new distributions are computed counting just inlinks and just outlinks respectively. To compute the inlink distribution, each node's inlinks are counted and accumulated in a histogram; outlink distributions follow the same process except that outlinks are counted. In what follows, any reference to a degree distribution is assumed to be a *combined* distribution unless stated otherwise.

4.1.2 Within-module versus between-module links

As noted in Chapter 2, the purpose of modularization is to minimize the propagation of change through a network. Nodes that are highly interactive are placed within the same module, so that any waves of change propagation that result from changes to individual nodes are contained within the module itself. However, because systems are only *nearly* decomposable, between-module interaction cannot be completely eliminated. It is important to distinguish within-module links from between-module links because the latter create the potential for change to propagate from one module to another. If the module under consideration is a class, between-module links represent class coupling relationships as described in Section 2.3.3.

Because coupling is a measure of strength of interaction between modules, research questions that involve coupling require studying the degree distributions of between-module links. These distributions are constructed by examining each node and only counting links that cross a module boundary. In what follows, any reference to degree distributions assumes that all links—both within-module and between-module—are counted, unless explicitly stated otherwise.

4.1.3 Link aggregation

Much of the analysis of coupling in object-oriented systems focuses on class-level interactions (Chidamber and Kemerer, 1991; Li and Henry, 1993; Chidamber and Kemerer, 1994; Lee et al., 1995; Briand et al., 1999b), because classes are the basic unit of modularization. Analysis of links, therefore, is not focused on individual nodes, but rather on an aggregation of links per class. Some of our hypotheses focus on individual nodes, because nodes are the source of all links in a software network, but others consider links in aggregated form.

The method for constructing aggregate distributions differs from those already presented in that only aggregate nodes are examined. For each such node, its children are identified and the links to them and from the aggregated node are counted. Degree distributions can be aggregated for any node that has children, but only aggregates at the class level are considered in this research. In what follows, any reference to a degree distribution assumes node-level analysis unless explicitly stated otherwise.

4.1.4 Combining degree distribution perspectives

Degree distributions can be computed using any combination of the above perspectives. For example, to examine issues surrounding coupling between classes, an aggregate between-module distribution would be appropriate. If that question were expanded to differentiate inlinks and outlinks, aggregate between-module inlink and aggregate between-module outlink distributions would be used. Combining the three perspectives—

combined/inlink/outlink, all links/between-module links, and node-level/aggregate-level—results in twelve possible degree distributions.

4.1.5 Hierarchical links

The source code for software systems is structured hierarchically, and this structure is represented in semgraphs using links that are categorized as *hierarchical*. The model of software evolution presented in Section 4.2.1 suggests that the hierarchical structure of software is imposed by the programming language grammar, which is external to the software systems themselves. Because this structure does not result from internal processes, hierarchical links are excluded from all degree distributions examined in this work.

4.2 Hypotheses

The questions introduced in Section 1.2 focus on the relationship between scale-free structure and coupling: does the presence of scale-free structure imply that high coupling is present? Existing coupling measures attempt to quantify the strength of association between modules in order to predict how robust they will be to change propagation. While there has been some empirical evaluation of these measures (Briand et al., 1998, 1999b; Brito e Abreu and Goulao, 2001), little work has focused on what comprises a typical distribution of coupling in software systems (Briand et al., 1999b; Brito e Abreu and Goulao, 2001). Furthermore, research on coupling metrics promotes the maxim of "high cohesion and low coupling," but there is little indication of what level of coupling is considered to be "high" (Brito e Abreu and Goulao, 2001).

Network theory suggests that complex systems exhibit scale-free structure (Barabási and Albert, 1999; Albert et al., 1999; Newman, 2005; Newman et al., 2006), and network analysis of software systems suggest that they are, ubiquitously, at least approximately, scale-free (Wheeldon and Counsell, 2003; Myers, 2003; Marchesi et al., 2004; Potanin et al., 2005; Baxter et al., 2006; Hyland-Wood et al., 2006; Concas et al., 2007; Ichii et al., 2008; Louridas et al., 2008; Gao et al., 2010). This suggests that

the network for a given software system contains areas of high connectivity where the maximum connectivity is substantially larger than the mean value for the network. For example, in the situations we will analyze it turns out to be two or three orders of magnitude larger. This suggests that high coupling may be common, but neither the coupling metrics literature nor the network analysis of software systems literature addresses this question directly. Sections 4.2.2 – 4.2.6 state five hypotheses as a means of exploring the question of whether software systems are scale-free and, if so, whether their scale-free structure indicates high coupling.

4.2.1 Proposed model

Source code networks evolve over time through the addition and deletion of nodes and links. When new nodes are added, there are two types of attachment that have to be resolved. In the first, programmers have to resolve where in the network the new nodes are to be placed. In the second, programmers need to resolve usage relationships between the newly added node and existing ones.

Because software networks are hierarchical, new nodes are always added as children of existing nodes, and valid hierarchical structure is maintained through a language grammar. For example, in the Java programming language, a method declaration node can only be the child of a class declaration node; otherwise, the language grammar would not allow it.

The second type of node attachment is optional because new nodes do not always require the use of functionality that already exists in the network. When required, the connection between a newly created node and an existing one is driven by whatever functionality is required for the new node to achieve its goals.

The above description of the software development process is used as a starting point to modify the BA model. When a programmer wishes to add new code, he must first determine its location. It may be within an existing container, such as a class or method declaration, or it may require a new container which the programmer must place appropriately.

A programmer's decision about location is based upon his current understanding of the network's hierarchical structure and requires him to exercise judgement as to which location is best. Once the location has been determined, he must decide whether the newly created node requires interaction with nodes already existing in the system.

The BA model specifies that nodes will preferentially attach to other nodes based on a probabilistic attachment function. For hierarchical structure, this function is determined by the language grammar. For usage relationships, the preferential nature of the attachment function is based on these criteria:

1. Different nodes offer different functionality.
2. Programmers have an incomplete understanding of the network.
3. Programmers are more likely to use nodes that they trust.

Functions that are more generally applicable are usable in a larger number of contexts, and will garner more incoming links than functions that are more specific. Because programmers have an incomplete understanding of the network, their choices will be limited to nodes of which they are aware and ones they have used before. Programmers are less likely to use a node that is known to have bugs and are more likely to use one that has a reputation for robustness. In each of these cases, a programmer is likely to give preference to some nodes over others.

Keller (2005) notes that in order to produce a scale-free network, the constraints on an evolutionary process are quite minimal. In general, an attachment function that has a non-uniform distribution will suffice (Simon, 1955). The model demonstrates that programmers have reasons to choose some nodes over others, which would cause a non-uniform distribution of usage. It is also important to recognize that programmers do make mistakes. A programmer may choose a module for a newly created node for which another module may be better suited, but is unknown to the programmer. Mistakes based on inadequate knowledge of the network serve to consolidate linking, thereby increasing the likelihood that some nodes will receive more links than others. For these reasons, a source code network with scale-free properties is expected to emerge.

4.2.2 Hypothesis 1: Scale-free structure in source code networks

Based on the proposed model of source code evolution, our first hypothesis is stated as follows:

> **Hypothesis 1**: The source code network of a software system exhibits a long-tailed degree distribution.

It is recognized that software systems below a certain size may not exhibit scale-free like structure due to their limited number of nodes. In these cases, however, there is little concern about modifiability because the systems are small enough to be rewritten.

4.2.3 Hypothesis 2: Scale-free structure and coupling

Should Hypothesis 1 hold true for a given software system, the presence of highly connected nodes—as implied by the presence of scale-free structure—may not necessarily indicate high coupling. Since design theory considers the primary criterion of modularization is to group highly interactive nodes (Alexander, 1964), we expect to find within-module interaction. However, it seems unlikely that the level of interaction that is implied by scale-free structure can be fully resolved within-module as this would require that the module contains a large number of nodes. As outlined in Section 2.1.2, larger modules are less modifiable because they take longer to stabilize in light of change propagation, so there is pressure to keep modules as small as possible.

Given this pressure to limit module size, Hypothesis 2 is stated as follows:

> **Hypothesis 2**: The network made up of between-module connections exhibits approximately scale-free structure.

4.2.4 Hypothesis 3: Outlink constraints

Hypothesis 1 does not differentiate between inlinks and outlinks. However, there is reason to expect that these may not contribute uniformly to

the degree distribution. First, differences have been observed between inlink and outlink distributions (Myers, 2003). Second, constraints are imposed on node outlinking by programming languages and by practical usage. For example, Java allows only single inheritance, thereby constraining classes to have a single link to their superclass. Variable declarations may have only one type, for simple declarations, or just a few types, in the case of generics. While programming languages do not impose limits on the number of outlinks from statement nodes, most programmers conform to coding standards that limit statements to a single line of source code, and this serves to constrain the number of outgoing links that statements have.

None of the aforementioned constraints on outlinks are imposed on inlinks. There is no limit on how many times an abstraction may be used, and because of the mechanism for defining links—from source node to target node—there is no indication in the source code of the frequency of use of a given abstraction. Therefore our third hypothesis is stated as follows:

> **Hypothesis 3**: The connectivity network of inlinks for source code entities exhibits a long-tailed degree distribution, while the connectivity network for outlinks is constrained by an upper bound.

4.2.5 Hypothesis 4: Aggregate measures of coupling

Hypotheses 1 through 3 relate to linking at the source code node level, and do not consider the aggregation of links based on the class structure of the software. However, it is useful to examine aggregate interaction at the class level, for two reasons:

1. In a software system, there are fewer classes than statements and variables. It easier for programmers to consider dependency between classes at an aggregate level than to consider dependency between individual source code nodes.

2. It is difficult to ascertain the level of outward dependency of a module based on source code nodes, because of the constraints placed

on outlinks for individual nodes. Understanding how much one class depends on others requires the examination of all outlinks of all its child nodes.

If Hypothesis 1 holds true, that means that the underlying source code structure is scale-free. This suggests that an aggregation of these nodes at the class level will also exhibit scale-free properties, although the distribution is likely to look different because of the aggregation. Specifically, aggregating nodes will shift the distribution to the right. Because the number of class nodes is small when compared with the total number nodes in the system, a distribution that only considers classes will have fewer data points, so the resulting distribution will be more noisy and less well defined. Smaller systems may not have enough nodes to generate a sufficiently defined distribution to perceive the aggregate structure as scale-free. However, the range of connectivity will be on the same scale as the non-aggregate distributions, which implies that high coupling will be present.

Hypothesis 3 suggests that there is a difference between inlink and outlink distributions because of the constraints placed on node outlinks. However, no such constraints exist at the class level, so it is expected that the outlink distribution for classes will not exhibit an upper bound, as Hypothesis 3 expects. Given these considerations, our fourth hypothesis is:

> **Hypothesis 4**: The network of classes for software systems of sufficient scale exhibits a long-tailed degree distribution. Aggregate outlink distributions are not bounded.

4.2.6 Hypothesis 5: Aggregate outlink distributions

Hypothesis 3 predicts that the distribution of outlinks on a per node basis will be bounded. Hypothesis 4 predicts that because the constraints on outlinks at the node level do not exist at the aggregate level, the resulting distribution will not exhibit the same bounding as is predicted for node level outlinks. The question arises, with the constraints on outlinks removed, what distribution is expected?

Since most individual nodes have relatively few outlinks, the aggregation of a small number of nodes will also have relatively few outlinks—albeit a larger number than an individual node. As a module increases in size, the aggregate number of outlinks is likely to increase, because each new node adds to the count. Given this relationship, it is reasonable to expect the distribution of outlinks aggregated at the class level to be highly correlated with the size of the module. The literature reports that class size is scale-free (Baxter et al., 2006; Louridas et al., 2008; Hatton, 2009), so it is expected that the aggregate outlink distribution will also be scale-free. Our fifth hypothesis is stated as follows:

> **Hypothesis 5**: The degree distribution for outlinks aggregated at the class level is correlated with class size and, for systems of sufficient scale, will exhibit approximate scale-free structure.

4.3 Experimental design

Hypotheses 1 through 5 will be tested against the Qualitas corpus (described in Section 3.2) using the tools described in Sections 3.3 – 3.5. Semantic graphs are computed for each system, from which the degree distributions required to test the hypotheses are extracted.

4.3.1 Computing module boundaries

While Hypothesis 1 involves all links of a software system, Hypotheses 2, 3, 4, and 5 utilize degree distributions that only reflect between-module links, which requires that module boundaries are explicitly defined. Recall from Section 2.1.2 that a module is any formal grouping of nodes. However, to remain consistent with existing coupling measures, the *class* is used as the module boundary for this experiment (Chidamber and Kemerer, 1991; Li and Henry, 1993; Chidamber and Kemerer, 1994; Lee et al., 1995; Briand et al., 1999b). Whether a specific link crosses a module boundary is determined by examining the hierarchical ancestry of the source and target nodes associated with the link. The closest class in a node's parent hierarchy is marked as the containing class for that node.

A link is determined to cross a module boundary if the containing classes for its source and target nodes are not the same.

The Java programming language allows for the definition of anonymous inner classes, which are the same as ordinary inner classes except that they are not explicitly named. Inner classes are ones that are defined within the context of a containing class, as illustrated in Figure 4.1. This Figure shows an inner class (*CInner*) defined within a method (*M1*) that is contained within another class (*C1*). A statement (*S1*) accesses an instance variable (*V1*) that is defined within *C1*, but outside *CInner*. Although both *V1* and *S1* are contained within the same class (*C1*), the interaction between the two nodes crosses the *CInner* class boundary, so it is considered to be between-module. One could argue that there are conditions under which the relationship between *V1* and *S1* should be considered within module. However, it is not the purpose of this thesis to speculate why programmers chose specific structures, but rather to report what structures are present.

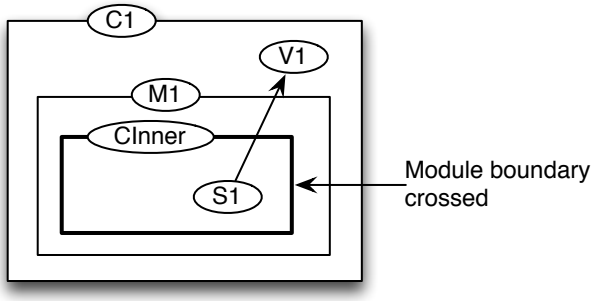

Figure 4.1: Inner class definition and coupling.

4.3.2 Testing the hypotheses

To test Hypotheses 1 through 4, 12 degree distributions are created for each system in the corpus. Table 4.1 illustrates which ones are used to test each hypothesis. The table is separated into two sections: node-level distributions and class-level distributions. Each section has two dimensions. In the first, distributions are computed for inlinks, outlinks, and inlinks and outlinks combined. In the second, one distribution includes

all links and the other includes only between-module links.

Hypothesis 1 is tested by examining the combined node-level degree distribution for system in the corpus. To satisfy the test, the resulting distributions must be left skewed and have a "heavy tail," which means that the tail of the distribution contains nodes that have high connectivity with respect to the mean. When observed on a log-log plot, each distribution should approximate a straight line for values above a minimum x value, below which the distribution is not observed (Clauset et al., 2009).

Hypothesis 2 is tested by examining the degree distribution for all between-module links in a software network. Because areas of high connectivity are not expected to be resolved within-module, these distributions should maintain the characteristic heavy tail that Hypothesis 1 postulates. When distributions computed for Hypothesis 2 are compared with those computed for Hypothesis 1, the nodes that made up the heavy tail for Hypothesis 1 should also make up the heavy tail for Hypothesis 2.

To test Hypothesis 3, the degree distributions used to test Hypotheses 1 and 2 are recomputed for both inlinks and outlinks, resulting in four node-level degree distributions for each system. Hypothesis 3 suggests that the outlink distributions will be constrained by an upper bound, while the inlink distributions will retain approximate scale-free structure. This structure can be identified by comparing the inlink distributions to the outlink distributions.

Hypothesis 4 suggests that connectivity aggregated at the class level will demonstrate scale-free structure. To test this, the six degree distri-

Node Connectivity			
Links Considered	Combined links	Inlinks Only	Outlinks Only
All links	H1	H3	H3
Between-Module Links	H2	H3	H3
Aggregate Connectivity			
Links Considered	Combined Links	Inlinks Only	Outlinks Only
All links	H4	H4	H4
Between-Module Links	H4	H4	H4

Table 4.1: Distributions required to test Hypotheses 1 – 4

butions used to test Hypotheses 1 through 3 are recomputed, but aggregated at the class level. The resulting distributions will have fewer data points, so it will be more difficult to identify them, but characteristics of approximate scale-free structure are expected in all distributions.

Hypothesis 5 predicts that, due to constraints on outlinks at the node level, the distribution of outlinks at the aggregate class level will correlate with the number of nodes in the class. This will be tested by computing the Pearson correlation coefficient (r) between aggregate outlink degree and class size, measured as the number of nodes.

4.4 Results

Testing Hypotheses 1 through 4 against the 97 systems contained in the Qualitas corpus results in 1164 degree distributions. This collection is too large to present here, so representative plots are provided. The full set of plots is available from the University of Waikato Institutional repository.[1]

4.4.1 Hypothesis 1

Three plots are chosen for discussion based on system size (total node count): *derby-10.1.1.0*, *jung-1.7.6* and *picocontainer-1.3*. These represent the largest, median, and smallest systems, respectively. Figure 4.2 shows their distributions on a single plot. The similarity in shape is striking: a positive slope is observed between the first two data points, followed by a linear negative trend. Note that each distribution is noisy toward the right hand side, which is expected because they are produced from discrete data points and naturally have fewer points exhibiting high values.

All the plots exhibit characteristics of heavy-tailed distributions. They are left skewed and have a total range that is at least an order of magnitude larger than the mean. The mean degree for each one is shown in Figure 4.2, and the means are approximately the same despite the enormous different in system size. To demonstrate this effect for all systems,

[1] http://hdl.handle.net/10289/6373

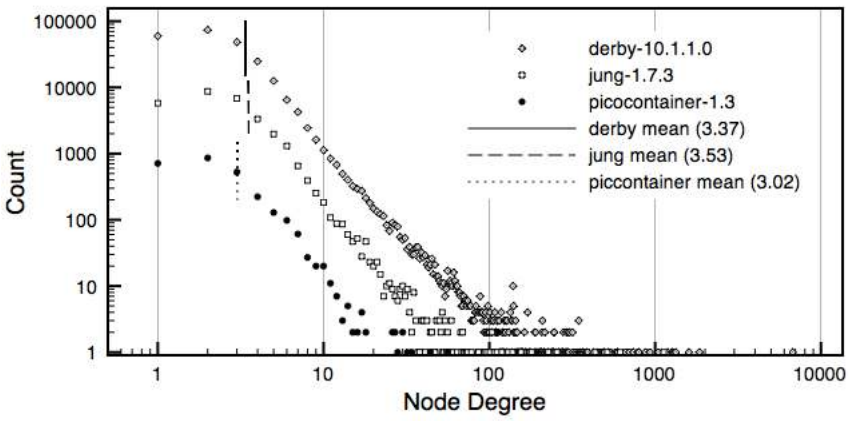

Figure 4.2: Distribution of overall connectivity for three sample systems.

the mean and maximum degrees of each system are computed and plotted with a logarithmically-scaled y-axis in Figure 4.3, where the systems are sorted into descending order of maximum degree. The mean degree over all systems remains fairly constant, while the maximum is between 10 and 1000 times the mean for almost all systems.

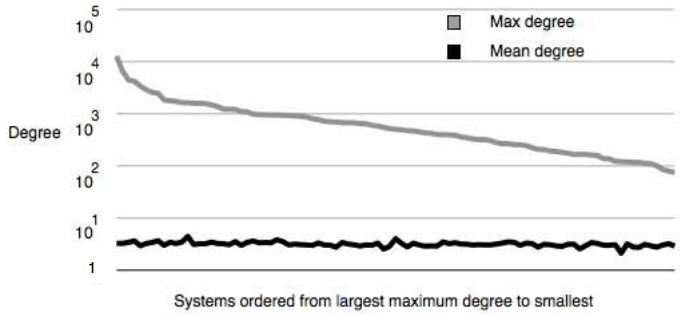

Figure 4.3: Mean degree vs. maximum degree for all systems.

Further study of the distributions in Figure 4.2 reveals a clear differentiation of the three systems, except to the right where the distributions are noisy. This is consistent with the expectations of a power-law distribution Clauset et al. (2009). The probability of a node with a high degree decreases proportionally to the degree; therefore, given a fixed α, the number of nodes with higher degrees increases with the node count. Based on these observations, we conclude that Hypothesis 1 is satisfied:

overall connectivity for source code entities follows a heavy-tailed distribution for all systems in the corpus.

4.4.2 Hypothesis 2

The between-module connectivity distributions for the three sample systems are shown in Figure 4.4. These show the overall coupling present in each system, and are similar in shape to the distributions for Hypothesis 1. The between-module connectivity distributions are less well defined, which is due to the lack of between-module interaction: less interaction equates to fewer data points, thereby producing noisier distributions. All the between-module distributions have similar shape, including a heavy tail that is apparent in Figure 4.4.

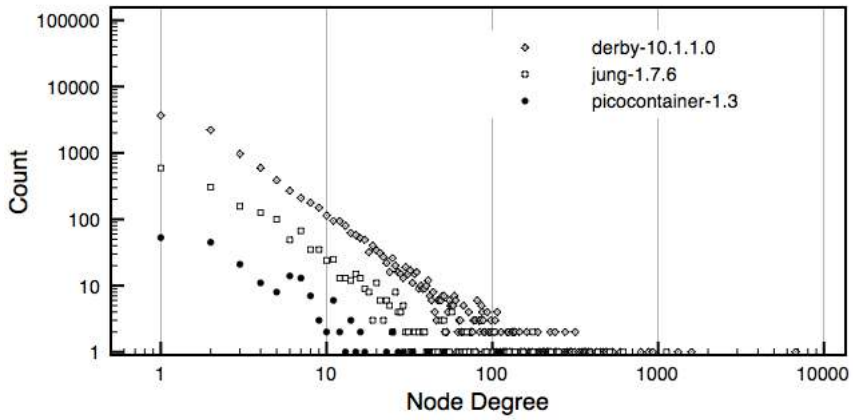

Figure 4.4: Between-module degree distributions for three sample systems.

Figure 4.4 demonstrates that the between-module connectivity distributions are similar in shape to those for overall connectivity (Figure 4.2). The between-module distributions are less well defined, which is due to programmers favoring within-module interaction over between-module interaction: less interaction equates to fewer data points, producing noisier distributions. All the between-module connectivity distributions have similar shape, including a heavy tail.

Figure 4.5 shows the full and between-module distributions plotted together, for each of the sample systems. An overlap in the heavy tail is

observed in each case. If the links for nodes with high overall connectivity were primarily resolved within-module, data points would migrate towards the left in the between-module distributions, and they would not exhibit a heavy tail. However, this migration is not observed. Instead, both distributions have heavy tails, which overlap when plotted on the same graph. This demonstrates that the nodes that appear in the heavy tail of the overall connectivity distributions are these nodes that appear in the heavy tail of the between-module connectivity distributions, and are responsible for the presence of high coupling.

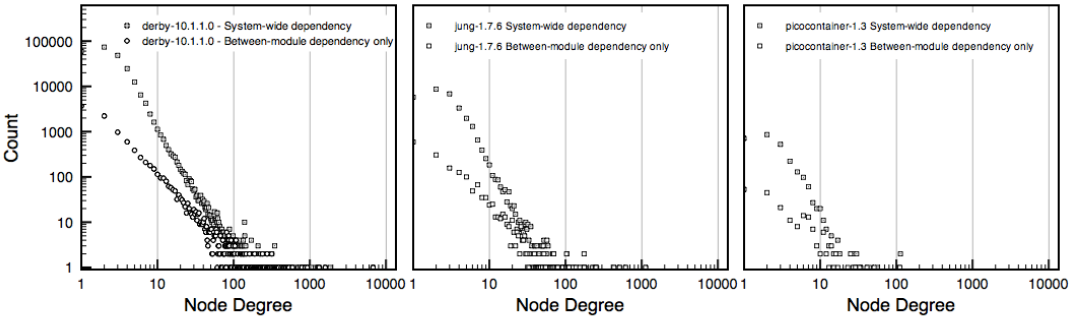

Figure 4.5: Comparison of overall and between-module connectivity distributions for three sample systems.

In Figure 4.5, there is a difference in slope of the linear portions of the distributions. Because the full connectivity distributions have more data points on their left side, the slope in the overall connectivity distributions are steeper than those for the corresponding between-module distributions. Using the process outlined in Clauset et al. (2009), α—the slope of the linear portion of the distribution—is estimated for distributions using $x_{min} = 1$ for between-module distributions and $x_{min} = 2$ for all full distributions; data below x_{min} are ignored. Figure 4.6 compares the estimated α between overall and between-module distributions for all systems, sorted in descending order by largest α estimate. The α for between-module connectivity distributions is lower than the overall connectivity distribution for the same system. Based on the above analysis, we conclude that Hypothesis 2 is satisfied.

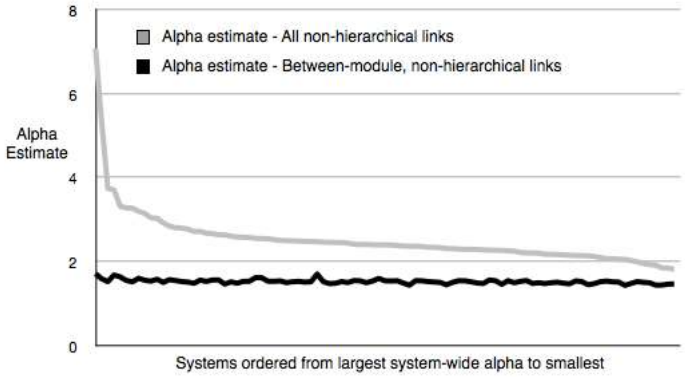

Figure 4.6: Comparison of α estimates for all systems.

4.4.3 Hypothesis 3

Because of the large number of distributions to be considered, only those for the system *derby-10.1.1.0* will be shown here. This is the largest system in the corpus, with the largest number of data points, and therefore produces distributions that highlight key properties most clearly. Figures 4.7 and 4.8 illustrate the distributions for all inlinks and between-module inlinks respectively. Both show approximate scale-free structure, as predicted by Hypothesis 3. The two distributions of inlinks for *derby-10.1.1.0* exhibit the same patterns observed for Hypotheses 1 and 2 for that system.

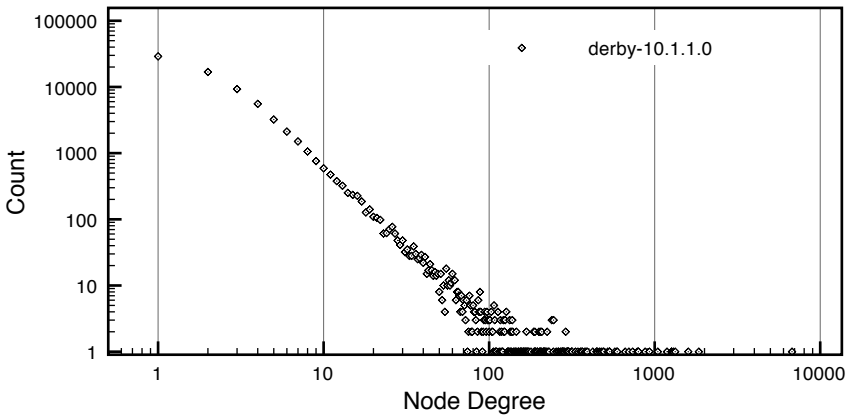

Figure 4.7: All inlinks for *derby-10.1.1.0*.

Figures 4.9 and 4.10 show the full outlink and between-module degree

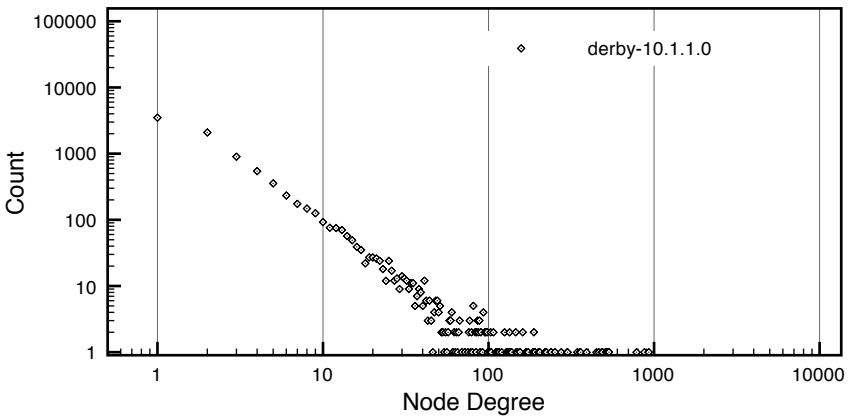

Figure 4.8: Between-module inlinks for *derby-10.1.1.0*.

distributions for *derby-10.1.1.0*. Both show a straight-line segment on the log-log plot, but their range is truncated when compared to the same distribution plots for inlinks. This can be seen in Figures 4.11 and 4.12.

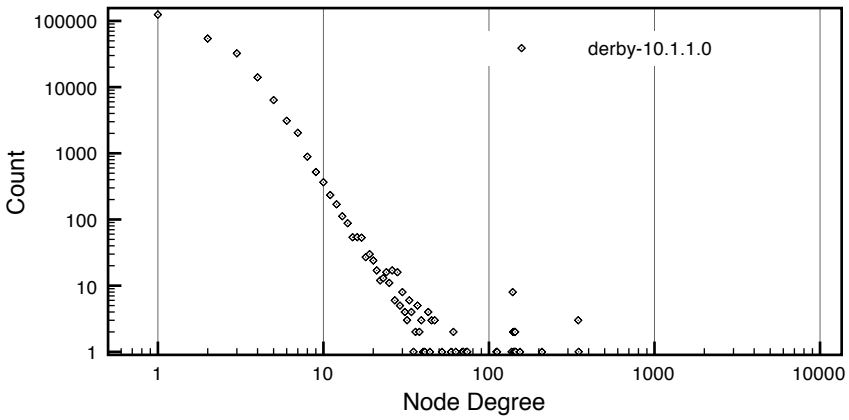

Figure 4.9: All outlinks for *derby-10.1.1.0*.

Hypothesis 3 predicts that the inlink distributions for software systems exhibit scale-free structure. This is clearly shown by Figures 4.7 and 4.8. It also predicts that the outlink distributions for software systems are truncated, which is shown by Figures 4.9, 4.10, 4.11 and 4.12. Hypothesis 3 holds true for the system *derby-10.1.1.0*.

While *derby-10.1.1.0* is reasonably representative, the distribution of its between-module outlinks does not fully convey the amount of truncation in outlinks that is observed in many of the smaller systems. However,

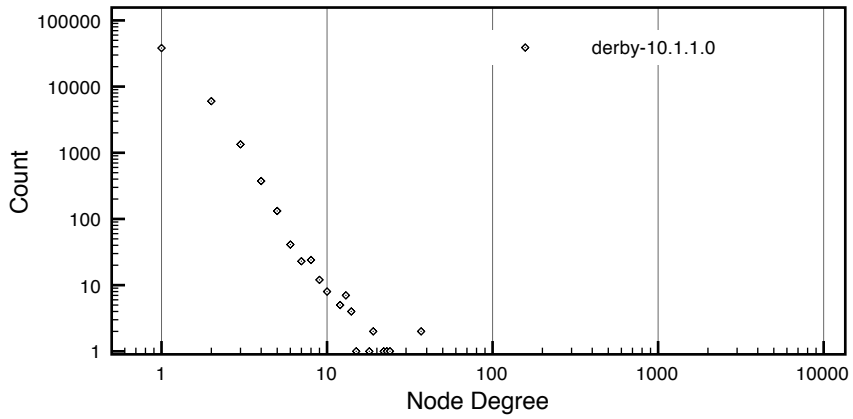

Figure 4.10: Between-module outlinks for *derby-10.1.1.0*.

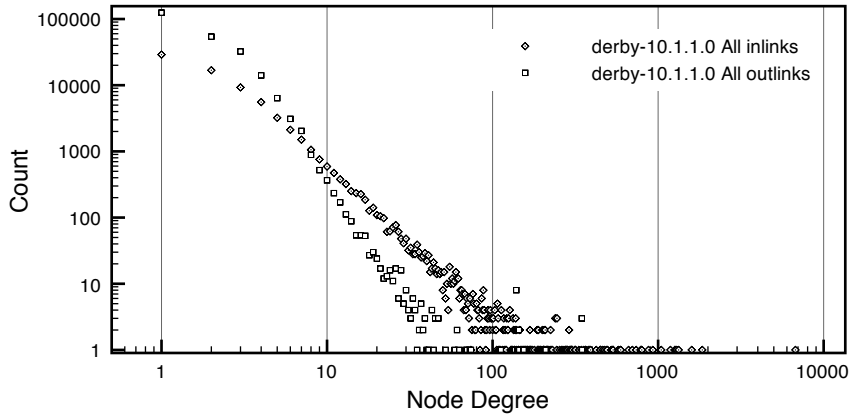

Figure 4.11: Comparison of all inlinks vs. all outlinks for *derby-10.1.1.0*.

45 of the 97 systems in the corpus have between-module outlink distributions for which the maximum value is ten or less. For example, the between-module outlink plot is shown for *jgraph-5.9.2.1* in Figure 4.13, and shows that the between-module outlink maximum does not exceed 10 links. While it may be tempting to look at the outlink distributions for *derby-10.1.1.0* (Figures 4.9 and 4.10) and argue that they illustrate approximate scale-free structure, this is clearly not the case for *jgraph-5.9.2.1* (Figure 4.13).

To test Hypothesis 3 for all systems in the corpus, the range for between-module inlinks and for between-module outlinks is shown in Figure 4.14. Along the x-axis are the systems in the corpus, sorted into descending

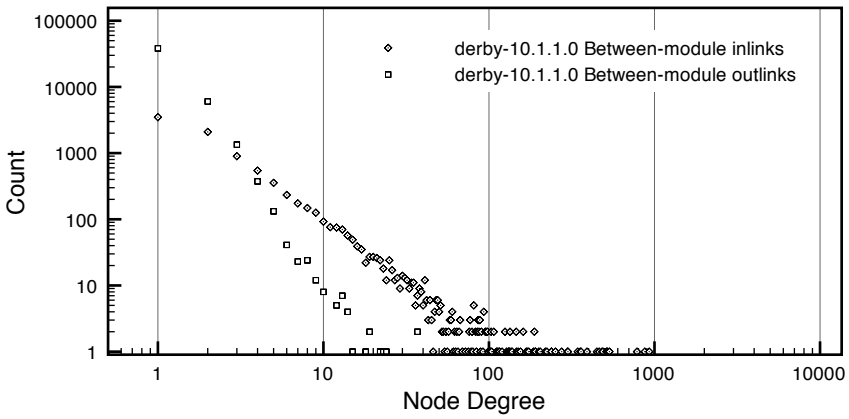

Figure 4.12: Comparison of between-module inlinks vs. between-module outlinks for *derby-10.1.1.0*.

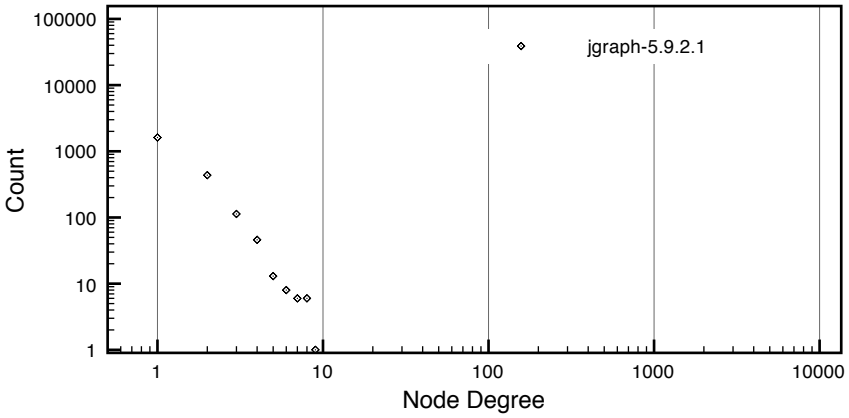

Figure 4.13: Between-module outlinks for *jgraph-5.9.2.1*.

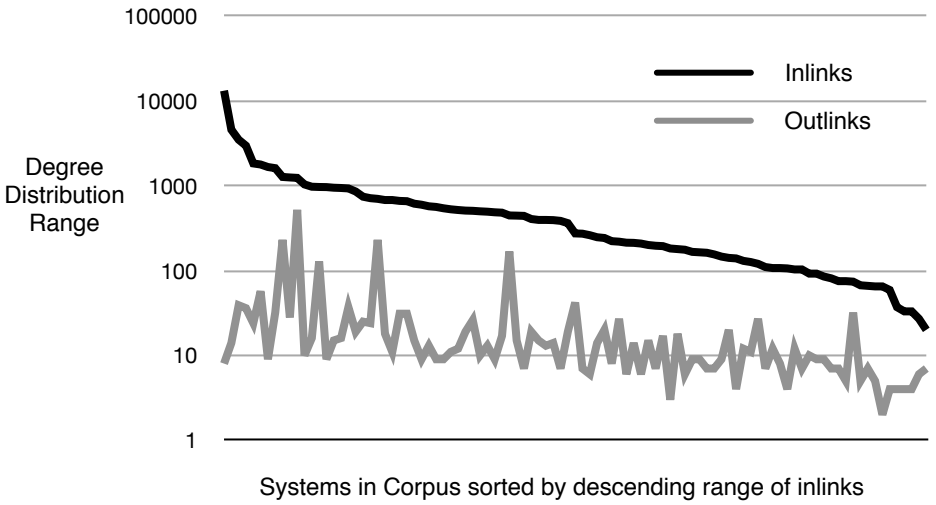

Figure 4.14: Degree distribution ranges of inlinks and outlinks for all systems.

order of inlink distribution range. With the exception of some peaks in the outlink plot, the range of inlinks typically exceeds the range of outlinks by a factor of ten. In the case of the peaks in the outlink plot, the range of the inlinks exceed that of the outlinks by a factor of between 2.5 and 7. Based on the evidence presented in this section, we conclude that Hypothesis 3 holds true for all systems in the corpus.

4.4.4 Hypothesis 4

A post-hoc analysis of node categories in the Qualitas corpus reveals that the mean class declaration count is 1.40% of the total system node count, with a standard deviation of 0.64 across all systems. Whereas the distributions used to test Hypotheses 1 through 3 contained 10,000 to 300,000 points, the distributions required to test Hypothesis 4 contain 50 to 2000 points. Because of the limited number of data points, it is difficult to observe scale-free structure by examining the degree distribution. While the aggregate combined distribution for *springframework1.2.7* (Figure 4.15) clearly shows the expected structure, this level of clarity is not present for *fitjava-1.1* (Figure 4.16).

With scale-free structure, we expect the range of the distribution to be much larger than the mean. Figure 4.17 plots the mean and maximum

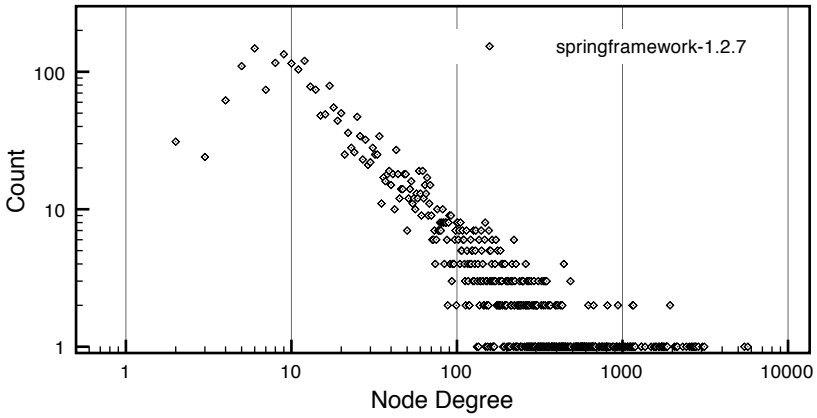

Figure 4.15: Aggregate distribution for all links for *springframework-1.2.7*.

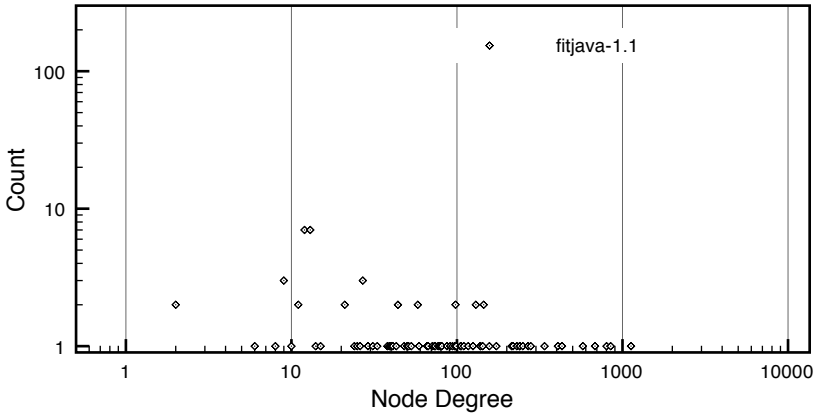

Figure 4.16: Aggregate distribution for *fitjava-1.1*.

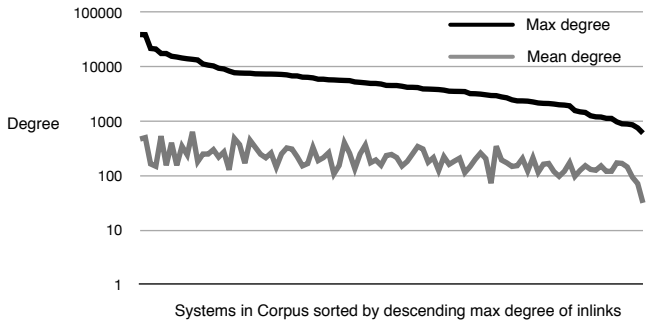

Figure 4.17: Max degree and mean degree for aggregate distributions.

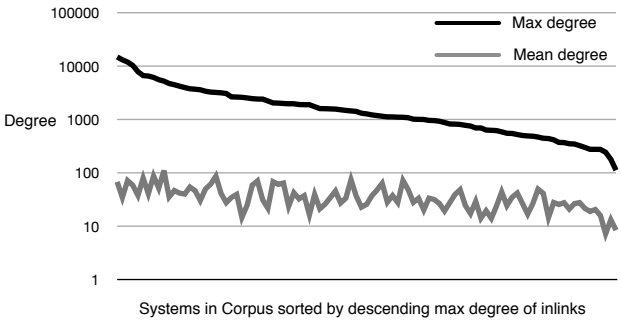

Figure 4.18: Max degree and mean degree for between-module aggregate distributions.

degree for aggregate distributions, and Figure 4.18 plots the mean and maximum degree for between-module aggregate distributions. In both plots, the range of the distribution exceeds the mean degree by at least an order of magnitude, which shows that even though the distributions have few data points, they exhibit scale-free structure. From this, we conclude that approximate scale-free structure is observed in aggregate measures of class connectivity and class coupling.

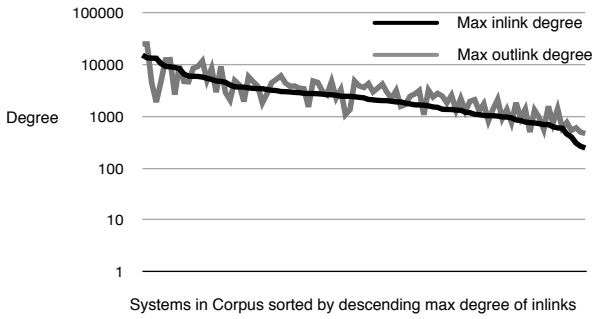

Figure 4.19: Max inlink and outlink degrees for aggregate distributions.

Hypothesis 4 also predicts that because there are no constraints on creating outlinks at the class level, outlink distributions will not be bounded as they are for Hypothesis 3. Figure 4.19 plots the maximum inlink degree and maximum outlink degree for the aggregated distributions. Figure 4.20 shows the same values for the between-module aggregate distributions. Whereas Figure 4.14 shows a clear separation between maximum inlink and outlink degrees, this is not seen in either of the

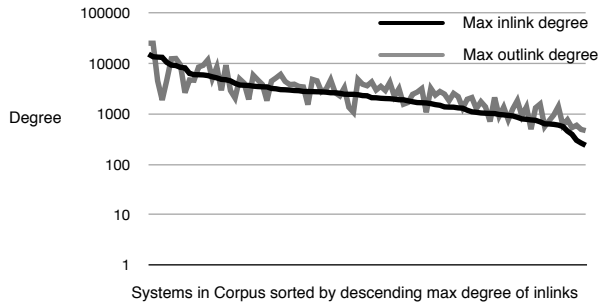

Figure 4.20: Max inlink and outlink degrees for between-module aggregate distributions.

comparison plots for the aggregate distributions. Indeed, both aggregate distributions have a similar range. We conclude that the structure observed in the aggregate measures is observed for both inlink and outlink distributions, and this satisfies Hypothesis 4 for all systems in the corpus.

4.4.5 Hypothesis 5

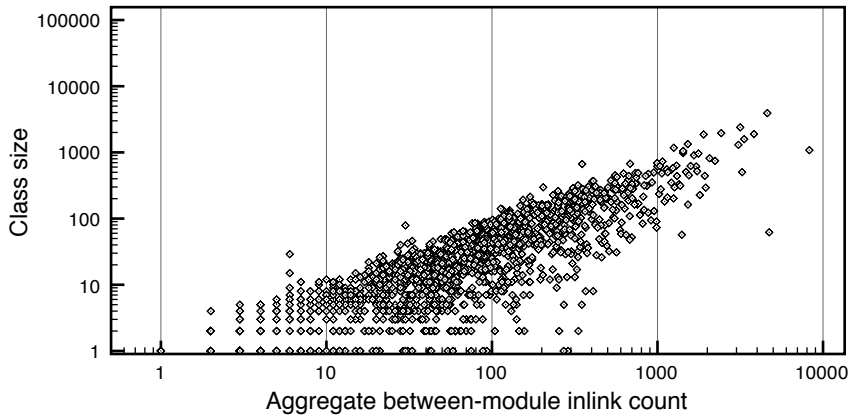

Figure 4.21: Class size versus aggregate between-module outlink count for *derby-10.1.1.0* (*r=0.89*)

For each system, the Pearson correlation coefficient (r) is computed for class size and aggregate between-module outlinks. The mean value of r across all systems is 0.83, with a standard deviation of 0.09. The

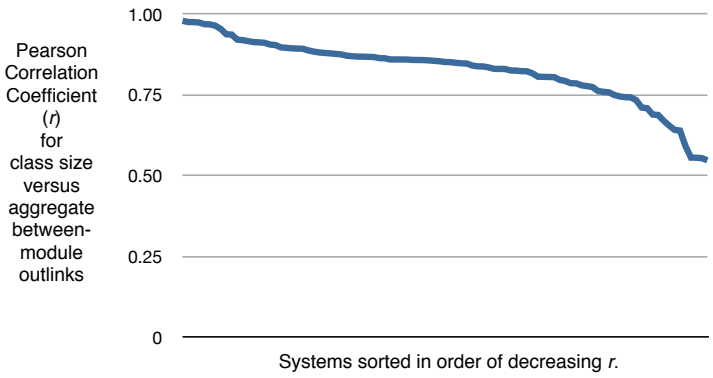

Figure 4.22: Correlation (r) between class size and aggregate between-module outlink count for all systems.

system with the lowest value of r, namely $r = 0.55$, is *junit-4.5*. Figure 4.21 plots class size against aggregate outlink count for *derby-10.1.1.0* ($r = 0.89$). Figure 4.22 shows r for each system in the corpus, sorted into descending order. Three quarters of the systems have correlation exceeding 0.75 and the remaining have correlation of at least 0.55. Based on these results, we conclude that Hypothesis 5 is satisfied.

4.4.6 Post-hoc analysis for Hypothesis 5

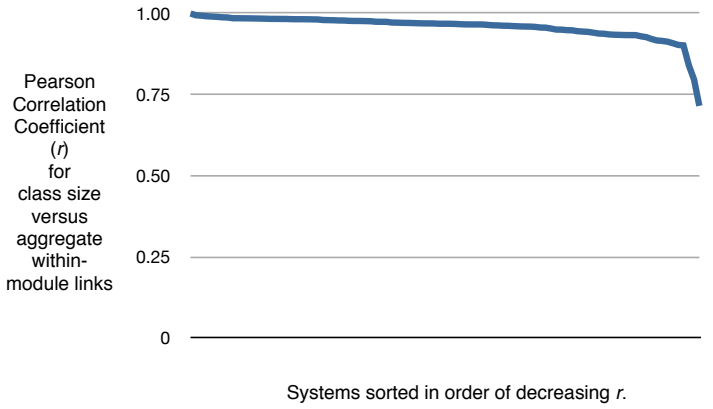

Figure 4.23: Correlation (r) between class size and aggregate between-module outlink count for all systems.

The observed correlation between class size and aggregate between-module outgoing links raises the question as to whether there is a cor-

relation between class size and links that are internal to the class. A post-hoc analysis was performed that computed the Pearson correlation coefficient for class size and aggregate within-module links. This yields a much higher correlation: the mean correlation across all systems is r = 0.92, with σ = 0.04. Figure 4.23 shows the Pearson correlation coefficient between class size and aggregate within-module links for all systems. These results show that when new nodes are added to a class, they are more likely to be accompanied by within-module links than by between-module links. For the corpus, the programmers, in general, favoured cohesive links over coupling links, which is expected.

4.5 Discussion

This empirical investigation demonstrates five properties of software structure that are present for all systems in the Qualitas corpus:

1. The node-level networks are approximately scale-free.
2. Scale-free structure at the node level translates to areas of high coupling.
3. At the node level, inlink distributions differ from outlink distributions in that the latter show evidence of an upper bound.
4. Class-level networks are approximately scale-free.
5. Class size is correlated with aggregate between-module outlink count.

4.5.1 High coupling caused by node-level interaction

All systems in the corpus contain areas of high coupling, which is a direct result of scale-free structure in the source code network. High coupling was predominantly caused by node inlinks, because outlinks are highly constrained in comparison. This suggests that the reduction of high coupling caused by node-level interaction depends on reducing the number of inlinks for highly connected nodes.

An obvious question arises concerning the presence of high coupling due to node inlinking: can it be effectively eliminated by placing arbitrary

limits on the number of inlinks for each node? Consider the case where a programmer required the use of the functionality offered by node *n*, but where that node had already reached its maximum connectivity limit. To make use of its functionality, the programmer has two options:

1. Replicate the functionality of *n*.

2. Use another node *m* to link to *n*, effectively acting as a proxy for it.

Replicating functionality within a software system is known as *code cloning* (Baxter et al., 1998). Although there are cases where using code clones is unavoidable (Kim, Sazawal, Notkin and Murphy, 2005), it is considered poor design because clones are semantically related, but their relationship is not obvious to the programmer, thereby making it possible to update one clone independently of the others. This can result in inconsistent behaviour between different parts of the system. Replacing high coupling with code replication is not a suitable solution.

In the case of node proxies, node *m* would link to node *n*, thereby using up only one of *n*'s inlinks. Nodes that needed to use the functionality of *n* would link to node *m*, which would delegate to node *n*. Should *m* reach its limit of inlinks, another proxy node *k* could be created to act in place of *m*. Theoretically, a proxy structure could be devised to allow for an arbitrary number of uses of *n*'s functionality while limiting to the number of direct connections to *n*.

The problem with a proxy solution, however, is that the proxy nodes are semantically equivalent to the nodes they represent. In the example above, because node *m* is just a proxy of node *n*, any changes to node *n* will be replicated at node *m*. Proxy nodes do not provide a mechanism to reduce the probability of change propagation, so the only point of having them is to avoid limits placed on connectivity. This begs the question as to why limitations should be imposed at all.

4.5.2 High coupling caused by aggregate interaction

Hypotheses 3 and 4 demonstrate that while outlinking does not cause high coupling at the node level, it does contribute to high coupling at the

aggregate level. Hypothesis 5 demonstrates that the number of aggregate outlinks is highly correlated with the number of nodes contained in the aggregate. An aggregate structure that contains few nodes has limited opportunity to generate a large number of outlinks because of the upper bound on outlinking for individual nodes. As an aggregate structure grows, the data suggests that the number of outlinks also grows; therefore the probability of high coupling due to aggregate outlinking increases as the aggregates grow.

The same question arises as to whether high coupling could be effectively managed by limiting the number of outlinks that can occur at the aggregate level. An outlink from a class implies that that class depends on some functionality that is defined externally. If a class were unable to utilize some external functionality because it had reached its outlinking limit, that functionality would have to be brought into the class in order to be usable. This would cause the class to grow, but Hypothesis 5 suggests that increasing the size of a class increases its dependency on external functionality, thereby increasing the pressure for more outlinks. This creates a positive feedback loop where an increase in class size attracts more outlinks, which causes more functionality to be brought into the class, thereby increasing its size again. This, too, is not a viable way to reduce coupling.

4.5.3 Internal validity

It is difficult to use these findings as a basis for an argument about the effectiveness of coupling as a design measure. There is no *a priori* information about the quality of the designs in the corpus, so one cannot correlate measurements taken against systems that are known to be well (or poorly) designed. However, the results hold across the entire set, which means that the properties measured do not differentiate systems in the corpus. Chapter 2 covered the reasoning behind which high coupling is considered to be poor design, but the findings of this investigation suggest that the mere presence of high coupling is not synonymous with poor design. If it were, one would have to conclude that all systems in the Qualitas corpus are poorly designed, but many of these systems

are actively maintained and utilized in production environments and, as a result, satisfy the basic criterion of design quality outlined in this thesis: that the system remains responsive to change pressures. It seems unlikely that all of the systems are poorly designed. A much more plausible argument is that the presence of high coupling is indeed consistent with good design.

4.5.4 Construct validity

Keller (2005) noted that the presence of a particular distribution does not imply a particular underlying or generational process at work. The fact that the empirical results support the five hypotheses does not prove that the proposed evolutionary model is the correct one for software evolution. It does, however, demonstrate that the BA model can be modified to be consistent with the evolution of software systems, and should not be dismissed without a deeper investigation. Since the proposed model is not specifically targeted to software written in the Java programming language, these findings may be applicable to other languages and paradigms, and this idea is supported by the literature (Myers, 2003).

4.5.5 External validity

While this investigation demonstrates that high coupling can be consistent with good design, one should exercise caution when extrapolating the results to systems other than those in the corpus. These systems are all open source and no effort has been made to compare open source systems to proprietary ones.

However, the differing functionality, size, maturity, and modification histories of the systems in the Qualitas corpus suggests some generalizability of these findings. None of the systems in the corpus were immune to the hypothesized effects, thereby suggesting that scale-free structure is independent of these properties.

4.5.6 Open research question

This work suggests that despite the maxim of "high cohesion and low coupling," high coupling is common to all software systems. Are we to conclude that most software systems are poorly designed, or is it possible to reconcile the presence of high coupling with good design practice? Chapter 2 described why high coupling is considered to be poor design: it facilitates the ripple effect, whereby small changes to a part of a system can necessitate system-wide changes. Since all systems in the Qualitas corpus exhibit high coupling, high levels of change propagation should be observable between releases for those parts of systems for which coupling is high. This is addressed in Chapters 5 and 6.

Chapter 5

Patterns of change

> The times they are a-changin'
>
> —Bob Dylan

This chapter describes the mechanisms used to measure change in software systems, and identifies the patterns of change that occur. We begin with a detailed description of the method used to match classes between system versions, which is based on Origin Analysis (Section 2.9).

Measuring the changes that occur between different versions of software systems requires a method of quantification. The proposed method uses eight change measures, which are derived from the main unit of modularization in object-oriented systems—the class. They are described in detail, and to aid in the identification and classification of higher levels of change patterns, a method of visualizing class lifetimes is presented.

The corpus is analysed using the eight measures. These analyses are used to identify and classify the kinds of change that are observed between versions of software. The first analysis examines the performance of the proposed matching heuristic by measuring the number of matched classes and the number of unmatched classes to assess the likelihood that unmatched classes represent errors in the matching process. The second uses a summary of the eight measures computed over the corpus as a whole to identify the frequency of change events. The third analysis investigates the correlation of change measures to identify how they change in relation to each other. The final analysis classifies patterns of change based on how classes change over their lifetime.

The chapter ends with a discussion of the implications of the observed results and their relationship with the structural properties identified in Chapter 4. This discussion provides the basis for the investigation described in Chapter 6.

5.1 Automated Matching Method

Automated Matching Method (AMM) is an automated process that is based on Origin Analysis. It is performed in two passes, where the first pass matches classes whose internal structure has had limited change, and the second pass matches classes based on interactions with other classes.

Because of the size of the corpus under investigation, a semiautomatic approach is unfeasible for two reasons. First, the volume of matches that are expected in a corpus of this size makes it unlikely that the work could be completed within a reasonable schedule. Second, no researcher will have enough experience with all systems in the corpus to exercise consistent judgment.

5.1.1 First pass—Applying matchers

The first pass uses four matchers, which each compare independent properties of classes—*Nameset similarity*, *Abstract/concrete*, *Class/interface*, and *Simple name*. The Nameset similarity matcher implements the primary similarity measure that is used to identify candidate matching classes. Classes that satisfy its criteria are included as possible candidates, and those that satisfy the criteria of the remaining three matchers are excluded as candidates.

Semgraphs represent classes by a subtree where the class declaration node is the root. A class's Nameset is the set of all names used by the class declaration node and its children. Named nodes are variable, parameter, method and non-anonymous inner class declarations. The names that are given to the nodes in a class are associated with its semantics (Church, 1943), so they are only likely to change when the semantics of the class change or when a programmer refactors the nodes

so that they are better aligned with the semantics of the class.

Using the measure of similarity stated by Bunge (1977), we compute the similarity between namesets as:

$$N.similarity(A, B) = \frac{|NS(A) \cap NS(B)|}{Min(|NS(A)|, |NS(B)|)} \tag{5.1}$$

where $NS(A)$ is a function that computes the nameset for class A.

The *Abstract/concrete* matcher excludes candidate classes that are not exclusively *abstract* or *concrete* between system versions. A concrete class represents a fully defined entity in a software system that can be instantiated. An abstract class, however, is an artefact of a factoring process that provides a single repository to house code that would otherwise be duplicated across multiple classes. Because abstract classes maintain code that is common to many classes, they tend to possess a similar structure to their subclasses, which can confuse the Nameset matcher and cause it to identify too many candidate classes. This matcher ensures that concrete classes can only be matched with concrete classes, and that the same is true for abstract classes.

Like the Abstract/concrete matcher, the *Class/interface* matcher does not allow classes to match with interfaces. An interface defines a set of methods that must be supplied by implementing classes. The use of interfaces can produce a series of classes that possess similar structure, making it difficult to match using Namesets.

The final matcher excludes classes that do not have the same *Simple name* between versions. Software systems have several classes that exhibit similar structure—such as code clones, sibling classes in inheritance hierarchies, and classes that implement interfaces—so matchers based on structure are likely to produce too many candidates. Since the aim of the first pass is to match classes that have exhibited little or no change, this matcher will only allow candidates whose Simple name has not changed. Classes whose Simple name has changed between system versions must be matched by pass 2.

Given these four matchers, the first pass works as follows. The similarity measure is computed for each pair of classes between the classes

in version A—the earlier version—and those in version B. Class pairings are grouped when

$$N.similarity(A, B) > 0.8.$$

Once all groupings have been made, the remaining matchers are applied to the groups, and those classes with a single remaining candidate are matched. Any classes that do not have exactly one candidate, or two or more candidates are forwarded to pass 2 of the process.

To determine an appropriate threshold, an *ad-hoc* sensitivity analysis was performed. The matchers were applied to corpus with an initial threshold value of 0.25. For each subsequent application, the threshold value was increased by 0.05, until a threshold value of 1.0 was achieved. The analysis showed that the number of matches remained most stable between threshold values 0.6 and 0.9. Based on these results, the threshold value of 0.8 was chosen because it provided a reasonable number of matches, and was high enough to provide reasonable confidence that it would produce fewer false positives than false negatives.

5.1.2 Second pass—Dependency analysis

The second pass uses dependency analysis to resolve matching for classes that exhibit more than the minimal internal change. Figure 5.1 illustrates how dependencies are used to match classes. In this diagram, there are two system versions (*A* and *B*) and three classes (*C1*, *C2*, and *C3*), where the first pass matched *C1.A* with *C1.B*, and *C3.A* with *C3.B*. However, *C2.A* was not matched by the first pass to *C2.B*.

Pass two computes the *inlink set* and *outlink set* for each unmatched class in system version *A* and for all classes in system version *B*. A class's inlink set is the set of classes that use its functionality, and its outlink set is the set of classes that it uses. In the above example, class *C2.A* is used by class *C3.A* and uses classes *C3.A* and *C1.A*. Although both *C1.A* and *C3.A* are in the earlier version of the system, they have both been matched to classes in the later version, and it is those matched classes that are added to the inlink and outlink sets of *C2.A*.

Class similarity is computed by comparing the similarity between inlink

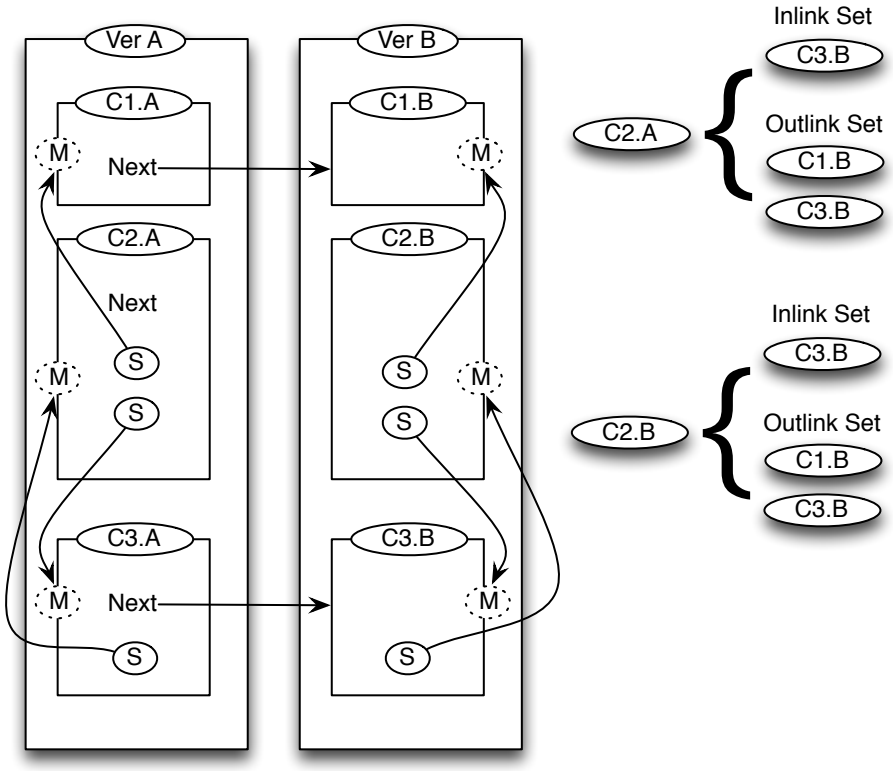

Figure 5.1: Matching classes between versions—pass 2.

and outlink sets. This comparison is performed using the same formula that is used to compare Namesets in pass 1.

$$I.similarity(A,B) = \frac{|IS(A) \cap IS(B)|}{Min(|IS(A)|, |IS(B)|)} \tag{5.2}$$

where $IS(A)$ is a function that computes the inlink set for class A. The similarity of outlink sets ($O.similarity$) is computed in the same way.

For classes to be candidates, they must satisfy the following criteria:

$$Min(I.similarity(A,B), O.similarity(A,B)) \geq 0.5$$

and

$$Max(I.similarity(A,B), O.similarity(A,B)) \geq 0.7.$$

Using these thresholds allows one set of links to change more than the other, but at least one of the sets must remain stable for a candidate to

be considered.

To perform the actual matching, each unmatched class in the earlier system version is compared against all classes in the later version, and classes whose similarity measures exceed the specified thresholds are grouped. As in pass 1, AMM judges two classes to be matched only in the case where there is a single matching candidate. In cases of no matching candidates or two or more candidates, no match is made.

To determine appropriate threshold values, an *ad-hoc* sensitivity analysis was performed on all classes that weren't matched by the first pass. All combinations of minimum and maximum thresholds were tested using increments of 0.05, where $0.25 \leq threshold_{min} \leq 0.95$, $0.3 \leq threshold_{max} \leq 1.0$ and $threshold_{min} < threshold_{max}$. The number of matches decreased significantly for values $threshold_{max} \geq 0.75$ and increased significantly for values $threshold_{min} \leq 0.45$. Based on these findings, the thresholds $threshold_{min} = 0.5$ and $threshold_{max} = 0.7$ were chosen.

5.2 Automated Matching Method performance evaluation

AMM was evaluated using three criteria:

1. The number of unmatched classes between release pairs.

2. The number of release pairs that have a large percentage of unmatched classes.

3. The number of unmatched classes that are highly coupled.

AMM was applied to the "e" release of the Qualitas corpus, which contains 334 versions of 12 independent software systems. The systems in the corpus were ordered based on their version number and release date, which is maintained as part of the corpus metadata (Tempero et al., 2010).[1] A full list of software system versions and pairings can be found in Appendix C.

[1] The "alpha" and "beta" versions of the *hibernate* system were excluded because they are not part of the normal release stream.

5.2.1 Unmatched classes

When a class in an earlier version of a pairing fails to match a class in the later version, there are two possible reasons:

1. The class was deleted from the system and no matching class exists in the later version.

2. The heuristic failed to find the appropriate matching class.

In case 2, a matching error has occurred. These represent the difficult cases for which an expert's judgment would be used in the semiautomated methods.

Table 5.1 shows the total number of matching opportunities, the number of classes matched (94%), and the number of unmatched classes (6%). Unmatched classes are broken down into five categories, each representing the reason that a match did not occur. Most unmatched cases were because of either one or both of the inlink and outlink sets were empty. In the final two cases, there were either no candidates or two or more candidates.

When using dependency analysis, the similarity of inlink and outlink sets must exceed 0.5 for both sets and 0.7 for one of the sets. In the case where one (or both) of the sets is empty, no suitable match can be found because of insufficient matching criteria. When a class has an empty inlink or outlink set, that signifies that none its interactions could be matched against classes in the later version, or that the class itself has no interaction with other classes. In each of these cases, dependency analysis cannot identify a match because there is no criteria for computing similarity. Of the 20,124 unmatched classes, failure to match

Total	Matched	Unmatched				
351335	331211 (94%)	20124 (6%)				
		I	O	B	N	>1
		11582 (3%)	449 (<1%)	7014 (2%)	51 (< 1%)	1028 (< 1%)

Table 5.1: Measures to assess quality of the matching process. I=Empty Inlink Set, O=Empty Outlink Set, B=Both sets empty, N=No match found, >1=Too many candidates

because of insufficient criteria accounts for 95% (19,045). If these cases were brought before experts in a semiautomated analysis, they would have similar difficulty in assessing whether a match should occur, because lack of information is a weak criterion for making an assessment.

The number of classes that were unmatched because there were too many candidate classes is 1028. These cases are most likely indicative of small code clone classes whose evolution has been limited, so they retain similar structure and interaction patterns to other clones. Here, it is likely that an expert would perform far better than AMM, but these cases represent less than 0.3% of the total cases and 5% of the unmatched ones.

In the remaining 51 cases, no matches could be found through both a comparison of structure and dependency analysis. It is likely that these cases represent classes that have been removed. Of the 20,124 classes that could not be matched, it is likely that more than 51 classes were removed, but their removal cannot be confirmed by this measure. Godfrey and Zou (2005) reported deletion rates as high as 18% for one pairing of the PostgreSQL system, with a mean rate of 3.5%. Since class deletions do occur, it is unlikely that all of the unmatched classes are due to failure of the matching heuristic.

5.2.2 Unmatch rates for individual pairings

The average number of classes that were unmatched between system versions is 6%. However, there are 14 system version pairings, shown in Table 5.2, whose unmatched classes exceed 15%. In the first six, the number of total classes decreases, so it is only to be expected that many classes will be unmatched. The next four saw a large increase in the number of classes along with an increase in major version number, suggesting a significant modification to the system. The next two cases are small systems, so the 15% match represents a small number of actual classes, and there appears to be a gap in the data between *junit 3.0* and *junit3.4*. In the last two cases there was an increase in the number of total classes, but a subsystem containing a large number of classes was removed.

Earlier Version	Later version
antlr 2.7.7	*antlr 3.0*
jmeter 2.2	*jmeter 2.3-rc3*
jung 1.2	*jung 1.3*
jung 1.7.6	*jung 2.0*
junit 3.8.2	*junit 4.0*
lucene 2.2.0	*lucene 2.3.0*
azureus2.0.8.5	*azureus 2.1.0.0*
hibernate 0.8.1	*hibernate 1.0*
hibernate 1.1	*hibernate 2.0-rc2*
hibernate 2.1.8	*hibernate 3.0-rc2*
junit 2.1	*junit 3.0*
junit 3.0	*junit 3.4*
ant 1.3	*ant 1.4*
junit 4.4	*junit 4.5*

Table 5.2: List of release pairs with > 15% unmatched classes.

In most of these cases, a large number of classes were removed, which explains the high rate of unmatched classes. In the four cases where the systems increased in size, they appear to have significant modification, which could either indicate that the classes were deleted or that there was a large amount of class splitting, which could have confused AMM.

5.2.3 Unmatched classes correlated with coupling

We have already determined that coupling exhibits approximate scale-free structure, so the dataset has a large number of classes that exhibit low coupling and fewer that exhibit high coupling. Since the focus of the analysis of change is to correlate levels of change with measures of

Reason for failure	High Coupling
Empty inlink set	1423
Empty outlink set	99
Both sets empty	424
Too many matches	370
No matches	37
Total	2353

Table 5.3: Number of unmatched classes with > 50 between-module links.

coupling, a loss of data points that exhibit low coupling will have little impact on the analysis because there are many. However, because there are fewer data points that exhibit high coupling, we should first ask whether a significant number of those points may be lost because of unmatched classes.

To make an assessment, the aggregate between-module incoming and outgoing links are recorded for each unmatched class. If either the incoming or outgoing links exceed 50—a conservative threshold—the class was judged to exhibit high coupling. Table 5.3 shows the number of highly coupled classes that remained unmatched for each of the reasons noted in Section 5.2.1. The total of 2353 represents 12% of the unmatched classes and < 1% of the total matching opportunities.

5.2.4 Discussion

It is not known how many of the unmatched classes represent ones that have been deleted by programmers and how many are errors caused by a failure of AMM. If all unmatched classes were in error, the total error rate represent 6% of the data points. If only classes that are highly coupled are considered, the error rate drops below 1%. These numbers represent an upper bound to the number of false negatives.

In cases where there were insufficient criteria for making a match, the default was to make no match. Also, in the case where too many possibilities were available, no match was made. These characteristics, in conjunction with the high thresholds used in the similarity metrics, makes it likely that there are fewer false positives than false negatives.

AMM possess some advantages over using an expert-based approach. Subjective judgment makes the matching process less reproducible because the underlying process is unquantifiable. The approach used here is described fully, and can be reproduced by other researchers.

Another problem associated with human participants are issues of fatigue and cognitive bias, which can impair their judgment (Tversky and Kahneman, 1974). While some biases can be controlled by experimental design, it is unlikely that fatigue would not be a factor for a corpus of the size used.

5.3 Measuring change

We now describe the eight measurements that are used to represent how classes change over time.

5.3.1 Computed measures

Figure 5.2 illustrates the eight change measures used. Each is numbered, but to simplify the diagram, only between-module measures are labelled. Measure 1 represents the number of the nodes contained by the module, and measures 2 through 8 represent links. While modules can be any hierarchical element that contains child nodes, this section assumes that modules are classes.

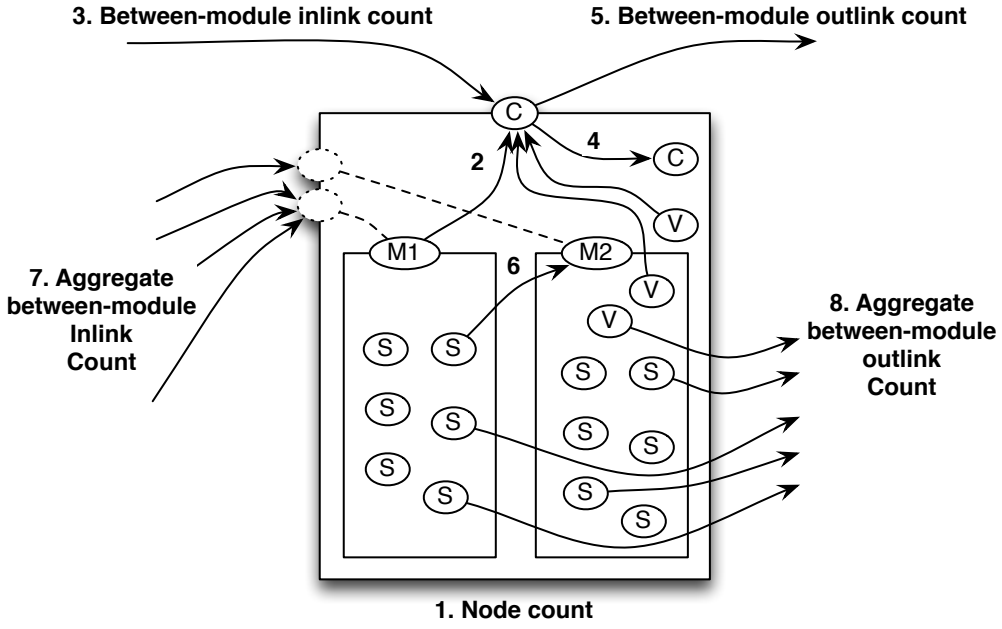

Figure 5.2: Eight parameters of change for a module.

Measure 1—Node count

The first measure is the number of nodes contained in the module, which can be seen as the number of nodes within the class rectangle in Figure 5.2. The addition and removal of nodes are reflected by changes to this measure.

Measure 2—Within-module inlink count

This measure is the number of inlinks to the module's root node that originate from within the module itself, which is shown as links from nodes within the rectangle to the class's root node on Figure 5.2. Links of this type are variable and parameter declarations whose type is the class, and method declarations whose return type is the containing class. Less common cases are inner classes that are defined as a subclass of the containing class. Changes to this measure indicate that the level of usage that originates from within the class has changed.

Measure 3—Between-module inlink count

This measure represents the number of inlinks to the module's root node that originate from outside the module's boundary, which reflects the level of usage of the class as a type by other nodes in the system. On Figure 5.2, this is shown as links to the class's root node that originate from outside of the rectangle.

Measure 4—Within-module outlink count

This measure reflects the number of outlinks from the module's root node that are resolved within the module itself. Outlinks from a class's root node are superclass and interface relationships. Since it is very unusual for such relationships to be resolved within the class itself, it is expected to be zero in most cases. On Figure 5.2, this is shown as links originating from the class's root node and terminating at a node within the rectangle.

Measure 5—Between-module outlink count

This measure represents the amount of dependence between the module's root node and nodes that are external to the module. Since the Java programming language only allows single inheritance and classes are likely to implement few interfaces, its value is expected to remain low. On Figure 5.2, this is shown as links originating from the class's root node and terminating with a node outside the rectangle.

Measure 6—Within-module aggregate link count

This measure represents the number of links between nodes contained within the module, not including links to the module's root node, which is indicative of the level of internal dependence exhibited by the class. On Figure 5.2, this is shown as a relationship between two nodes within the class's rectangle.

Measure 7—Between-module aggregate inlink count

This measure reflects the number of inlinks to nodes contained by the module that do not originate from within the module itself, which reflects the level of dependence upon the class's public interface by the rest of the system. While links to both method and variable declarations can be included, object-oriented design rules encourage encapsulating variable declarations (Meyer, 2000), so they are less likely to be accessed from external sources. On Figure 5.2, this is shown as a relationship between external nodes and class interface nodes.

Measure 8—Between-module aggregate outlink count

This measure reflects the number of aggregate outlinks from nodes contained in the module to nodes defined outside the module's boundaries, which indicates the level of dependence between the class and the rest of the system. On Figure 5.2, this is shown as a relationship between nodes within the rectangle and external nodes.

5.3.2 The lifetime of a class

Figure 5.3 shows measures for an example class over eight system versions, and Table 5.4 illustrates the corresponding change matrix. The table has nine columns, the first column showing the version pair number and the remaining columns the class's change measures. The first and last rows show the initial and ending values for the measures, and the intervening rows show the amount of change of each measure for each version pairing. The purpose of the change matrix is to illustrate

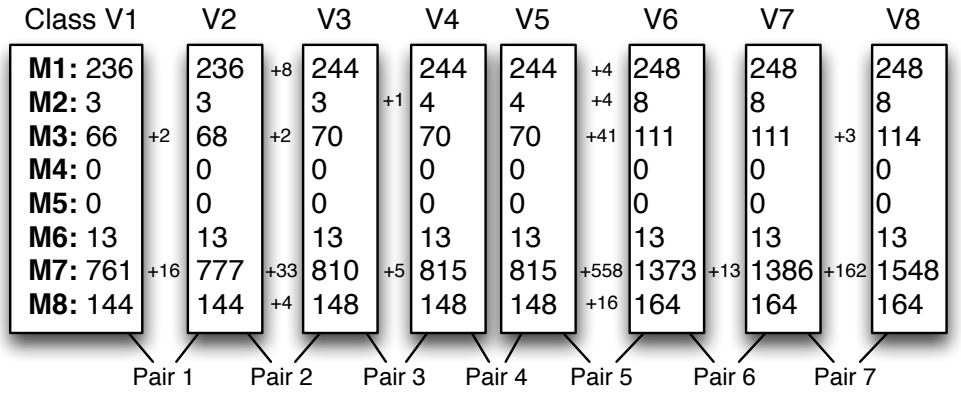

Figure 5.3: Change measures for eight versions of the same class.

Pair	M1	M2	M3	M4	M5	M6	M7	M8
	236	3	66	0	0	13	761	144
1	0	0	2	0	0	0	16	0
2	8	0	2	0	0	0	33	4
3	0	1	0	0	0	0	5	0
4	0	0	0	0	0	0	0	0
5	4	4	41	0	0	0	558	16
6	0	0	0	0	0	0	13	0
7	0	0	3	0	0	0	162	0
	248	8	114	0	0	13	1548	164

Table 5.4: Change matrix for example class

the amount of change that occurred for each step in a class's lifetime.

The initial values for this class indicate that it is well established, with a large number of nodes (M1=236). It has moderate coupling to the class declaration (M3=66), high aggregate inlink coupling (M7=761), and high aggregate outlink coupling (M8=144). There is little interaction between nodes within the class (M6=13). The change matrix has seven version pairing rows indicating that the class was matched in eight successive versions of the software. Throughout those releases, the internal structure remains relatively stable as there are no changes to internal links, and only 12 nodes (M1) and 20 aggregate outgoing links (M8) are added. However, there is more change in the usage of this class by the rest of the system. Links to it increase by a total of 48 (M3), and the number of aggregate incoming links increases by a total of 787 (M7).

5.4 Analysis of change

AMM was applied to the "e" release of the Qualitas corpus. 18,758 unique class lifetimes were identified in the corpus, and their change matrices computed. Three analyses are performed. First, the number of zero-value change measures is computed. Second, the correlation of each change measure pair is computed. Finally, a nomenclature for describing patterns of change is defined and their frequencies computed.

The semgraph files for some software versions can exceed 200 MB. Combined with multiple system versions, this makes it impractical to perform the analysis within the memory space of a single process. Therefore, change analysis is performed on each system version pairing and the results are stored in a file. After the change analysis has been performed for each pair, the files are aggregated to produce change matrices for all classes across all system versions. Change analysis is computed for each pairing independently on the computing cluster using the tools described in Chapter 3.

5.4.1 Confounding factors

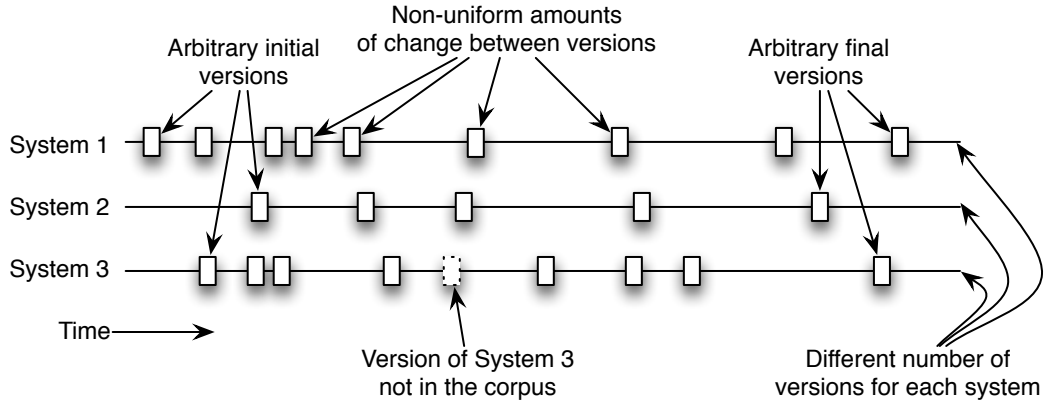

Figure 5.4: Non-uniformity in the Qualitas corpus.

The variability between releases and between systems in the corpus represents confounding factors. Irregularities in the systems are shown in Figure 5.4. Each system has a different number of versions, which makes comparison difficult in the absence of normalization. Also, the ini-

tial and final versions of each system are not constrained to a particular point in the system's lifetime. For example, *hibernate*'s initial version in the corpus is 0.8.1 where *antlr*'s initial version is 2.4.0. For some systems, some versions were released, but are not present in the corpus because they do not meet the inclusion criteria, creating gaps in the dataset. Finally, there is no normalization of the amount of change represented by each version, so direct comparison between versions must be done with care.

5.4.2 Change measures that equal zero

A cursory analysis suggests that the change matrices are sparse. In Table 5.5, the column *CR* shows the number of change rows for each system, along with the total for the corpus. Column *ZR* shows the number of rows that contain only zero-valued entries, and the remaining columns show the number of zero-valued entries for each measure.

Of the 331,211 change rows, 81.2% contained all zero entries. Columns M2, M4, and M5 reveal that those measures do not change very often: they have 99.6%, 99.9%, and 99.3% zero-valued entries respectively. Measure M8 has the fewest zero-valued entries at 88.8%, followed by M7 at 91.5%. Both M7 and M8 represent coupling through inlinking and outlinking respectively, and this suggests that of the changes that can occur, changes in coupling are most likely. Contrast M7 and M8 with M6, which is zero-valued 92.3% of the time. The structure of internal class in-

System	CR	ZR	M1	M2	M3	M4	M5	M6	M7	M8
ant	13919	10704	11817	13839	13113	13919	13850	12012	12591	11737
antlr	3589	2493	3094	3563	3105	3589	3575	3186	3042	3035
argouml	15703	9667	12091	15502	14508	15703	15301	12817	13179	11417
azureus	138429	119131	129221	138244	132718	138429	137953	131409	128964	127568
freecol	8857	5859	6797	8774	7866	8857	8768	7217	7384	7002
hibernate	76827	66864	71807	76650	73708	76827	76525	72776	72138	71758
jgraph	2962	2434	2705	2955	2775	2962	2962	2739	2693	2718
jmeter	12556	8970	10199	12403	11596	12556	12479	10826	11204	10477
jung	7367	5587	6732	7344	6571	7367	7284	6818	6367	6533
junit	2913	2093	2387	2890	2617	2913	2879	2616	2572	2466
lucene	17621	13437	15121	17491	16276	17621	17436	15468	15768	14924
weka	30468	21676	26492	30335	28572	30468	29902	27693	27313	24349
Total	331211	268915	298463	329990	313425	331211	328914	305577	303215	293984
Percent of Total		81.2	90.1	99.6	94.6	99.9	99.3	92.3	91.5	88.8

Table 5.5: Number of change measures that equal 0. CR=Change Row Count, ZR=Count of rows where all entries = 0, MX=Count of rows with measure X = 0

teraction appears to be slightly more stable than that of between-module interaction.

M3 shows that the number of links to class declaration nodes remain unchanged 94.6% of the time. That M3 changes less often than M7 suggests that programmers do not always have to declare new variables when they choose to invoke methods, but rather use existing links. M1 remains unchanged 90.1% of the time, which suggests that between system versions, most classes do not change in size.

5.4.3 Correlation of change measures

The proposed model of software evolution and the correlation between class size and between-module links revealed by Hypothesis 5 suggests that some of the measures are correlated. Table 5.6 shows the Spearman rank correlation coefficient (r_s) for each pair of the eight computed change measures. This table shows that most of the measures have a low correlation, with p < 0.001 for all entries. Measure M4 has <0.06 correlation with all other measures, M5 has <0.18 correlation with all other measures, and M2 has <0.22 correlation with all other measures.

M3 has a low correlation with all other measures (< 0.16), with the exception M7 (0.44). It is expected that M3 would have a higher correlation with M7 because a reference to a class declaration (M3) is required in order to invoke methods on that class (M7). If a programmer needs to invoke a method on a class and a variable of the necessary type is not available within its scope, a new variable is required, increasing both M3

	M1	M2	M3	M4	M5	M6	M7	M8
M1	–	–	–	–	–	–	–	–
M2	0.09	–	–	–	–	–	–	–
M3	0.06	0.15	–	–	–	–	–	–
M4	<0.01	<0.01	0.02	–	–	–	–	–
M5	0.14	0.10	0.03	0.05	–	–	–	–
M6	0.84	0.21	<0.01	< 0.01	0.17	–	–	–
M7	0.11	0.20	0.44	< 0.01	0.07	0.09	–	–
M8	0.78	0.08	0.08	< 0.01	0.14	0.71	0.09	–

Table 5.6: Correlation of change measures across the Qualitas corpus

and M7. If method invocations are removed, the references to the class are no longer necessary, decreasing both M3 and M7.

M1 has low correlation (< 0.15) with measures M2, M4, and M5, which suggests that there is little relationship between a class's size and is internal links to the class declaration node. Similarly, M1 has a very low correlation with M3 and M7, which implies that changes in class size do not accompany changes in how the class is used. However, M1 has a higher correlation with M8 (0.78) and M6 (0.84), which is predicted by Hypothesis 5. When nodes are added to an existing classes, they are accompanied by both within-module and between-module links.

The correlation between M1, M6, and M8 shows that programmers tend to favor within-module links over between-module links. This is an important finding as it demonstrates that the high coupling identified by Hypotheses 1 through 4 occurs in spite of a greater focus on within-module links. The high correlation between M1 and M8 provides some support for Lehman's laws of software evolution (Lehman, 1996; Lehman, Ramil, Wernick, Perry and Turski, 1997). In the face of continuing change (law 1), programs tend to grow in size (law 6), while at the same time their complexity increases (law 2) and their quality declines (law 7), unless rigorous attempts are made to maintain complexity and quality. As classes grow in response to change pressures, complexity is added through increased between-module dependencies, which increases the probability of change propagation.

5.4.4 Discussion of global observations

Over the lifetimes of the systems in the corpus, there is little change to existing code. Class size and incoming and outgoing between-module links changes in only 8%–11% of the cases, which demonstrates that most classes remain stable despite the change pressures that have been applied. That between-module incoming links to the class declaration change less often than incoming links to child nodes implies that the increased use of classes occurs in contexts where the class is already being used.

The two areas of correlated change appear to be independent. The us-

age of classes is independent of their internal structure because changes in size, internal links, and external outgoing links are uncorrelated with incoming links. This shows that increases in usage of a class do not accompany changes to its structure, which suggests that as classes are used more they tend to change less frequently.

5.5 Observed patterns of change

In this section, a method for classifying the changes observed within classes is proposed and applied to the corpus. We first describe pattern classification, which is supported by examples from the corpus, and then present the frequency of occurrence of the various classifications at both the measure and the class levels.

5.5.1 Pattern classification

We use five labels to represent the frequency of change that is observed over a class's lifetime: *unchanging*, *stable*, *moderately stable*, *moderately unstable* and *unstable*. Labels are assigned to each measure in the change matrix based on the percentage of zero entries that are present, which is shown by Table 5.7.

The label assigned to a class is the same as that assigned to the majority of its measures, and in the event of a tie, the label denoting the least change is assigned. This means that the class label describes the pattern of change of most of its measures, so that only those measures that are different need to be expressed. For example, a class with all measures having zero entries is called *unchanging*. However, a similar class with a

Classification	Percentage of Zero values (p)
Unchanging	p = 100
Stable	100 > p $\geq$ 75
Moderately Stable	75 > p $\geq$50
Moderately Unstable	50 > p $\geq$ 25
Unstable	25 > p$\geq$ 0

Table 5.7: Pattern classification based on the percentage of zero-valued entries.

stable M1 and an *unstable* M8 is called an *unchanging* class with *stable* M1 and *unstable* M8. Because measures M2, M4, and M5 exhibit little change, they are not considered when the class label is assigned, and are not presented in any examples.

5.5.2 Example classifications

Tables 5.8, 5.9, and 5.10 show the change matrices for three example classes taken from the *ant* system.

The first is the class `org.apache.tools.ant.BuildEvent`, which is an *unchanging* class with *stable* M1, *moderately unstable* M3, and *unstable* M7. This highlights several important aspects of the class. First, its internal structure has changed little, which means that little modification has occurred throughout its lifetime in the corpus. Second, there are many changes in how the class is used, so it likely holds a key role in the system. Finally, M3 is more stable than M7, which means that the

Pair	M1	M3	M6	M7	M8
	52	31	27	26	12
1	0	7	0	54	0
2	0	44	0	46	0
3	0	8	0	56	0
4	0	0	0	0	0
5	0	61	0	127	0
6	0	0	0	2	0
7	0	0	0	2	0
8	0	0	0	1	0
9	0	0	0	–1	0
10	0	14	0	32	0
11	0	0	0	0	0
12	0	8	0	3	0
13	0	7	0	8	0
14	0	0	0	0	0
15	0	0	0	–2	0
16	0	–11	0	6	0
17	1	8	0	2	0
18	0	30	0	23	0
19	0	0	0	–1	0
	53	207	27	385	12

Table 5.8: Unchanging class with stable M1, moderately unstable M3, and unstable M7

usage of this class is not only changing in new code, but in existing code as well.

The second is the class `org.apache.tools.ant.BuildException`, which is *moderately stable* class with *stable* M8 and unstable M3 and M7 (see Table 5.9). This is similar to the first class except that its internal structure does change, and both M3 and M7 change in similar ways. This suggests that the class sees minimal modification, and is heavily used by both new and existing classes in the system.

The third (Table 5.10) is the class `org.apache.tools.ant.Path`, which is *moderately unstable* with *unstable* M7. This class is heavily used and has a reasonably high degree of change. Unlike the first two examples, the internal structure of this class changes throughout its lifetime while at the same time its usage increases. This class has the potential to be a problem in the system because it is highly coupled and exhibits a high frequency of change.

Pair	M1	M3	M6	M7	M8
	40	146	13	195	7
1	5	153	2	203	1
2	23	101	15	187	1
3	0	156	2	236	0
4	0	–1	0	1	0
5	0	317	0	329	0
6	3	1	1	2	0
7	0	12	0	22	0
8	0	0	0	3	0
9	0	3	0	3	0
10	0	100	0	312	0
11	0	5	0	23	0
12	0	13	0	64	0
13	0	41	0	28	0
14	0	1	0	–1	0
15	0	–1	0	–3	0
16	0	68	0	128	0
17	1	15	0	19	0
18	–28	43	–22	25	–1
19	0	–2	0	–3	0
	44	1171	11	1773	8

Table 5.9: Moderately stable class with stable M8 and unstable M3 and M7

Pair	M1	M3	M6	M7	M8
	97	49	70	88	8
1	165	48	115	93	61
2	53	48	41	40	8
3	–1	75	2	135	2
4	0	0	0	0	0
5	77	83	44	97	72
6	0	4	0	4	0
7	10	4	10	12	6
8	0	0	0	–1	0
9	0	0	0	0	0
10	27	43	16	68	11
11	0	0	0	–1	0
12	4	8	2	7	1
13	–8	6	–8	3	–2
14	0	0	0	1	0
15	0	0	0	0	0
16	–58	32	–63	–40	–20
17	23	5	7	4	3
18	7	9	3	1	2
19	0	1	0	2	0
	396	415	239	513	152

Table 5.10: Moderately unstable class with unstable M7

5.5.3 Pattern frequency

Table 5.11 shows the frequency of occurrence of each pattern for the measures M1, M3, M6, M7, and M8, and for classes. The table shows 8,262 instances of *unchanging* classes, which represents 44% of the cases. Classes are *stable* 38% of the time and *moderately stable* 15% of the time. Classes are *moderately unstable* slightly less than 2% and *unstable* < 1% of the time.

M3 exhibits the least amount of change, with an *unchanging* rate of 55% and a *stable* rate of 32%. M6 and M7 have similar rates, with an

Measure	Unchanging	Stable	Mod. Stable	Mod. Unstable	Unstable
M1	6682 (36%)	8022 (43%)	3235 (17%)	587 (3%)	232 (1%)
M3	10378 (55%)	5961 (32%)	1956 (10%)	318 (2%)	145 (1%)
M6	8336 (44%)	7020 (37%)	2736 (15%)	474 (3%)	192 (1%)
M7	8370 (45%)	7040 (38%)	2528 (13%)	541 (3%)	279 (1%)
M8	6642 (36%)	7310 (39%)	3796 (20 %)	728 (4%)	282 (1%)
Class	8262 (44%)	7208 (38%)	2666 (15%)	445 (2%)	177 (1%)

Table 5.11: Frequency of patterns observed for each measure and for classes

unchanging classification accounting for 44%–45% and *stable* accounting for 37%–38%. M1 and M8 have the lowest *unchanging* rates at 36% each. For all of the measures, *unchanging*, *stable*, and *moderately stable* account for approximately 95%–96% of all observable patterns.

5.6 Discussion

We have seen that 81.2% of the change rows in the corpus contain all zero entries (Section 5.4.2). The measures, when grouped by class lifetimes, contain all zero entries 36%–55% of the time, depending on the measure. Similarly, the measures have either *unchanging*, *stable*, or *moderately stable* 95%–97% of the time (Section 5.5.3). Classes that are *moderately unstable* and *unstable* account for only 3%–5% of the cases, which suggests that the frequency of overall change to existing classes is relatively low. Since Chapter 4 revealed that all of the systems in the corpus exhibit areas of high coupling, these observations raise questions about the relationship between high coupling and the ripple effect.

The primary concern about highly coupled classes is that they have the ability to propagate change to a large number of classes. However, the observations presented in this chapter suggest that it has not occurred in any of the systems in the corpus. The modification of existing classes is reflected by changes in the number of nodes, within-module links, and aggregate between-module links. However, the corresponding change measures—M1, M6, and M8—do not show evidence of widespread change. Indeed, these measures show a high degree of stability.

The analyses performed in sections 5.4.2 and 5.5.3 addresses the frequency that classes change, and do not consider change that is propagated between connected classes. Questions about the relationship between coupling and change propagation are addressed in Chapter 6.

Chapter 6

High coupling and software evolution

> We come and go just like ripples in a stream.
>
> —John V. Politis

In Chapter 4 we determined that high coupling is a common property of software systems, and in Chapter 5 we observed that as software grows, changes to existing code are infrequent. Now we examine structural changes in relation to coupling. First, do changes in coupling measures provide support for the software evolution model proposed in Chapter 4? Second, can we observe the ripple effect that is predicted by the presence of high coupling?

6.1 The evolution of inlink coupling

While there are studies that have measured coupling in software systems (Chidamber and Kemerer, 1991; Li and Henry, 1993; Chidamber and Kemerer, 1994; Briand et al., 1999b; Baxter et al., 2006; Gao et al., 2010), I have found no investigation of how coupling itself evolves. Consider the change matrix in Table 6.1, which shows the lifetime of a *moderately stable*, highly coupled class. The class is *weka.core.Instances*, and it is the most used class in the *weka* system (Witten, Frank and Hall, 2011). It begins as a large class with 1251 children nodes (M1), and it is highly coupled with 370 inlinks to its declaration (M3), 1267 inlinks to its chil-

dren nodes (M7), and 401 outlinks from its children to other nodes in other classes (M8).

The first five rows show little change. However, in the remaining history, M7 has only three zero entries and M3 has only five, which makes those measures *unstable*. Measures M1 and M8 also exhibit change, but have more zero rows, so they are classified as *moderately unstable*. Perhaps surprising, in light of the level of change seen in those other measures, is the limited amount of change observed in M6; the internal link structure of the class itself is *moderately stable*. The main area of change is the degree to which the class is used.

Further examination reveals that although M1 and M8 are *moderately unstable* and M6 is *moderately stable*, the starting and ending values have not changed much when compared to the magnitude of their values. They have changed frequently, but have maintained a kind of equilibrium over the class's lifetime. However, M3 and M7 have changed considerably, with M3 and M7 increasing by a factor of six and five times respectively. Although this class has had internal modifications, it has seen a steady change in how it is used.

An examination of column M7 reveals several numbers that appear large when compared to the change matrices of other classes. In most cases, the numbers are increasing, but there are some negative numbers, including a large loss of inlinks in the last version of the system. For a class to lose 2551 links it must have had at least that many prior to the change, so large negative values like this one could only exist in the change matrix for a highly coupled class. However, for a class to receive a large number of new inlinks does not require that it had previously possessed a large number. The consistent presence of the large numbers in this change matrix compared to the relatively small number seen in others raises the question as to whether or not the current level of coupling has an effect on the number of new incoming links that a class receives. Is a heavily used class likely to garner more links than less heavily used ones, which is a form of preferential attachment? How does the coupling of a highly coupled class evolve?

Pair	M1	M2	M3	M4	M5	M6	M7	M8
	1251	26	370	0	0	1278	1267	401
1	0	0	0	0	0	0	1	0
2	0	0	–2	0	0	0	–6	0
3	0	0	5	0	0	0	8	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	–105	1	222	0	0	–73	662	21
7	72	0	55	0	0	67	128	11
8	25	–1	52	0	0	20	185	6
9	16	2	35	0	0	12	59	14
10	3	0	48	0	0	1	41	–1
11	0	0	1	0	0	0	33	0
12	0	0	0	0	0	0	0	0
13	0	0	22	0	0	0	68	0
14	42	0	173	0	0	44	397	30
15	0	0	121	0	0	0	184	12
16	0	0	66	0	0	0	163	0
17	0	0	6	0	0	0	19	6
18	146	1	76	0	0	203	394	34
19	2	0	43	0	0	6	361	5
20	–156	–1	21	0	0	–212	42	–35
21	14	2	20	0	0	10	100	–3
22	1	0	121	0	0	0	248	0
23	0	0	67	0	0	0	264	0
24	0	0	3	0	0	0	–4	0
25	50	0	0	0	0	56	5	6
26	0	0	2	0	0	0	23	–4
27	0	0	1	0	0	0	4	0
28	0	0	7	0	0	0	28	0
29	0	0	0	0	0	0	4	0
30	5	0	0	0	0	11	0	4
31	0	0	14	0	0	0	24	0
32	–3	0	0	0	0	–3	0	–1
33	0	0	2	0	0	0	3	0
34	–2	0	93	0	0	–8	166	–9
35	68	3	48	0	0	107	129	36
36	0	0	5	0	0	0	5	0
37	6	–1	211	0	0	5	563	1
38	0	0	256	0	0	0	460	8
39	9	0	35	0	0	11	138	16
40	–321	1	115	0	0	–347	218	-115
41	0	0	83	0	0	0	315	0
42	46	0	114	0	1	48	342	23
43	0	0	448	0	0	0	1578	0
44	6	0	–27	0	0	7	–176	15
45	–33	0	13	0	0	–38	42	–14
46	41	1	54	0	0	57	146	19
47	24	0	192	0	0	5	565	–44
48	13	0	–862	0	0	10	–2551	4
	1220	34	2329	0	1	1277	6645	446

Table 6.1: Moderately stable class with moderately unstable M1 and M8 and unstable M3 and M7

6.1.1 Hypothesis 6: Distribution of changes to M7

Because all the systems in the corpus possess approximate scale-free structure, it is likely that the distribution of increases to M7 are not uniform. Similarly, for the structure to remain scale-free over time, some classes must consistently receive higher increases while others consistently receive lower ones.

> **Hypothesis 6**: The distribution of aggregate between-module inlink increases is non-uniform and classes that have higher inlink counts are more likely to receive larger increases.

6.1.2 Testing Hypothesis 6

Hypothesis 6 is tested by extracting the non-zero values from change rows for M7 and the corresponding count of aggregate between-module inlinks from the full corpus. To determine whether the distribution is non-uniform, the frequency of change values are plotted. To determine whether there is a relationship between historical usage and future usage, the Pearson correlation coefficient (r) is computed between the count of aggregate between-module inlinks and the corresponding increase.

This method of evaluation is valid with respect to the non-uniformity of the data in the corpus, as discussed in Section 5.4.1. Although the amount of change between versions is not normalized, the change values for M7 are not being directly compared to each other. Rather, the first test demonstrates that across the corpus, the distribution of change values is non-uniform. The second test demonstrates that the distribution of high change values is preferential with respect to the current value of M7.

6.1.3 Results

Figure 6.1 shows the distribution of the change values for M7, which is clearly non-uniform. Most increases to inlink counts are small, and larger values occur decreasingly often, much like a power law distribution.

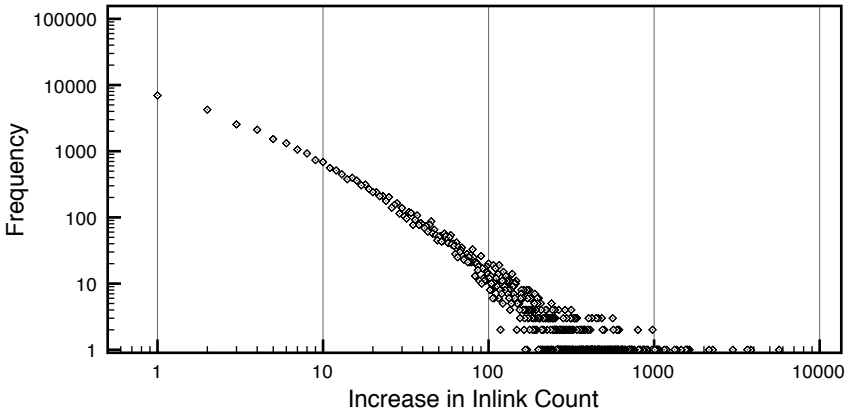

Figure 6.1: Frequency of changes in aggregate between-module inlink count.

Figure 6.2 plots the measured increase in M7 against its current values. This shows that most of the increases are less than 200 new inlinks, which can be seen as the large body of points at the bottom of the plot. For the current inlink count range of 10 to 100, the range of the corresponding increase values rises from around 75 new inlinks to 200. The range of increases stays at around 200 new inlinks until the current inlink count reaches 1000, at which point the increase values decrease. For the group of points on the lower portion of the plot, the distribution of increase values is not uniform across current inlink values. Classes that have more inlinks are more likely to receive a larger increase than those with a smaller number of inlinks.

The scattering of points across the plot for increases larger than 200 is more difficult to interpret. While current inlink counts between 1 and 100 do have some of these larger increases, there are more points for current inlink values larger than 100. Again, this suggests that classes with larger inlink counts do tend to receive larger increases.

The Pearson correlation coefficient for this plot is $r = 0.27$. This rather small value shows that a large M7 value does not necessarily mean that the class will always receive large increases. However, the correlation does show a tendency for classes with large M7 values to receive larger increases when increases of that magnitude occur.

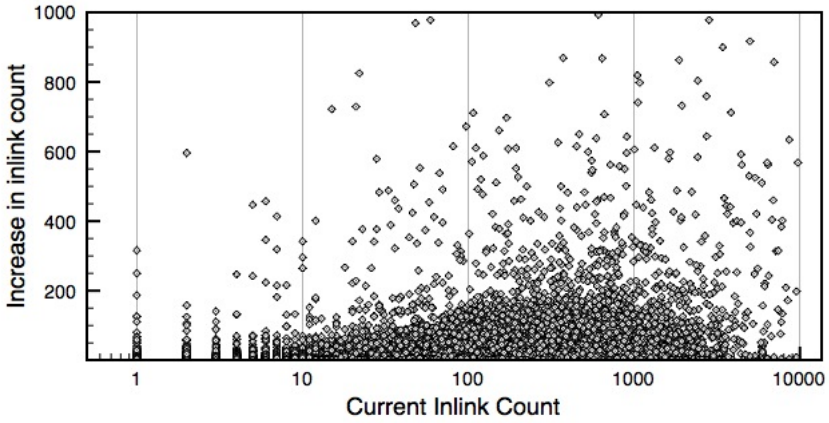

Figure 6.2: Increase in inlink count versus current inlink count.

6.1.4 Discussion

These findings mirror those presented in Chapter 5, but from a different perspective. In Chapter 5, the frequency of non-zero change values was investigated to show that most of the time, classes do not change. In this section, when change did occur, it was the magnitude that was considered. Figure 6.1 shows that most of increases are small and the frequency of occurrence decreases as change values become larger. While one could argue that the non-uniform distribution revealed by Figure 6.1 is due to the non-uniform degree of change between version pairs, Figure 6.2 shows that larger magnitudes of change are more likely to be observed for classes that already possess higher measures. The non-uniform increases are applied in a non-uniform way, thereby resulting in scale-free structure. This shows that when creating new links, programmers show a preference for some classes over others.

The BA model of network evolution has been described as “the rich get richer” (Barabási and Albert, 1999) and there is evidence of that phenomenon in these results. However, this model is not directly applicable to software evolution because it derives its causal factors from the intrinsic structure of the network, and in software systems the factors that cause change do not originate from within the network but rather from external sources. Predicting how a software system might change based

on its structure is only valid if future change pressures are not independent from its past. If there is a dependency between future and past change pressures, then how the structure changed in the past is likely to reflect how it is going to change in the future.

From this perspective, the scale-free structure of software systems may be the result of the dependence between change pressures over the lifetime of a software system. For example, a software system for word processing may be subjected to different change pressures over its lifetime, but those pressures will always be related because they are constrained by the problem that the system addresses. No matter what feature is added, provided that it relates to word processing, the resulting change pressures will likely result in changes that are non-uniformly distributed, and those patterns of non-uniformity are likely to be repeated.

6.2 High coupling and the ripple effect

From Chapter 2, we recognize that the ripple effect is a primary concern when considering the evolvability of a complex network. Too much interdependence between nodes increases the probability of change propagation to the point that small changes result in large-scale change propagation. Mens and Demeyer (2001) state “obviously, highly coupled parts of the software are very sensitive to changes because they typically consist of software entities that are strongly connected with one another.” In their definition, highly coupled nodes are *evolution-sensitive*, and “whenever something is changed in these [evolution-sensitive] parts, it may have a high impact on many other parts.”

Also from Chapter 2, we recognize that programmers have to make design decisions based on an anticipation of what they believe is likely to change. With information hiding, nodes that are likely to change are hidden within a module, and access is only granted through a defined interface. With DSM, the effects of between-module interaction are mitigated through the introduction of a design rule. In both of these cases, the node that is expected to change is hidden by one that is less likely to change. Nodes that are less likely to change make up module interfaces.

Consider the case where a programmer was able to successfully predict the change pressures for a system, to the extent that his design decisions minimized change propagation. Is it possible that such a design might still possess areas of high coupling? Inherent in the maxim of "high cohesion/low coupling" is that any high coupling reduces the quality of design and should be avoided Dhama (1995). Not considered, however, is the case where a design may be well suited to change pressures and still possess high coupling. In an investigation of the evolution of the Eclipse system, Hou (2007) claims that the architecture protects the system from ripple effects and design changes.

This section investigates whether evidence of ripple effects that propagate to a large proportion of classes can be found in the corpus. Since all the systems contain high coupling, it is expected that numerous instances of the ripple effect have occurred.

6.2.1 Criteria for the ripple effect in software

The literature about ripple effects in software focuses on simulating ripple effects (Yau and Collofello, 1985; Tsantalis et al., 2005; Sharafat and Tahvildari, 2007; Li et al., 2009), and there is no accepted way of identifying systems that have been subjected to system-wide changes because of change propagation. To determine whether high coupling contributes to ripple effects, we need the ability to detect them in software networks. Recall the SIR model of disease propagation (Section 2.6.1), where the network exhibits scale-free structure and nodes can be in one of three states: *susceptible*, *infective*, and *removed*. The rate of transmission through a scale-free network is not determined solely by the number of infective nodes, but also depends on the number of links between infective and susceptible nodes. This is the same basis upon which the maxim of high cohesion/low coupling is based: highly coupled classes have the ability to propagate change to a large number of other classes Watts (2002). Applying these principles from the SIR model to software change propagation suggests three criteria for ripple effects:

1. a large proportion of classes must exhibit change

2. the changed classes must be connected

3. the subnetwork of changed classes must contain highly coupled nodes.

While it is possible for change to propagate between a small number of classes, this investigation is concerned with ripple effects that propagate change to a large proportion of classes. However, because systems can vary in size, it is difficult to ascertain what constitutes a large proportion of classes. In this investigation this determination is made by comparing the number of changed classes to the number of unchanged.

In order for change to propagate based on coupling, the affected classes must be connected. This investigation does not consider change that may propagate for reasons other than through direct connection, such as co-change (Ball, Adam, Harvey and Siy, 1997b; Gall, Hajek and Jazayeri., 1998; Zimmermann et al., 2004).

Since our purpose is to study the role that coupling plays in the ripple effect, some of the classes that change must be highly coupled. If there is high propagation of change through a system and none of the classes involved are highly coupled, one cannot conclude that high coupling was a contributing factor.

6.2.2 Hypothesis 7: Identifying ripple effects in software

Unfortunately, it is not possible to conclusively identify the ripple effect in the corpus. If two connected classes change, there is insufficient information to determine whether the changes are the result of change propagation, or whether the classes have changed coincidentally. However, if one or more of the conditions outlined are not met, then we can conclude that ripple effects that match our criteria have not occurred.

To determine whether a high proportion of classes have changed, we check the ratio between the total number of changed classes and the total number of unchanged classes. Instances where this ratio exceeds one are considered significant for this investigation.

The ratio of changed to unchanged classes is not sufficient to determine if highly coupled classes are included in the set of changed classes.

To make this determination, classes must be weighted by their level of coupling. This can be accomplished by using the *degree centrality* measure, which computes the level of *influence* of a node based on its degree. While there are other measures for computing a node's influence within a network—examples include *betweenness centrality* (Newman, 2010) and *pagerank* (Page, Brin, Motwani and Winograd, 1998)—these other measures do not compute influence based on coupling, which is relevant here. The degree centrality of a node is computed by dividing its degree by the number of links in the network (Wasserman and Faust, 1994). Classes that are highly coupled have a greater ability to propagate change within the software system, so they will have a larger degree centrality.

To use the degree centrality measure, the system is split into two subnetworks, one network the changed classes and the other the unchanged classes. For each subnetwork, the disjoint sets of connected classes are identified, and for each set, its influence is computed by summing the degree centrality for each node. The results are considered significant if the sum of influence for changed sets exceeds those for unchanged.

This leads to:

> **Hypothesis 7**: In systems that have high coupling, we expect to find evidence of the ripple effect, which is characterized by a significant proportion of changed classes, whose influence exceeds that of unchanged classes.

Since all systems in the corpus possess high coupling, it is expected that they should exhibit characteristics of the ripple effect when subjected to change pressures.

6.2.3 Results

Figure 6.3 plots both measures used to test Hypothesis 7, sorted by descending order of the influence measure. The two are highly correlated, with a Pearson correlation coefficient $r = 0.95$. The percentage measure is consistently lower than the degree centrality measure except for a few

points and for cases where there are few changed classes, which are on the right side of the plot.

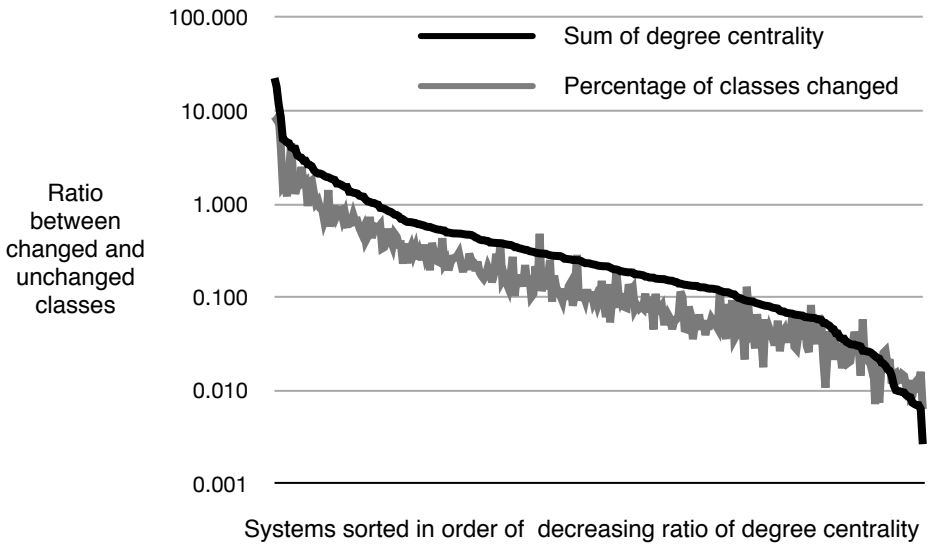

Figure 6.3: Ratio of changed and unchanged classes for both degree centrality and percentage change.

For the influence measure, there are 49 system pairs for which the ratio is considered significant, and for the percentage measure, there are 22 pairs. Out of the total of 334 pairs, this gives rates of 15% and 7% respectively where there is a possibility that the ripple effect has occurred. Even though the two measures are highly correlated, the degree centrality measure identified twice as many version pairs that indicate the possibility of system-wide change to existing classes.

For many of the version pairs, the graph of changed classes contains two or more disjoint sets of nodes. In all cases, there was a single set that had most of the nodes. Since the sum of degree centrality for the smaller sets was always a small fraction of that for the main set, those sets are not reported on Figure 6.3.

A post-hoc analysis was performed to identify which version pairs exhibit the potential for ripple effect. The pairs were grouped by system and are shown in Table 6.2. The system with the largest number of version pairings is *freecol*, which has 9 release pairings out of a total of 22 releases. *Ant* was second on the list with 7 release pairings out of a to-

System	High influence count
freecol	9
ant	7
lucene	6
hibernate	6
weka	5
argouml	4
jmeter	3
antlr	3
junit	2
jgraph	2
jung	1
azureus	1

Table 6.2: Number of version pairs where influence of changed classes exceeds that of unchanged classes.

tal of 19 releases, and *lucene* was third with 6 release pairings out of 20. These three systems show a high influence of changed classes in a large percentage of their releases when compared to the other systems, and this suggests the possibility that these systems may not have been well designed for the kinds of change pressures to which they have been subjected.

6.2.4 Discussion

The primary concern of high coupling is that it facilitates the ripple effect through software systems as they are being modified in response to change pressures (Mens and Demeyer, 2001). All systems in the Qualitas corpus have areas of high coupling, which suggests that ripple effects should be common, but the results show that evidence for the possibility of this effect was only observed in 15% of the version pairs, based on the criteria proposed. Neither measure indicated any significant effect in 85% of the cases. Because these measures do not confirm the presence of a ripple effect, the percentage of cases where it actually occurred may be smaller than 15%.

Overall, the results cannot confirm Hypothesis 7 in 85% of the version pairs, and this finding is supported by the observations made in Chapter 5, which showed that most classes in the corpus are either unchanged or remain stable between releases. This suggests that while high coupling

may be necessary for the ripple effect to occur, it is not sufficient, and that mechanisms such as abstraction may help programmers to mitigate the effects of change pressures. Thus it does not appear as though the presence of high coupling is necessarily indicative of poor design. Instead, the systems appear to be reasonably well designed with respect to the change pressures that they faced. It must be noted, however, that the post-hoc analysis does reveal that the systems *freecol*, *ant*, and *lucene* have the largest number of release pairings that meet the criteria for exhibiting the ripple effect. Further investigation of these systems is warranted.

These findings contradict those of some studies that attempt to assess the impact of change propagation through simulation. Li et al. (2009) proposed a method of evaluating the impact of change propagation using a graph model of software, and concluded that coupling is "a significant quality attribute" because their simulations demonstrated that the presence of high coupling led to the ripple effect. Their simulations were for a single system (*JEdit*), and it is not clear that they represented realistic change pressures or that their primary control parameter (α) reflects reasonable probability of propagation of change. Abdi et al. (2009) proposed a probabilistic approach based on Bayesian networks to predict the impact of change in object-oriented systems, and also concludes that coupling is a good indicator of change propagation. However, their simulation was tested on a single system (*BOAP*) and only attempts to determine whether links are going to be *weak*, *average* or *strong*, and does not go on to show that high coupling causes ripple effects.

Sharafat and Tahvildari (2007) developed an approach to compute the probability that a class will change in the future based on its source code structure and dependencies extracted from UML models. They evaluate it against a single system (*JFlex*), which they chose because it is small enough to be visualized easily. Unfortunately, because small systems have few classes, changes can appear to have a higher impact because changing even a small number of classes can represent a high percentage of the total classes. Tsantalis et al. (2005) developed an approach to detecting the *change proneness* of classes and tested it on both *JFlex* and

a system called *JMol*. In their results, they state that the change histories of *JFlex* and *JMol* have a ripple effect of 25% and 50% respectively. These high figures reflect their chosen measurement method, which is based on the work of Yau, Collofello and McGregor (1978). In this measure ripple effect computation is based on how coupling of variables can propagate changes to the rest of the program. It is not clear how they used this measure to measure an object-oriented system.

6.2.5 Limitations of this analysis

While the findings presented here show that high coupling has had less impact on the evolution of software systems than was expected there are some scenarios that cannot be identified through this analysis. We have only considered the ripple effect for classes that are connected on the class graph. However, the literature on co-change demonstrates that classes that are not connected can show a high correlation of change, thus demonstrating that a change relationship does not require direct connection in the source code (Hassan and Holt, 2004; Zimmermann et al., 2004). These kinds of relationships cannot be detected using the Qualitas corpus because they are based on a temporal analysis of a fine grained dataset—for example, every time a particular class is changed and committed to the system repository, it is accompanied by changes to another class.

While some cases of co-change cannot be detected using static analysis (Zimmermann et al., 2004), it is possible to determine whether co-change effects would have caused these results to be different. For the centrality measure, the influence was computed for each subgraph of connected nodes and compared against the maximum importance for the subgraphs of unchanged nodes. A post-hoc analysis that computes the aggregate influence of all changed nodes revealed that there was no change to the number of systems that may have exhibited ripple effects. If co-change had occurred, the effects were either too small, or the classes were already part of the largest subgraph of changed nodes.

One other case that is undetectable by this investigation is when change pressures are rejected because of their perceived impact on the system.

Programmers decide that a particular change request is too difficult to perform because of the level of dependency in the system, so the changes are never made. In this case, the level of coupling has had an impact on the evolvability of the system by preventing the changes from occurring at all.

Chapter 7

Conclusions and Future Work

> "Begin at the beginning,", the King said, very gravely, "and go on till you come to the end: then stop"
>
> —Alice's Adventures in Wonderland, Lewis Carroll

This thesis investigates the structure of software and how it might have an impact on system modifiability. From design theory, we recognize that the interconnectivity between the various pieces of a system can render it inflexible to change because of the ripple effect. Small changes can propagate in such a manner that they have system-wide effects, and this kind of structure is difficult to modify.

To minimize the probability of ripple effects, systems are modularized and their between-module interaction is regulated. However, network theory postulates that complex systems are scale-free, which means that some parts exhibit very high levels of connectivity. Network analysis has been applied to software systems, and findings have shown that many properties of software systems are scale-free. The goal of this research was to perform an empirical investigation to determine whether software systems are scale-free, and, if so, whether scale-free structure has a negative impact on their modifiability.

7.1 Research questions

The investigation addresses three research questions:

1. Is the structure of software scale-free?

2. Does scale-free structure result in high coupling?

3. Can the ripple effect be observed in software that has high coupling?

Each of these questions is considered in turn.

7.1.1 Is the structure of software scale-free?

While there have been several analyses that have identified scale-free structure in software (Wheeldon and Counsell, 2003; Myers, 2003; Marchesi et al., 2004; Potanin et al., 2005; Baxter et al., 2006; Hyland-Wood et al., 2006; Concas et al., 2007; Ichii et al., 2008; Louridas et al., 2008; Gao et al., 2010), many have been purely exploratory in that they do not attempt to ascertain why scale-free structure emerges (Baxter et al., 2006; Concas et al., 2007; Louridas et al., 2008). Those that have proposed reasons for the emergence of scale-free structure have used class and method level analyses (Wheeldon and Counsell, 2003; Myers, 2003; Marchesi et al., 2004; Potanin et al., 2005; Valverde and Sole, 2005; Hyland-Wood et al., 2006; Ichii et al., 2008; Gao et al., 2010). However, classes and methods are an aggregation of simpler nodes, and it is the interaction of those nodes that forms the basis for the patterns of interaction observed at the higher levels. The goal of the first research question was to determine whether source-code network at the level of statements and variables are also scale-free.

To address this question, CodeNet was used to create directed-graph representations of 97 open-source software systems. The degree distribution histogram for each was computed, and examined for evidence of scale-free structure (Hypothesis 1). The histogram for each system examined possessed evidence of scale-free structure.

Other researchers have observed different degree distributions for inlinks and outlinks (Myers, 2003; Louridas et al., 2008), so the inlink and outlink distributions were computed (Hypothesis 3). In each system, the former was found to exhibit scale-free structure, while the latter was found to be truncated by an upper bound. This demonstrates that the presence of scale-free structure at the level of statements and variables results from node inlinking. The number of outlinks from each node are

subjected to programming language and practical constraints—for example, each class can only have one superclass, and coding standards tend to limit the length of source-code lines, and therefore limits the number of outlinks that can be defined. However, no such limitations affect the frequency of node use; a node is used as many times as programmers require the utility that it provides.

To help ascertain why scale-free structure emerges in source-code, a model of software evolution was presented (Section 4.2.1). It is based on the well-known BA model of network evolution (Barabási and Albert, 1999), modified to be applicable to source code evolution. In the BA model, the probability of inlinking to a specific node is based on the number of links that the node already has: nodes with more links are more likely to attract new inlinks. In the proposed model, nodes are more likely to attract inlinks based on three criteria: the utility provided by the node, whether or not the node was know to programmers, and the trust that programmers had in the node. Nodes that provide more general functionality and that are known and trusted by programmers are more likely to receive inlinks. For scale-free structure to emerge, the probability distribution for inlinking must be non-uniform (Simon, 1955; Keller, 2005), and this is observed with Hypothesis 6.

Scale-free structure was also found at the class level (Hypothesis 4), which supports the findings of others (Baxter et al., 2006; Louridas et al., 2008; Gao et al., 2010). However, at the class level, both inlink and outlink distributions were found to be scale-free.

As noted in Section 2.5.1, the definition of a scale-free network requires that it's degree distribution is a power-law. However, other distributions exhibit similar connectivity properties (Simon, 1955; Keller, 2005; Clauset et al., 2009) that are of interest to this investigation, so the strict definition of scale-free structure is too restrictive here. Since other researchers have found power-laws in software software networks Potanin et al. (2005); Baxter et al. (2006); Concas et al. (2007); Louridas et al. (2008); Hatton (2009), the aim was not to replicate those findings, but rather to investigate the broader implications between high coupling and software modifiability. In light of this context, the term *scale-free*

is used loosely here to mean any network whose degree distribution approximates a power-law.

7.1.2 Does scale-free structure result in high coupling?

Having ascertained that the source-code network of software systems is scale-free, the next question is whether scale-free structure results in high coupling. Since coupling is defined as the strength of association between modules, it is not necessarily the case that the high connectivity found in scale-free structure results in high coupling. One of the criteria used for modularization is the grouping of nodes based on high levels of interaction (Alexander, 1964; Brito e Abreu and Goulao, 2001), so it is possible that areas of high connectivity may be resolved within the containing module, which is indicative of high cohesion rather than high coupling. To test whether this was the case, the degree distributions for each system were computed, but only links that crossed class boundaries were considered (Hypothesis 2). In all cases, the resulting degree distributions showed evidence of scale-free structure.

This finding is important because it shows that scale-free structure observed at the class level results directly from scale-free structure of the underlying source-code network. Nodes that contribute to high coupling between modules are the same nodes that exhibit high connectivity in the source-code network. In all cases where high connectivity was observed due to scale-free structure, high coupling was also observed.

This investigation also considered high coupling that resulted from the aggregation of nodes into classes (Hypothesis 4). Here too, degree distributions also showed evidence for scale-free structure, which means that some classes exhibited connectivity patterns that were much larger than the mean connectivity between classes. This is true for both inlink and outlink distributions.

7.1.3 Can the ripple effect be observed in software with high coupling?

Based on the design theory presented in Chapter 2, it is believed that high coupling can facilitate change propagation between modules in such a manner that small changes propagate system-wide. Software in this state has limited changeability because of the effort required to make simple changes. Since high coupling is observed in all of the systems, it is expected that successive version pairs will see internal change to a high percentage of connected classes. Due to the nature of the data, however, the presence of change propagation cannot be conclusively confirmed because two connected classes that both exhibit internal structural change may have simply changed coincidentally. However, if two classes are connected and one of them exhibits no internal structural changes, it can be confirmed that no change propagation occurred.

There is no accepted method of determining the severity of change propagation in software systems. Research that focuses on ripple effects in software is prospective and attempts to answer the question, "what is the impact if this change is made," rather than retrospectively considering the changes that were actually made (Black, 2001, 2006, 2008; Abdi et al., 2009). Other research is based on simulation of change propagation, which are also prospective (Yau and Collofello, 1985; Tsantalis et al., 2005; Sharafat and Tahvildari, 2007; Li et al., 2009).

Because no accepted measure is available, an assessment was made by considering two separate analyses. The first, described in Chapter 5, is an exploratory investigation that identified and categorized the patterns of change based on the frequency of changes to structural measures. The second, described in Chapter 6, identified classes that exhibited internal structural changes, and computed a ratio between the number of changed versus unchanged classes. The second analysis is based on the SIR model of disease propagation (Ball et al., 1997a), which includes properties that are expected in software systems that exhibit system-wide change propagation.

The exploratory analysis revealed that in response to change pressures, the majority of classes remained unchanged between software

versions. In 81.2% of the cases, all of the class measures remained unchanged, and in 88.8% to 99.9% of the cases, individual measures remained unchanged, depending on the measure. Classes were categorized as either *unchanging*, *stable*, *moderately stable*, *moderately unstable*, or *unstable*, based on the frequency of change observed in their structural measures. In 95% of the cases, classes were either *unchanging*, *stable*, or *moderately stable*, which suggests that most classes exhibited low frequency of change.

While these measures are coarse-grained, they do show that classes are more likely to remain unchanged between system versions than to show modification. This suggests that if ripple effects are occurring, they are not propagating far into the class network, because we would expect to see evidence of a larger number of classes exhibiting change.

This observation is confirmed by the second analysis (Hypothesis 7), which examined two ratios per system version pair. The first ratio is between the number of changed versus unchanged classes, and only exceeded 1 in fewer than 7% of the total cases.

The second ratio is based on network properties extracted from the SIR model of disease propagation. In this model, the impact of the disease on the population is not based on the number of infective persons, which are equivalent to changed nodes, but rather on the number of links between the infective and susceptible persons. Nodes with more links (i.e. more highly coupled) are seen to have greater *influence* in the network because they have a greater chance of propagating infection to susceptible portions of the network. This is similar to the propagation of change within a software system. Classes that have low connectivity have little ability to affect the network, but highly coupled classes can have a large impact. To determine whether the classes that were modified include those that have greater ability to propagate change, the influence of each class was measured, and the total influence of connected subgraphs of changed and unchanged classes was compared. For all systems, the ratio of collective influence of changed classes exceeds that for unchanged classes in only 15% of the cases. There is a high degree of correlation between this measure and the previous measure based on percentages.

While these measures do not conclusively prove that ripple effects did not occur, they show that for each system version pair, changes appear to be confined to a subset of classes. In the majority of version pairs, it would be difficult to conclude that change propagation involving a large number of classes has occurred. Based on the observations made here, this investigation cannot confirm system-wide change propagation for systems that are confirmed to possess areas of high coupling.

While these measures have limitations, it is important to recognize that they represent a systematic and transparent method of attempting to ascertain the presence and impact of change propagation due to high coupling, which can be applied to a large corpus of software systems. Ripple effects in software systems have been postulated for over three decades and design principles have been devised to address the their effects, yet no method of conclusively demonstrating the phenomenon in existing software systems currently exists. I believe that this is because the phenomenon is complex. The measures introduced here represent an initial step in addressing the problem, and further research is clearly warranted.

7.2 Closing remarks

In light of the literature presented and the investigations conducted, the question arises, “how is it possible for high coupling to be consistent with good design?” The design of complex systems requires that designers balance the tradeoffs between different design decisions. When a module is constructed, two aspects must be considered. The first is the part of the module that remains hidden. This is the part of the system that the designer believes has a high probability of changing based on anticipation of future change pressures. The second is the part of the module that is exposed. This is the part that the designer believes will not need to change in light of anticipated change pressures.

If the designer has been successful in anticipating future change pressures, the system will remain flexible because those pieces that need to change remain hidden and those that are exposed do not need to change.

In this case, if the exposed portions are highly coupled, the impact of that coupling will be minimal because it is precisely those parts of the system that are expected to change the least. Because the analyses presented here could not conclusively demonstrate some theorized effects of high coupling, the door is opened to the possibility that the systems are well designed from the perspective of their modifiability, in spite of their high coupling.

7.3 Contributions

This thesis makes four contributions. First, it demonstrates, using a large corpus of open source software systems, that the source-code network is scale-free. This shows that when programmers build software from statements, and class, method and variable declarations, the majority of nodes have limited connectivity, but some have very high connectivity. This is important because design theory and design principles stress the avoidance of high connectivity. In a scale-free network, most nodes have low connectivity, and that accords with design principles. However, scale-free structure shows that areas of high connectivity cannot be eliminated completely. Because scale-free structure is observed in all the systems, this suggests that it is a common property of software.

Second, this work demonstrates that the high connectivity of source-code nodes results in high coupling. The pressure to keep modules small limits the ability for highly connected nodes to resolve their links within their containing module, which is the reason why high connectivity results in high coupling. Since this characteristic was observed in all examined systems, and it is also observed in the literature (Gao et al., 2010), it is likely to be a common property of software systems. This finding is important because modularization is seen as the prime method of avoiding high levels of change propagation, and high coupling between modules can limit its effectiveness.

Third, this investigation is unable to confirm some theorized effects of high coupling. While this suggests that high coupling may not necessarily be a conclusive indicator of poor design quality, this finding is largely

due to the lack of methods available to demonstrate the phenomenon. Because design principles are devised, in part, to address issues of high change propagation, this highlights a significant gap in the field. This investigation introduces an initial measure that draws from network analysis, which is used by other fields that also track propagation through complex networks.

Finally, this investigation provides some evidence that programmers generate links to nodes preferentially, and this is observed by the evolution of class usage. Increases in class usage are non-uniformly distributed, and are more likely to be applied to nodes that already exhibit higher levels of usage. While this work conjectures that this preferential attachment is based on three criteria, those criteria are not specifically tested, so no conclusions are drawn about what causes programmers to link preferentially.

7.4 Future work

It has been said that good research generates more questions than it does answers. Two avenues of future research are presented in this section.

7.4.1 Ripple effect detection and impact analysis

It has been stressed in this investigation that even though there is considerable effort expended to address the impact of large change propagation, no method of detection exist. As the field expands its understanding of the structure of software, and as analysis techniques become more sophisticated, it is clear that in order to determine if our methods of dealing with the problem are effective, we need means to identify and characterize the phenomenon. If we cannot demonstrate that our methods cause favorable changes, then the effectiveness of those methods remain largely theoretical.

This research provide two bases for further research. First, there are other fields from which ideas about propagation through complex networks can be drawn. This investigation considered the SIR model, which models the propagation of infectious disease through a population.

Ripple effects in software have similar properties—for example, change propagates from node to node through interactions. This research shows how properties of that model can be tailored to the problem of change propagation in software, and applied to a large corpus of software systems. While the model presented here is immature, I hope that it can be used as a basis for further development and refinement, so that ripple effects can be detected and their impact quantitatively assessed.

Second, this research provides a network model of software that is not limited to classes and methods. Many investigations in object-oriented systems focus on class-level analysis. This has the advantages of dealing with fewer entities, but is a disadvantage if important properties of change propagation are based in internal class structures. In these cases, those properties will be masked by analyses at the class level. The model presented here provides a more complete representation of software than class-based or method-based model, thereby making it possible to identify aspects of change propagation due to internal class structure.

7.4.2 Research support and reproducibility

The availability of the Qualitas corpus has had a significant impact on this research. First, it eliminated the time that would be required to collect a large corpus of systems. Second, it provides a stable series of software systems upon which the analyses can be based. Since this corpus is accessible to all researchers, its use makes the research more reproducible. Indeed, since its formal release in 2010, several researchers have begun to incorporate the corpus in their own research (Beckman, Kim and Aldrich, 2011; Chow and Tempero, 2011; Ducasse, Oriol and Bergel, 2011; Taube-Schock, Walker and Witten, 2011; Vasilescu, Serebrenik and van den Brand, 2011; Zaparanuks and Hauswirth, 2011b,a).

However, the Qualitas corpus does not solve all the problems that face a researcher who is performing empirical studies on software systems. It is large (currently 42GB), so obtaining it through a network connection can take a considerable amount of time. Similarly, once it has been obtained, considerable effort must be expended in processing it to get

it into a form that is suitable for analysis. The experience of building the tools for this research provide evidence enough of that, and this perspective is also shared in the literature (Murphy and Notkin, 1996). As more effort is put towards processing the corpus prior to analysis, the reproducibility of the work decreases, because researchers would have to commit more of their own resources to obtain the same results. As more resources are required, there is less likelihood that researchers will have the ability to make this effort.

A solution to this problem is to make Qualitas available online through a web service that provides data from the corpus in different forms. For those researchers who wished to perform lexical analyses, the web service can serve the source code as it appears in the corpus. For researchers interested in performing a semantic analysis, the web service could return AST or semgraph forms of the requested information. A centralized service of this kind could save many researchers considerable time through the elimination of complex and redundant work.

Another benefit of such a service is that of searchability. To perform searches within the corpus, researchers are limited to the lexically-based indices, unless they go through the costly process of building indices based on their search requirements. With the use of semgraphs, indices based on structure can be built, which would allow researchers the ability to not only search on lexical properties of source code, but on structural properties as well. This could significantly reduce the time it takes for researchers to find examples of source code that satisfy a specific structural property.

References

Abdi, M., Lounis, H., Sahraoui, H. (2009). Predicting change impact in object-oriented applications with bayesian networks. In *The 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09)*, pp. 234–239.

Brito e Abreu, F., Goulao, M. (2001). Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Fifth European Conference on Software Maintenance and Reengineering*, pp. 47–57.

Abreu, F., Goulão, M., Esteves, R. (1995). Toward the design quality evaluation of object-oriented software systems. In *Proceedings Fifth International Conference on Software Quality, Austin, Texas*.

Aho, A. V., Johnson, S. C., Ullman, J. D. (1976). Code generation for expressions with common subexpressions. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pp. 19–31. New York, NY, USA: ACM.

Albert, R., Jeong, H., Barabási, A.-L. (1999). Diameter of the World Wide Web. *Nature, 401*, 130–131.

Alexander, C. (1964). *Notes on the Synthesis of Form*. Harvard University Press.

Allen, E. B., Khoshgoftaar, T. M., Chen, Y. (2001). Measuring coupling and cohesion of software modules: an information-theory approach. In *Proceedings of 7th International Software Metrics Symposium*, pp. 124–134.

Ashby, W. (1952). *Design for a Brain*. John Wiley and Sons.

Bailey, N. T. J. (1975). *The mathematical theory of infectious diseases and its applications 2nd edition*, vol. 413. Griffin.

Baker, B. S. (1995). On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pp. 86–95.

Baldwin, C. Y., Clark, K. B. (2000). *Design Rules: the power of modularity*. The MIT Press.

Ball, F., Mollison, D., Scalia-Tomba, G. (1997a). Epidemics with two levels of mixing. *Annals of Applied Probability*, *7*, 46–89.

Ball, T., Adam, J.-M. K., Harvey, A. P., Siy, P. (1997b). If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*.

Bansiya, J., Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17.

Barabási, A.-L., Albert, R. (1999). Emergence of scaling in random networks. *Science*, *286*, 509–512.

Basili, V., Briand, L., Melo, W. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, *22*(10), 751 –761.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E. (2006). Understanding the shape of java software. *SIGPLAN Notes*, *41*, 397–412.

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pp. 368 –377.

Beckman, N., Kim, D., Aldrich, J. (2011). An empirical study of object protocols in the wild. In Mezini, M. (Ed.), *ECOOP 2011 – Object-Oriented*

Programming, vol. 6813 of *Lecture Notes in Computer Science*, pp. 2–26. Springer Berlin / Heidelberg.

Bieman, J., Kang, B.-K. (1995). Cohesion and reuse in an object-oriented system. In *Proceedings of ACM Symposium on Software Reusability (SSR'95)*, pp. 259–262.

Black, S. (2001). Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, *13*, 263–279.

Black, S. (2006). Is ripple effect intuitive? a pilot study. *Innovations in Systems and Software Engineering*, *2*, 88–98.

Black, S. (2008). Deriving an approximation algorithm for automatic computation of ripple effect measures. *Information and Software Technology*, *50*(7–8), 723–736.

Booch, G., Rumbaugh, J., Jacobson, I. (2005). *Unified Modeling Language User Guide, The (2nd Edition)*. Addison-Wesley Professional.

Bosak, J., Bray, T., Connolly, D., Maler, E., Nicol, G., Sperberg-McQueen, C. M., Wood, L., Clark, J. (1998). W3c xml specification dtd. http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm

Briand, L., Arisholm, E., Counsell, S., Houdek, F., Thévenod–Fosse, P. (1999a). Empirical studies of object-oriented artifacts, methods, and processes: State of the art and future directions. *Empirical Software Engineering*, *4*, 387–404.

Briand, L., Daly, J., Wüst, J. (1999b). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, *25*(1), 91–121.

Briand, L., Devanbu, P., Melo, W. (1997). An investigation into coupling measures for c++. In *Proceedings 19th International Conference on Software Engineering, ICSE97, Boston*, pp. 412–421.

Briand, L., Morasca, S., Basili, V. (1999c). Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, pp. 722–743.

Briand, L., Wüst, J., Daly, J., Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *The Journal of Systems and Software, 51*, 245–273.

Briand, L. C., Daly, J. W., Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering, 3*, 65–117.

Bunge, M. (1977). *Treatise on Basic Philosophy, Volume 3: Ontology I, The furniture of the world*. D. Reidel Publishing Company, Dordrecht-Holland.

Burn, O. (2011). Checkstyle 5.5. http://sourceforge.net/projects/checkstyle/

Chatzigeorgiou, A., Tsantalis, N., Stephanides, G. (2006). Application of graph theory to OO software engineering. In *Proceedings of the 2006 International Workshop on Interdisciplinary Software Engineering Research*, WISER '06, pp. 29–36. ACM.

Chen, T., Gu, Q., Wang, S., Chen, X., Chen, D. (2008). Module-based large-scale software evolution based on complex networks. In *8th IEEE International Conference on Computer and Information Technology, CIT 2008*, pp. 798 –803.

Chidamber, S., Kemerer, C. (1991). Towards a metrics suite for object-oriented design. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA91*, pp. 197–211.

Chidamber, S., Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Chidamber, S. R., Darcy, D. P., Kemerer, C. F. (1998). Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering*, *24*(8), 629–639.

Chow, J., Tempero, E. (2011). Stability of java interfaces: a preliminary investigation. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, pp. 38–44. New York, NY, USA: ACM.

Church, A. (1943). Carnap's introduction to semantics. *The Philosophical Review*, *52*(3), 298–304.

Clauset, A., Shalizi, C. R., Newman., M. E. J. (2009). Power-law distributions in empirical data. *SIAM Review*, *51*(4), 661–703.

Concas, G., Marchesi, M., Pinna, S., Serra, N. (2007). Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, *33*(10), 687 –708.

Cubranic, D., Murphy, G. (2003). Hipikat: recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 408 – 418. doi: 10.1109/ICSE.2003.1201219.

Demeyer, S., Ducasse, S., Nierstrasz, O. (2000). Finding refactorings via change metrics. *SIGPLAN Notices*, *35*, 166–177.

Dhama, H. (1995). Quantitative models of cohesion and coupling in software. *The Journal of Systems and Software*, *29*(1), 65–74.

Dixon-Peugh, D. (2011). PMD: Java source code analysis. http://pmd.sourceforge.net/

Ducasse, S., Lanza, M., Tichelaar, S. (2000). Moose: An extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*.

Ducasse, S., Oriol, M., Bergel, A. (2011). Challenges to support automated random testing for dynamically typed languages. In *International Workshop on Smalltalk Technologies*. Edinburgh, Royaume-Uni.

Eclipse Foundation (2011). The Eclipse Foundation community website. http://www.eclipse.org

Eder, J., Kappel, G., Schrefl, M. (1992). Coupling and cohesion in object-oriented systems. *Conference on Information and Knowledge Management*.

Eder, J., Kappel, G., Schrefl, M. (1994). *Coupling and Cohesion in Object-Oriented Systems*. Technical Report, Univ. of Klagenfurt.

El-Emam, K., Benlarbi, S., Goel, N., Rai, S. (1999). *A Validation of Object-Oriented Metrics*. National Research Council of Canada, NRC/ERB 1063.

Erdős, P., Rényi, A. (1959). On random graphs. *Publicationes Mathematicae, 6*, 290–297.

Erdős, P., Rényi, A. (1960). On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences, 5*, 17–61.

Etzkorn, L., Delugach, H. (2000). Towards a semantic metrics suite for object-oriented design. In *Proceedings of 34th International Conference on Technology of Object-Oriented Languages and Systems*, pp. 71–80.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Gall, H., Hajek, K., Jazayeri., M. (1998). Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM'98)*, pp. 190–198.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Gao, Y., Xu, G., Yang, Y., Niu, X., Guo, S. (2010). Empirical analysis of software coupling networks in object-oriented software systems. In *Proceedings IEEE International Conference on Software Engineering. Service Sci.*, pp. 178–181.

German, D. M. (2006). An empirical study of fine-grained software modifications. *Empirical Software Engineering*, *11*, 369–393.

Gesser, J. V. (2008). javaparser. http://code.google.com/p/javaparser/

Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R., Daniel, R. (2007). Using concept analysis to detect co-change patterns. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, pp. 83–89. New York, NY, USA: ACM.

Gîrba, T., Ducasse, S., Marinescu, R., Raţiu, D. (2004). Identifying entities that change together. In *the Ninth IEEE Workshop on Empirical Studies of Software Maintenance ICSM2004*.

Gladwell, M. (2002). *The Tipping Point: How Little Things Can Make a Big Difference*. Back Bay Books.

Godfrey, M., Zou, L. (2005). Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, *31*(2), 166–181.

Gosling, J., Joy, B., Steele, G., Bracha, G. (2005). *The Java Language Specification, third ed.* Addison-Wesley.

Gyimothy, T., Ferenc, R., Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, *31*(10), 897–910.

Haney, F. M. (1972). Module connection analysis: a tool for scheduling software debugging activities. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, AFIPS '72 (Fall, part I), pp. 173–179. New York, NY, USA: ACM.

Harrison, R., Counsell, S., Nithi, R. (1998). An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, *24*(6), 491–496.

Hassan, A., Holt, R. (2004). Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp. 284–293.

Hatton, L. (2009). Power-law distributions of component size in general software systems. *IEEE Transactions on Software Engineering*, *35*(4), 566–572.

Henderson-Sellers, B. (1996). *Software Metrics*. Prentice Hall, U. K.

Hitz, M., Montazeri, B. (1995). Measuring product attributes of object-oriented systems. In Schäfer, W., Botella, P. (Eds.), *Software Engineering — ESEC '95*, vol. 989 of *Lecture Notes in Computer Science*, pp. 124–136. Springer Berlin / Heidelberg.

Holmes, R., Walker, R. J., Murphy, G. C. (2005). Strathcona example recommendation tool. *SIGSOFT Software Engineering Notes*, *30*, 237–240.

Hou, D. (2007). Studying the evolution of the eclipse java editor. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pp. 65–69. New York, NY, USA: ACM.

Hovemeyer, D., Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Notices*, *39*, 92–106.

Hyland-Wood, D., Carrington, D., Kaplan, S. (2006). *Scale-free nature of Java software package, class and method collaboration graphs*. Technical Report TR-MS1286, University of Maryland, College Park.

Ichii, M., Matsushita, M., Inoue, K. (2008). An exploration of power-law in use-relation of java software systems. In *19th Australian Conference on Software Engineering, 2008, ASWEC 2008*, pp. 422–431.

Jenkins, S., Kirk, S. (2007). Sotware architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences*, *177*(12), 2587–2601.

Jing, L., Keqing, H., Yutao, M., Rong, P. (2006). Scale free in software metrics. In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, vol. 1, pp. 229–235.

Kamiya, T., Kusumoto, S., Inoue, K. (2002). Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, *28*(7), 654–670.

Keller, E. F. (2005). Revisiting "scale-free" networks. *BioEssays*, *27*(10), 1060–1068.

Kelly, D. (2006). A study of design characteristics in evolving software using stability as a criterion. *IEEE Transactions on Software Engineering*, *32*(5), 315–329.

Kim, M., Sazawal, V., Notkin, D., Murphy, G. (2005). An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, *30*, 187–196.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, *28*(8), 721–734.

Koenig, A. (1995). Patterns and antipatterns. *Journal of Object-Oriented Programming*, *8*(1), 46–48.

Komondoor, R., Horwitz, S. (2001). Using slicing to identify duplication in source code. In Cousot, P. (Ed.), *Static Analysis*, vol. 2126 of *Lecture Notes in Computer Science*, pp. 40–56. Springer Berlin / Heidelberg.

Kontogiannis, K. (1997). Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pp. 44–54.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, *50*(4), 37–42.

Lee, Y.-S., Liang, B.-S., Wu, S.-F., , Wang, F.-J. (1995). Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of the International Conference on Software Quality, Maribor, Slovenia*.

Lehman, M. (1996). Laws of software evolution revisited. In Montangero, C. (Ed.), *Software Process Technology*, vol. 1149 of *Lecture Notes in Computer Science*, pp. 108–124. Springer Berlin / Heidelberg.

Lehman, M., Ramil, J., Wernick, P., Perry, D., Turski, W. (1997). Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pp. 20–32.

Li, D., Han, Y., Hu, J. (2008). Complex network thinking in software engineering. In *International Conference on Computer Science and Software Engineering*, vol. 1, pp. 264–268.

Li, L., Qian, G., Zhang, L. (2009). Evaluation of software change propagation using simulation. In *The WRI World Congress on Software Engineering (WCSE '09)*, pp. 28–33.

Li, W., Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, *23*(2), 111–122.

Liskov, B. (1987). Keynote address – data abstraction and hierarchy. *SIGPLAN Notices*, *23*, 17–34.

Liu, J., Lü, J., He, K., Li, B., Tse, C. K. (2008). Characterizing the structure quality of general complex software networks. *International Journal of Bifurcation and Chaos*, *18*(2), 605–613.

Louridas, P., Spinellis, D., Vlachos, V. (2008). Power laws in software. *ACM Transactions on Software Engineering Methodology*, *18*, 2:1–2:26.

Marchesi, M., Pinna, S., Serra, N., Tuveri, S. (2004). Power laws in Smalltalk. In *Proceedings of European Smalltalk User Group Joint Event*.

Marcus, A., Poshyvanyk, D. (2005). The conceptual cohesion of classes. In *ICSM'05. Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 133–142.

Martin, R. (1994). OO design quality metrics – an analysis of dependencies. In *Proceedings of the Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA94*.

Mens, T., Demeyer, S. (2001). Future trends in software evolution metrics. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pp. 83–86. New York, NY, USA: ACM.

Mens, T., Lanza, M. (2002). A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, *72*(2), 57–68.

Meyer, B. (2000). *Object-Oriented Software Construction SECOND EDITION*. Prentice Hall.

Meyers, T. M., Binkley, D. (2004). Slice-based cohesion metrics and software intervention. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE'04)*, pp. 256–265.

Milgram, S. (1967). The small world problem. *Psychology Today*, *2*, 60–67.

Monasson, R. (1999). Diffusion, localization and dispersion relations on "small-world" lattices. *The European Physical Journal B - Condensed Matter and Complex Systems*, *12*, 555–567.

Murphy, G. C., Notkin, D. (1996). Lightweight lexical source model extraction. *ACM Transactions on Software Engineering Methodology*, *5*, 262–292.

Myers, C. R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E, 68*, 046116.

Newman, M., Barabási, A.-L., Watts, D. J. (2006). *The Structure and Dynamics of Networks*. Princeton University Press.

Newman, M. E. J. (2005). Power laws, pareto distributions and zipf's law. *Contemporary Physics, 46*, 323–351.

Newman, M. J. E. (2010). *Networks: An Introduction*. Oxford University Press.

Newman, M. J. E., Strogatz, S. H., Watts, D. J. (2001). Random graphs with arbitrary degree distributions and their applications. *Physical Review E, 64*(2), 1–17.

Page, L., Brin, S., Motwani, R., Winograd, T. (1998). *The pagerank citation ranking: Bringing order to the web.* Technical Report, Stanford University.

Pan, W., Li, B., Ma, Y., Liu, J. (2011). Multi-granularity evolution analysis of software using complex network theory. *Journal of Systems Science and Complexity, 24*, 1068–1082.

Pandit, S. A., Amritkar, R. E. (2001). Random spread on the family of small-world networks. *Physical Review E, 63*, 041104.

Parasoft (2011). Java static analysis, code review, unit testing, runtime error detection.
www.parasoft.com/jsp/products/jtest.jsp

Pareto, V. (1897). *Cours D'économique Politique*. Macmillan.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*, 1053–1058.

Parnas, D. L. (1994). Software aging. In *Proceedings of the International Conference on Software Engineering*, pp. 279–287.

Parnas, D. L., Siewiorek, D. P. (1975). Transparency in the design of hierarchically structured systems. *Communications of the ACM*, *18*(7), 401–408.

Potanin, A., Noble, J., Frean, M., Biddle, R. (2005). Scale-free geometry in OO programs. *Communications of the ACM*, *48*, 99–103.

Selby, R. W., Basili, V. R. (1991). Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, *17*(2), 141–152.

Semmle Limited (2011). On demand software analytics. http://semmle.com

Sharafat, A. R., Tahvildari, L. (2007). A probabilistic approach to predict changes in object-oriented software systems. In *The 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*.

Simon, H. (1953). Causal ordering and identifiability. In *Studies in Econometric Method (edited by W.C. Hood and T.C. Koopmans)*. Cowles Commission Monograph 14, New York.

Simon, H. A. (1955). On a class of skew distribution functions. *Biometrika*, *42*, 425–440.

Simon, H. A. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society*, *106*(6), 467–482.

Simon, H. A. (1996). *The Sciences of the Artificial*. The MIT Press.

Software-Tomography GmbH (2011). Sonargraph architect. http://www.hello2morrow.com/products/sonargraph

Solomonoff, R., Rapoport, A. (1951). Connectivity of random nets. *Bulletin of Mathematical Biology*, *13*, 107–117.

Stevens, W. P., Myers, G. J., Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, *13*(2), 115 –139.

Subramanyam, R., Krishnan, M. (2003). Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, *29*(4), 297–310.

Succi, G., Pedrycz, W., Djokic, S., Zuliani, P., Russo, B. (2005). An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering*, *10*, 81–104.

Sullivan, K. J., Griswold, W. G., Cai, Y., Hallen, B. (2001). The structure and value of modularity in software design. *SIGSOFT Software Engineering Notes*, *26*, 99–108.

Taube-Schock, C., Walker, R., Witten, I. (2011). Can we avoid high coupling? In Mezini, M. (Ed.), *ECOOP 2011 – Object-Oriented Programming*, vol. 6813 of *Lecture Notes in Computer Science*, pp. 204–228. Springer Berlin / Heidelberg.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J. (2010). The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pp. 336–345.

Travers, J., Milgram, S. (1969). An experimental study of the small worls problem. *Sociometry*, *32*, 425–443.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. (2005). Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, *31*(7), 601–614.

Tu, Q., Godfrey, M. (2002). An integrated approach for studying architectural evolution. In *Proceedings of the 10th International Workshop on Program Comprehension*, pp. 127–136.

Tversky, A., Kahneman, D. (1974). Judgment under uncertainty: Heuristics and biases. *Science*, *185*(4157), 1124–1131.

University of Waikato (2011). Symphony: High performance computing cluster.
http://symphony.waikato.ac.nz/

Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P. (1999). Soot—a Java optimization framework. In *Proceedings of CASCON 1999*, pp. 125–135.

Valverde, S., Cancho, R. F., Sole, R. V. (2002). Scale-free networks from optimal design. *EPL (Europhysics Letters), 60*(4), 512.

Valverde, S., Sole, R. V. (2003). Hierarchical small-worlds in software architecture. *ArXiv preprint cond-mat/0307278 – arxiv.org*.

Valverde, S., Sole, R. V. (2005). Logarithmic growth dynamics in software networks. *EPL (Europhysics Letters), 72*(5), 858.
http://stacks.iop.org/0295-5075/72/i=5/a=858

Vasa, R., Lumpe, M., Branch, P., Nierstrasz, O. (2009). Comparative analysis of evolving software systems using the gini coefficient. In *IEEE International Conference on Software Maintenance, ICSM 2009*, pp. 179–188.

Vasilescu, B., Serebrenik, A., van den Brand, M. (2011). You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 313–322.

Verelst, J. (2005). The influence of the level of abstraction on the evolvability of conceptual models of information systems. *Empirical Software Engineering, 10*, 467–494.

Wasserman, S., Faust, K. (1994). *Social Network Analysis*. Cambridge University Press.

Watts, D. (2002). A simple model of global cascades on random networks. *Proceedings of the National Academy of Sciences, 99*(9), 5766–5771.

Watts, D. J., Strogatz, S. H. (1998). Collective dynamics of “small-world” networks. *Nature*, *393*(6684), 409–410.

Wheeldon, R., Counsell, S. (2003). Power law distributions in class relationships. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 45–54.

Witten, I. H., Frank, E., Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*. Morgan Kaufmann.

Yau, S., Collofello, J. (1985). Design stability measures for software maintenance. *IEEE Transactions on Software Engineering*, *SE-11*(9), 849–856.

Yau, S., Collofello, J., McGregor, T. (1978). Ripple effect analysis of software maintenance. In *Proceedings of the Computer Software and Applications Conference (COMPSAC ’78)*, pp. 60–65.

Ying, A. T., Murphy, G. C., Ng, R., Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, *30*(9), 574–586.

Zaparanuks, D., Hauswirth, M. (2011a). The beauty and the beast: Separating design from algorithm. In Mezini, M. (Ed.), *ECOOP 2011 – Object-Oriented Programming*, vol. 6813 of *Lecture Notes in Computer Science*, pp. 27–51. Springer Berlin / Heidelberg.

Zaparanuks, D., Hauswirth, M. (2011b). Vision paper: The essence of structural models. In Whittle, J., Clark, T., Kühne, T. (Eds.), *Model Driven Engineering Languages and Systems*, vol. 6981 of *Lecture Notes in Computer Science*, pp. 470–479. Springer Berlin / Heidelberg.

Zhou, Y., Lu, J., Xu, H. (2004). A comparative study of graph theory-based class cohesion measures. In *Proceedings of the 18th IEEE International Conference on Software Maintenance*, pp. 44–53.

Zhou, Y., Xu, B., Zhao, J., Yang, H. (2002). ICBMC: an improved cohesion measure for classes. In *Proceedings of International Conference on Software Maintenance*, pp. 44–53.

Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A. (2004). Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pp. 563–572.

Zwillinger, D. (Ed.) (2003). *CRC Standard Mathematical Tables and Formulae, 31st Edition*. CRC Press.

Appendix A

Example parse file

The following text is the parsefile for the Java listing on Figure 3.2 given in Chapter 3.

```
project Demonstration
module org.afox.codenet
entity class [TypeDeclaration-Resolved]
attribute pathName Entity
attribute fullName org.afox.codenet.Entity
attribute lineNumber 5
attribute superclass java.lang.Object
attribute modifier public
entity variable [VariableDeclarationFragment]
attribute type java.lang.String
attribute pathName name
attribute lineNumber 6
attribute modifier private
end variable
entity variable [VariableDeclarationFragment]
attribute type java.util.HashMap
attribute pathName attributes
attribute lineNumber 7
attribute modifier private
end variable
entity methodDeclaration
attribute pathName Entity
attribute parameterCount 1
attribute lineNumber 9
attribute declaredSignature Entity(java.lang.String)
attribute constructor true
attribute modifier public
```

```
entity variable [SingleVariableDec]
attribute lineNumber 9
attribute type java.lang.String
attribute pathName name
end variable [SingleVariableDec]
entity block
attribute lineNumber 9
entity statement [ExpressionStatement]
attribute lineNumber 10
entity expression [Assignment]
attribute lineNumber 10
attribute operator =
attribute stateChange true
entity expression [LHS]
entity expression [FieldAccess-Resolved]
attribute fieldTarget org.afox.codenet.Entity
attribute stateChange true
attribute this org.afox.codenet.Entity
entity expression [SimpleName]
attribute variableUsage name
attribute stateChange true
end expression [SimpleName]
end expression [FieldAccess]
end expression [LHS]
entity expression [RHS]
entity expression [SimpleName]
attribute variableUsage name
end expression [SimpleName]
end expression [RHS]
end expression [Assignment]
end statement [ExpressionStatement]
entity statement [ExpressionStatement]
attribute lineNumber 11
entity expression [Assignment]
attribute lineNumber 11
attribute operator =
attribute stateChange true
entity expression [LHS]
entity expression [SimpleName]
attribute variableUsage attributes
attribute stateChange true
```

```
end expression [SimpleName]
end expression [LHS]
entity expression [RHS]
entity expression [ClassInstanceCreation]
attribute targetType java.util.HashMap
attribute signature HashMap()
end expression [ClassInstanceCreation]
end expression [RHS]
end expression [Assignment]
end statement [ExpressionStatement]
end block
end methodDeclaration
end class [TypeDeclaration]
```

Appendix B

System structural measures

The following data are the structural measures of the systems in the Qualitas corpus that were examined. Column titles are as follows: Nod=nodes;Lnk=links; Pkg=Packages; Cls=classes; Mth=methods; Blk=blocks; Sta=statements; Var=variables.

Name/Version	Nod	Lnk	Pkg	Cls	Mth	Blk	Sta	Var
derby-10.1.1.0	318831	809952	135	1805	25067	56357	160555	74910
gt2-2.2-rc3	256838	651522	219	3453	26347	52556	106738	67523
weka-3.5.8	248704	682151	91	2019	19169	47561	124152	55710
jtopen-4.9	230394	593240	18	1940	20559	42206	112259	53410
tomcat-5.5.17	177249	433523	149	1777	17152	36247	80214	41708
compiere-250d	155379	388859	43	1260	18128	25458	73472	37016
jrefactory-2.9.19	145131	339539	152	2377	14360	28069	69583	30588
springframework-1.2.7	134537	345844	175	3359	15190	24186	58898	32727
jena-2.5.5	131297	344557	99	2491	16087	22670	53850	36098
xalan-j.2.7.0	116465	299784	45	1253	10261	22459	46033	36412
ant-1.7.1	112705	267462	87	1570	12485	26538	46413	25610
xerces-2.8.0	110658	284063	59	960	9078	23259	52158	25142
squirrel.sql-2.4	103945	263161	149	2508	11620	20755	42450	26461
aoi-2.5.1	100684	327557	25	806	6553	12553	52454	28291
exoportal-v1.0.2	96607	245273	470	2165	12386	17066	38103	26415
megamek-2005.10.11	95350	270531	29	634	6572	16304	53314	18495
jruby-1.0.1	90464	237725	50	1166	8816	17310	40733	22387
aspectj-1.0.6	89315	212129	47	1580	12293	18713	34534	22146
mvnforum-1.0-ga	88992	226592	61	617	7208	15728	43692	21684
azureus-3.1.1.0	88525	210525	134	1438	8874	19818	35455	22804
jgroups-2.6.2	88493	226551	23	1007	8310	16151	43564	19436
sandmark-3.4	88073	245897	126	1131	7429	13929	39713	25743
itext-1.4.5	87308	233801	32	710	6537	15168	42079	22780
jedit-4.3pre14	87088	224173	57	1153	6953	15245	42922	20756
hibernate-3.3.1-ga	86974	212863	100	1314	11876	18330	31594	23758
jmeter-2.3.2	77818	203577	134	950	8139	15647	34778	18168
poi-2.5.1	72893	195343	45	780	8510	12413	32430	18713
c.jdbc-2.0.2	71312	187144	155	796	5994	12749	33678	17938
jfreechart-1.0.1	69042	183005	55	713	6581	13227	30019	18445

Name/Version	Nod	Lnk	Pkg	Cls	Mth	Blk	Sta	Var
hsqldb-1.8.0.4	68174	175169	15	382	4897	13802	32831	16245
ireport-0.5.2	67994	195718	37	1323	5290	10122	38661	12559
struts-1.2.9	67756	150767	73	791	9483	15048	28553	13806
freecol-0.7.4	64207	174633	31	778	4794	13448	30208	14946
columba-1.0	63416	162011	199	1341	6913	12082	26459	16420
drjava-20050814	63160	164795	28	1356	7085	11339	29894	13456
jasperreports-1.1.0	61926	161668	42	815	7673	12782	26158	14454
findbugs-1.0.0	56686	150501	43	990	5795	9655	25295	14906
galleon-1.8.0	51840	135837	36	724	3819	9762	25066	12431
jext-5.0	51776	140724	56	818	3940	8222	27815	10923
antlr-2.7.7	49316	120770	14	298	3308	11002	25161	9531
pmd-3.3	45635	103427	57	677	4398	8952	22614	8935
heritrix-1.8.0	44892	113124	46	553	4521	9601	18537	11632
colt-1.2.0	44612	117946	27	592	4143	6978	18358	14512
rssowl-1.2	42674	125329	23	715	2464	4130	28909	6431
pooka-1.1-060227	42588	106430	27	802	4166	9338	18435	9818
jung-1.7.6	37741	101931	74	801	4133	6262	14896	11573
roller-2.1.1-incubating	35911	78393	53	427	3771	7097	14442	10119
james-2.2.0	35078	87966	38	503	3236	7774	15133	8392
trove-1.1b5	32353	73515	4	555	4664	7600	10633	8895
proguard-3.6	29563	75305	21	409	3977	5762	9012	10380
drawswf-1.2.9	26520	71057	44	325	2747	4040	10661	8701
velocity-1.5	25620	59716	39	353	2556	4914	11917	5839
ganttproject-1.11.1	24613	67425	37	549	2724	4219	10463	6619
quartz-1.5.2	24575	61428	42	185	2215	4945	10616	6570
axion-1.0-M2	24505	56203	14	261	2978	5693	10656	4901
lucene-1.4.3	23928	61307	17	333	2096	3769	11856	5855
htmlunit-1.8	23456	56642	14	362	3112	4778	8392	6796
sablecc-3.1	22929	57684	7	267	2191	5058	8959	6445
joggplayer-1.1.4s	22082	58094	17	311	1790	3724	10408	5830
jgraphpad-5.10.0.2	21630	62411	27	442	1879	3761	9289	6230
sunflow-0.07.2	21579	64717	22	239	1507	2862	10306	6641
ivatagroupware-0.11.3	21214	50862	87	226	1996	4510	8256	6137
jspwiki-2.2.33	20759	54309	26	314	1979	4060	8686	5692
jparse-0.96	19933	47566	4	75	905	3824	11349	3774
emma-2.0.5312	19647	49134	30	325	1813	3542	8189	5746
jgraph-5.9.2.1	18855	52349	18	256	2021	3489	7571	5498
quickserver-1.4.7	18798	47198	33	260	1692	3560	9880	3371
log4j-1.2.13	18185	43805	49	349	1948	3694	7643	4500
jsXe-04.beta	17890	47195	19	270	1445	3482	8089	4583
xmojo-5.0.0	17303	43089	28	168	1457	3528	8008	4112
cobertura-1.9	16639	39569	22	110	1239	2486	10468	2312
freecs-1.2.20060130	16184	43826	11	129	981	2927	8965	3169
openjms-0.7.7-alpha-3	15604	37055	27	275	1974	3593	5853	3880
jhotdraw-5.3.0	15065	38506	15	273	2283	3192	5387	3913
jag-5.0.1	15001	39581	23	259	1399	2909	6782	3627
javacc-3.2	14739	40712	8	140	715	2460	9036	2378
displaytag-1.1	14731	34959	29	335	1701	3104	5910	3650
jgrapht-0.7.3	13223	37994	19	279	1280	2109	6313	3221
informa-0.6.5	12980	32712	20	221	1581	2574	5125	3457
jrat-0.6	12440	32003	51	249	1463	2377	4719	3579

Name/Version	Nod	Lnk	Pkg	Cls	Mth	Blk	Sta	Var
jpf-1.0.2	12158	30827	12	152	1271	2895	5065	2761
marauroa-2.5	11245	28188	20	159	1223	2322	4809	2710
argouml-0.24	9802	23975	17	122	1087	1946	4239	2389
webmail-0.7.10	9244	23352	24	122	1071	1947	3694	2384
oscache-2.3-full	8049	18863	16	119	779	1869	3358	1906
quilt-0.6-a-5	7735	18808	16	114	820	1571	3270	1942
fitlibrary-20050923	7706	17826	17	234	1103	1444	2940	1966
jFin.DateMath-R1.0.0	6866	17277	26	112	816	1234	2878	1798
nekohtml-0.9.5	6606	17153	7	54	422	1453	2887	1781
jmoney-0.4.4	6310	17618	6	193	713	996	2989	1411
junit-4.5	6308	14766	33	400	1233	1362	2112	1166
jchempaint-2.0.12	5757	15844	8	125	419	1146	2696	1361
jasml-0.10	5482	15419	8	53	256	895	3011	1257
fitjava-1.1	3862	10296	5	96	462	786	1564	947
picocontainer-1.3	3771	9117	5	99	540	842	1155	1128

Appendix C

Version pairs

C.1 *ant* release pairs

Pair Number	Version 1	Version 2
1	1.1	1.2
2	1.2	1.3
3	1.3	1.4
4	1.4	1.4.1
5	1.4.1	1.5
6	1.5	1.5.1
7	1.5.1	1.5.2
8	1.5.2	1.5.3.1
9	1.5.3.1	1.5.4
10	1.5.4	1.6.0
11	1.6.0	1.6.1
12	1.6.1	1.6.2
13	1.6.2	1.6.3
14	1.6.3	1.6.4
15	1.6.4	1.6.5
16	1.6.5	1.7.0
17	1.7.0	1.7.1
18	1.7.1	1.8.0
19	1.8.0	1.8.1

C.2 *antlr* release pairs

Pair Number	Version 1	Version 2
1	2.4.0	2.5.0
2	2.5.0	2.6.0
3	2.6.0	2.7.0
4	2.7.0	2.7.1
5	2.7.1	2.7.2
6	2.7.2	2.7.3
7	2.7.3	2.7.4
8	2.7.4	2.7.5
9	2.7.5	2.7.6
10	2.7.6	2.7.7
11	2.7.7	3.0
12	3.0	3.0.1
13	3.0.1	3.1
14	3.1	3.1.1
15	3.1.1	3.1.2
16	3.1.2	3.1.3
17	3.1.3	3.2

C.3 *argouml* release pairs

Pair Number	**Version 1**	**Version 2**
1	0.16.1	0.18.1
2	0.18.1	0.20
3	0.20	0.24
4	0.24	0.26
5	0.26	0.26.2
6	0.26.2	0.28
7	0.28	0.28.1
8	0.28.1	0.30
9	0.30	0.30.2

C.4 *azureus* release pairs

Pair Number	Version 1	Version 2
1	2.0.8.2	2.0.8.4
2	2.0.8.4	2.1.0.0
3	2.1.0.0	2.1.0.2
4	2.1.0.2	2.1.0.4
5	2.1.0.4	2.2.0.0
6	2.2.0.0	2.2.0.2
7	2.2.0.2	2.3.0.0
8	2.3.0.0	2.3.0.2
9	2.3.0.2	2.3.0.4
10	2.3.0.4	2.5.0.4
11	2.5.0.4	3.0.0.8
12	3.0.0.8	3.0.1.0
13	3.0.1.0	3.0.1.2
14	3.0.1.2	3.0.1.4
15	3.0.1.4	3.0.1.6
16	3.0.1.6	3.0.2.0
17	3.0.2.0	3.0.2.2
18	3.0.2.2	3.0.3.0
19	3.0.3.0	3.0.3.4
20	3.0.3.4	3.0.4.0
21	3.0.4.0	3.0.4.2
22	3.0.4.2	3.0.5.0
23	3.0.5.0	3.0.5.2
24	3.0.5.2	3.1.0.0
25	3.1.0.0	3.1.1.0
26	3.1.1.0	4.0.0.0
27	4.0.0.0	4.0.0.2
28	4.0.0.2	4.0.0.4
29	4.0.0.4	4.1.0.0
30	4.1.0.0	4.1.0.2
31	4.1.0.2	4.1.0.4
32	4.1.0.4	4.2.0.0
33	4.2.0.0	4.2.0.2
34	4.2.0.2	4.2.0.4
35	4.2.0.4	4.2.0.8
36	4.2.0.8	4.3.0.0
37	4.3.0.0	4.3.0.2

Continued on next page

Continued from previous page

Pair Number	Version 1	Version 2
38	4.3.0.2	4.3.0.4
39	4.3.0.4	4.3.0.6
40	4.3.0.6	4.3.1.0
41	4.3.1.0	4.3.1.2
42	4.3.1.2	4.3.1.4
43	4.3.1.4	4.4.0.0
44	4.4.0.0	4.4.0.2
45	4.4.0.2	4.4.0.4
46	4.4.0.4	4.4.0.6
47	4.4.0.6	4.4.1.0
48	4.4.1.0	4.5.0.0
49	4.5.0.0	4.5.0.2
50	4.5.0.2	4.5.0.4

C.5 *freecol* release pairs

Pair Number	Version 1	Version 2
1	0.3.0	0.4.0
2	0.4.0	0.5.0
3	0.5.0	0.5.1
4	0.5.1	0.5.2
5	0.5.2	0.5.3
6	0.5.3	0.6.0
7	0.6.0	0.6.1
8	0.6.1	0.7.0
9	0.7.0	0.7.1
10	0.7.1	0.7.2
11	0.7.2	0.7.3
12	0.7.3	0.7.4
13	0.7.4	0.8.0
14	0.8.0	0.8.1
15	0.8.1	0.8.2
16	0.8.2	0.8.3
17	0.8.3	0.8.4
18	0.8.4	0.9.0
19	0.9.0	0.9.1
20	0.9.1	0.9.2
21	0.9.2	0.9.3
22	0.9.3	0.9.4

C.6 *hibernate* release pairs

Pair Number	Version 1	Version 2
1	0.8.1	1.0
2	1.0	1.1
3	1.1	2.0-rc2
4	2.0-rc2	2.0-final
5	2.0-final	2.0.1
6	2.0.1	2.0.2
7	2.0.2	2.0.3
8	2.0.3	2.1-rc1
9	2.1-rc1	2.1-final
10	2.1-final	2.1.1
11	2.1.1	2.1.2
12	2.1.2	2.1.3
13	2.1.3	2.1.4
14	2.1.4	2.1.5
15	2.1.5	2.1.6
16	2.1.6	2.1.7
17	2.1.7	2.1.8
18	2.1.8	3.0-rc1
19	3.0-rc1	3.0
20	3.0	3.0.1
21	3.0.1	3.0.2
22	3.0.2	3.0.3
23	3.0.3	3.0.4
24	3.0.4	3.0.5
25	3.0.5	3.1-rc1
26	3.1-rc1	3.1-rc2
27	3.1-rc2	3.1-rc3
28	3.1-rc3	3.1
29	3.1	3.1.1
30	3.1.1	3.1.2
31	3.1.2	3.1.3
32	3.1.3	3.2-cr1
33	3.2-cr1	3.2-cr2
34	3.2-cr2	3.2.0-cr3
35	3.2.0-cr3	3.2.0-cr4
36	3.2.0-cr4	3.2.0-cr5
37	3.2.0-cr5	3.2.0.ga

Continued on next page

Continued from previous page

Pair Number	Version 1	Version 2
38	3.2.0.ga	3.2.1-ga
39	3.2.1-ga	3.2.2-ga
40	3.2.2-ga	3.2.3-ga
41	3.2.3-ga	3.2.4-ga
42	3.2.4-ga	3.2.4-sp1
43	3.2.4-sp1	3.2.5-ga
44	3.2.5-ga	3.2.6-ga
45	3.2.6-ga	3.3.0.cr1
46	3.3.0.cr1	3.3.0-cr2
47	3.3.0-cr2	3.3.0-ga
48	3.3.0-ga	3.3.0-sp1
49	3.3.0-sp1	3.3.1-ga
50	3.3.1-ga	3.3.2-ga
51	3.3.2-ga	3.5.0-cr-1
52	3.5.0-cr-1	3.5.0-cr-2
53	3.5.0-cr-2	3.5.3-final
54	3.5.3-final	3.5.5-final

C.7 *jgraph* release pairs

Pair Number	Version 1	Version 2
1	5.4.4-java1.4	5.5
2	5.5	5.5.1
3	5.5.1	5.6.2.1
4	5.6.2.1	5.6.2
5	5.6.2	5.6.3
6	5.6.3	5.7
7	5.7	5.7.1
8	5.7.1	5.7.3
9	5.7.3	5.7.3.1
10	5.7.3.1	5.7.4
11	5.7.4	5.7.4.1
12	5.7.4.1	5.7.4.2
13	5.7.4.2	5.7.4.3
14	5.7.4.3	5.7.4.4
15	5.7.4.4	5.7.4.5
16	5.7.4.5	5.7.4.6
17	5.7.4.6	5.7.4.7
18	5.7.4.7	5.8.0.0
19	5.8.0.0	5.8.1.1
20	5.8.1.1	5.8.2.0
21	5.8.2.0	5.8.2.1
22	5.8.2.1	5.8.3.1
23	5.8.3.1	5.9.0.0
24	5.9.0.0	5.9.1.0
25	5.9.1.0	5.9.2.0
26	5.9.2.0	5.10.0.0
27	5.10.0.0	5.10.0.1
28	5.10.0.1	5.10.2.0
29	5.10.2.0	5.11.0.0
30	5.11.0.0	5.11.0.1
31	5.11.0.1	5.12.0.0
32	5.12.0.0	5.12.0.1
33	5.12.0.1	5.12.0.4
34	5.12.0.4	5.12.1.0
35	5.12.1.0	5.12.2.1
36	5.12.2.1	5.13.0.0

C.8 *jmeter* release pairs

Pair Number	Version 1	Version 2
1	1.8.1	1.9.1
2	1.9.1	2.0.0
3	2.0.0	2.0.1
4	2.0.1	2.0.2
5	2.0.2	2.0.3
6	2.0.3	2.1-rc1
7	2.1-rc1	2.1
8	2.1	2.1.1
9	2.1.1	2.2
10	2.2	2.3-rc3
11	2.3-rc3	2.3-rc4
12	2.3-rc4	2.3
13	2.3	2.3.1
14	2.3.1	2.3.2
15	2.3.2	2.3.3
16	2.3.3	2.3.4
17	2.3.4	2.4

C.9 *jung* release pairs

Pair Number	Version 1	Version 2
1	1.0.0	1.1.0
2	1.1.0	1.1.1
3	1.1.1	1.2.0
4	1.2.0	1.3.0
5	1.3.0	1.4.0
6	1.4.0	1.4.1
7	1.4.1	1.4.2
8	1.4.2	1.4.3
9	1.4.3	1.5.0
10	1.5.0	1.5.1
11	1.5.1	1.5.2
12	1.5.2	1.5.3
13	1.5.3	1.5.4
14	1.5.4	1.6.0
15	1.6.0	1.7.0
16	1.7.0	1.7.1
17	1.7.1	1.7.2
18	1.7.2	1.7.4
19	1.7.4	1.7.5
20	1.7.5	1.7.6
21	1.7.6	2.0
22	2.0	2.0.1

C.10 *junit* release pairs

Pair Number	Version 1	Version 2
1	2.0	2.1
2	2.1	3.0
3	3.0	3.4
4	3.4	3.5
5	3.5	3.6
6	3.6	3.7
7	3.7	3.8
8	3.8	3.8.1
9	3.8.1	3.8.2
10	3.8.2	4.0
11	4.0	4.1
12	4.1	4.2
13	4.2	4.3.1
14	4.3.1	4.4
15	4.4	4.5
16	4.5	4.6
17	4.6	4.7
18	4.7	4.8
19	4.8	4.8.1
20	4.8.1	4.8.2

C.11 *lucene* release pairs

Pair Number	Version 1	Version 2
1	1.2-final	1.3-final
2	1.3-final	1.4.3
3	1.4.3	1.9-rc1
4	1.9-rc1	1.9-final
5	1.9-final	1.9.1
6	1.9.1	2.0.0
7	2.0.0	2.1.0
8	2.1.0	2.2.0
9	2.2.0	2.3.0
10	2.3.0	2.3.1
11	2.3.1	2.3.2
12	2.3.2	2.4.0
13	2.4.0	2.4.1
14	2.4.1	2.9.0
15	2.9.0	2.9.1
16	2.9.1	2.9.2
17	2.9.2	2.9.3
18	2.9.3	3.0.0
19	3.0.0	3.0.1
20	3.0.1	3.0.2

C.12 *weka* release pairs

Pair Number	Version 1	Version 2
1	3.0.1	3.0.2
2	3.0.2	3.0.3
3	3.0.3	3.0.4
4	3.0.4	3.0.5
5	3.0.5	3.0.6
6	3.0.6	3.1.7
7	3.1.7	3.1.8
8	3.1.8	3.1.9
9	3.1.9	3.2
10	3.2	3.2.1
11	3.2.1	3.2.2
12	3.2.2	3.2.3
13	3.2.3	3.3
14	3.3	3.3.1
15	3.3.1	3.3.2
16	3.3.2	3.3.3
17	3.3.3	3.3.4
18	3.3.4	3.3.5
19	3.3.5	3.3.6
20	3.3.6	3.4
21	3.4	3.4.1
22	3.4.1	3.4.2
23	3.4.2	3.4.3
24	3.4.3	3.4.4
25	3.4.4	3.4.5
26	3.4.5	3.4.6
27	3.4.6	3.4.7
28	3.4.7	3.4.8
29	3.4.8	3.4.9
30	3.4.9	3.4.10
31	3.4.10	3.4.11
32	3.4.11	3.4.12
33	3.4.12	3.4.13
34	3.4.13	3.5.0
35	3.5.0	3.5.1
36	3.5.1	3.5.2
37	3.5.2	3.5.3

Continued on next page

Continued from previous page

Pair Number	**Version 1**	**Version 2**
38	3.5.3	3.5.4
39	3.5.4	3.5.5
40	3.5.5	3.5.6
41	3.5.6	3.5.7
42	3.5.7	3.5.8
43	3.5.8	3.6.0
44	3.6.0	3.6.1
45	3.6.1	3.6.2
46	3.6.2	3.7.0
47	3.7.0	3.7.1
48	3.7.1	3.7.2