System Design and the Cost of Architectural Complexity

by

Daniel J. Sturtevant

S.M. Engineering and Management, Massachusetts Institute of Technology, 2008
B.S. Computer Engineering, Lehigh University, 2001
B.A. Political Science, Lehigh University, 2000

Submitted to the Engineering Systems Division
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

Signature of Author .....
Engineering Systems Division
February, 2013

Certified by .....
Alan D. MacCormack, Ph.D.
MBA Class of 1949 Adjunct Professor of Business Administration, Harvard Business School
Thesis Supervisor

Certified by .....
Steven D. Eppinger, Sc.D.
General Motors Leaders for Global Operations Professor of Management Science and Engineering Systems
Thesis Committee Chair

Certified by .....
Christopher L. Magee, Ph.D.
Professor of the Practice of Engineering Systems
Thesis Committee Member

Certified by .....
Daniel Jackson, Ph.D.
Professor of Computer Science
Thesis Committee Member

Accepted by .....
Oliver L. de Weck, Ph.D.
Associate Professor of Aeronautics and Astronautics and Engineering Systems
Chairman, Engineering Systems Division Education Committee

Table of Contents

Table of Contents 3

List of Figures 7

List of Tables 8

ABSTRACT 9

1 Acknowledgements 11

2 Motivation 13

2.1 Research Questions 17

3 Literature Review 19

3.1 Complexity in Systems 19

3.2 Complexity in Software 21

3.3 The Importance of Controlling Complexity 23

3.3.1 Uncontrolled Complexity in Design Projects 23

3.3.2 Uncontrolled Complexity in Operational Systems 25

3.4 Architecture Controls Complexity 26

3.5 Important Architectural Patterns 28

3.5.1 The Benefits of Hierarchy 28

3.5.2 The Benefits of Modularity 29

3.5.3 The Benefits of Abstraction Layers and Platforms 31

3.6 Network Representations of Important Architectural Patterns 34

3.7 Networks in System Architecture 38

3.7.1 Design Structure Matrix Methods for Complex Systems 39

3.7.2 DSMs and System Structure 41

3.8 Exploring the Network Structure of a Software System 42

3.9 Some Definitions of Complexity 44

3.10 Measures of Software Complexity **46**

3.10.1 Component-Based Complexity Metrics 46

3.10.2 Structural Complexity Metrics 49

3.10.3 Criticism of Complexity Metrics 49

3.11 The Difficulty of Placing a Value on Redesign **50**

4 Conceptual Model, Gap in the Literature, and Hypotheses **53**

4.1 Conceptual Model **53**

4.2 Complexity and Quality **54**

4.3 Complexity and Productivity **57**

4.3.1 Software Cost and Schedule Estimation Techniques 58

4.3.2 The Impact of Architecture on Productivity 60

4.3.3 The Productivity of Individual Software Developers 61

4.4 Complexity and Human Capital **63**

5 Research Methods **67**

5.1 Measuring Architectural Complexity in Software **67**

5.1.1 Networks and DSM Architecture Representations 68

5.1.2 Procedure for Assigning Architectural Complexity Scores to Source Code Files 70

5.1.3 Component Architectural Categories: Peripheral, Utility, Control, Core 79

5.1.4 The Relationship Between Hierarchy, Modularity, and MacCormack's Metrics 80

5.2 Measuring Costs of Software Development and Maintenance **83**

5.2.2 Means of Capturing Complexity and Cost Data 87

6 Research Setting **91**

6.1 Organization Under Study: Large Scale Commercial Software Firm **91**

6.2 The Software Development Process at Iron Bridge **92**

6.3 Data Selected for Use in Studies **95**

6.3.1 Software Source Code Files and Developers Chosen for Study 95

6.3.2 Procedures Used To Clean Data Samples 96
6.4 Structure and Complexity of Files in the Sample 99
7 Result 1: Link Between Architectural Complexity and Defects 103
7.1 Descriptive Statistics on Files and Complexity 103
7.2 Modeling Architectural Complexity and Defects 105
7.3 Regression Models 108
7.4 Interpretation of Results 111
8 Result 2: Link Between Architectural Complexity and Productivity 115
8.1 Descriptive Statistics on Developer Productivity 115
8.2 Modeling Architectural Complexity and Developer Productivity 119
8.3 Regression Models 122
8.4 Interpretation of Results 127
9 Result 3: Link Between Architectural Complexity and Staff Turnover 133
9.1 Descriptive Statistics on Developer Turnover 133
9.2 Modeling Architectural Complexity and Staff Turnover 134
9.3 Regression Models 138
9.4 Interpretation of Results 141
10 Discussion & Conclusions 145
10.1 Contributions to Academic Literature 146
10.2 Contributions to Managerial Practice: 149
10.3 Limitations of This Work 151
10.4 Directions for Future Work 153
10.5 Concluding Remarks 156
11 Bibliography 159

List of Figures

Figure 1: Hierarchical Tree as Network 34
Figure 2: Hierarchical Tree as DSM 34
Figure 3: Note about DSM and Network Conventions 35
Figure 4: Modules as Network 36
Figure 5: Modules as DSM 36
Figure 6: Layers as Network 36
Figure 7: Layers as DSM 36
Figure 8: Hierarchy of Modules as Network 38
Figure 9: Hierarchy of Modules as DSM 38
Figure 10: DSM of a Laptop Computer (From Baldwin & Clark) 40
Figure 11: Conceptual Model 53
Figure 12: Simple Network 69
Figure 13: Simple DSM 69
Figure 14: Moscow Subway Map 70
Figure 15: Mozilla DSM 70
Figure 16: Simple Pseudocode 72
Figure 17: Hierarchical Software System 73
Figure 18: "Core-Periphery" Software System 73
Figure 19: Simple Network (Direct) 75
Figure 20: Simple Network (Transitive Closure) 75
Figure 21: Simple DSM (Direct) 75
Figure 22: Simple DSM (Transitive Closure) 75
Figure 23: Hierarchy of Modules 78
Figure 24: Hierarchy of Modules with Unwanted Links 78
Figure 25: DSM of Hierarchy of Modules 78
Figure 26: DSM of Hierarchy of Modules with Unwanted Links 78
Figure 27: Transitive Closure DSM of Hierarchy of Modules 78
Figure 28: Transitive Closure DSM of Hierarchy of Modules with Unwanted Links 78
Figure 29: Distribution of Visibility Scores and Cutoff Points for a "Core Periphery" Network 79
Figure 30: Primary Developer Workflow 84
Figure 31: Change Tracking, Version Control, and Integration Between the Two 86
Figure 32: Infrastructure for Extracting, Relating, and Analyzing Data 89
Figure 33: Fixed Development Window 92
Figure 34: Primary Developer Workflow and Release Cycle 93
Figure 35: DSM for Release 1 96
Figure 36: DSM for Release 8 96
Figure 37: Release 7 DSMs and Visibility Plots 101
Figure 38: Expected Number of Bugs Fixed in a File (1) 112
Figure 39: Expected Number of Bugs Fixed in a File (2) 112
Figure 40: Expected Number of Lines to Fix Bugs in a File (1) 113
Figure 41: Expected Number of Lines to Fix Bugs in a File (2) 114
Figure 42: Histogram of Activity in High McCabe Files 119
Figure 43: Histogram of Activity in Core 119
Figure 44: Developer Productivity and Architectural Complexity 127
Figure 45: Predicted Developer Turnover (1) 143
Figure 46: Predicted Developer Turnover (2) 143

List of Tables

Table 1: Mapping of Visibility Scores to Architectural Complexity Classification 80
Table 2: Propagation Cost of Structured and Random Networks 82
Table 3: File-Based Metrics Captured 88
Table 4: Developer-Based Metrics Captured 88
Table 5: File Count Broken Down by Complexity Classification 102
Table 6: Measures of Development Activity During Each Release 104
Table 7: Averages for File Size, Change Size, and Development Activity During Each Release 104
Table 8: Variables Included In Statistical Models Predicting Defect Proneness 106
Table 9: Predicting Number of Changes in a File to Fix Bugs. (Negative Binomial Model) 109
Table 10: Predicting LOC Changed in a File to Fix Bugs. (Negative Binomial Model) 110
Table 11: Expected Value For the Number of Bug-Fix Changes in the "Typical" File 111
Table 12: Expected Value For the Number of Lines Submitted to Fix Bugs in "Typical" File 113
Table 13: Developers and Activity in Each Release 117
Table 14: Activity For the Average Developer by Task and Location in Codebase For Each Release 118
Table 15: Variables Included In Statistical Models Predicting Developer Productivity 121
Table 16: Predicting LOC Produced per Developer to Implement Features For One Release (Neg Binomial Panel Data Model) 124
Table 17: Predicting LOC Produced per Developer to Fix Defects For One Release (Neg Binomial Panel Data Model) 125
Table 18: Predicting LOC Produced per Developer For One Release. (Neg Binomial Panel Data Model) .. 126
Table 19: Predicted Number of Lines Produced by Average Developer at Various Architectural Complexity Levels if Only Implementing Features or Doing Other Non-Bug Related Tasks 130
Table 20: Predicted Number of Lines Produced by Average Developer at Various Architectural Complexity Levels if Only Fixing Bugs 131
Table 21: Predicted Number of Lines Produced by Average Developer at Various Architectural Complexity Levels 132
Table 22: Comparing the Population of Stayers and Leavers 134
Table 23: Variables Included in Statistical Model Predicting Developer Turnover 137
Table 24: Predicting Turnover Among Developers Based on Rankings (Logistic Model) 139
Table 25: Predicting Turnover Among Developers (Logistic Model) 140
Table 26: Predicted Probability of Leaving the Firm For Developers Based On Their Relative Amount of Work in Core 142
Table 27: Predicted Probability of Leaving the Firm at Various Architectural Complexity Levels 142

System Design and the Cost of Architectural Complexity

by

Daniel J. Sturtevant

Submitted to the Engineering Systems Division
on February 3rd, 2013 in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy
In Engineering Systems

ABSTRACT

Many modern systems are so large that no one truly understands how they work. It is well known in the engineering community that architectural patterns (including hierarchies, modules, and abstraction layers) should be used in design because they play an important role in controlling complexity. These patterns make a system easier to evolve and keep its separate portions within the bounds of human understanding so that distributed teams can operate independently while jointly fashioning a coherent whole.

This study set out to measure the link between architectural complexity (the complexity that arises within a system due to a lack or breakdown of hierarchy or modularity) and a variety of costs incurred by a development organization. A study was conducted within a successful software firm. Measures of architectural complexity were taken from eight versions of their product using techniques recently developed by MacCormack, Baldwin, and Rusnak. Significant cost drivers including defect density, developer productivity, and staff turnover were measured as well. The link between cost and complexity was explored using a variety of statistical techniques.

Within this research setting, we found that differences in architectural complexity could account for 50% drops in productivity, three-fold increases in defect density, and order-of-magnitude increases in staff turnover. Using the techniques developed in this thesis, it should be possible for firms to estimate the financial cost of their complexity by assigning a monetary value to the decreased productivity, increased defect density, and increased turnover it causes. As a result, it should be possible for firms to more accurately estimate the potential dollar-value of refactoring efforts aimed at improving architecture.

Thesis Supervisor: Alan D. MacCormack
Title: Adjunct Professor of Business Administration, Harvard Business School

Thesis Chairman: Steven D. Eppinger
Title: Professor of Management Science and Engineering Systems

Keywords: System architecture; System design; Software engineering; Quality; Technical debt; Complexity; Modularity; Design structure matrix; Network

1 Acknowledgements

There are many people who cannot be appropriately thanked for the love and support they provided over the past 5 years. First and foremost is Jessica, without whom none of this would be possible. Her loving support, encouragement, excessive workload, and statistical expertise were more than I deserved. I would also like to thank our son Trent and twin daughters Maggie and Alexis for putting up with me. All three arrived while I was in graduate school, and each deserves a pony or a jetpack to make up for some of the time I could not spend with them. Thank you for being wonderful little people. I look forward to watching you grow. I would also like to thank my parents, Betty & Dave, and my parents-in-law, John & Carolyn, for moral support, advice, and occasional childcare.

I would also like to thank Alan MacCormack, who is the best advisor and collaborator that a graduate student could hope for. He is brilliant but unassuming, helpful but unobtrusive, and is generally just an all-around great mentor and person. I also enjoyed the helpful guidance of a Steven Eppinger, Chris Magee and Daniel Jackson, each of whom provided valuable feedback and support during the development of this thesis. I was incredibly lucky to have the opportunity to work with and learn from a fantastic group of people at MIT and Harvard over many years. I owe a debt of gratitude to a great many of them. In particular, I would like to thank Richard Larson, Michael Davies, Carliss Baldwin, Oli de Weck, Joel Moses, Dan Whitney, Chuck Hastings, Eric Klopfer, Pat Hale, Whitman Richards, Brad Morrison, Stewart Madnick, Nazli Choucri, Helen Trimble, and the indomitable Beth Milnes for various types of generous guidance and support.

I would like to thank "Iron Bridge Software" for allowing me access to their management systems and software for research purposes. Individuals at Iron Bridge who deserve specific credit include "Clark Kent," "She-Ra," and "Zephrin" for providing technical resources and insights into company culture, processes, software architecture, metrics, and tools. I must also thank them for generously agreeing to review this work and for offering feedback and suggestions for improvement. (I wish I could acknowledge these individuals by name, but pseudonyms will have to suffice.) It is a rare researcher who is afforded the opportunity to do this type of collaborative work with a commercial firm. I am very lucky to have had this experience and wish Iron Bridge and its employees future success and happiness.

Finally, I would like to thank the organizations that have helped fund my research or teaching assistantships including Siemens, IARPA, NASA/NOAA, and MIT's Engineering Systems Division, Computer Science and Artificial Intelligence Laboratory, and Sloan School of Management.

2 Motivation

Designers of today's large systems struggle with the fact that those systems are both *complicated* and *complex*. Modern systems are *complicated* in the sense that they have far exceeded the bounds of human understanding. [1-3] The collective knowledge required to complete a design is much broader than could be internalized by a single person over the course of a lifetime. Hundreds or thousands of engineers now make intellectual contributions to the design of single artifacts. Secondly, modern systems are *complex*. One of the defining features of complex systems is that they are often interconnected in ways that enable unanticipated behavior to emerge as a result of unexpected interactions between system components. Because of this *emergent behavior*, the whole often does not behave in a manner that logically follows from the independent functioning of its parts.

System architects often take great pains to keep complexity under control because overly complex systems carry a variety of costs and risks. They are more expensive to design, harder to maintain, and can be more prone to failure. System complexity clearly adds value as well, however. Over the last century, increasingly complex machines, services, processes, and infrastructures have done old jobs better and provided new capabilities that were previously unimaginable. While complexity can be costly, a higher-complexity system may very well be worth the price. [4] A natural tradeoff therefore exists between enabling valuable functionality or performance characteristics and keeping complexity under control.

Complexity across systems, and the complexity of different regions within the same system, can vary widely. In the battle to constrain and channel the behavior of a large system so that complexity is appropriately managed, a principal weapon in the designer's arsenal is the *architectural pattern*. Architects striving to make large systems tractable make them hierarchical, compose them of independent modules, separate them into conceptual layers, and reuse parts. These types of architectural patterns endow systems with inherently beneficial properties [5], and also "addres[s] basic human limitations in dealing with complexity." [6] Design is not easy or straightforward, however. Weighing the costs and benefits of alternative architectural choices is difficult. Designers must choose between multiple competing ways to decompose a system into hierarchical

structures and competing criteria for determining which functionality should be clustered in each module. [7] They must determine how big each module should be and how interfaces between them should be structured. In addition, hierarchy and modularity are not free – they impose their own costs, may impact performance, and can limit the scope of future decision-making. A designer must trade performance requirements against complexity controlling features across the system being designed. These choices will have a profound impact on how complicated and complex different portions of a system will be. As a result, a system may have regions bearing widely varying costs and risks.

Unfortunately, there has been is little quantitative work to help managers and designers understand the cost of complexity in an architecture. Because of this, it is hard for them to place a value on hierarchy and modularity in a system design. It is hard for them to understand the burden that a company is forced to shoulder when architectural patterns degrade over the life of a long-lived system. Finally, it is hard for them to objectively weigh the value of refactoring efforts aimed at asserting (or reasserting) various principles of large-scale system design.

The purpose of the study described in this thesis is to begin to fill this gap. In this report, we describe research that was done to measure costs incurred by a successful commercial software development firm during the ongoing development of a mature software product. Tens of thousands of individual software source-code files were assigned complexity scores related to their network positions within the software's architecture. A variety of costs were measured over a eight back-to-back development windows. These included the number of defects in complex files, differences in the productivity of engineers working with complex files, and differences in staff turnover among engineers working in complex files. By presenting a well-rounded consideration of the costs of complexity in different regions of a mature technical architecture, we hope to provide insights that might allow development organizations to weigh important tradeoffs in a structured manner. Complexity is neither good nor bad. It provides value, but is also costly. An ability to measure the multidimensional costs of complexity could help to inform important decisions made during the design and maintenance of today's large software systems.

Over the course of this discussion, we will focus on a specific type of complexity: *architectural complexity.* Regions within a system that are more *architecturally complex* have fewer hierarchical and modular structures governing the relationships between system elements. Hierarchy and modularity are well known patterns employed by man and nature to keep complexity under control even as systems grow. Technical architectures in which these patterns are judiciously applied often have a variety of evolutionary advantages. They tend to be more stable, of higher quality, safer, and benefit from other "ilities" over the course of their lifecycles. During this research, we operationalize the concept of architectural complexity by using network algorithms and metrics devised by MacCormack, Baldwin, and Rusnak [8, 9] that classify system elements based on their level of coupling (both direct and indirect) with the rest of the system. Because these procedures are designed to identify regions of an architecture containing large system spanning cycles of dependencies, they can identify regions in which hierarchical structure and modular isolation are relatively absent. These high-complexity regions may exist because they were originally designed to be high coupled, or because of a subsequent degradation of complexity controlling patterns.

A software system was chosen for analysis because software has several unique properties that enable this study. Firstly, software is an artifact that embodies pure function unencumbered by the burden of physical form. Software development firms engage in design but have no need for manufacturing or assembly. By observing a large software firm, we are measuring a design process unencumbered by many economic constraints, such as the large fixed costs found in an aerospace plant, and with few serious constraints imposed by physics. By looking at a software firm, we are in some senses isolating the impact of complexity on the cost of maintaining a design. Secondly, software is grown, evolved, and reconfigured more rapidly than other systems precisely because of this relative dearth of economic and physical constraints. Empirical observations of the same phenomena can be made over shorter time-scales. Thirdly, successful software systems have the property that they are extremely long-lived. MacCormack tells us "mature products often contain significant amounts of code from their earliest versions, even if major evolutions in design have since been made." He tells us that as they grow and take on new functionality they "bare the consequences of decisions made long ago." [9] This continuity between successive versions supports longitudinal analysis.

Finally, the architecture of a software system can be automatically extracted from its source code, and important cost-related information can be assembled from version control systems, bug-tracking systems, and other databases commonly used in software development, thus making it feasible to explore the cost of complexity in very large systems.

In this study, we found that differences in architectural complexity accounted for differences in developer productivity of 50%, three-fold differences in defect density, and order-of-magnitude differences in staff turnover. Costs of this magnitude make a strong case for the benefits of design patterns that manage complexity and the value of system redesign efforts aimed at imposing (or re-imposing) those patterns.

2.1 Research Questions

This study aims to address one overarching question:

- **What costs does architectural complexity within a software system impose on the design organization that develops and maintains it?**

There are several costs that architectural complexity within a design might impose on the development organization. We choose to focus on a subset with a direct and measurable impact on payroll – the principal expenditure within many software and engineering design firms:

- **Quality: Do software components in more architecturally complex regions of a codebase experience more defects?** Every hour that an engineer spends correcting a defect is pure waste. It is an activity that the organization must perform, but derives no value from. Quality also impacts customers, and therefore impacts firm reputation, adoption, and market-share.

- **Productivity: Are engineers who work in software components with higher levels of architectural complexity less productive?** System components that are harder to work with waste designers' time. If designers are twice as productive working in components where architectural complexity is low, we can say that high complexity consumes half of that individual's effort.

- **Human Capital: Is there a higher rate of staff turnover among engineers working in software components with higher levels of architectural complexity?** If complexity is related to higher turnover, we can say that the cost of that complexity is the cost of recruiting and training replacements. Furthermore, if complexity is a causal factor in staff turnover, then it clearly must play some role in harming morale as well.

3 Literature Review

An interesting and diverse body of work has looked at many of the issues around the relationship between complexity, architecture and costs. In this section, we review pertinent literature on complexity in systems, and more specifically on software complexity. We explore the dangers that complexity in a technical system can pose. We then turn to the role of system architecture and the means by which common architectural patterns (such as hierarchy, modularity, and abstraction layering) in a design keep complexity under control. We look at how these canonical architectural patterns can be measured and reasoned about by exploring high-level network representations of real systems. We then look at past work that has explored software designs using network-analysis and review the history of complexity measurement in software. We conclude by looking at the problems an organization faces when trying to decide whether (or how) to reduce complexity within an existing system's architecture.

3.1 Complexity in Systems

Two things that make today's systems hard to design and maintain are that they are *complicated* and they are *complex*. By *complicated*, we mean that they are so large or detailed that no single individual can understand how they work. By *complex*, we often mean that interactions between parts can result in strange behavior that is hard to anticipate and which can threaten safe and reliable operation.[1] This was not always the case. During the time period since the beginning of the Industrial Revolution until the advent of complex systems in the early twentieth century, those individuals running design and manufacturing organizations were capable of understanding their processes and products. During this "epoch of great inventions and artifacts" [10] large hierarchically structured organizations grew by taking advantage of differentiated labor and interchangeable parts. [11-13] The design process, however, remained in the hands of small groups of people. Once a

[1] This distinction between complexity and complicatedness is used by Crawley [1], while many others use the word complexity to denote both meanings. In this report, we will sometimes distinguish between the two where the distinction is appropriate, but will also use the term complexity to refer to both psychological aspects and properties inherent to a system elsewhere.

problem was understood, managers coped with the demands of accomplishing a large task by dividing it until each sub-task was small enough for a person or team to handle. Hierarchical control, division of task, and assembly of standard parts led from Adam Smith [13] to Taylor's system of Scientific Management [14], Ford's assembly line, and Edison's electrification at the turn of the twentieth century. Then something began to change. Systems such as the telephone network [15] and automated gunfire control systems [16] seemed to increasingly resist reductionist approaches. The process of designing and operating modern machines began to change in fundamental ways. [10]

The technical knowledge required to complete a modern system's design is much larger than could be learned by a single person over the course of a lifetime. These systems have far exceeded the bounds of human understanding. [1-3] Complicated systems sometimes consist of billions of parts connected in countless ways. Hundreds or thousands of engineers make intellectual contributions to the design of these artifacts. As a result, it is no longer only the organization, the product, and the production process that must be decomposed. The *design process* itself must be subdivided and allocated to large groups of people with different skills. Those charged with designing and evolving a complicated system must grope for means of managing the *structure of the design process* (the layout of teams and the communication channels between them) even though everyone involved is at least partially blind. A recent British Royal Academy of Engineering report says that "[o]n a large software project one is lucky if one person in 50 has anything resembling an overall understanding of the conceptual structure of the project, and divinely blessed if that person has the ability to explain it in lay terms." [17] It is often now impossible for a group of engineers to really know if a flaw in the decomposition of the design organization will lead them to miss opportunities to create a good *technical structure*, or if the collective "unknown-unknowns" will wreak havoc on the end result. [5] Further compounding this issue is the fact that today's large system designs are often created by teams that span firms, institutions, and continents. Somehow, mysteriously, many of our complex systems work even though no one can truly claim to know how or why.

In the past hundred years our systems also became *complex*. One of the defining features of complex systems is that of *emergent behavior* – the idea that the whole often does not behave in a manner that logically follows from the independent functioning of its parts. The need to avoid emergent behavior (to prevent defects or disaster) or the desire to find and exploit it requires modern organizations to employ strategies, processes, and structures beyond hierarchical reductionism. Some properties that we require in our complex systems – such as safety – cannot be obtained by assigning responsibility to a single group because they are systemic in nature. Accidents often result from unanticipated interactions *between* parts, not from problems identifiable within individual components. [18, 19]

3.2 Complexity in Software

The invention of software adds a new twist because it *is* function unencumbered by the burden of form. During the industrial revolution, *production* was distributed but *design* was not. In complex electro-mechanical systems, both the *design* and *production* were separate things to be decomposed. Now, with the advent of large software products, *design* is distributed but the notion of a *production* process entirely eliminated. There are no large fixed costs and no serious physical constraints. By examining software evolution and development activity in a large software firm, we are observing complex design evolution in a strikingly pure form.[2] The evolutionary forces guiding the development of programming languages and methodologies since the 1970s have often favored those technologies that allow humans to better manage complexity. Fred Brooks forcefully makes this point:

---

[2] Software developers continuously encounter semi-routine tasks. They write programs to automate them as part of their normal workflow. As these programs become more general, they are often shared. The software industry as a whole responds in a similar manner as it evolves. Whenever a concept becomes sufficiently well understood that it can be turned into a repeatable process and routinized, a program is written to perform the task. If a concept is sufficiently abstract, it will become a design pattern used in existing languages, and might later become a syntactical construct embedded in new languages. There is no room in software for non-design professionals because non-creative tasks are mechanized as a matter of course.

> Software entities are more complex for their size than perhaps any other human construct because no two parts are alike... In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound... Likewise, a scaling-up of a software entity is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly... Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability. From complexity of function comes the difficulty of invoking function, which makes programs hard to use. From complexity of structure comes the difficulty of extending programs to new functions without creating side effects. From complexity of structure come the unvisualized states that constitute security trapdoors. [20]

Brooks is not alone in his belief in the essential complexity of software. In a paper reviewing the evolution of the field from its inception until 1997, Shapiro concluded "[f]rom the 1960s onward, many of the ailments plaguing software could be traced to one principal cause - complexity." Shapiro begins his paper with the assertion that the "fundamental nature of software—Involv[es] basic and poorly understood problem-solving processes combined with unprecedented and multifaceted complexity", and then proceeds to employ the word 107 more times. [6] In his review paper, complexity fell into two general categories: "program complexity" [21] related to of control, size, modularity, information content, and data structures and "psychological complexity," [22] related to "problem comprehension, translation, and system design." Basili refined this notion of psychological complexity by saying that complexity is "a measure of the resources expended by another system while interacting with a piece of software. If the interacting system is people, the measures are concerned with human efforts to comprehend, to maintain, to change, to test, etc., that

software." [23]

Shapiro said that as hardware became more capable during the 1970's, program efficiency took a backseat to complexity management and human understandability as the primary evolutionary force within the field. [6] Since that time, advocates for many successful software engineering innovations such as stepwise refinement, information hiding, top-down design, structured design, and object orientation each put their newly-preferred method forward as a better means of coping with complexity, complicatedness, or both.

3.3 The Importance of Controlling Complexity

System designers must place great importance on controlling complexity during the design process and in systems once they are operational. When complexity in a design is well managed it makes the design process proceed more smoothly and makes the resulting system more reliable. Unfortunately, "Complexity in large scale IT systems remains an area which is insufficiently well understood. The degree of complexity entailed in achieving a particular objective can be very difficult to estimate at the project outset." [17] Complexity in a design always has the potential to be problematic. Even when it remains manageable, it causes project delays, defects and other forms of waste. When we lose control of complexity, project failures can sink firms and accidents can cause property destruction and loss of life.

3.3.1 Uncontrolled Complexity in Design Projects

Complexity in the design process has led to many of high-profile cost overruns, project failures and bankruptcies. Lyneis says that "a major reason for continued schedule and budget performance problems is that while projects are fundamentally complex dynamic systems, most project management concepts and tools either (1) view a project statically or (2) take a partial, narrow view in order to allow managers to cope mentally with the complexity. Traditional tools and mental models are inadequate for dealing with the dynamic complexity of projects." [24] A few notable examples of failed software projects include a multi-billion dollar attempt to create an FAA air-control system [25] and an automated baggage handling system in the Denver airport that "mutilated and lost bags." [26]

Coupled interactions and feedback between quality, productivity, rework, and turnover are often responsible for success or failure on large projects. A family of empirically validated differential-equation-based project management models has been created to link and explore these causal influences. [24] Many of these feedback models have also been designed and calibrated specifically for the purpose of understanding project and organizational dynamics in software. [27-29] These system dynamics models simulate a variety of interrelated pathologies that can sink large projects. Quality problems can lead to low productivity because designers must create workarounds and sometimes build on top of functionality that is eventually reworked or scrapped. Low productivity leads to project delays, time pressure, and further quality problems. Delays lead to overtime and overwork, leading to fatigue, mistakes, burnout and turnover. Turnover leads to the hiring of rookies, whom are both more likely to introduce defects, and less productive than veterans. In addition, rookies require mentorship, which takes time away from the veterans who would otherwise be working in the system. Overworked mentors might spend less time helping new employees mature. These project dynamics models tell us that some projects operate within acceptable thresholds allowing them to complete in a reasonable manner, while others succumb to these interrelated pathologies. The causal relationships influencing the evolution of a large program combine to form a system that is highly non-linear and can be unstable. Seemingly minor policy choices can be the only thing separating a smoothly running project from a "death march."

Another body of work – built around task structure matrices - explored the strong link between unanticipated rework and coupling in technical designs. Because the link between task dependencies in a development project and coupling in a design is very strong, rework on a design project is highly related to complexity in a product's architecture. This body of work tells us that dependencies that are unmanaged, unanticipated, cyclical, or architecture spanning can result in project overruns, organizational dysfunction, and failure to converge on a workable design. [5, 30-36]

3.3.2 Uncontrolled Complexity in Operational Systems

Complexity has been responsible for a variety of high-profile system failures and accidents that have lead to loss of life. [18, 37] Since the advent of complicated systems, we have been forced make ourselves content with the fact that design *processes* must sometimes be distributed and self-organizing [38-41]. Complexity-based methods are used in system analysis. [42-49] Successful designs must also be structured in such a way that they can evolve over time in response to learning, new requirements, and new opportunities. [48, 50, 51] Designers still have little desire to be surprised by the behavior of large systems that are in operation - when this happens, the results are generally undesirable.

A major goal of a designer is to manage *structural complexity* in a design so as to keep the *dynamic and emergent* complexity of a system in operation well understood and controlled.[3] This is because accidents are often caused by unanticipated interactions *between* parts. [18, 19] For example, the Tacoma Narrows bridge collapse in 1940 was caused by harmonic properties of the bridge as a whole. This scenario was hardly considered by its designers. More recently, a cascading power failure in India affected over a half billion people in July of 2012 [54]. While the exact cause is presently unclear, what is evident is that the power distribution network was structured in such a way that a positive feedback loop could amplify a local problem and cripple an entire country. These types of failures are pernicious because they result from the structure of the system as a whole that results in insufficiently constrained behavior. When *emergent* properties of the system as a whole are unanticipated, they often cause *emergencies*.

---

[3] The system architecture community focuses more on structural complexity while the system dynamics community [37, 52, 53] focuses more on behavioral complexity (or dynamic complexity), both agree that the structure of a system is a key driver of its long term behavior and dynamic characteristics. Both communities view structures and the resulting dynamic interactions as directly responsible for how a system performs during periods of stability and how likely various catastrophic events might be.

3.4 Architecture Controls Complexity

The fact that modern system are so complicated that the design itself must be decomposed and distributed, and the idea that modern systems are so complex that unexpected behaviors can emerge (often due to interactions between separately designed components) leads to a new role for a design professional known as a system architect. The role of this person is to analyze stakeholders, study their needs, and serve as their representative throughout the design process. The architect must use this information to devise a concept for a complex system and ensure that when built, it functions appropriately. In 2004, the Massachusetts Institute of Technology 'ESD Architecture Committee' [55] said that "[m]an-made... systems are intended to have certain primary functions, plus other properties that we call "ilities:" durability, maintainability, flexibility, and so on... The primary functions have immediate value while the ilities tend to have life-cycle value.... Complex systems have behaviors and properties that no subset of their elements have... Some of these are deliberately sought as the product of methodical design activity... While achieving these behaviors, the designers often accept certain undesirable ... side effects... In addition, systems have unanticipated behaviors commonly called emergent. Emergent behaviors may turn out to be desirable in retrospect, or they may be undesirable." The architect must design a system that can function properly, appropriately control and channel its complexity, and make hard tradeoffs. The architect must identify important system-level properties that must be managed centrally, decompose the design into manageable chunks, create design rules [5] and promote design patterns [56] which allow engineers to manage complexity within and between their subsystem boundaries. Up-front choices that an architect makes constrain the design space, but also reduce ambiguity and reduce the time that will be required to converge to a workable system if done properly. The role of the architect is to identify inherent tensions in the space of possible designs that will lead to chaos if left unmanaged, manage those choices centrally, and leave teams free to innovate independently within the necessary constraints. Important design rules may seem sub-optimal at the local level. In such cases, the role of the architect is to represent the global perspective. The architect must also represent the long-term perspective by carefully considering total lifecycle costs because maintenance costs sometimes exceed the cost

of development by a ratio of ten to one. [57] Pimmler and Eppinger say "[t]he choice of product architecture has broad implications for product performance, product change, product variety, and manufacturability." [58] Brooks says that establishing a common vision around a system plan with "conceptual integrity" is of utmost importance. [59, 60] Crawley says that of all activities that go on during the creation of a system, architecture "has some of the greatest impact on eventual success." [1]

The architect must "design the design." [59, 60]Architects decompose a design by allocating similar functionality to modules with high cohesion, creating controlled interfaces isolating a module's internals from its external environment, devising shared utilities for common use, making communication protocols clear, and arranging modules into a hierarchies and layers. The system must be structured in such a way that design teams can operate independently much of the time, know when coordination is required, and be able to coordinate effectively when necessary. The overall structure should be set up in such a way that individual teams evolving separate chunks rarely cause unwanted side-effects elsewhere, designers can recognize threats when they do occur, and the overall conceptual integrity of the system is maintained. All of this involves a number of hard choices. The architect must contend with multiple competing ways of decomposing a system into hierarchical structures [7] and competing criteria for determining which functionality should be clustered in each module. [58] The architect must also determine how big each module should be and how interfaces between them should be designed. These choices will have a profound impact on how complex different portions of a system will be.

Successful architecture process will yield an abstract description of a large system that, if designed and built, would function appropriately, control complexity, and have the ability to scale and evolve over time. Ulrich and Eppinger define this as "the scheme by which ... decomposed elements are arranged in chunks." [61] Ulrich chooses to "define product architecture more precisely as: (1) the arrangement of *functional elements;* (2) the mapping from *functional elements* to *physical components;* (3) the specification of the *interfaces* among interacting physical components." [62] Crawley defines architecture as the "The embodiment of concept, and the allocation of physical/informational function to elements of form, and definition of interfaces among the

elements and with the surrounding context." [1] Moses says that the architecture of a system "is a skeleton that connects the components of the system. A skeleton does not fully describe the human body or an engineering system, but it is a necessary and crucial part of the system's description." [63] When analyzing systems, Moses sees value in creating architectural descriptions emphasizing *structural complexity* (a description of parts and their connections). He finds it useful "to study the relationships between certain generic architectures (e.g., tree structures, layered structures, networks), their structural complexity and the non-traditional properties of systems, such as flexibility." [63] Ultimately, the high level structural patterns that are built into a design from its inception will have a profound effect on the evolutionary trajectory and lifetime cost of a complex system.

3.5 Important Architectural Patterns

Certain well-known patterns are employed by man and nature to keep complexity under control even as systems grow. These patterns include hierarchy, modularity, and abstraction layering. Technical architectures in which these patterns are judiciously applied tend to be of higher quality, safer, and benefit from other "ilities" over the course of their life cycles.

3.5.1 The Benefits of Hierarchy

Formally, the term hierarchy denotes any directed acyclic graph (DAG). While, it may not contain cycles, it can contain multiple source and sync nodes, and can both diverge and converge. A tree is a very common type of hierarchy that fans out from a single root (or controller node) and never converges. A layered system is a different kind of hierarchy. Hierarchies are pervasive organizing patterns in many large real-world systems because they endow systems with a variety of useful properties. Herbert Simon tells us that "[h]ierarchy ... is one of the central structural schemes that the architect of complexity uses." [51] In The Sciences of the Artificial, he says "complex systems might be expected to be constructed in a hierarchy of levels, or in a boxes-within-boxes form. The basic ideas it that several components in any complex system will perform particular sub-functions that contribute to the over-all function... To design such a complex

structure, one powerful technique is to discover viable ways of decomposing it to semi-independent components corresponding to its many functional parts. The design of each component can then be carried out with some degree of independence of the design of others, since each will affect the others largely through its function and independently of the detail of the mechanisms that accomplish the function." [48, 51] Leveson says "a general model of complex systems can be expressed in terms of a *hierarchy* of levels of organization, each more complex than the one below, where a level is characterized by having *emergent* properties. Emergent properties do not exist at lower levels; they are meaningless in the language appropriate to those levels... Thus, the operation of the processes at the lower levels of the hierarchy result in a higher level of complexity... [A]t a given level of complexity, some properties characteristic of that level (emergent at that level) are irreducible." [19] Simon says that hierarchical systems are commonly found in the natural world because nearly decomposable hierarchies with stable subsystems (or intermediate forms) enable evolutionary processes, allowing highly ordered systems to grow, acquire new capabilities, and adapt. Simon notes that hierarchical patterns in systems manage complexity because they "have a high degree of redundancy, hence can often be described in economical terms." Hierarchical organization assist designers by reducing the cognitive burden placed on the human mind when examining a system from any one vantage point. Hierarchies also facilitate top-down control and the imposition of safety constraints [19, 64]. They are useful structures for classifying, storing, and searching for information. [65] Finally, the requirement that a hierarchy contains no cyclic connections reduces the possibility that feedback loops will be formed between widely separated components. These feedback loops, or cycles, can hindering change [31] or lead to unintended non-linear dynamic behavior. [53]

3.5.2 The Benefits of Modularity

Modular is a term used to describe architectures composed of distinct modules – semi-autonomous structures with formal boundaries that separate their internal environment from the outside world. Robust modules have the property of "homeostasis" – their internal functioning is not easily disrupted by fluctuations in the external environment. [49] Modular systems contain many independent components, each of which can

change or evolve separately with minimal impact on each other or on the system as a whole. [66] Modules *hide information* in the sense that the use of one only requires a client to understand its interface, not its complex internals. Modularity was recognized as a critical means of controlling software complexity as early as the 1960s. [67] In the early 1970s, Wirth proposed a process of "stepwise refinement" for designing software in a manner that would result in modularized code. [68] Parnas, whose work anticipated object-orientation, followed up by contributing reasonable advice on what criterion should be used when decomposing a software design into modules. [69, 70] As computer science developed, increasingly sophisticated types of modules were invented including an extreme form, known as Object-Orientation, which combined data-type abstraction and access control. [6, 71, 72] Modularity is similarly important in physical product design. Ulrich's classic paper contrasted modularity with *integrality* and discussed multiple types of modularity including slot modularity, bus modularity, and sectional modularity. He described the relative advantages of each type and illustrated his case by showing alternative design concepts for a trailer, a personal computer, and an office desk. [62] Ulrich defined a perfectly modular product as one in which every internal function is performed by a distinct part. He said "[a]t one extreme, modular products allow each functional element of the product to be changed independently by changing only the corresponding component. At the other extreme fully integral products require changes to every component to effect change in any single functional element. The architecture of a product is therefore closely linked to the ease with which change to a product can be implemented." Ulrich noted that modular product architectures are more flexible, adaptable, and manufacturable. Baldwin and Clark tell us why modularity is important in economic terms. They say "modularity in design - an observable property of design and design processes – dramatically alters the mechanism by which designs can be changed. A modular design in effect creates a new set of *modular operators,* which open up new pathways of development for the design as a whole." [5] When designers are working inside a modular technical design, they can independently transform modules in six important ways. Modules in a system can be *split, substituted, augmented, excluded, inverted, and ported.* Baldwin and Clark demonstrate through computer simulation how the option to independently modify subcomponents within a modular system accelerates innovation by creating "real options" in the system's

design that behave similarly to *options* in the world of finance. Essentially, driving modularity into an integral architecture can transform a single large investment into a much more valuable basket of small investments. Baldwin and Clark tell us that modular designs can "evolve via a decentralized search by many designers for valuable options," each experiment conducted independently within a bounded span of responsibility and control. In general, modularity is a useful attribute, but not all systems can be modular. Whitney points out that many systems cannot achieve the level of modularity often achievable in software and digital systems due to fundamental power constraints or because multiple types of system-spanning relationships (power, informational, electrical, physical, force translation, etc.) each suggest alternative modularizations, none of which are fully satisfactory. [7] Whitney also makes note of the current movement towards integrality in computer design brought on by the need to dissipate heat in laptops. Furthermore, modularity is not free. Modular designs contain overhead in the form of "design rules" that add cost to the front-end of a design process and potential recurring costs in the form of ongoing performance limitations. [5] This may be a worthwhile investment in software systems with volatile codebases or complicated problem domains. In systems that are less volatile or complicated, the investment in modularity may not pay off. [73]

3.5.3 The Benefits of Abstraction Layers and Platforms

Layers combine the notion of hierarchy and modularity in a manner that serves to contain complexity and endow a system with a variety of beneficial properties. Layers in systems provide services to components above them while relying on services provided by those below. They combine the notion of directionality found in hierarchies with the notion of information hiding found in modules. Conceptual layers in a design are sometimes called *abstractions*. In other contexts, technology layers are called *platforms*. Although layers are themselves inflexible, Moses says that layered structures can make an overall system more flexible. [74] Baldwin and Woodward tell us that platforms form rigid structures in an architecture that create stable interfaces upon which modules can rapidly evolve. [75] Moses says, "Layered systems are common in large scale hardware/software systems. For example, a personal computer will have a layer for the microprocessor, several layers for the operating system including a user interface layer, possible layers for a database system,

and additional layers for application software." [74] Layering hides information in a stronger manner than modularity does because it partitions a complex network of components into two distinct regions that may be considered independently. In addition to hiding details, abstraction layers may embody new higher-level concepts by aggregating diverse facilities into a useful coherent whole.[4] Abstraction layers can also partition systems by engineering discipline or be responsible for defining the boundaries between disciplines. The transistor, for instance, creates a useful barrier that allows electrical engineers to study quantum mechanics while computer engineers can study Boolean logic. Krueger says that the creation of new abstraction layers is the primary means by which large-scale reuse is achieved in software. [76] Functionality that is repeatedly found to be useful ends up buried beneath layers of abstract symbols in subsequent generations. Daniel Jackson says that the creation of conceptual abstractions is central to the design of software:

> "Software is built on abstractions. Pick the right ones, and programming will flow naturally from design; modules will have small and simple interfaces; and new functionality will more likely fit in without extensive reorganization. Pick the wrong ones, and programming will be a series of nasty surprises: interfaces will become baroque and clumsy as they are forced to accommodate unanticipated interactions, and even the simplest of changes will be hard to make. No amount of refactoring, bar starting again from scratch, can rescue a system built on flawed concepts. Abstractions matter to users too. Novice users want programs whose abstractions are simple and easy to understand; experts want abstractions that are robust and general enough to be combined in new says. When good abstractions are missing from the design, or erode as the system evolves, the resulting program grows barnacles of complexity. The user is then forced to master a mass of spurious details, to develop workarounds, and to accept frequent, inexplicable failures. The core of software development, therefore, is the

[4] For instance computer operating systems present an abstraction known as the *single-threaded process* - which serves as a container for all applications running on the computer. Under typical circumstances, this container allows each program to behave as if it were the only program running on a perfectly deterministic machine, even though it is continually contending for resources on a machine whose behavior is far from predictable or repeatable in practice.

design of abstractions. An abstraction ... is a structure, pure and simple – an idea reduced to its essential form. [77]

3.6 Network Representations of Important Architectural Patterns

We have created the following figures to depict hierarchy, modularity, and layering as embodied in different types of directed network graphs. Each network is shown using two representations. Pictures on the left are traditional network views with letters as nodes and directed arrows as arcs. Pictures on the right are square matrix views with nodes contained in an ordered list, and arcs represented as dots in the square matrix at ("from node", "to node.") This ordered matrix representation (known as a Design Structure Matrix or DSM) is widely used in engineering design circles. [30, 78]

Figure 1 and Figure 2 show a *hierarchical tree* with node L as a root node. Trees are a common type of hierarchy. Note that in an appropriately sorted DSM, a tree-hierarchy appears as a band starting somewhere along the middle of the left hand side and moves towards the lower right corner.

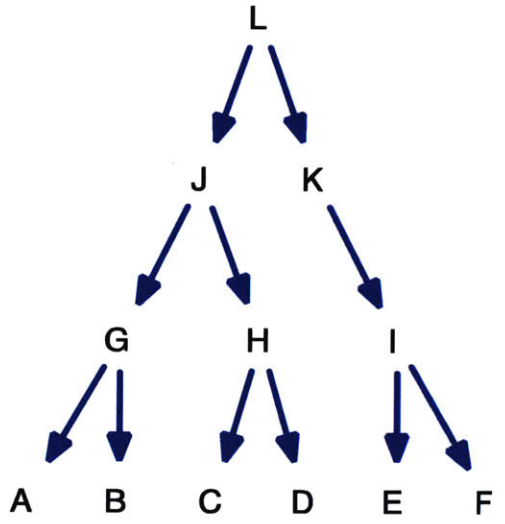

Figure 1: Hierarchical Tree as Network

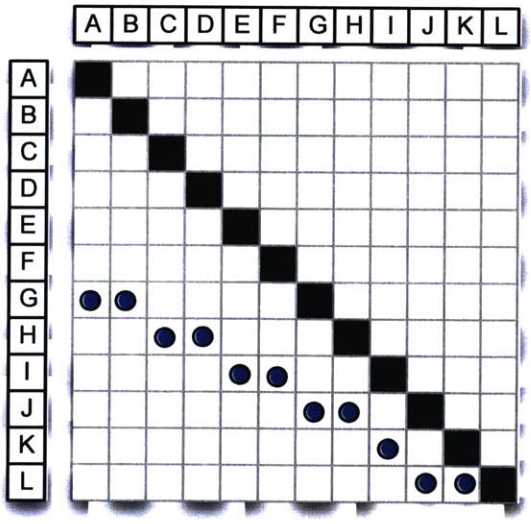

Figure 2: Hierarchical Tree as DSM

Important note about network notation: For those already familiar with DSMs, it is very important to note that the conventions used in this report are somewhat different than those used in much of the existing DSM literature. This is because DSMs originated in the hardware community, but the software community follows slightly different network drawing conventions. While the DSMs shown in this report (e.g. Figure 2) follow typical DSM conventions, the corresponsding "traditional network views" (e.g. Figure 1) have arrows pointing in the opposite direction of what is typically shown. This arises from the fact that software *call graphs* show arrows pointing from a *calling* function to one that is *called.* Figure 1 and Figure 2 could both represent a call from within function L that invokes function J. In this scenario, function J does useful work and returns the result of that work back to function L. In electro-mechanical network representations, however, arrows are typically drawn from a *provider* J to the *recipient* L.

The convention used in this thesis is consistent with the call graph convention that is often used in the software DSM literature (see MacCormack's conventions [79] for an example) and different that what is found the hardware DSM literature (see figures 1.3a and 1.3b in Eppinger's book[78].)

Figure 3: Note about DSM and Network Conventions

Figure 4 and Figure 5 show *four independent tightly coupled modules*[5]. Codependence between nodes within each module shows up as bidirectional arrows in the network view and symmetry in the DSM. Modules have high internal cohesion and low external coupling. In an appropriately sorted DSM, modules appear as distinct and identifiable blocks arranged along the diagonal. (If the nodes in the DSM were sorted differently, these modules might not be evident.)

[5] For simplicity, we have not shown links between modules. In order to be one system rather than four separate ones, these modules should be loosely connected in some way.

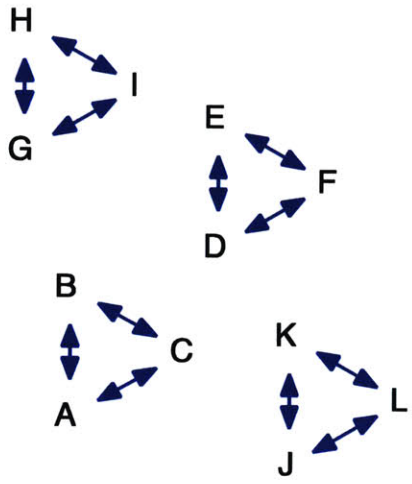

Figure 4: Modules as Network

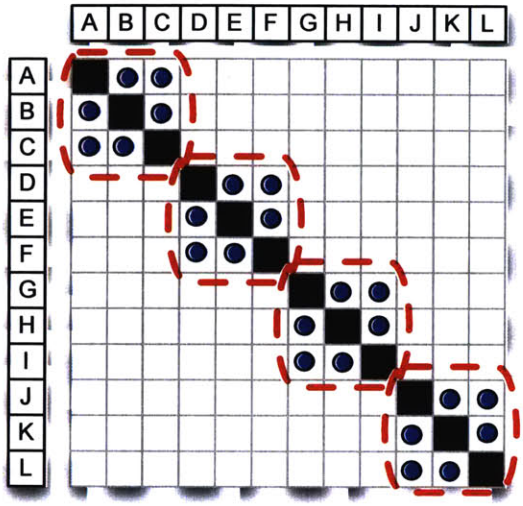

Figure 5: Modules as DSM

Figure 6 and

Figure 7 show layers, platforms, or abstractions. As can be seen in the DSM, in some senses this structure is a combination of hierarchy and modularity. Nodes in a higher layer can directly access nodes in a lower layer, but connections do not generally skip layers. (Modular access within the same layer is omitted for the sake of clarity.)

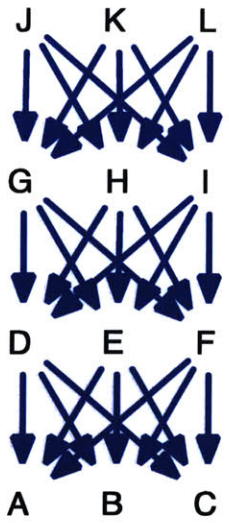

Figure 6: Layers as Network

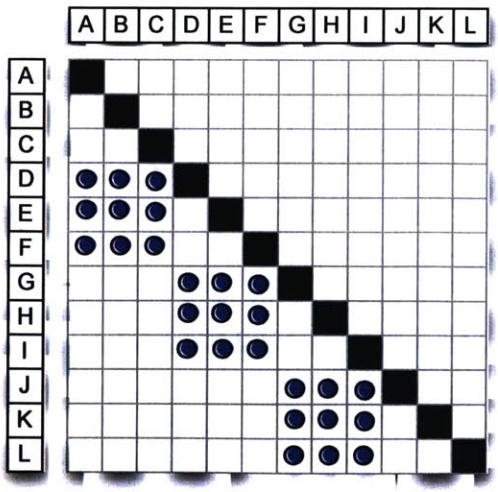

Figure 7: Layers as DSM

Figure 8 and Figure 9 show a *hierarchy of modules*. This mixed system depicts an organization of components that characterizes many complex systems. It incorporates some of the principles of large-scale system design articulated in different ways by Simon, Parnas, and others. Herbert Simon describes a nearly decomposable hierarchy of "intermediate forms." [48, 51] Simon says that natural and artificial systems are often organized in this manner because these structures provide stability and other evolutionary advantages. These intermediate forms are modules with high internal cohesion and low external coupling, thereby endowing a system with what David Parnas called "information hiding" properties [66, 69]

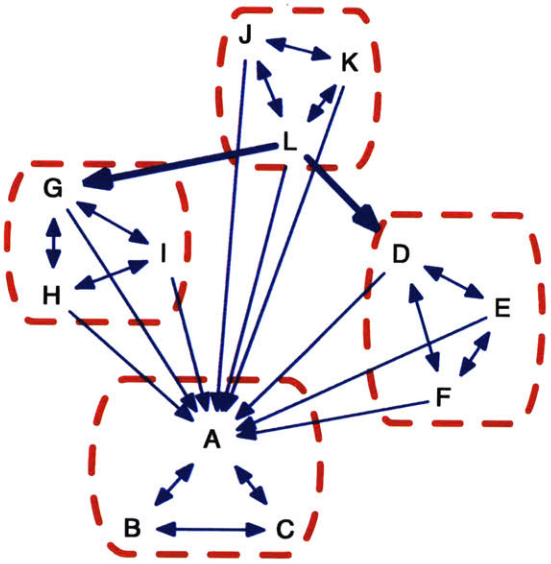

Figure 8: Hierarchy of Modules as Network

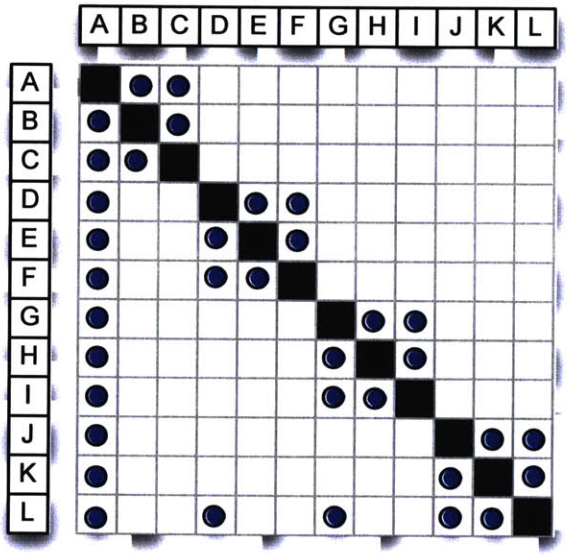

Figure 9: Hierarchy of Modules as DSM

3.7 Networks in System Architecture

Because complexity controlling features in system architectures – notably hierarchies, modules, and layers – can all be thought of as types of networks, many in the systems community have begun to reason about products as networks of interconnected components. Strogatz says that "from the perspective of nonlinear dynamics, we would also like to understand how an enormous network of interacting dynamical systems — be they neurons, power stations or lasers — will behave collectively, given their individual dynamics and coupling architecture." [80] System architects have begun to lean heavily on the language of networks when

describing engineering systems and their properties. The recent report about system architecture released by MIT [55] employs the term *network* 40 times. This makes sense because many important engineering systems, including the Internet, are literally networks. [81, 82] In addition, the report contains references to *types* of networks including *hierarchy* which appears 11 times, *tree* which appears 9 times, *layer* which appears 11 times, *module* which appears 59 times, and *hub* which appears 6 times. The MIT Architecture committee tells us that "[s]ome architectures can be represented fairly completely as networks. In such cases, a lot can be determined about their behavior from graph theory." After all, if architecture is an "abstract descriptions of entities... and [their] relationships", then a network - defined by its nodes and arcs, is a natural corollary. In addition, the group argues persuasively that many of the properties we care to measure and manage over a system's lifecycle including "robustness, adaptability, flexibility, safety, and scalability... might be measured using network models of a particular architecture." [55]

3.7.1 Design Structure Matrix Methods for Complex Systems

The Design Structure Matrix (DSM) is a square matrix network representation that has been used to capture project dependencies[6] and coupling in a product architecture. DSMs represent project tasks or system elements as network nodes and represent task dependencies or coupling between parts as arcs. Nodes are represented as an ordered list of length N. Arcs are stored in a square matrix with size $<N, N>$. An arc is added to the network by inserting an entry at a point *<from node, to node>*, thereby creating an association between two nodes in the ordered list. [30, 31, 58, 78, 83] This representation makes certain features visible that cannot be seen in a traditional "node and arcs" view. Eppinger and Browning call the DSM "a network modeling tool used to represent the elements comprising a system and their interactions, thereby highlighting the system's architecture." [78] Unlike typical pictures of networks, DSMs allow engineers to see the topography and density of interconnections in and between different parts of a system. To illustrate, Baldwin and Clark's figure of a laptop shown in Figure 10 [5] highlights the interconnections between different parts,

[6] When used to represent project dependencies, these square matrices might be called 'Task Structure Matrices' (TSMs) rather than DSMs.

shows how those parts are clustered into modules, and points out circular interactions between parts in different modules that should ideally prompt cross-team coordination.

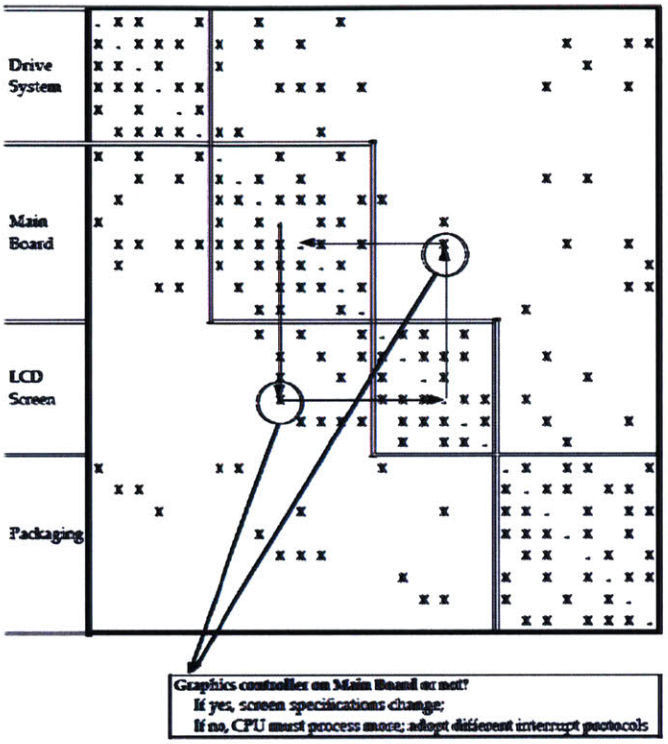

Figure 10: DSM of a Laptop Computer (From Baldwin & Clark)

The DSM community was not the first to use matrix-based representations of technical networks. (Prior examples of square matrices depicting modularity turn up in Bergland's and Simon's work for instance. [51, 84]) The DSM community has, however, pioneered the application of square matrices as a practical tool in large-scale systems analysis, engineering design, and project management. Browning tells us that the "[s]ystems engineering of products, processes, and organizations requires tools and techniques for system decomposition and integration. A design structure matrix (DSM) provides a simple, compact, and visual representation of a complex system that supports innovative solutions to decomposition and integration problems. The advantages of DSMs *vis-à-vis* alternative system representation and analysis techniques have led to their increasing use in a variety of contexts, including product development, project planning, project management, systems engineering, and organization design." [85] Eppinger and Browning survey the field in

Design Structure Matrix Methods and Application. [78] This book gives many examples of their application including a Pratt & Whitney jet engine [86-88], Xerox digital printing technology [89], and the Mozilla internet browser. [8]

3.7.2 DSMs and System Structure

Although DSMs are now commonly used to represent product architectures, they were first used in project management to represent tasks and their dependencies. [30]. By using a network representation with the ability to capture feedback, Steward overcame an important limitation of the commonly used 'Critical Path Method.' [90] These "task structure matrices" (TSMs) made it easier to explore rework and design iteration. [31] A number of subsequent innovations allowed TSMs to be used in project estimation. [32-34]

The use of DSMs as tools for analyzing the interconnectivity patterns inside a technical architecture began with Pimmler and Eppinger. They devised a method using the DSM to identify and evaluate alternative modularizations. [58] Their method involved:

- Identifying parts, their connections, and the functions that those parts perform.
- Documenting the various physical and functional interconnections in a DSM.
- Ranking those interconnections based on their desirability or undesirability.
- Clustering or "chunking" the connections or interactions into different possible groupings or "modularizations" based on the strength of interactions between parts.

Pimmler and Eppinger focused on generating many alternative architectures early in the design process so that various schemes for decomposing design tasks, defining modular boundaries, and allocating design tasks to teams could be considered early in the process.

In Design Rules: The Power of Modularity, Baldwin and Clark use DSMs to illustrate the means by which modularity creates value for innovators. [5] This book illustrates its case using many DSMs starting with a coffee mug and moving up to examples including a graphics controller, a motherboard, and a laptop

computer. Baldwin and Clark use these DSMs to reason about the economic value of modularity in a design and illustrate a means by which modular designs create that value. Baldwin and Clark argue persuasively that the modular design of the IBM/360 [91] fundamentally transformed not only computer architecture, but also the structure and scale of the hardware and software industry by increasing the return on investment of R&D activities.[7] Others have built on this line of research by exploring how network coupling impacts the value of an architecture [92] and the rate at which systems can improve. [93]

Novak and Eppinger explore the relationship between product complexity and vertical integration. [94] They define the complexity of a system as a function of its component count, component interaction count, and novelty. Their complexity calculations include coupling because integral systems are more likely to experience change propagation and include novelty because new systems likely contain design interactions that have not yet been discovered. They find "that in-house production is more attractive when product complexity is high, as firms seek to capture the benefits of their investment in the skills needed to coordinate development of complex designs." Their findings seem consistent with Baldwin and Clark's observation that the modularity built into the IBM/360 reduced architectural complexity to such an extent that it caused the vertically integrated industry to disintegrate and restructure in a manner consistent with the modular [5] and layer/platform [75] structures we observe in computers and software today. (IBM's innovation was enormously profitable in the short term, but opened the door for new competitors and ultimately caused them to lose control of some of the most valuable subcomponents within the architecture.) Novak and Eppinger's observations are also consistent with a study describing the re-integration of the bicycle drive-train industry in response to a novel and highly integral innovation [4].

3.8 Exploring the Network Structure of a Software System

---

[7] Prior to the development of the IBM/360 the computer industry was vertically integrated with each manufacturer creating highly integral designs. Every generation of computer defined a new set of interfaces, operating systems, and programming languages - each incompatible with the last.

Researchers have recently begun using networks to explore the architecture of large software systems. Valverde found that software dependency networks have "small world" characteristics and are "scale-free" for nodal in-degree (but not out-degree). [95, 96] Sullivan et al. applied design rules theory to Parnas' classic modularity example to illustrate the value of information hiding. [97] Settas and Stameleos used DSMs to explore software "anti-patterns." [98]. MacCormack, Rusnak, and Baldwin conducted multiple studies exploring large-scale system evolution and modularity. [8, 9, 79, 99]

When researchers investigate software architectures empirically, they generally construct networks by using "call graph extractors" to pull entities and relationships out of software source-code. Dependencies are then fed into network or matrix manipulation software for analysis. [100, 101] Call graph extractors automate the process mining software dependencies, making the process much more efficient than manual construction and eliminating human error from the process. Once software DSMs are extracted, they can be analyzed alongside the contents of other databases that store information about the development process such as "bug-tracking" and "version control" systems. Combining information from these data-sources makes longitudinal analysis possible by giving researchers the ability to "track the evolution of a design over time." [8]

MacCormack et al. construct DSMs by using source-code files as network nodes and inserting arcs between them when software dependencies (such as *function calls)* span file boundaries. Once a software DSM is constructed, MacCormack computes "visibility" metrics (based on transitive closure) designed to reflect the modularity of architectures and the coupling of individual components. These methodological innovations allowed MacCormack, Rusnak, and Baldwin to empirically measure the level of modularity in a large system, to compare levels of modularity over different versions of the same system, and to compare modularity across systems. Because MacCormack's metrics and techniques are used in this study, they will be described in detail in the "Methods" section of this report rather than being elaborated upon here.

MacCormack, Baldwin, and Rusnak used software DSM methods to explore the evolution of the Mozilla Internet browser, finding a substantial decrease in coupling after a refactoring effort that was undertaken to

make its source code more maintainable. [8] They expanded upon these techniques in a variety of other studies. In one, they explored the structure of matched pairs of comparably sized open-source and proprietary systems that performed identical tasks, finding that open-source systems were consistently more modular than their proprietary counterparts. [99] In another study, they tracked changes over six releases of a software system, finding that more tightly coupled components are "harder to kill" (more likely to persist from version to version), "harder to maintain," and "harder to augment." [9] In another study, they extracted the architecture of over 1000 software releases of 19 applications to survey system evolution. They found that a strong majority of systems (approximately 80% in their sample) have a "core-periphery" structure – characterized by a network in which some files formed a tightly and cyclically coupled grouping – or core - while others sat on the periphery. Some cores remained stable while structure grew around them, while others grew with the size of the system. [79] MacCormack also used software DSMs to study the duality between organizational and product architecture. [99] Other studies have followed in this research vein. Lamantia et al. conducted two case studies finding evidence that software modularity was valuable because it allowed different regions of code to evolve at different rates and allowed firms to substitute "at risk" modules with minimal impact elsewhere. [102, 103] Akaikine explored the impact that the complete rewrite of a major commercial software product had on maintenance costs. He found both a substantial decrease in coupling and a substantial improvement in the performance of the maintenance organization. [104] Sosa, Browning, and Mihm explored the dynamic evolution of the Ant system architecture. [105] They describe an experimental phase before software architecture settled, the emergence of a dominant configuration, and the appearance of limits to growth and complexity saturation "which might call for a refactoring of parts of (or the entire) product architecture."

3.9 Some Definitions of Complexity

Summers and Shah (both mechanical engineers) usefully summarize a few of the many perspectives on system complexity in the following quoted bullets [106]:

- "The whole exceeds the sum of the parts. Complexity should include how the parts are assembled into the whole. This coupling between the parts leads to the view that complexity does not include a simple additive property from the components to the assembly, but rather there are emergent properties that are only found collectively in the assembly. This view is predominantly for studying the complexity of the design product."
- "Complexity is a measure of the minimum amount of information (bits) required to describe in the given representation. The amount of information required minimally to describe something in a specific representation suggests a lower bound of complexity for that which is described. Any information in excess of this is superfluous or redundant. This view of complexity ignores the possible interconnectedness of the information and how hard or difficult it is to parse the minimal representation."
- "Complexity is the amount of effort required to manufacture or design. This view of complexity looks at complexity from how difficult it is to solve a problem, be that manufacture or design. As more effort is required, the complexity increases. This suggests that the complexity of a design product or design problem are related to the design process."
- "Complexity is a measure of the tasks required to achieve some function (or components). This view of complexity is equivalent to algorithmic complexity. A complexity measure should be developed with respect to the number of functions and the level at which they are found that are required for satisfying the design requirements."
- "Complexity is a measure of the phase change between order and randomness (entropy). This view of complexity accounts only for the information compaction. It does not address the difficulty associated with reconstructing the minimal chain of information into the original string. Consider the complexity (entropy) associated with a random string of letters and the complexity (entropy) associated with this paper. The two strings may be of equivalent size, thus yielding equivalent levels of complexity. However, the effort required to produce the paper extended across years, while a four line random number generator may be used to create the random string."

- "Complexity is a measure of the number of basic operations required for solving a problem. The complexity of the problem is associated with the computational complexity of the best-known algorithm for solving the problem. This computational complexity is proportional to the time it takes for an implemented algorithm to solve the design problem given that the basic operations are roughly equivalent in time."

These varied perspectives (or others like them) often form theoretical foundations on which many existing complexity metrics rest.

3.10 Measures of Software Complexity

In response to the general consensus that complexity in its various forms had a tremendous influence on quality and cost, practitioners in the 1970s began to try to devise software complexity metrics. This effort has continued through the present day. While each metric paints only an incomplete picture, only capturing some important features of the code while ignoring others, each attempts to measure some important concept related to program or psychological complexity. The oldest and most commonly used metrics are *component* based. They measure some property internal to identifiable elements within a software system. Agresti says that "Most of this work... has been focused on a single characteristic or oriented toward program modules rather than large software systems or subsystems as units of observation." [107] More recently, some less commonly used *architectural* or *structural* metrics have been devised to measure the interrelationships between those elements. These structural measures look at coupling, cohesion, modularity, interfaces, cycles and other system-level attributes in a software system. Some component-based and structural metrics devised over the years have been found to relate to quality, maintainability, and other desirable project and product attributes.

3.10.1 Component-Based Complexity Metrics

The simplest (but also crudest) measure of complexity employed is the raw count of lines of code (LOC) in a program, file, function, class, method, or other programming construct. LOC is a reasonable metric for use in project estimation because all else being equal, it will be strongly correlated with to the effort required to create a working piece of software. LOC based metrics have earned a bad reputation in some quarters, not

because they are not useful, but because their use creates perverse incentives when applied inappropriately for rewarding individual productivity. The software engineering community has generally abandoned this practice for the simple reason that the metric is easily gamed in a way that leads to bad programs. LOC metrics are still valuable for estimation purposes, and for estimating productivity in a statistical sense, so long as the metric is not simultaneously used to judge individuals. Some studies have found a relationship between module size (in LOC) and defect density, possibly because more complex code requires more room or because larger modules are more complicated. [108]

Two early metrics that received wide attention were those devised by Halstead and McCabe. Halstead proposed a set of interrelated metrics, also based on a static analysis of code that measured a program's operands and operators and derived measures of its "volume," difficulty to understand, and the effort required to write it. [109] Unfortunately, Kan says that "empirical studies provide little support to the [Halstead] equations." [108]

McCabe proposed a "cyclomatic" complexity metric that has proved more successful. McCabe assigns a number to a "structured program" or block of executable code based on a static analysis of the number linearly independent execution paths that can be followed as a program executes. In modern programming languages, McCabe scores typically apply to procedures (called functions in C or C++) or class methods. Alternative paths through a procedure result from conditional branching statements (*if statement, switch/case statement, while loops, etc.).* [110] Gill and Kemerer provide the following four-step recipe for computing the original version[8] of McCabe's metric. [111]. We will quote from their definition:

1. Increment one for every IF, CASE or other alternate execution construct
2. Increment one for every Iterative DO, DO-WHILE or other repetitive construct
3. Add two less than the number of logical alternatives in a CASE

---

[8] A common variant (the one used as a control variable in this study) excludes switch/case statements from consideration in the McCabe score. This is often referred to as "Modified McCabe cyclomatic complexity."

4. Add one for each logical operator (AND, OR) in an IF

McCabe asserted that his number could be used to estimate the effort required in test coverage. He also suggested that cyclomatic complexity for functions or methods should be kept below the value 10 so that they remain understandable and testable. A classification scheme has been devised to bin functions into four general types based on their McCabe scores. [112] Functions with scores of:

- 1-10 have "low" complexity
- 11-20 have "medium" complexity
- 21-50 have "high" complexity
- 51 and above are considered "untestable"

McCabe's metric has been positively related to defect density [113-115] and the productivity of developers doing maintenance on previously shipped code. [111] Many firms now use McCabe's scores as a means of identifying problematic code.

A variety of similar syntactic metrics have emerged with the advent of new languages and methodologies. [116, 117]. Many studies have been conducted to test the relationship between complexity metrics and quality, maintainability, and other non-functional attributes of code. Some complexity metrics have been found to correlate with quality [118, 119] or maintenance effort [120, 121] Due to the realization that LOC, McCabe's, Halstead's, and other syntactic constructs each might capture certain limited aspects of code complexity, composite metrics have been devised as well. Card and Agresti found that a metric combining information about control flow (as measured by fan-out) and information flow (as measured by the size of the interface as measured by I/O variables) correlated strongly with defect density. [119, 122] Another index was created that has been shown to be a good indicator of software maintainability. [123] A variety of other indices have been created, some of which have been calibrated to predict various *ilities*. Some of these indices are expressed as complex polynomial equations and can serve as indicators pointing developers to modules that may be in need of attention.

3.10.2 Structural Complexity Metrics

In addition to syntactical metrics, a number of lesser-used structural metrics exist, some of which relate to levels of coupling and cohesion. [124, 125] Two metrics of note are "fan-in" and "fan-out" [126] which are measures of the *direct* structural connectivity between components. Fan-in counts the number of components that depend upon a component, while Fan-out counts the number of components that are depended upon by a component. When looking at the degree distributions of software networks, Valverde has found that software dependency networks have "small world" characteristics and are "scale-free" (obeying a Power-law distribution) for nodal in-degree (but not out-degree). Kan cites studies indicating that Fan-out correlates with defects (because it is a measure of the number of upstream components) but suggesting that Fan-in does not. [108]

MacCormack, Baldwin, and Rusnak devised a procedure for classifying software components (such as files) based on their level of direct and indirect coupling with the rest of a software system. [8, 9] Their directed-network-based classification scheme identifies "core" nodes - those that are contained within the largest network cycle. In some senses, this classification scheme can differentiate between software source code files whose interactions with the rest of the system are mediated through hierarchical or modular constructs and those that are more tightly coupled to disparate parts of the system. (This metric only captures some important properties related to hierarchy and modularity.) Because the MacCormack approach is used to operationalize the notion of "architectural complexity" in this study, we will give a detailed description later in this report.

3.10.3 Criticism of Complexity Metrics

Complexity metrics have generated a large amount of debate for a number of reasons. Some have highlighted the limitations of these types of metrics. Curtis et al. conducted experiments to test their relationship with "psychological complexity." [22] Under a limited set of experimental conditions, Curtis found that program size, cyclomatic complexity, and Halstead metrics correlated with programmer accuracy and time to task

completion. His results held only when the programs were unstructured, contained no comments, and the developers being tested were inexperienced, however. Curtis concluded that things like good variable names, comments, and indentation play a strong role in reducing psychological complexity even if they are not considered by the McCabe and Halstead metrics. Furthermore, a lack of correlation for experienced developers led the researchers to theorize that senior developers have mental schemas that they apply during program analysis in the same way a chess-master evaluates a board. Because these developers 'see' higher-level structures and behaviors rather than individual symbols in the code, measures that are easily calculated from small pieces of code might have limited explanatory power. Curtis said that to be effective, metrics should consider aspects of psychological complexity in their formulation. Another limitation of metrics is that there are many important desirable properties of software that nobody knows how to measure. Furthermore, we lack a clear explanation for why some complexity metrics correlate with quality attributes even when they do. This is especially true for composite metrics. Because many metrics lack an underlying theory of programming or psychology, many believe that they are only crude indicators that are often useless due to a lack of robustness, normativeness, or prescriptiveness. [108, 127]. Another important limitation of those traditional metrics is that they are local in nature. Due to computational limitations, most component-based metrics quantify properties of a single function, method, class, or source file in isolation without considering the complexity of the relationships between those isolated units. These metrics fail to capture the complexity and complicatedness created by inter-component coupling patterns – both of which are important factors in software development and maintenance.

3.11 The Difficulty of Placing a Value on Redesign

In an ideal world, engineers evaluating system designs would judge alternatives by doing a full accounting of the long-term financial and other stakeholder value that would be generated by each. This would require the designer to estimate the benefits of system functionality and performance characteristics as well as the costs, including the cost of complexity, within the design. Multiple objectives could be defined, and the net present value (NPV) of design alternatives could be estimated, reasoned about, and optimized. Some promising

efforts have been made to apply financial valuation techniques to software project decision-making. [128] Unfortunately, the use of objectively rational decision-making models is often unrealistic or impractical under most circumstances. As a result, normal decision-making biases can tip scales in favor of sub-optimal choices because some costs and benefits are salient to designers and managers while others are not. While the benefit of additional system performance might be immediate, clear, and calculable, some costs of complexity are hard to understand, long-term in nature, and may be borne by someone other than the original system creator. Under these circumstances, an organization may be biased against incorporating complexity-controlling (but more expensive or performance limiting) structures in an initial design even if the NPV of this alternative were objectively superior.

Similar issues arise during ongoing maintenance of complex systems, many of which are incredibly long-lived. Whitney et al. tell us that "[s]ometimes, architectures are designed or evolve to minimize complexity, but, as systems grow in size, a point is usually reached where a system's complexity becomes overwhelming, imposing a limit on what one can do to operate the system, predict its behavior, or change it." [55] Scaling limits, [105] changing requirements, and other stresses during maintenance can cause systems to "decay" over time. Natural entropy can erode modular boundaries or connect components whose interactions were previously mediated through hierarchies and layering schemes. These systems will become brittle and increase the likelihood of unwanted change propagations [129, 130] A system may become less coherent, defects might become more frequent, and the productivity of engineers maintaining it might decline. When this occurs, some engineers will call for a design overhaul, known as a "refactoring" [131] in the software community, to improve the situation going forward. Unfortunately, managers dividing scarce resources between developing new features, defects correction, and redesign will tend to favor the first two options. Although those advocating redesign may intuitively understand that refactoring is in the long-term interest of the firm, they have no good way to support this intuition in a quantifiable manner. They cannot easily draw a clear line between complexity reduction today and fewer defects or improved effectiveness tomorrow.

There are other reasons for an underinvestment in redesign. The costs imposed on the development organization by problems in an existing architecture may be invisible – hidden in the form of tacit and unquestioned performance expectations, norms, processes, or routines. [132] A company may have no basis for comparison – no means of weighing the cost of maintaining their current system against some hypothetical alternative implementation. The true cost of *not* confronting complexity within an existing design might be unconsidered and unknown. Calls to refactor might only come after high-profile defects or a noticeable erosion of productivity. Under this type of threat, however, the group may feel pressure to engage in "fire fighting" to improve short-term performance or display heroics under scrutiny. Because redesign is a 'worse-before-better' proposition, it runs counter to these instincts. [133-135] Although refactoring might be incredibly valuable in the long-term, it will offer few immediate rewards and will consume the attention of many highly skilled employees. Because such an endeavor often defies short-term individual, managerial, and investor instinct, pursuing this course requires visible and sustained executive-level support.

4 Conceptual Model, Gap in the Literature, and Hypotheses

In this section, we will articulate a conceptual model linking the complexity within an architecture to specific costs incurred by an organization including quality problems, lower productivity, and higher staff turnover. We will also look at prior work that has been done to establish this link, identify gaps in that literature, and state three testable hypotheses.

4.1 Conceptual Model

Some designers of real-world complex systems may encounter persistent problems in parts of the system they interact with. Problem components might require more defect corrections. They might be rigid, inflexible, or hard to change. They might be brittle – subject to break if perturbed. They might be unstable – requiring continual change to accommodate various demands unrelated to improved functionality. They may also be incomprehensible – hard to understand and work with. Software developers working in problematic files may have trouble keeping track of how the code operates. Their productivity may slip. They may trigger more unintended side effects and may introduce more defects. They may have trouble making educated guesses about when work will be completed. Their vantage point may lead them to perceive their portion of the system as inelegant, inconsistent, or conceptually incoherent relative to other parts in the system. It is conceivable that designers who routinely interact with these issues may even have slower learning curves, lower job-satisfaction, and a higher likelihood of leaving the firm.

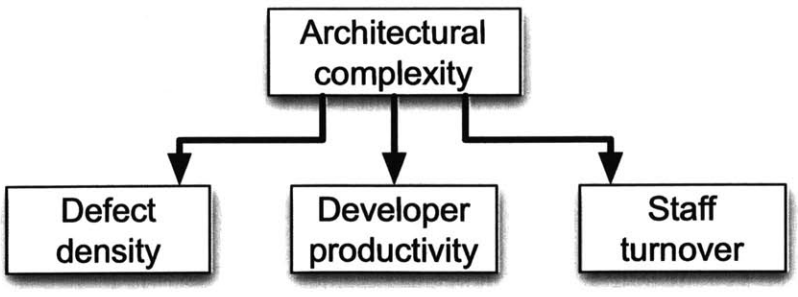

Figure 11: Conceptual Model

In this work we hypothesize that these problems can be caused by complexity in the system's architecture. We differentiate this *architectural complexity* from more traditional inward-looking metrics, such as McCabe's score, that are designed to capture the complexity contained within individual components, files, functions, classes, or methods when considered in isolation. Architectural complexity is outward facing in that it aims to capture the complexity of the connectivity structure between a component and the rest of the system.

The architectural complexity of a component is higher when it is in a positioned in a region of the system with a relative absence of canonical structures known to keep systems stable and within the bounds of human understanding: hierarchy and modularity, among others. Architecturally complex components may have been initially positioned within integral parts of the system, or entropy in the development process may have degraded interfaces and connected formerly isolated segments of the system. If interconnectivity patterns contain architecture spanning cycles, feedback may cause design iterations or gridlock during ongoing maintenance and development. Files that are positioned in more complex regions of a system architecture should be harder to work with, and hence more costly to the development organization.

Useful metrics for operationalizing the concept of architectural complexity are the visibility metrics developed by MacCormack, Baldwin, and Rusnak. These metrics can be used to categorize files based on how reachable they are within the network of components and whether they are positioned within large cycles. [8, 9]

4.2 Complexity and Quality

In the first analysis in this thesis we explore the link between architectural complexity and defect density. Many previous studies have looked at determinants of software defects for good reason. Quality problems have plagued software since its invention, leading many academics and practitioners to repeatedly declare the field of software engineering to be in a state of crisis. Major project failures and the loss of human life due to software errors are seen as a much too common occurrence. [6, 116, 136-139]

Complex software is difficult to design correctly. While some defects are attributable to bugs introduced during the coding process, many are caused by incorrect, incomplete, or inelegant requirements and specifications. To complicate matters further, Brilliant et al. found that many defects are caused by complexity in the problem domain itself. Certain required processes or algorithms are harder to code properly than others, regardless of programming methodology, decompositional strategy, or language employed during software design. [140] Subtle errors at architectural interfaces may be especially damaging because they may be found late, be harder to fix, and therefore cost substantially more to correct. These types of issues may only become apparent during system integration. [141] Unfortunately, the cost of fixing a defect has a geometrical relationship to the development stage in which it is found, which potentially makes them extremely expensive. [142]

Complex software is exceedingly hard appropriately validate. Some of these difficulties arise from properties unique to software that make it exceedingly difficult to test. Many traditional means of managing the quality of electromechanical systems do not work in the software domain. Firstly, because software has a discrete (rather than continuous) nature, tracing the possible paths through code or enumerating the possible system states results in rapid combinatorial explosion. In physical systems, the continuous nature of products allows testers to collapse the state-space by making various logical inferences. Furthermore, most risk models developed for physical products focus on the probability of individual part failures due to wear. In software there is no such thing as "wear." Software defects are in some sense always caused by design or specification failure. As a result, statistical models imported from the electro-mechanical domain are sometimes useless. [6, 143]

Over the years software engineers and computer scientists have taken different approaches to addressing quality issues. One school of thought, pushed by Dijkstra, Hoare, and other mathematically oriented computer scientists, has advocated for the use of formal methods and proofs of program correctness. [144, 145] Such methods have gained traction in the kernels of mission critical applications, but have been viewed as impractical for larger general-purpose software. Most large software projects rely instead upon system and

unit testing regimes. [146-148] While such testing is useful, it can never be perfect. Due to combinatorial explosion it impossible to test even many reasonable conditions. Given the impossibility of full test coverage, an organization must consider the level of acceptable risk when determining how many resources to dedicate to testing. [149] Disagreements between the formal methods and testing camps have subsided to a large degree, and advances have been made in both the theory of testing [150] and the practical utility of formal methods [77] over time.

Various studies have been conducted to find relationships between development process measures, software product measures (including complexity of various sorts), and defect density. Unsurprisingly, more defects are found in files that are larger and that experience more overall development activity. [151] These strong predictors are often included as controls in statistical models looking at defects in a file. There is disagreement about the relationship between a file's age and its defect density. Graves found that older files had fewer bugs [152] but Kemerer and Slaugher [115] found that they had more. A variety of development process metrics have been found to correlate with defect density as well. Mockus et al. found that changes that modify files in multiple subsystems are more likely to introduce defects, suggesting that misalignment between functional relationships and structural relationships in the code increases risk. [153] Eaddy et al. obtained a similar result, finding that code implementing "cross cutting concerns" has more defects. [154] Cataldo et al. found that files that have many "logical" dependencies (i.e., over a defined time period, changes to those files are submitted together) have more defects. [151] Interestingly, however, they also found that if groupings of files with logical dependencies form stable clusters, then defect density goes down substantially. This suggests that natural modules that successfully contain functional changes within their borders successfully manage complexity. Cataldo also found that changes that must be jointly implemented by multiple people are associated with greater defect density in files.

As previously noted, McCabe's cyclomatic complexity number has been found to correlate positively with defect density. [113-115] Functions and class methods with high McCabe scores are hard to understand. They are therefore more likely to contain errors and more likely to be modified incorrectly. They are also

harder to test appropriately due to a combinatorial explosion of execution paths. Card and Agresti found that a metric combining information about control flow (as measured by fan-out) and information flow (as measured by the size of the interface as measured by the number of I/O variables) correlated strongly with defect density. [119, 122] Code with high intra-module cohesion and low inter-module coupling has been found to have fewer defects as well. [155-157]

As previously noted, large-scale systems structured as nearly decomposable hierarchies of loosely coupled modules are thought to possess a number of significant evolutionary advantages that should lead to fewer defects. [48, 51, 69] Much of that work that establishes this link has been theoretical and descriptive in nature however. Despite a large body of literature that has taken a quantitative approach to exploring the relationship between various process, syntactic, and structural predictors of defects, little of work has been done to quantitatively explore the relationship between the defects experienced by a software component and the high-level architectural properties of the system in which it rests. One notable exception is a recent study conducted by Sosa, Mihm, and Browning in which cyclical relationships (which are disallowed in hierarchies) between Java classes in an open-source software project were found to predict future defects. [158] No similar studies have been conducted on a large commercial codebase developed by a large group of paid software engineers, however.

This leads us to our first hypothesis:

Hypothesis 1: *Software files with higher levels of architectural complexity are more error prone.* Complex files will be modified to fix defects more often than other files.

4.3 Complexity and Productivity

In our second analysis we explore the link between architectural complexity and developer productivity. The need to understand productivity has been driven to some extent by the project management community. Managers need the ability to do reliable cost and schedule estimation for planning and contracting purposes.

Within this context, the focus has been on team or organizational productivity. Another large body of work, much of it theoretical, qualitative, and descriptive, has looked at the impact of system architecture on the effectiveness of design and maintenance organizations. In addition, some researchers have done empirical and experimental work to explore determinants of individual productivity. A portion of that work has looked at the link between productivity and various forms of complexity.

4.3.1 Software Cost and Schedule Estimation Techniques

A variety of techniques have been devised to help project planners estimate cost and schedule for software development projects. Kemerer [159], Jorgenson [160], and Boehm [161] have each written papers surveying activity in this field. In 1987, Kemerer noted that Boehm's COCOMO model [162] and the SLIM model [163] were in wide use and had general applicability. Other techniques that have come along since that time include Delphi, a rule-based approach, system-dynamics-based approaches pioneered by Abdel-Hamid and Madnick [27, 28], COCOMO II (an update to make COCOMO suitable for estimating Object-Oriented software projects among other things) [161], and Agile estimation techniques. [164] Boehm says that people devising these estimation techniques "all faced the same dilemma: as software grew in size and importance it also grew in complexity, making it very difficult to accurately predict the cost of software development." [161] Boehm continues:

> Just like in any other field, the field of software engineering cost models has had its own pitfalls. The fast changing nature of software development has made it very difficult to develop parametric models that yield high accuracy for software development in all domains. Software development costs continue to increase and practitioners continually express their concerns over their inability to accurately predict the costs involved. One of the most important objectives of the software engineering community has been the development of useful models that constructively explain the development life cycle and accurately predict the cost of developing a software product.

This problem is far from academic. In 2006, Laird and Brennan reported that the average project overran its budget by 45%, only 28% were delivered on time, and 23% of projects were killed. [165, 166] Boehm says [161] that planners using COCOMO or other estimation models glean information from past projects and combine it with information about the current design, the available workdays, team size, team skills, and labor costs to produce estimates for effort, schedule and cost. This process generally takes place during or immediately after the architecture phase of a project. Many of these models take as inputs:

- Estimates of a project's size in lines of code (LOC), instructions, function points, or object-oriented constructs
- Estimates of an average developer's productivity in terms their ability to deliver LOC, instructions, function points, etc. per unit time.
- Crude multipliers or scaling factors to account for customer complexity, system complexity, novelty, etc.

When studying the accuracy of estimation models, Kemerer found that "models developed in other environments do not work very well uncalibrated." Organizations wishing to use models must collect historical data internally before they can be very useful. He noted that the best models, once calibrated, worked reasonably well, "explain[ing] 88 percent of the behavior of the actual man-month effort [159].

Kemerer concludes by noting what he believes to be the fundamental weakness of these estimation approaches. [159]

> [A]lthough improving estimation techniques within the industry is a worthwhile goal, the ultimate question must concern how the productivity of software developers can be improved. These estimation and productivity questions are related in that the estimation models contain descriptions of what factors their developers believe affect productivity. How well do these models identify and reflect these factors?... [T]he models, although an improvement over their raw inputs for estimating project effort, do not model the factors

affecting productivity very well. One possible extension of this research is to... attempt to determine the causes for the wide swings in accuracy of the estimates across projects. What characteristics make a project amenable to this type of estimation? What factors could be added to the models to enable them to do a better job on all of the projects? On the productivity side... projects show a large amount of variance in terms of such traditional metrics as SLOC per man-month. Can these variations be traced to productivity factors controllable by the software manager?... Further research needs to be done to isolate and measure these factors affecting systems professionals' productivity if the profession is to meet the challenges of the future.

4.3.2 The Impact of Architecture on Productivity

Architectural factors strongly influence developer productivity considered in the aggregate. In fact, Printz makes a compelling argument that the equations underlying the COCOMO model presume that the system being developed has been appropriately modularized according to Parnas' principles, thereby allowing developers to operate independently. If a design is not sufficiently modular, Printz asserts that estimation models lack a solid theoretical foundation. [167] A substantial body of work in design and systems theory explores the means by which hierarchical decomposition and modularity eliminate feedback in the design process, thereby enabling this independence of action and reducing the likelihood of change propagation or rework. [5, 30, 31, 34, 36, 46, 55, 58, 66, 69, 70, 168-172] (This work has already been discussed and will not be revisited in great depth here.) The architecture of a product being developed or maintained has a substantial impact on the productivity of a development organization. Misalignment between the structure of a complex product and the needs of a development team can impair organizational performance. Recent case studies have shown that refactoring software to reduce architectural complexity can be extremely rewarding because it can make developers more productive when implementing new functionality, and because it reduces defect-proneness, thereby allowing the developer to expend less effort correcting, testing for, avoiding, and preventing defects. [8, 104]

4.3.3 The Productivity of Individual Software Developers

Some experimental and empirical work has explored the determinants of individual and team productivity. Controlled experiments have been conducted to study the time it takes developers them to complete tasks (or their ability to complete the task at all) under a variety of conditions. (Unfortunately, many of these studies have used computer science students rather than professionals as test subjects.) Industry studies have combined information from change tracking, accounting, and version control systems to determine the rate at which various tasks have been performed.

Most productivity studies have looked at factors that contribute to project or team productivity. Blackburn reports that smaller teams, projects with shorter cycle times, and teams that spend more time on requirements and prototyping are more productive. [173] McCormack et al. found that developers on larger projects, projects with complete functional specifications, and projects that employed prototyping were more productive. They also found a weak relationship between productivity and the completeness of detailed design specifications. [174] It is well known that defect correction is more difficult and time consuming than the development of new features. Banker et al. tell us that 50% of effort expended during a maintenance task is consumed trying to understand the code, and that complexity is strongly related to understandability. [175] The effort required to correct a defect grows dramatically as time passes, partially due to a need to reconstruct a mental model of the code and partially due to the "knock-on" effects of quality problems. [24, 59] Practices aimed at minimizing defect introduction and finding defects early have an extraordinary impact on individual and team productivity as a result.

Some studies have looked at determinants of individual productivity. Many researchers have noted substantial variations in developer ability. The productivity of individuals has been shown to vary by an order of magnitude. [162, 176-183] (Although not highlighted in this report, we should note that a 10x difference in developer productivity between the top and bottom quartile was observed in the data used for this study as well). The impact of individual variation is very strong, likely exceeding the impact of most project or software related factors. Due to this highly skewed individual variation, studies exploring other factors must

take care to appropriately control out this noise. Other important relationships have been found. Individuals with more experience and a diversity of experiences tend to be more productive. Unsurprisingly, developers are also more productive when they have done prior work that that is similar to the current task. [175] Evidence of learning curves in software development has been found. [184, 185] In addition to age and experience, individual practices matters as well. Littman et al. did an experiment and found a strong relationship between an individual's problem solving strategy and performance on experimental maintenance tasks. They identified developers who used "systematic" and "as-needed" strategies during problem solving. "As the names imply, maintainers employing a systematic strategy attempted to construct a mental model of how the program worked, and then used that mental model in the performance of their maintenance task. Others only examined the program code when necessary to check specific hypotheses. The systematic maintainers were the only ones who successfully completed the maintenance tasks. Recently, Robson et al. have noted that this finding may be an artifact of the small program used in the experiment, and that on large programs the systematic approach may be infeasible, leaving no preferred strategy." [186] This last point is important. It suggests that if complexity (architectural or otherwise) is so great that the mental model required to use the "systemic" strategy is too large to fit into an individual's head, the "as needed" strategy is the only one available.

A few studies have looked at the relationship between code complexity and productivity. By combining data from software tracking systems and billing systems at a defense contractor, Gill found McCabe's cyclomatic complexity to be negatively associated with developer productivity when performing software maintenance [111] Chen reported that an entropy-based program-control-complexity metric correlated negatively with productivity under experimental conditions (but a sample size of 8 prevents us from drawing conclusions.) [187] In another experiment, Curtis found that program size, McCabe cyclomatic complexity, and Halstead metrics were negatively correlated with programmer accuracy and time to task completion. His results held only when the programs were unstructured, contained no comments, and the developers being tested were inexperienced, however. For experienced developers, no effect was found. [22] Akaikine and MacCormack recently found that after a commercial software application was completely rewritten in a different language,

resulting in a system with much lower architectural complexity, the company's maintenance organization was capable of patching critical defects more rapidly. [104]

There is a large and diverse body of descriptive and theoretical work describing the virtues of hierarchy, modularity, and other architectural patterns in system designs. Much of that work tells us that engineers should be more productive when working in well structured architectures. Unfortunately, there is currently no empirical research validating this link between a system's architectural properties and the productivity of engineers doing feature development or maintenance. This represents a serious gap in the literature on complex system design management. Our second hypothesis is therefore the following:

Hypothesis 2: *When working in more complex regions of a system design, developers are less productive.* Developers working in complex regions will produce fewer lines of code overall, fewer lines of code when implementing features, and fewer lines of code when correcting defects, than they would when working in less architecturally complex regions of the system.

4.4 Complexity and Human Capital

In our third analysis we explore the link between architectural complexity and human capital issues. More specifically, we explore the link between the architectural complexity of the code a developer works in and the probability that that developer leaves the firm. Turnover is costly in its own right. In addition, measures of staff turnover can also be viewed as a proxy for other costs related to the performance issues, morale issues, or burnout that might precede a voluntary or involuntary termination. While some amount of turnover in any organization is healthy, a causal link between code-complexity and turnover could hardly be seen as positive. In this section we will review pertinent literature on potential human costs of complexity such as morale and staff turnover among technologists and state our third hypothesis.

Most research on motivation and turnover among technical professionals is aimed at technology managers, seeking to improve managerial practice. (An excellent reference is The Human Side of Managing

Technological Innovation by Katz [188])

Software development staff turnover is very costly for a development organization. Westlund says, "Retaining information technology employees has been a problem in many organizations for decades. When key software developers quit, they depart with critical knowledge of business processes and systems that are essential for maintaining a competitive advantage." Kemerer relays a similar insight provided by Dean and McCune: "a survey of Air Force maintainers reported that the top three problems in software maintenance were all comprehension related: (1) a high rate of personnel turnover requiring that unfamiliar maintainers work on the systems, (2) difficulty in understanding the software, particularly in the absence of good documentation, and (3) difficulty in determining all of the relevant places to make changes due to an inadequate understanding of how the program works." [120]

Because turnover is costly, it is important to understand the factors that lead to it. In a survey of software developers, Westlund looked at the relationship between "turnover intentions" (an employee's willingness to leave) and nine factors including satisfaction with pay, opportunity for promotion, quality of supervision, benefits, contingent rewards, working conditions, happiness with coworkers, satisfaction with the nature of the work, and satisfaction with communication. Multiple linear regression models were used to isolate the impact of each determinant. While all of these predictors correlated with turnover intentions, some effects were stronger than others. The strongest predictors of turnover were satisfaction with contingent rewards (the acknowledgement of a job well done), communication, and supervision. The weakest were satisfaction with benefits and working conditions. [189]

Motivation (and demotivation) are strongly linked to turnover. It is therefore important to understand the factors that motivate technologists. In 2007, Beecham, Baddoo, and Hall surveyed the literature on motivation among software engineers. [190] Their summary of the topic included a number of interesting insights: Characteristics of the personality makeup of software engineers include "the need for growth and independence... The need for growth may be due to the engineer's internal make up, and... the need to keep up with the fast changing technology. The need for independence is possibly linked to the type of person

attracted to software engineering that is sometimes seen as a creative task that is not helped by overbearing management." [190] The "most frequently cited motivators in the literature are, 'the need to identify with the task' such as having clear goals, a personal interest, understanding the purpose of a task, how it fits in with the whole, having job satisfaction; and working on an identifiable piece of quality work." [190-192] In addition, "An experienced Software Engineer is more likely to be motivated by challenges, opportunities for recognition and autonomy." "Poor working conditions" and "lack of resources" are the most commonly discussed demotivators. The effect of work type – new feature development vs. maintenance tasks – on happiness was somewhat ambiguous. Maintenance activity was cited as demotivational in some studies, while other studies showed that some engineers were happy working to maintain legacy code (provided that that maintenance involved the evolution of functionality rather than strictly being corrective in nature). [190]

It is widely understood that people do not leave firms; they leave their bosses. Supervisors who inhibit the productivity of creative professionals or who behave erratically can create toxic work environments. It stands to reason that technical systems that inhibit productivity or that behave erratically may have a similar demotivational effect. After all, software engineers spend more time with the code than with their supervisors. If architectural complexity causes more defects (as stated in hypothesis 1) and impairs productivity (as stated in hypothesis 2) then it may very well impair morale and lead to voluntary turnover. Additionally, because architectural complexity is neither directly observable nor well understood, a manager may fail to appreciate its influence. As a result, lower productivity or higher defect introduction rates may lead a manager to inaccurately conclude that a subordinate is a poor performer, leading to an increased probability of involuntary turnover. Unfortunately, there is no body of literature exploring the link between architecture or complexity on morale and turnover. This leads us to our third hypothesis:

Hypothesis 3: *Developers working in more complex regions of an architecture are more likely to leave their job voluntarily or involuntarily.* Assuming hypotheses 1 and 2 hold, these developers will be frustrated by the increased propensity towards defects, be frustrated by the lower productivity they experience, or be evaluated negatively by managers or peers as a result.

5 Research Methods

A variety of research methods were chosen to explore the links between costs and complexity. This section describes the means used to represent architecture, the means of defining and measuring complexity, and the means by which architecture and complexity metrics were extracted from a large software system. We then demonstrate how the complexity metrics used in this study relate to hierarchy and modularity. We go on to describe the workflow and tools used by software developers, the historical data that is produced as a side-effect of their work, and the means by which we can exploit historical databases to measure defect density, productivity, and staff turnover within an organization.

5.1 Measuring Architectural Complexity in Software

Well-known patterns are employed by man and nature to control complexity as systems scale. These include hierarchy and modularity. Technical architectures in which these patterns are judiciously applied tend to be of higher quality, be safer, and to benefit from other "ilities" over the course of their lifecycles. "Judicious" does not always imply *more,* however, and *more* is not always *better.* There are certainly well designed systems that are integral where these patterns may be less pronounced. Integral systems are more architecturally complex than comparably sized systems with hierarchical structure or modular boundaries, and that architectural complexity may also make them more costly. Within the same system, different regions will be more interconnected or less so, and will therefore have varying levels of intra-system architectural complexity. Components in a position to affect many other components, or that can be affected by many other components, have high levels of architectural complexity relative to their less well-connected peers. Affecting or being affected by other components need not be direct – it may be done through an intermediate connection. Architectural patterns are global features of a design. To capture the impact that these architectural patterns have on a single component we must therefore use metrics that take both direct and indirect relationships into account.

Networks are established tools for representing and analyzing system architectures because they are a natural means of capturing hierarchical relationships, modularity, coupling, cohesion, and cyclicality, and other architecturally important patterns. When networks are used for this purpose, network *nodes* designate parts, components, or system elements. *Arcs* or *lines* between nodes designate functional or structural relationships between those parts. If these arcs are *directed* they can represent one-way dependencies between parts. Directed networks are appropriate abstractions for use in this study because they can be used to represent relationships between software constructs that are often unidirectional. In this research, we extract networks from a software product's source code. We then compute network metrics as a means of assigning architectural complexity scores to each file within that codebase.

5.1.1 Networks and DSM Architecture Representations

A network is a means of representing entities and the paths or relationships between them. In network terminology, an entity is a "node" while a path or relationship is an "edge" or "arc." The following figures show the same simple network in two different ways. Figure 12 is a traditional network view. Nodes A, B, and C, are connected by the arc $\overrightarrow{BA}$ and the bi-directional line $\overline{CA}$. In Figure 13, this same network is represented as a square matrix. Nodes have the same ordering down columns and across rows. Dots in the matrix the same purpose as the arrows in the traditional view. A directed arc is read by looking first at a row (the "from" dimension"), finding a blue dot indicating an arc, and then scanning up the column to see which node the arc goes "to." In the traditional representation, each node is a single point. In the DSM representation, each arc is a single point. While encoding identical information, each view can highlight different network features.

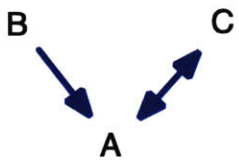

Figure 12: Simple Network

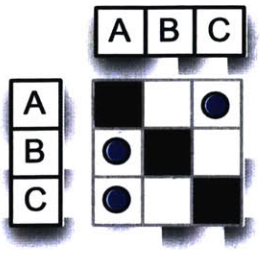

Figure 13: Simple DSM

It is important to note that this report uses network conventions that are slightly different than those used in much of the DSM literature due to differences between software and hardware. See Figure 3 for a detailed explanation.

Two real-world networks can illustrate the value of both traditional and DSM network notations. The Moscow Subway map shown in Figure 14 (a traditional view) can be used in navigation.[9] A matrix representation would not allow a person to reach a destination. On the other hand, the matrix representation of the Mozilla software system (shown in Figure 15) can be used to highlight the modularity and coupling patterns in the system. [9] A traditional view would not make these properties visible.

[9] Thanks to Dan Whitney for providing this example.

Figure 14: Moscow Subway Map

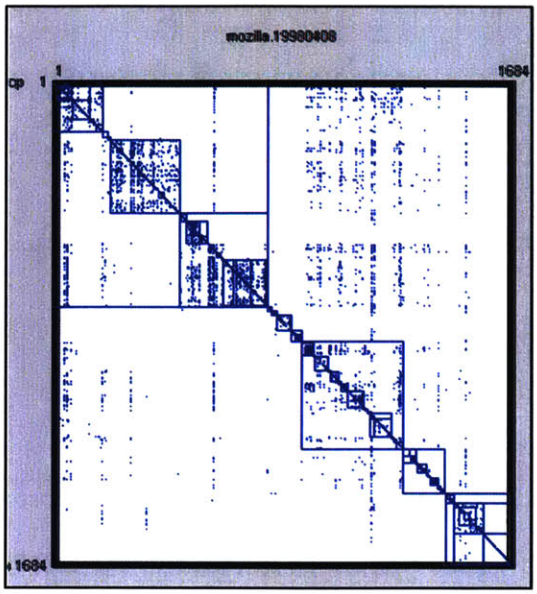

Figure 15: Mozilla DSM

5.1.2 Procedure for Assigning Architectural Complexity Scores to Source Code Files

In this research, we operationalize the concept of architectural complexity by using network metrics devised by MacCormack, Baldwin, and Rusnak [8, 9] that capture the level of coupling between each software file and the rest of the system, and therefore represent relative absence of modular isolation or hierarchical structure in a design.

The MacCormack approach can be described in the following 5 steps:

- Capture a network representation of a software product's source-code using dependency extraction tools
- Find all the indirect paths between files in the network by computing the graph's transitive closure
- Assign two visibility scores to each file that represent its reachability from other files or ability to reach other files in the network
- Use these two visibility scores to classify each file as one of four types: peripheral, utility, control, or core.

This procedure and its rationale will be described in detail in the next sections.

5.1.2.1 Extracting DSMs From a Software Codebase

Software developers can create and modify systems consisting of thousands of source code files and many millions of lines of code. Each source code file contains text written in a specific programming language. These files typically specify functions a computer should perform and data structures for those functions to operate on. Functions tell a computer's processor what instructions to perform while the data structures define information that will be stored in a computer's memory. Similar functionality is often grouped inside the same source code file, but some files will depend on functionality or data described in other files. When software development is complete, all of the files must be compiled and linked together.[10] Each file is translated into a machine-readable form, and cross-references between files are resolved. Once this linking step is complete, a program can be loaded and run.

In this research, we construct networks that represent software source-code files as nodes. When a relationship spans two files (such as the invocation of functionality or data access), it will be represented as an arc between two nodes. Figure 16 is a very simple illustration containing a program with two files, each containing two procedures. The first file defines procedures for calculating properties of a rectangle. The second defines generic mathematical procedures that multiply and add numbers. Calls from the "rectangle" functions to the "math" functions span these two files. This example would result in a network with two nodes, and a single arc from "rectangle_functions" to "math_functions".

[10] The same logic occurs with interpreted languages, but the translation and linking happen at run-time. In compiled languages, some linking may be performed dynamically at runtime as well.

```
// ====================================
// file name: rectangle_functions
// ====================================
procedure area = rectangle_area(length, height)
        area = multiply(length, height)
end

procedure perimeter = rectangle_perimeter(length, height)
        perimeter = add(multiply(length, 2), multiply(height, 2))
end
```

```
// ====================================
// file name: math_functions
// ====================================
procedure sum = add(num1, num2)
        sum = num1 + num2
end

procedure multiple = multiply(num1, num2)
        multiple = num1 * num2
end
```

Figure 16: Simple Pseudocode

Real software products are obviously much larger than this (admittedly absurd) example. For example, the DSMs shown in Figure 17 and Figure 18 represent two entirely different software systems of roughly comparable size. In both DSMs, an algorithm has reordered the files so as to move as much "mass" below the diagonal as possible. The software system shown in Figure 17 has a structure that is extremely hierarchical (As demonstrated by the fact that the algorithm moved almost all mass below the diagonal.) The software system in Figure 18, on the other hand, has a "core-periphery" structure. When a system has a core-periphery structure, the lower-diagonalization process naturally segments a DSM into four distinct regions. Figure 18 includes files that are utilities (relied upon by many others), a core with indirect and cyclical connectivity, files that are on the periphery of the network, and controller files that call out to many other files. MacCormack has found that approximately 80% of software systems have this "core-periphery" structure, while approximately 20% are more purely hierarchical. [79]

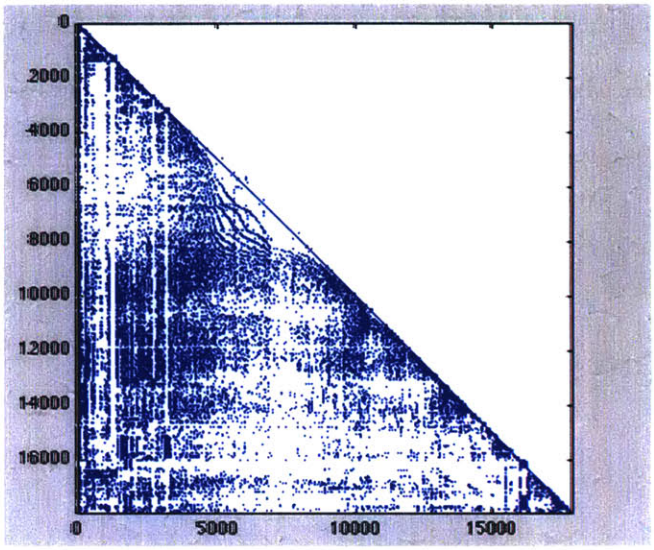

Figure 17: Hierarchical Software System

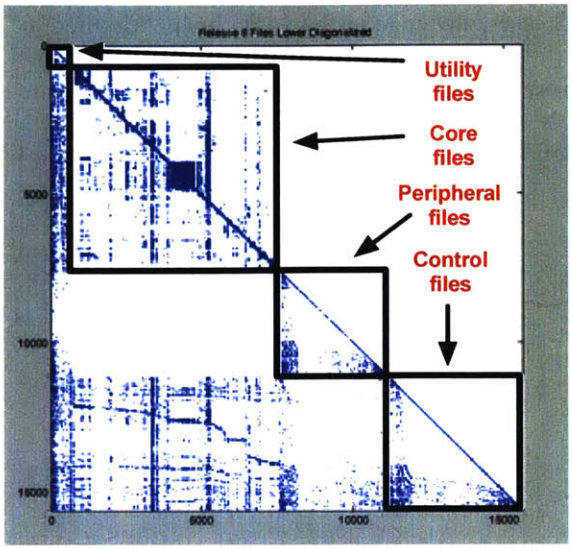

Figure 18: "Core-Periphery" Software System

In this study we extracted DSMs from portions of codebases composed of C and C++ language files. The *Understand* static analysis tool commercially available from *Scientific Toolworks, Inc.* was used to parse code and extract dependency information. We chose to add directed links in DSMs when the following types of file-spanning relationships were encountered:

- The site of function calls to the site of the function's definition
- The site of class method calls to the site of that class method's definition
- The site of a class method definition to the site of the class definition
- The site of a subclass definition to the site of its parent class' definition
- The site at which a variable with a complex user-defined type is instantiated or accessed to the site where that type is defined. (User-defined types include structure, union, enum, and class.)

The directionality of these arrows was chosen based on the likely direction of change propagation. (Change actually propagates in the opposite direction of the arrows given the way we have chosen to draw them.) In

all cases, a change in the entity being "pointed to" had a reasonable chance of requiring a change in the entity from which the arrow originates. The "to" node is a file which defines an interface, provides functionality, or defines the structure of data that the "from" node relies upon.

5.1.2.2 Finding the Indirect Dependencies in a Graph

Once a network of the software architecture is captured, a *transitive closure* [193, 194] algorithm is run to identify all direct and indirect links. The figures below illustrate how this is done. Figure 19 and Figure 21 are network and DSM representations of the same graph. Figure 20 and Figure 22 are the transitive closures diagrams of that same graph. Note that node "D" depends on "C" directly, and "A" and "B" indirectly. The transitive closure graph shows potential dependencies in the system. The indirect link from D to A is important because an unwise design change made to A could break D through the intermediary C. Unintended side-effects of design choices may be conveyed through intermediaries because indirect links are harder for the designer to track.

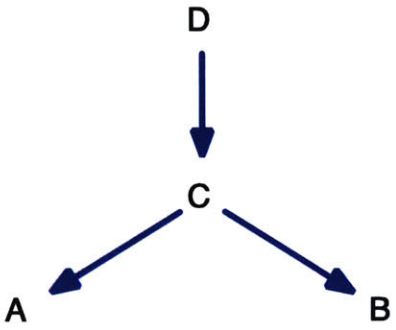

Figure 19: Simple Network (Direct)

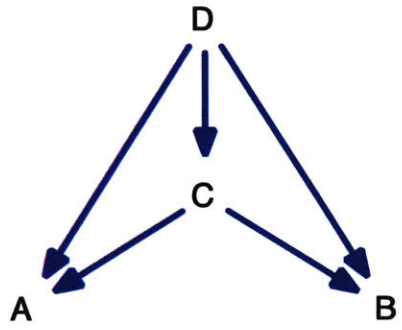

Figure 20: Simple Network (Transitive Closure)

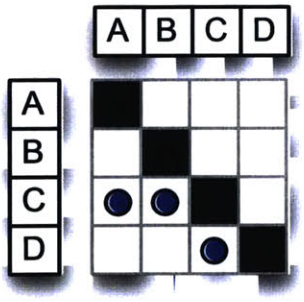

Figure 21: Simple DSM (Direct)

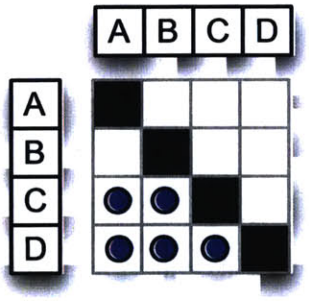

Figure 22: Simple DSM (Transitive Closure)

5.1.2.3 Computing visibility metrics for each file

Once the transitive closure graph is computed, visibility scores are computed for each node.

The following metrics are taken for **each node** in the **direct dependency DSM**:

- ***Fan In (FI):*** *How many other nodes depend upon it directly?* Computed by counting the number of arrows pointing into that node or counting entries (including the diagonal square) down the node's column in the DSM.

- ***Fan Out (FO):*** *How many other nodes does it depend upon directly?* Computed by counting the number of arrows pointing out from that node or counting entries (including the diagonal square) across the node's row in the DSM.

The following metrics are taken for **each component** from the **transitive closure DSM** and its nodes:

- ***Visibility Fan In (VFI):*** *How many other nodes depend upon it directly or indirectly?* Computed by counting the number of arrows pointing into that node in the transitive closure graph or counting entries (including the diagonal square) down the node's column in the DSM.

- ***Visibility Fan Out (VFO):*** *How many other nodes does it depend upon directly or indirectly?* Computed by counting the number of arrows pointing out from that node in the transitive closure graph or counting entries (including the diagonal square) across the node's row in the DSM.

The following metrics are taken for **the system as a whole**:

- ***Network Density:*** A system-wide metric determined by dividing the number of direct links in the graph by the total number of possible links. Computed by counting the number of dots and diagonal elements and dividing by the total number of squares.

- ***Propagation Cost:*** A system-wide metric determined by dividing the number of direct *and indirect* links in the graph by the total number of possible links. Computed by counting the number of dots and diagonal elements and dividing by the total number of squares.

To illustrate, Figure 23 and Figure 25 represent the same network with 12 nodes and 47 arcs (including self referencing arcs), while Figure 27 is its transitive closure. In these examples, node "H" has FI = 3, FO = 4, VFI = 6, and VFO = 6. For the system as a whole, Network density = 47/144 and Propagation cost = 81/144.

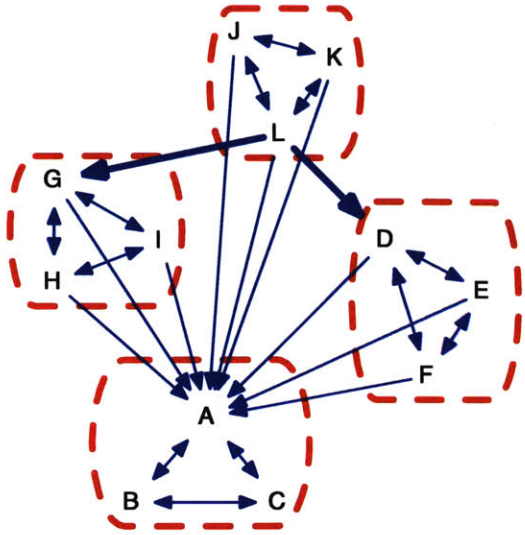

Figure 23: Hierarchy of Modules

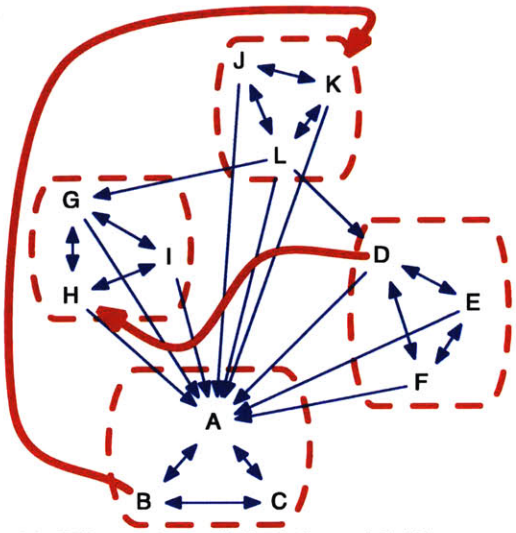

Figure 24: Hierarchy of Modules with Unwanted Links

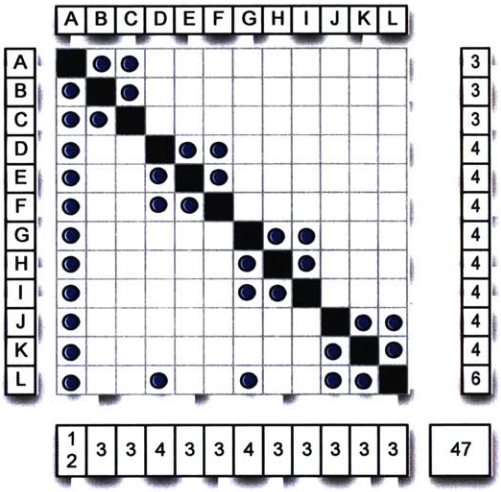

Figure 25: DSM of Hierarchy of Modules

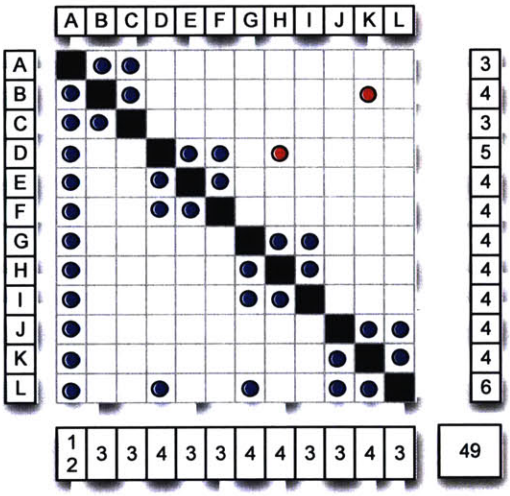

Figure 26: DSM of Hierarchy of Modules with Unwanted Links

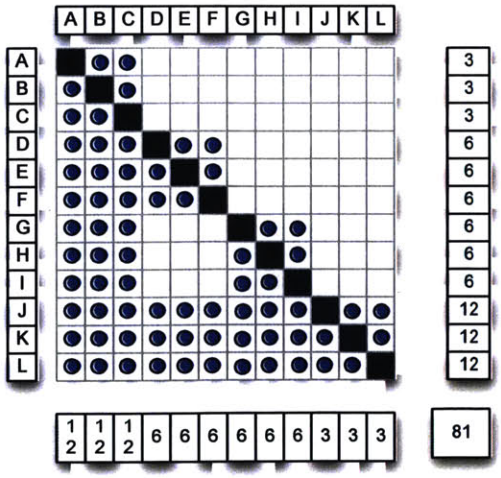

Figure 27: Transitive Closure DSM of Hierarchy of Modules

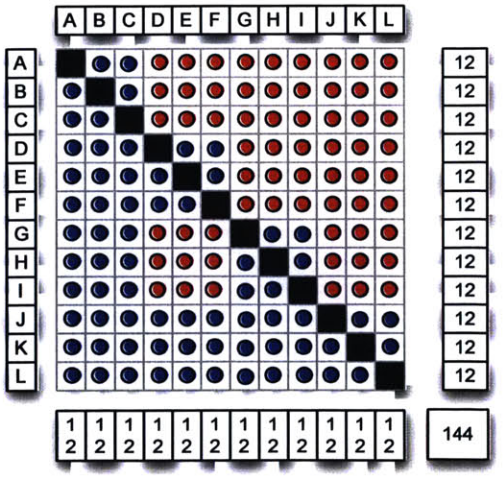

Figure 28: Transitive Closure DSM of Hierarchy of Modules with Unwanted Links

5.1.3 Component Architectural Categories: Peripheral, Utility, Control, Core

Once computed, VFI and VFO scores for components across a system can be rank-ordered and plotted to see their distributions. Figure 29 shows the distribution of visibility scores for one of Iron Bridge's releases. When systems have a core-periphery structure, these distributions tend to contain "cliffs" demarcating the boundary between peripheral files and those that are highly connected when indirect links are considered. [79] In the MacCormack approach, these cliffs are used to partition VFI and VFO scores into "low" or "high" bins.

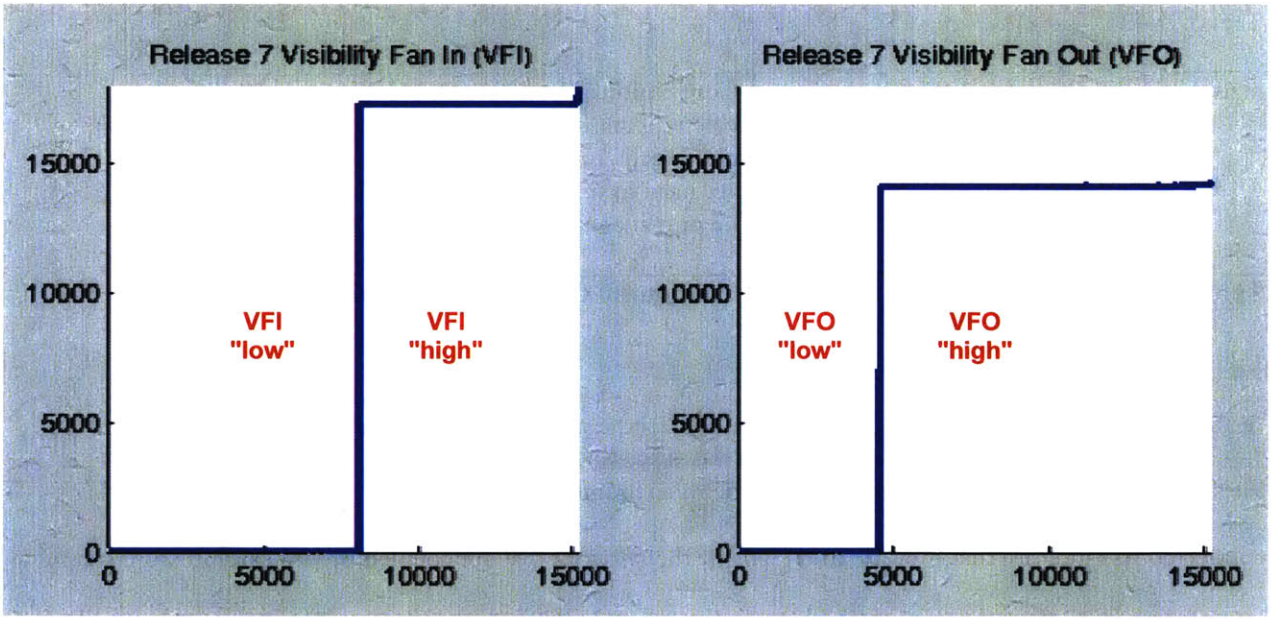

Figure 29: Distribution of Visibility Scores and Cutoff Points for a "Core Periphery" Network

Once visibility scores have been computed, and once those scores are classified as either "high" or "low", each component can be classified as *peripheral, utility, control,* or *core* according to the scheme laid out in Table 1.

If VFI is	and VFO is	then it is	Description
low	low	**peripheral**	Peripheral components do not influence and are not influenced by much of the rest of the system.
high	low	**utility**	Utility components are relied upon (directly or indirectly) by a large portion of the system but do not depend upon many other components themselves. They have the potential to be self-contained and stable.
low	high	**control**	Control components invoke the functionality or accesses the data of many other nodes. It may coordinate their collective behavior so as to bring about the system level function.
high	high	**core**	Core regions of the form highly integral clusters, often containing large cycles in which components are directly or indirectly co-dependent. They regions are hard to decompose into smaller parts and may be unmanageable if they become too large.

Table 1: Mapping of Visibility Scores to Architectural Complexity Classification

In this research, we use each file's classification as peripheral, utility, control, or core as an indicator of the file's architectural complexity. Core files are the most architecturally complex because their high levels of connectedness indicate that they are in regions of the network that are coupled by large architecture spanning cycles.

5.1.4 The Relationship Between Hierarchy, Modularity, and MacCormack's Metrics

5.1.4.1 How Visibility Metrics Capture Architectural Complexity

Figure 23, Figure 25, and Figure 27 show a network previously discussed in the literature section showing system structured as a hierarchy composed of modules. In some ways this DSM incorporates principles of design put forth by David Parnas, Herbert Simon, and others. [48, 51, 69] In order to understand how these

architectural patterns manage complexity and how MacCormack's visibility metrics capture this fact, we will explore a degenerate case represented by Figure 24, Figure 26, and Figure 28.

Imagine that during maintenance, engineers inadvertently added two additional links ($\overrightarrow{DH}$ and $\overrightarrow{BK}$) in violation of design rules. The network density rises slightly from (47/144) or 33% to (49/144) or 34%. $\overrightarrow{DH}$ causes multiple issues. First, D is interacting with a node that was not previously considered "public" by its module. Secondly, $\overrightarrow{DH}$ couples two modules that previously had no interaction. These modules can no longer co-evolve independently. Teams developing the separate modules may not be aware of this fact. It is possible that H's owner is unaware that D is now dependent upon it. $\overrightarrow{BK}$ is more problematic. It introduces a long cycle spanning several independent components. Not only does B directly depend upon K, K also indirectly depends upon B. Any functionality that indirectly depends on B (potentially all of it) now has a chance of getting into a recursive loop of dependence. Modular isolation of the system has broken down. Hierarchy has been eliminated because arrows no longer flow in one direction. *Homeostasis* has been eliminated. This fact is captured in transitive closure DSM shown in Figure 28. In the degenerate system, VFI and VFO for all nodes are now at a maximum. Propagation cost of this system has risen from (81/144) or 56% to (144/144), 100%.

5.1.4.2 How Hierarchy and Modularity Control Architectural Complexity

Hierarchy and modularity are tools that can control complexity and enable certain *ilities*. When these patterns are employed judiciously, a system can scale, side effects can be avoided, independent parts can co-evolve, and the development process can be managed by fallible and boundedly-rational human actors. Networks connected in these ways are far from random – rather, they are highly indicative of intentional or evolved order in a system.

To demonstrate this point, 10,000 random DSMs with 12 nodes and 47 arcs (12 on the diagonal) were generated. These DSMs had the same number of nodes and arcs as the network in Figure 23 (Parnas and Simon's *hierarchy of modules* from above.) After generation, the transitive closure of each random graph was

computed. The propagation cost for the random networks was compared against the propagation cost for network in Figure 23. Table 2 contains the results.

	Network density	Propagation cost
Hierarchy of Modules	**32.64%**	**56.25%**
Hierarchy of Modules (broken)	**34.03%**	**100.00%**
Random network MIN	**32.64%**	**51.39%**
Random network MAX	**32.64%**	**100.00%**
Random network Mean	**32.64%**	**93.84%**
Random network Median	**32.64%**	**92.36%**
Random network Mode	**32.64%**	**100.00%**

Table 2: Propagation Cost of Structured and Random Networks

Note that the average propagation cost for random networks was above 90 percent, while the propagation cost for the *hierarchy of modules* was very close to the minimum of the 10,000 randomly generated networks. By lower-diagonalizing and visually inspecting 50 randomly generated networks with low propagation cost (those with scores below 60%) we found that lowest scoring architectures were almost fully hierarchical while those with only one or two small cores had these low scores as well. Hierarchy and modularity seem to control structural complexity, and this fact is captured by MacCormack's visibility metrics.

It should be noted that MacCormack's classification scheme defines categories that only imperfectly capture *some* aspects of hierarchy and modularity. By definition, hierarchies are directed *acyclic* graphs. By identifying the largest system-spanning cycle and defining nodes captured that cycle it as "core," we identify a set of files that are clearly in a-hierarchical regions of the system. Some non-core files are also positioned within network-cycles, but those cycles tend to be localized rather than system spanning. Localized cycles can constitute modules that, so long as they are reasonably sized, manage complexity by isolating integrated functionality. MacCormack's classification scheme also captures some notions of modularity by differentiating between files that are tightly coupled and loosely coupled. Loosely coupled elements are easier to evolve and have greater "option-value." By employing a single metrics that imperfectly captures some aspects of both hierarchy and modularity, we can operationalize the concept of architectural complexity in a simple (if somewhat crude) manner. If the simple architectural complexity metrics employed in this study found to be important, subsequent work should explore the link between quality, productivity, and other

more specialized architectural measures. Examples of other architecture measures worth exploring in future work include hierarchy measures devised by Luo and Magee [195] and measures of network centrality. For an in depth review of hierarchy theory and metrics, see Luo's doctoral dissertation. [196] For a good discussion of modularity metrics, see Hölttä-Otto and de Weck. [197]

5.2 Measuring Costs of Software Development and Maintenance

An underlying premise of this work is that architectural complexity can cause significant costs that are traceable to specific software source-code files. In this study, file-level categories (peripheral, utility, control, and core) will be used as independent variables in regression analysis aimed at determining the cost of architectural complexity.

We will explore the relationship between complexity and various forms of cost that are incurred by the organization during development and maintenance. The following measures of cost are used as dependent variables in regression analysis:

- Defect correction activity required in files
- Productivity among developers working in different types of files
- Staff turnover among developers working in different types of files

We also use a variety of other file-, developer-, and activity-metrics as control variables in regression analysis.

In order to understand how these costs can be traced to specific software source-code files, we must understand some things about the workflow of a typical software developer and the tools and databases used in the development process.

5.2.1.1 The Typical Developer Workflow

In the typical workflow (for a project of reasonable size) a software engineer will interact with both a "Change Request" system and a "Version Control System." A *change request system* stores feature requests and

bug reports. It is used by developers to manage tasks and to track work progress. A *version control system* stores all versions of the source code and information about the changes that go into it over time. It allows developers to look at the history and evolution of every file it manages and determine who contributed each line of code.

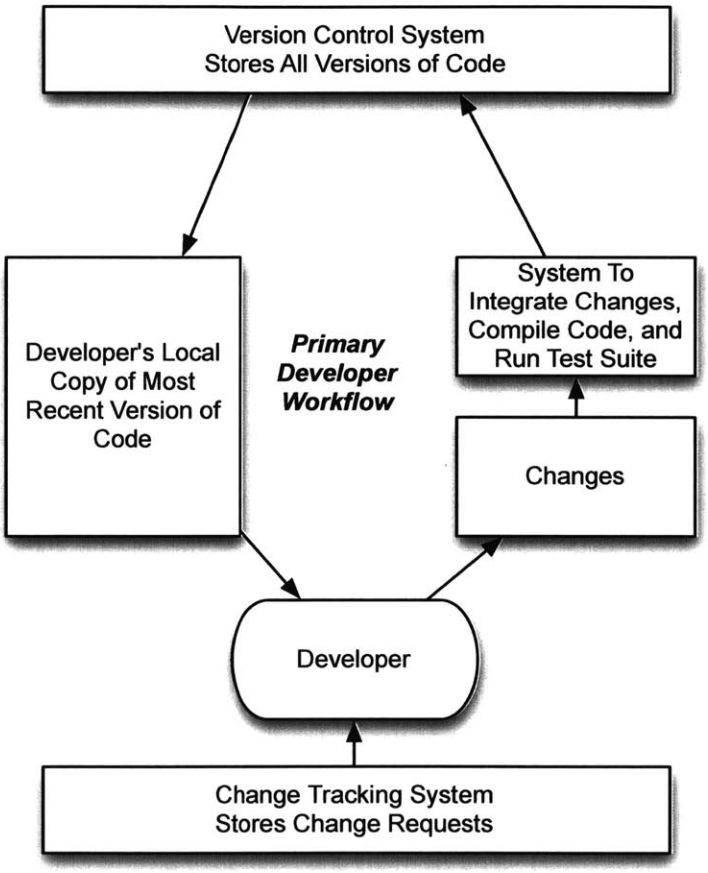

Figure 30: Primary Developer Workflow

To illustrate: Imagine that a developer named "Jill" has undertaken the task of making a change to the software. Jill chooses a task to work on from a list of tasks in the change request system. (If the task she wishes to work on is not in the system, she creates a new entry.) She engages in planning activities appropriate to the task such as requirements gathering, functional design, architectural design, and communication with other people and teams. When Jill is ready to begin coding, she creates a copy of the

most recent version of the code from the version control system's central repository. She makes a local copy – known as a "sandbox" that she is free to modify, recompile, and test without interfering with the work of others. Jill implements the required changes by modifying existing source code files or creating new ones.

When Jill believes the task is complete, she submits her new and modified files to the version control system for inclusion in a new "most recent" version of the code. The version control system compares Jill's locally modified files against the current version and and does two things:

- Creates *changes* which store information about specific lines that must be added-to and removed-from each modified file to incrementally update it from one version to the next.
- Inserts those changes into the version control system repository so that the next person to create a sandbox will obtain Jill's new version of the code.

Once this process is complete, Jill will modify the change request to indicate that the work has been completed and will begin the process anew on her next task.

5.2.1.2 Historical Data Available in Change Tracking and Version Control Systems

Each request stored in the typical change tracking system contains many fields, some of which can be extracted to enable this research:

- A unique identifier
- A means of knowing whether a request is to implement a feature, fix a bug, or perform some other task (such as refactoring)
- Dates on which a change request was created and the change was completed
- The name of the individual who performed the task

Logs in the version control system will generally store the following useful information about every change made to a file:

- A unique identifier
- The name of the file being changed
- The number of lines of code being added and deleted (Note that changing a single line will appear as one line added and one line deleted.)
- The date of the change
- The name of the individual that submitted the code

5.2.1.3 Required Linkage Between Changes and Change Requests

The following diagram shows those data fields that are generally stored in change request systems and version control systems that can be used for our research purposes:

Figure 31: Change Tracking, Version Control, and Integration Between the Two

One additional step is required of "Jill" to enable this analysis. It must be possible to link specific changes to specific change requests so that we can determine which changes were intended to fix bugs, and which were intended to implement features. Although every other piece of data described above is available for most software development projects, this linkage between version control history and change request history is not always present. To enable our historical analysis, Jill must have included unique identifiers associated with the change requests she works on in the version control logs for the changes she submitted so that the two could be associated.

5.2.2 Means of Capturing Complexity and Cost Data

Table 3 shows the type of file-level information that was assembled for use in analysis. Table 4 shows the type of developer-level information that was used in analysis. This data was obtained by mining corporate version control systems, change tracking systems, human-resources databases, and source code.

Per File Development Activity Metrics (captured for a range of dates)	File Based Metrics (captured from a snapshot of the source code)
Number of changes that went into a file	Architectural complexity metrics
Number of changes to fix bugs in file	McCabe cyclomatic complexity
Number of non-bug changes in file	File size (lines of code)
Lines of code (LOC) added and removed	File age (in years)
LOC added and removed to fix bugs	File language (C++, Java, etc.)
Non-bug LOC added and removed	File purpose (product vs. test)

Table 3: File-Based Metrics Captured

Per Developer Activity Metrics (captured for a range of dates)	Developer Metrics (captured on a specific date)
Number of changes made to files	Time with company (in years)
Number of file changes to fix bugs	Is manager?
Number of non-bug file changes	Department
Lines of code (LOC) added and removed	Role (Developer, Quality Engineer, Consultant)
LOC added and removed to fix bugs	Hire date
Non-bug LOC added and removed	Termination date

Table 4: Developer-Based Metrics Captured

Over the course of this research, software was developed to extract this information from multiple sources within a firm, insert it into a relational database, query the database to create tables for use in statistical analysis, and to perform statistical tests. Figure 32 shows a simplified diagram of the infrastructure that was developed during the course of this research:

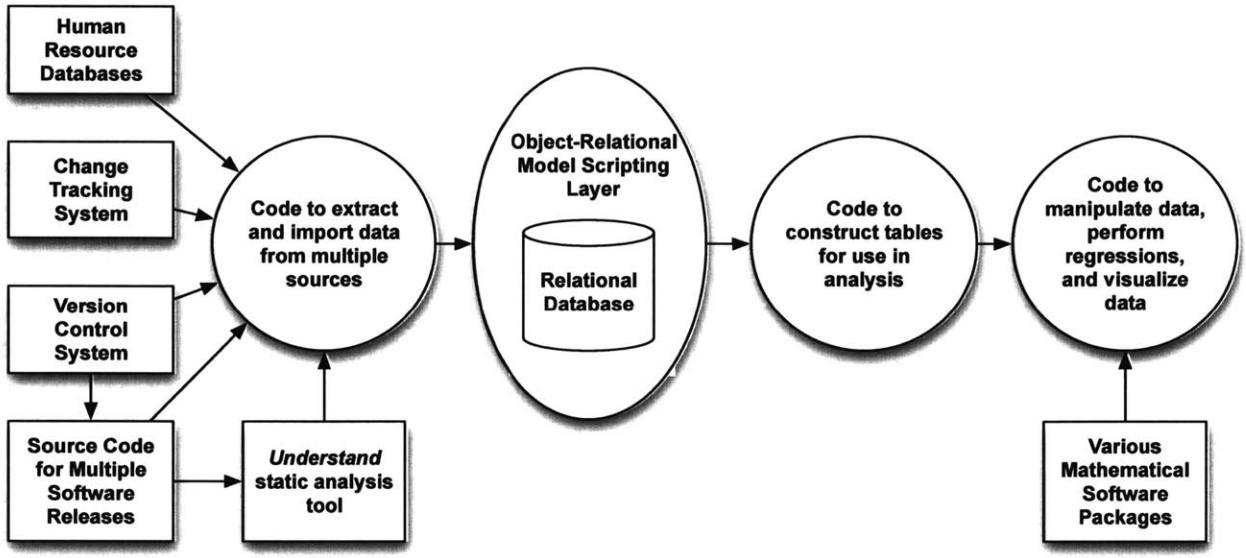

Figure 32: Infrastructure for Extracting, Relating, and Analyzing Data

6 Research Setting

This study set out to quantitatively measure the link between architectural complexity (the complexity that arises within a system due to a lack or breakdown of hierarchy or modularity) and a variety of costs incurred by a development organization. A study was conducted at a successful commercial software development firm within a large codebase. Measures of architectural complexity were taken from their software over 8 successive versions to enable longitudinal analysis. Multiple significant cost drivers including defect density, developer productivity, and development staff turnover were measured as well by extracting information from software version control, change tracking, and human resource databases. The link between cost and complexity was explored using a variety of statistical techniques. This section describes the organization being studied, the portions of their codebase that were chosen for study, and the methods used to extract and clean the data. It concludes by providing some descriptive statistics on the complexity of the code being studied.

6.1 Organization Under Study: Large Scale Commercial Software Firm

The software under examination in this study is a portion of a very large code-base owned by a successful commercial firm. Over time, thousands of professionals wrote software consisting of hundreds of thousands of files and tens of millions of lines of code in many different languages. Hereafter, we will refer to the firm by the pseudonym "Iron Bridge Software." This body of code forms a product platform – some products are required for others to run. Iron Bridge organizes development activity around a fixed release schedule. Within this cadence, teams have coordinated periods for planning, feature development, and quality control. Each development cycle concludes with the commercial release of a new version of the software to customers.

Information was extracted from software source code for eight successive released versions and information about periods of development activity leading up to each release. Architecture metrics were extracted from source code for each version. Information about development costs the organization incurred were extracted

from version control systems, change tracking systems, and human resource databases. The cost of complexity was explored by relating differences in complexity to differences in development costs across the codebase.

6.2 The Software Development Process at Iron Bridge

Iron Bridge's products are developed by hundreds of software professionals all working to improve the same codebase. Product development teams within Iron Bridge exercise a lot of independence when working in their regions of the source-code, and coordinate when they meet at system interfaces. These teams leverage centrally managed tools and processes however. The code-base is stored in a common version control system, compiled using a common build system, and tested using a common regression-testing framework. Teams use a shared change tracking system, version control system, code validation tools, and common project management processes.

Figure 33 depicts the timeline of development activities for one release. At the outset of a release there is a period of time for managerial goal setting and development planning. This is followed by a period of active development. Two important dates toward the end of this period include a deadline for completing product enhancements and a deadline by which all code changes are supposed to be completed. Following code-freeze, developers move on to planning for the next release. Some bugs may continue to be found and fixed, but these late fixes have the potential for negative side effects, and thus receive heightened scrutiny.

Figure 33: Fixed Development Window

Figure 34 shows a simplified picture of the workflow for a developer at Iron Bridge. In the lower left we find a customer receiving a new version of the software. This customer has unmet needs, encounters limitations, or discovers defects in the product. Through various planning processes, marketing activities, and technical support channels, customer needs are translated into prioritized feature requests and bug reports stored and tracked in the change tracking system. Requests are also entered by employees who need to track their own work, encounter bugs, or need functionality developed by other teams. They enter information about features they wish to develop, bugs they need to fix, and refactoring that should be done.

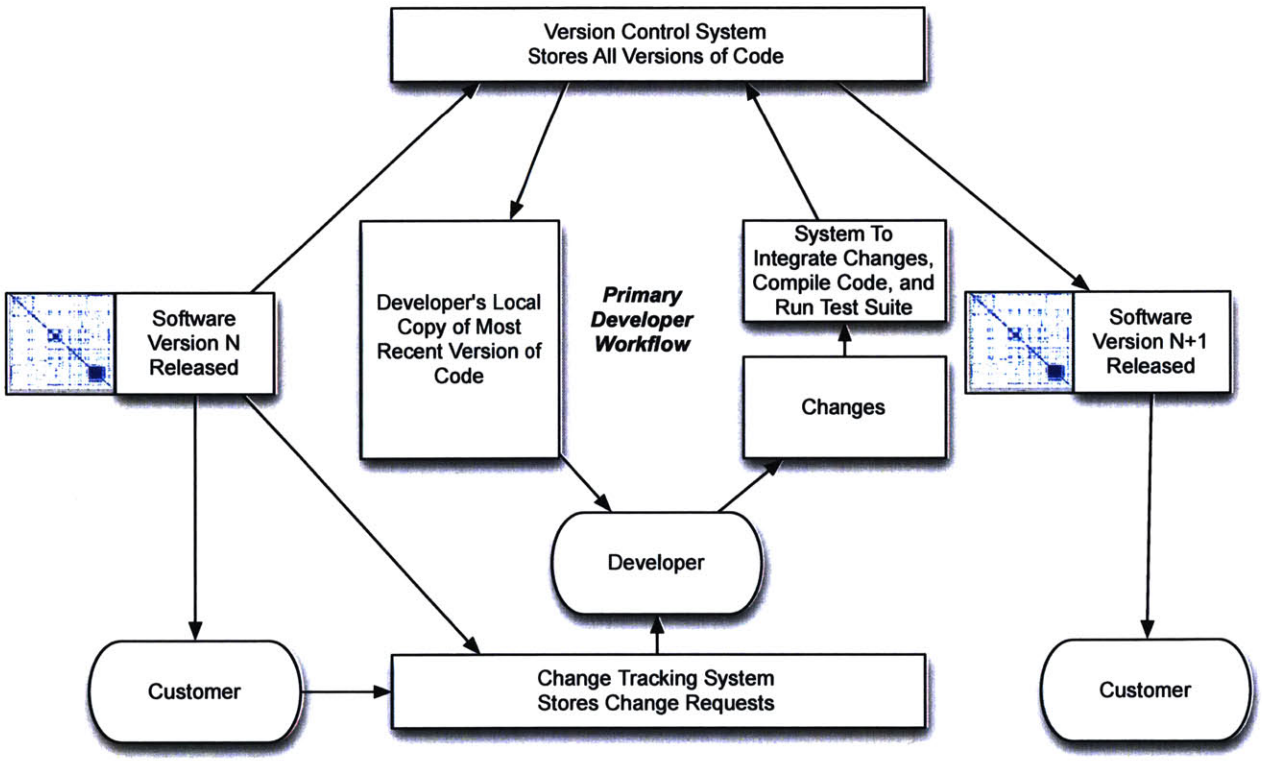

Figure 34: Primary Developer Workflow and Release Cycle

Developers use the change tracking system to monitor the progress of their work. Change requests are assigned owners and passed between people. They contain information about whether a request is to correct a bug, implement a feature, or do some other task such as refactoring. Each change request can put into a

number of states indicating development progress beginning with "New" and ending as either "Completed" or "Discarded."

Iron Bridge was chosen for investigation because it has conducted a good natural experiment. Because teams at Iron Bridge have independent control over software but centralized calendars and tools, the company has done a number of things that enable this research. First, the effect of process, tools, and schedule are controlled for. The impact of architecture on costs incurred by the organization when developing it can be isolated in a reasonable manner. Secondly, because developers within Iron Bridge use common tools, databases, processes and terminology, common measures related to productivity and quality could be established across teams. Thirdly, Iron Bridge's history of data-collection and for long periods without changes in its tooling allowed for longitudinal analysis. Fourthly, because Iron Bridge is a commercial firm we have the opportunity to study not only the software, but also the developers. Many research studies in this field look at open-source systems. Such studies can look at issues related to quality but cannot look at productivity because they cannot make a "40 hour assumption." Here we can measure the productive output of a large number of individuals and assume that they have worked a reasonably similar amount of time. No such assumption can be made when looking at open-source projects. In addition, access to human-resource databases allows us to control for time with the company and managerial status. Finally, and perhaps most crucially, Iron Bridge maintains integrated change tracking and version control systems. Policy dictates that developers include the identification number of specific features or bugs being tracked through the development pipeline when submitting changes into the version control system. Tooling is designed to support this workflow and various checks are put in place to enforce the policy. As a result, the link between feature requests, bug reports, and the code that is submitted to implement them is largely intact a substantial.

6.3 Data Selected for Use in Studies

In order to study the impact of architectural complexity on various costs, we collected samples of source-code files, samples of software developers who worked in those files, and a sample of changes that those developers submitted to files over time.

6.3.1 Software Source Code Files and Developers Chosen for Study

Iron Bridge's codebase consists of code written in multiple languages including C++ and Java. The C++ portion of this codebase was chosen for our study. The C++ portion of the codebase was chosen for several reasons. First, the C++ codebase was large enough that the number of source files, amount of development activity, and number of developers led us to believe that statistically significant results could be obtained for this study. Secondly, the C++ portion of the codebase contains a substantial portion of the historical development activity. Third, because C++ is a compiled language (rather than an interpreted language in which symbols are resolved at runtime) static analysis tools used to extract the dependency structure of the codebase could do a reasonably good job of accurately representing the architecture of the system. Fourthly, C++ code is the heart of the overall system. It implements the most important functionality and algorithms.

In the defect density analysis between 9937 and 13941 C++ files were examined for each of the 8 releases studied. The overall sample consists of 94364 C++ file-releases, including multiple observations of many of the same files. The average file in the sample is 4.2 years old and contains approximately 550 lines of code. During the development periods under study, the number of files in the sample grew by approximately 40% and the number of lines of code grew from 5.5 to 7 million lines. Figure 35 and Figure 36 show DSMs for releases 1 and 8. During this time new utility bands appeared and new modules were added, but the overall structure remained remarkably stable despite its growth.

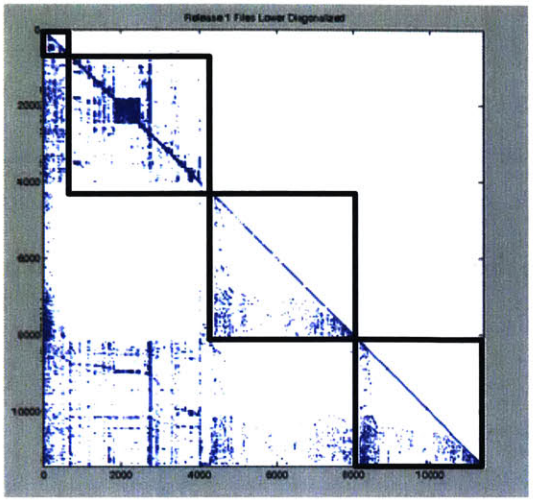

Figure 35: DSM for Release 1

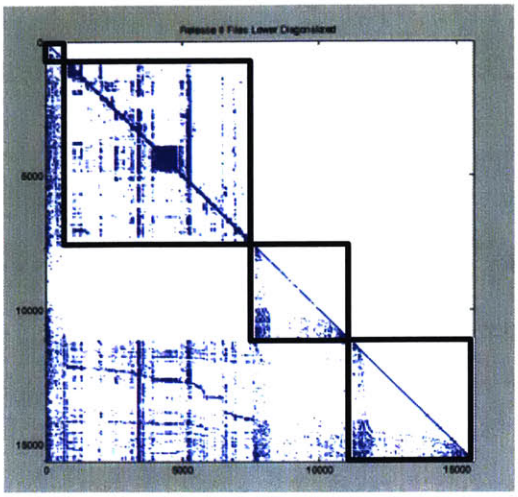

Figure 36: DSM for Release 8

The developer productivity analysis included 178 individuals in its sample. A panel-data approach was employed in that analysis. Developer data was taken on a per-release basis for each of the 8 releases. In total, 478 developer-release observations were included in that analysis. In the analysis of staff turnover, we included 108 individuals in the sample. The turnover analysis did not use a panel approach. In that analysis, information from multiple developer-releases was pooled to give aggregate information for each developer.

6.3.2 Procedures Used To Clean Data Samples

Multiple heuristics were devised to determine if specific files, changes, or developers should be removed from consideration in regression models. These heuristics were devised by examining hundreds of outlier files and changes, and by speaking with developers about potential sources of data or validity problems. Procedures were written to modify the database accordingly.

6.3.2.1 Rules Used to Include or Exclude Files From Consideration in Defect Analysis

Files were removed from consideration for a variety of reasons. In order to be included in the sample:

- Files had to be part of a product sold to customers. Steps were taken to remove files that implemented unit tests, system tests, or non-shipping infrastructure or tools code.

- Files had to be manually written by human developers. Steps were taken to remove code that appeared to be automatically generated rather than written. (Files were removed from the sample if they were abnormally large or if the submission patterns indicated that they were being completely replaced by each submission rather than being incrementally updated.)
- Header files were removed because their contents typically consist of interface descriptions rather than implementation details, and because they are generally much smaller than the files containing executed code. (Header files were present in the DSM during the transitive closure operation and were removed from the sample afterward. This allowed transitive dependency paths to be revealed.)

6.3.2.2 Rules used to Include or Exclude Changes From Consideration in All Studies

Information about changes submitted into the version control system was used to determine the amount of development that went into one file or that was performed by one individual. Procedures were devised to remove questionable changes from this sample. This following set of rules was used to determine which changes to source code files would be excluded from consideration:

- If a single change during a development window added as many lines of code as existed in the file at the time of the release, the change was removed from consideration. When examining these very large submissions, it became clear that the vast majority were caused either by automated changes in file indentation or by the submission of generated (rather than manually written) code.
- If the number of changes submitted by a single individual on one day was above the 99th percentile (135), all submissions by that individual on that day were omitted. While examining these cases it became clear that these "activity spikes" were caused by individuals making automated changes to a large number of files. Examples include submissions in which copyright notices were updated in every source file or small formatting changes were applied across the code.
- If the number of lines of code (LOC) submitted by a single individual on one day was above the 99th percentile (7800), all submissions by that individual on that day were removed from consideration.

When examining these cases, it became clear that these cases involved the renaming of large files or the movement of large pieces of functionality from one file to another.

- If the number of lines of code added and deleted relative to the size of the file is above the 99th percentile (lines added and lines deleted both more than 4 times the number of lines in the file), all changes to that file during the release were removed from consideration. In addition, the file itself was removed from the sample. Files with extremely high levels of code churn often contained automatically generated code – code in which any change to the generation process creates an entirely new file rather than incremental changes.
- If a file was above the 99th percentile on size (3600 lines) all of its changes were removed from the sample of changes and the file itself was removed from the sample of files.

6.3.2.3 Rules used to Include or Exclude Developers From Consideration in Productivity and Turnover studies

The following method was used to determine if a developer-release would be included in the sample:

- A person had to have a title indicating a primary responsibility for developing code and be in a department responsible for developing code in the shipping product. This excludes "testers" who also developed code, but were primarily responsible for creating unit or system tests. It excludes people whose primary responsibility was developing and maintaining internal infrastructure and tools. It also excludes consultants and individuals in training programs.
- A developer had to be employed for the entire duration of a development window to be considered for that time-period.
- For inclusion in the productivity analyses, a developer had to submit at least one change to a core file during every release they would be included in as a developer-release data point. For inclusion in the turnover analysis, a developer had to submit at least one core change during any of the 8 releases studied.

- In the productivity analysis, a developer had to have submitted more than 50% of their lines of code into C++ files represented in the DSM during a release for inclusion. In the turnover analysis, a developer had to submit more than 50% of their lines of code into files in DSMs over the entire time period. Because many developers wrote code in other languages in addition to C++, it was important to limit the developer sample to those who were most directly impacted by the architectural complexity being examined.
- In the productivity analysis, a developer had to be responsible for coding at least 10 features or bug fixes that were tracked in the change tracking system during every release they were included. In the turnover analysis, a developer had to code at least 10 tracked changes over the eight-release window. Because it is important to differentiate between feature development and bug-fixing activities in these studies, we only include individuals with sample sizes that allow us to reasonably estimate the proportion of their work dedicated to each activity.
- At least 70% of a developer's changes must have been deemed "valid" by the procedure outlined above. Developers with more invalid changes were removed from the analysis.

6.4 Structure and Complexity of Files in the Sample

Figure 37 shows DSMs and the distribution of visibility scores for release 7. It illustrates the means by which each C++ file in the sample was classified as peripheral, utility, control, or core. The upper-left DSM is sorted according to the directory structure. Bands of utility files are clearly visible, as are modules along the diagonal. The upper-right DSM is lower-diagonalized. The small box in the upper left contains utility files, followed by core, peripheral, and control file regions. The bottom two panels plot the visibility scores for files in a sorted order. When this is done, the bimodal nature of these visibility scores is apparent. Iron Bridge's C++ codebase has a core-periphery rather than a hierarchical structure. These charts also indicate how files were assigned architectural complexity classifications. The prominent ridge in the middle of each graph was chosen as the demarcation line between "low" and "high" scores for purposes of binning. Once files were assigned to "low" or "high" buckets on both visibility dimensions, classification was

straightforward. Table 5 shows the number of files and the number in each McCabe and architectural complexity classification for each of the 8 releases. Note the growth of the codebase and of the size of the core through time.

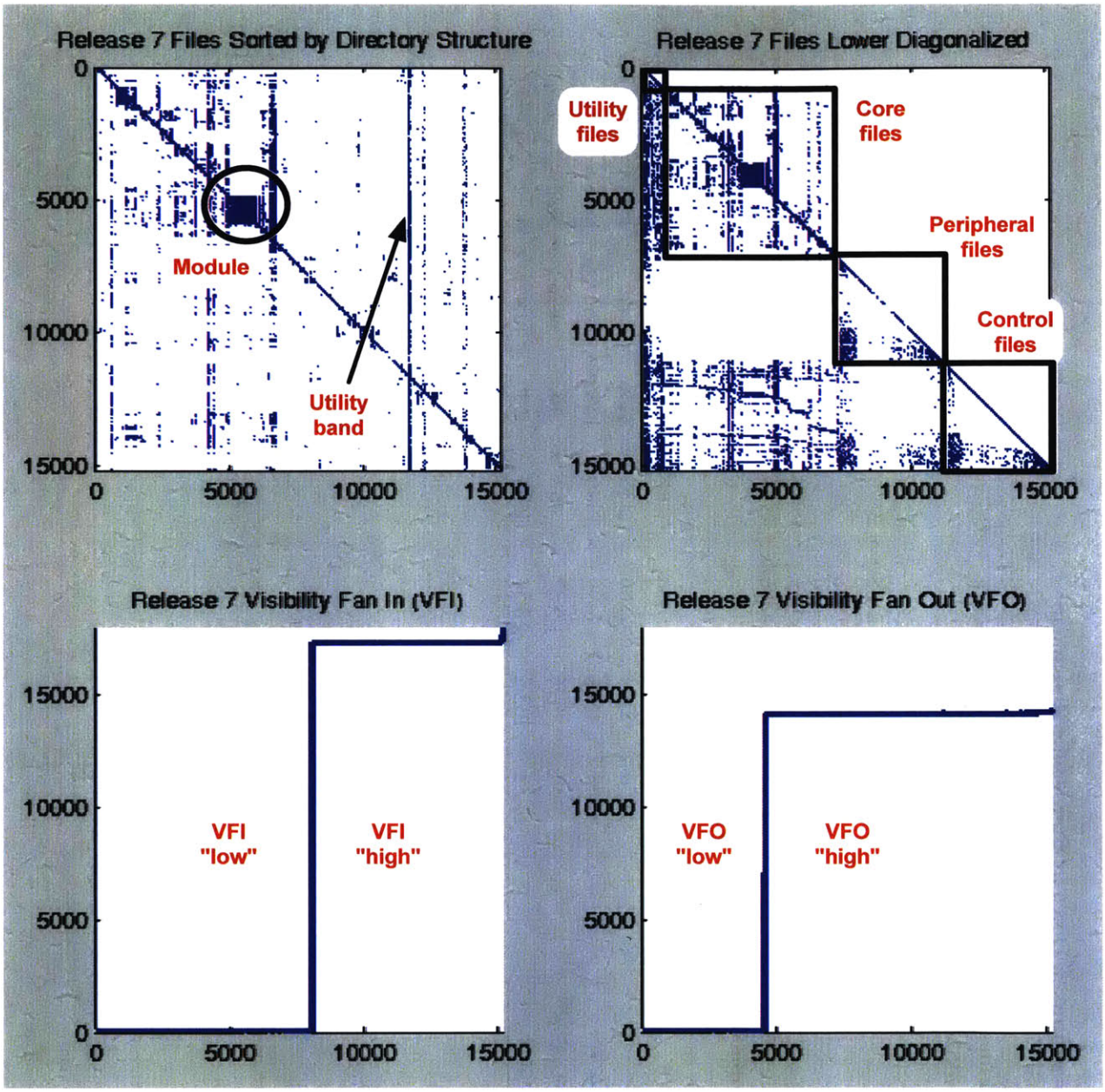

Figure 37: Release 7 DSMs and Visibility Plots

Table 5: File Count Broken Down by Complexity Classification

Release	1	2	3	4	5	6	7	8
Total number of files	9937	10447	10671	11576	12186	12311	13295	13941
Architectural complexity classification								
Peripheral	2691	2305	2158	2193	1835	2981	1975	1901
Utility	543	602	636	915	679	780	685	718
Control	3262	3503	3371	3564	3923	2704	4127	4461
Core	3441	4037	4506	4904	5749	5846	6508	6861
Component complexity classification								
McCabe Low	5973	6321	6534	7282	7702	7904	8691	9241
McCabe Mid	2076	2174	2206	2336	2436	2412	2565	2645
McCabe High	1448	1506	1499	1506	1586	1567	1611	1616
McCabe Very High	440	446	432	452	462	428	428	439
Mean McCabe Score	14.32	13.92	13.50	12.98	12.80	12.42	11.92	11.51

7 Result 1: Link Between Architectural Complexity and Defects

In our first analysis we explore the hypothesis that architecturally complex files experience more defects. We explore the relationship between the architectural complexity of individual files and the amount of bug-fix development activity that occurs in those files. We control for file size, age, and non-bug related code-churn, and McCabe cyclomatic complexity.

7.1 Descriptive Statistics on Files and Complexity

For each of the 8 releases studied, between 9937 and 13941 C++ files were examined. While the last release contained nearly 14,000 files, roughly 2,000 files were created or modified during any given release. These 2,000 files were changed approximately 15,000 times during each release. Those 15,000 changes contained approximately 600,000 line additions or deletions. The number of changes and lines affecting files is highly skewed. Table 6 shows activity during each release. Table 7 shows a variety of mean values related to file size, change size, and activity. The average file had 49 lines changed in it during the typical release cycle. 33 of those lines were changed to implement features and do other tasks, while 16 were changed to fix bugs.

Table 6: Measures of Development Activity During Each Release

Release	1	2	3	4	5	6	7	8
Total number of files	9937	10447	10671	11576	12186	12311	13295	13941
Number of files modified	2139	2844	3778	3414	3649	3452	3702	3150
for features & tasks	1462	2228	2840	2584	2834	2755	2882	2453
for bug fixes	1356	1502	1980	1847	1980	1759	2046	1612
Number of changes	6019	7728	10343	10515	10601	10085	11439	8203
for features & tasks	3153	4643	6210	6630	6273	6101	6473	5055
for bug fixes	2866	3085	4133	3885	4328	3984	4966	3148
Number of lines in files	5627014	5953113	6047870	6277627	6772103	6891070	7096692	7239940
Number of lines modified	360403	509704	632241	645008	713688	596700	674741	504823
for features & tasks	198508	332628	439290	445008	485146	407988	445833	350248
for bug fixes	161895	177076	192951	200000	228542	188712	228908	154575

Table 7: Averages for File Size, Change Size, and Development Activity During Each Release

Release	1	2	3	4	5	6	7	8
Mean file age	3.70	3.78	3.94	4.12	4.21	4.37	4.57	4.61
Mean file size	566.27	569.84	566.76	542.30	555.73	559.75	533.79	519.33
Mean change size	59.88	65.96	61.13	61.34	67.32	59.17	58.99	61.54
Mean changes per file	0.61	0.74	0.97	0.91	0.87	0.82	0.86	0.59
for features & tasks	0.32	0.44	0.58	0.57	0.51	0.50	0.49	0.36
for bug fixes	0.29	0.30	0.39	0.34	0.36	0.32	0.37	0.23
Mean lines changed per file	36.27	48.79	59.25	55.72	58.57	48.47	50.75	36.21
for features & tasks	19.98	31.84	41.17	38.44	39.81	33.14	33.53	25.12
for bug fixes	16.29	16.95	18.08	17.28	18.75	15.33	17.22	11.09

7.2 Modeling Architectural Complexity and Defects

Our first proposition is that all else being equal, files with more architectural complexity will have more defects. In order to analyze the determinants of defects in a file, we construct two statistical models using the C++ source-code file as the unit of analysis. In the first, a count of the number of changes submitted to fix defects was used as the dependent variable. In the second, the number of lines of code modified (added and deleted) to fix defects was used as the dependent variable. The independent variable under study was an indicator of whether each file was categorized as "peripheral", "utility", "control", or "core" according to the previously described architectural complexity classification devised by MacCormack, Baldwin, and Rusnak.

A variety of controls were included for each file including measures of activity that was not bug related (e.g. the addition of new features), a measure of the file's size, a measure of the file's age, and the McCabe cyclomatic complexity metric designed to measure the complexity contained *within* the file.

Parameters for both models were estimated using a Negative Binomial regression due to the count nature of the dependent variable and the fact that the conditional data is overdispersed, invalidating the assumptions of the simpler Poisson model. The Zelig framework was used to run regressions and subsequent simulations to estimate parameters values. [198-201] Data for files from each of the 8 releases was combined, and a dummy variable was included to indicate the release. Nearly 100,000 data points were used in these regressions. Each observation point represents a file-release version, including approximately 12,000 files from each of 8 releases.

Table 8 lists variables included in models in which the number of changes or lines of code required to fix bugs in a file is predicted.

Table 8: Variables Included In Statistical Models Predicting Defect Proneness

Changes submitted to fix bugs	Dependent Variable	Count	The number of changes that were submitted to fix bugs in a development window. If a change was associated with multiple change requests, only some of which were to fix bugs, then only a portion of the change will count as a bug fix.
Lines of code changed to fix bugs	Dependent Variable	Count	The number of lines changes associated with bug fix changes. If a change is allocated proportionally to bug fix and non-bug fix categories, then this line count is allocated proportionally as well
Changes submitted to implement features or do other non-bug related tasks	Control	Count	The number of changes that were submitted for other reasons than to fix bugs. These include changes to implement new features or perform tasks such as refactoring. If a change was associated with multiple change request, only some of which were not to fix bugs, then only a fraction of the change will count here.
Lines of code submitted to implement features or do other non-bug related tasks	Control	Count	The number of lines changed associated with non-bug fix changes. If a change is allocated proportionally to bug fix and non-bug fix categories, then this line count is allocated proportionally as well.
Number of lines of code contained in file	Control	Count	The number of lines of code in a source code file in the shipped (released to customers) version of that file.
Age of file	Control	Float	The age of the file (in years) on the date that of release to customers. Computed by subtracting the date of the file's first change from the release date.
McCabe cyclomatic complexity classification	Control	Categorical	Modified McCabe cyclomatic complexity scores were computed for every function/method in each file. Files were then assigned the score of the highest scoring function/method it contained. Based on these McCabe scores, each file was categorized as having McCabe complexity that was "undefined" (if 0), "low" (if between 1-10), "mid" (if between 11-20), "high" (if between 21-50), and "very high" (if above 50). [112]

Release index	Control	Categorical	Each file observation has dummy variables indicating which of the 8 development windows the observation was made for.
Architectural complexity classification	Independent Variable	Categorical	Based on the logic outlined in previous sections on architectural complexity, each file is categorized as being "peripheral," "utility," "control," or "core" using the transitive closure based techniques developed by MacCormack, Baldwin, and Rusnak [8, 9]

7.3 Regression Models

Table 9 shows the result of regressions predicting the number of *defect correction changes* that go into a file over the course of a release while Table 10 shows the result of regressions predicting the number of *lines of code changed to fix defects* in a file over the course of a release. Both sets of models behave similarly. As one might expect, the number of bug fixes in a file is correlated with the size of the file and the number of non-bug changes. Our intuition about the effect of a file's age is also confirmed. Older files have lower defect density. Also note that the number of defect changes submitted to a file generally increases with McCabe's cyclomatic complexity, but goes down as one moves from files in the "high" range (21-50) to the "untestable" range (>50). This discrepancy is eliminated in the models predicting the number of lines of code changed to fix defects. A few other control variables not shown in the tables were tested as well.[11]

Note that our first proposition holds. Files with high architectural complexity (those considered "core") have defect densities that are significantly higher than files that are "peripheral" and "utility" and somewhat higher than files classified as "control" files. All results are significant at the 0.1% level.

[11] When direct fan-in and fan-out are included in regression models fan-out is significant and fan-in tends not to be. These variables are not included in models presented here because they strongly correlate with visibility scores. Even when they are included, the statistical results presented below still hold.

Table 9: Predicting Number of Changes in a File to Fix Bugs. (Negative Binomial Model)

Parameter	Model 1: controls		Model 2: cyclomatic complexity		Model 3: architectural complexity		Model 4: combined	
LOC in file	0.00051873	***	0.000325008	***	0.000435369	***	0.000232755	***
Non-bug changes	0.47845193	***	0.456561328	***	0.432710048	***	0.407008559	***
File age	-0.05603372	***	-0.067722601	***	-0.052193625	***	-0.065597024	***
Cyclomatic: mid			0.621477523	***			0.626102941	***
Cyclomatic: high			0.792611777	***			0.851977307	***
Cyclomatic: very high			0.607481099	***			0.727127107	***
Architectural: utility					0.303542216	***	0.396496711	***
Architectural: control					0.898747327	***	0.860218904	***
Architectural: core					1.186554887	***	1.226177521	***
Residual Deviance	40809		41387		41004		41672	
Degrees of Freedom	94353		94350		94350		94347	
AIC	108713		107756		107242		106181	
Theta	0.28199		0.30896		0.31141		0.34638	
Std-err	0.00446		0.00509		0.00507		0.00591	
2 x log-lik	-108689		-107726.013		-107212.489		-106144.866	

N = 94364 files observations (from 8 releases)
Dummy variables for each of 8 releases omitted.
*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

Table 10: Predicting LOC Changed in a File to Fix Bugs. (Negative Binomial Model)

Parameter	Model 1: controls		Model 2: cyclomatic complexity		Model 3: architectural complexity		Model 4: combined	
LOC in file	0.00156486	***	0.0011712	***	0.00143183	***	0.00104115	***
Non-bug lines change	0.00372536	***	0.00353601	***	0.00355368	***	0.00335322	***
File age	-0.10050305	***	-0.11730352	***	-0.1026859	***	-0.11853279	***
Cyclomatic: mid			0.774729	***			0.70392074	***
Cyclomatic: high			0.93363115	***			0.95513134	***
Cyclomatic: very high			0.91923347	***			0.96444595	***
Architectural: utility					0.2018549	*	0.35797922	***
Architectural: control					0.94111466	***	0.84721344	***
Architectural: core					1.14823521	***	1.14683088	***
Residual Deviance	30370		30418		30428		30475	
Degrees of Freedom	94353		94350		94350		94347	
AIC	227861		227512		227403		227079	
Theta	0.030212		0.030692		0.030836		0.031295	
Std-err	0.000285		0.00029		0.000291		0.000295	
2 x log-lik	-227837.302		-227482.025		-227373.406		-227042.861	

N = 94364 files observations (from 8 releases)
Dummy variables for each of 8 releases omitted.
*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

7.4 *Interpretation of Results*

Simulations were run to explore effect sizes and look at the response seen in outcome variables in response to changes in the independent variables. In the following simulations, controls were set to their mean values. Lines of code in a file was set to 550, file age was set to 4.2 years, the number of changes submitted to files to implement features and do other non-bug related tasks was set to 0.47, and the number of feature & task lines of code churn was set to 33. Table 11 shows the result of 16 simulations that tested each possible combination of McCabe and architectural complexity classifications to determine the effect of both forms of complexity on the number of bugs appearing in a typical file. Figure 38 and Figure 39 contain plots of the simulation results shown in Table 11.

Table 11: Expected Value For the Number of Bug-Fix Changes in the "Typical" File

		Architectural			
		Peripheral	**Utility**	**Control**	**Core**
McCabe	**Low**	0.059	0.087	0.139	0.200
		(0.007)	(0.011)	(0.016)	(0.023)
	Mid	0.110	0.163	0.259	0.374
		(0.013)	(0.021)	(0.030)	(0.044)
	High	0.138	0.205	0.325	0.469
		(0.017)	(0.026)	(0.038)	(0.055)
	Very high	0.121	0.181	0.287	0.414
		(0.015)	(0.024)	(0.035)	(0.051)

Note: standard deviation in parentheses

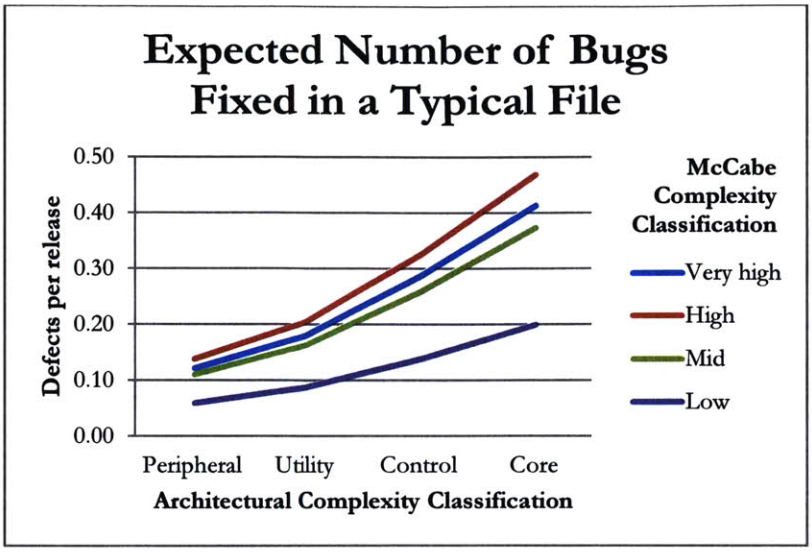

Figure 38: Expected Number of Bugs Fixed in a File (1)

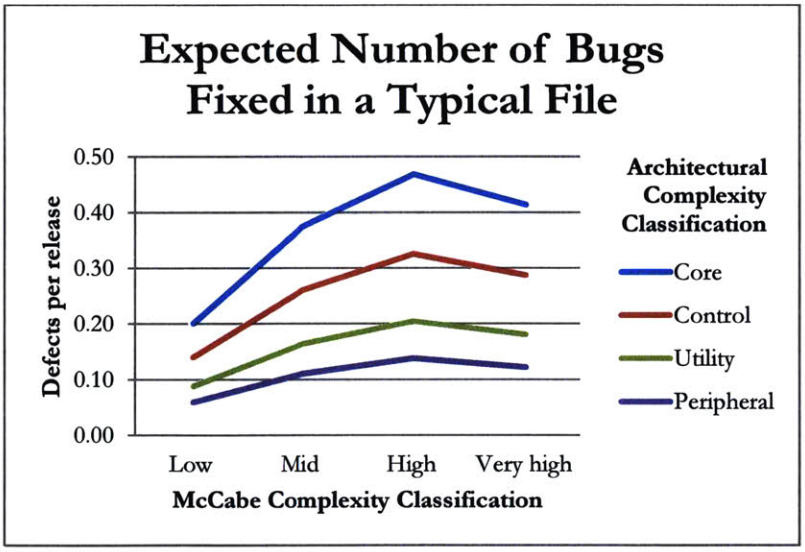

Figure 39: Expected Number of Bugs Fixed in a File (2)

Both architectural and McCabe's cyclomatic complexity seems to matter a great deal. Note that files with high McCabe scores are expected to have 2.1 times as many bug fixes as files with low McCabe scores. Changes in architectural complexity have an impact of roughly the same order of magnitude. When compared against the periphery, utility files have 48% more defects, control files have 2.4 times as many defects, and core files have 3.4 times as many defects. When both types of complexity are considered in

combination, the effects are quite large. A core file with a high McCabe score is expected to have 8 times as many defects as a peripheral file with a low McCabe score.

Because changes vary dramatically in size, simulations were run to estimate the expected number of lines of code changed to fix defects as well. Table 12, Figure 40, and Figure 41 show the results of these simulations.

Table 12: Expected Value For the Number of Lines Submitted to Fix Bugs in "Typical" File

		Architectural			
		Peripheral	**Utility**	**Control**	**Core**
McCabe	**Low**	2.220	3.183	5.178	6.985
		(0.366)	(0.563)	(0.844)	(1.130)
	Mid	4.490	6.441	10.470	14.132
		(0.756)	(1.166)	(1.737)	(2.336)
	High	5.776	8.276	13.473	18.182
		(0.993)	(1.524)	(2.286)	(3.080)
	Very high	5.853	8.379	13.658	18.413
		(1.141)	(1.711)	(2.630)	(3.562)

Note: standard deviation in parentheses

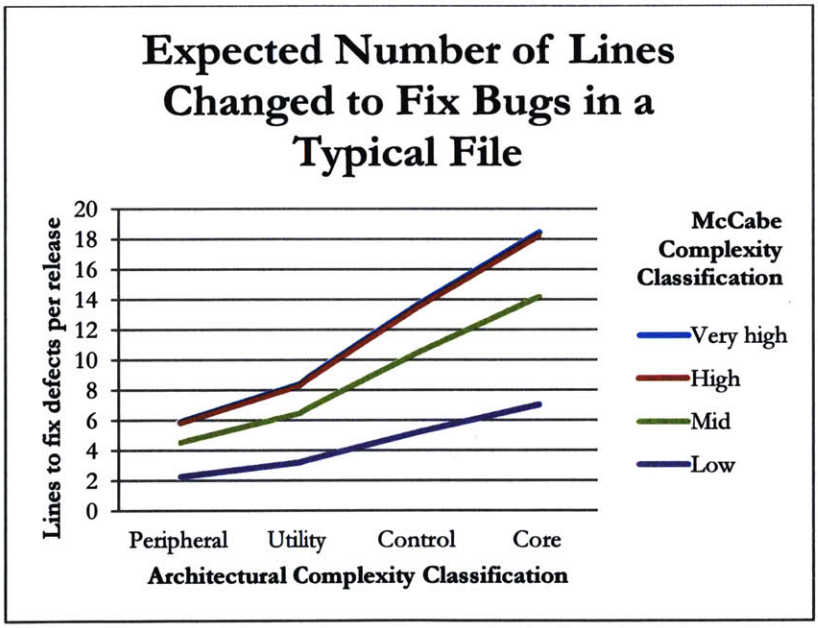

Figure 40: Expected Number of Lines to Fix Bugs in a File (1)

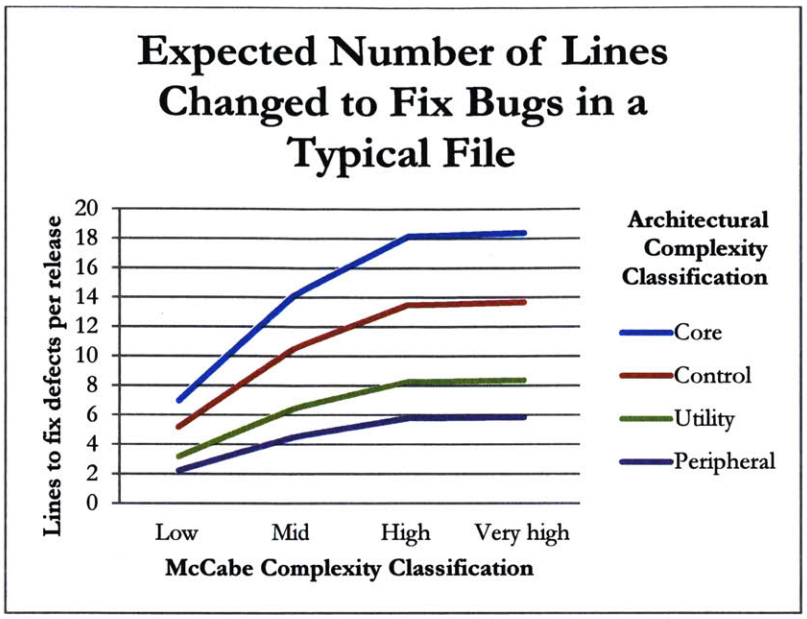

Figure 41: Expected Number of Lines to Fix Bugs in a File (2)

The overall picture remains similar. Core files are expected to have 3.1 times as many lines changed to fix bugs as peripheral files and files with high McCabe scores are expected to have 2.6 times as many bug fix lines as those with low McCabe scores. All else equal, core files with very high complexity are expected to have 8.3 times as many lines changed to correct defects as peripheral files with low McCabe scores.

8 Result 2: Link Between Architectural Complexity and Productivity

In our second analysis we explore the hypothesis that when developers work in architecturally complex files their productivity is impaired. We explore the relationship between the fraction of lines of code an individual contributes to "core" files during a release and their total number of lines of code produced during that release. In these models we control for a variety of other factors that could each be considered alternative explanations for why a developer's productivity may have declined. Controls tested include a developer's time with the firm, managerial status, fraction of activity working in new (rather than legacy) code, faction of activity spent fixing bugs, and fraction of activity working in files with high McCabe Cylomatic complexity. The goal of these models is to determine if architectural complexity has a significant impact on the productivity of developers, even when weighed against these alternative explanations.

8.1 Descriptive Statistics on Developer Productivity

The sample of developers used to explore productivity included 178 people who wrote a majority of their code in the C++ portion of Iron Bridge's codebase. Because 8 releases were measured, developers had the opportunity to appear in the dataset up to 8 times. Due to repeats, this sample consisted of 478 distinct developer-release observations for use in panel-data analysis. The sample included 388 observations of individual contributors and 90 observations of managers. The median amount of time a developer-release had been with the company was slightly over 4 years. Over the course of 8 releases, the developer-releases observed produced nearly 2 million lines of code as measured by the *addition* and *deletion* of lines in file changes. Of these 2 million lines produced, 1.1 million were created to implement features or perform some other non-bug related task such as refactoring. 800,000 lines were produced to fix bugs. Table 13 shows the number of developers in each sample, information about their tenure and managerial status, and information about the lines of code they produced on average to implement features and fix bugs.

The median developer produced 3,200 lines of code per release, while the mean developer produced 4,000 lines changed over the course of a release. Productivity between individuals was highly skewed. The top quartile has approximately 10 times the productivity as the bottom quartile. (This is a striking but generally understood phenomenon.)

Table 14 breaks down development activity by the type of work being performed (feature work vs. bug fix) and the location of that work. Approximately half of the lines coded are submitted to new files and half to legacy files. One third of activity takes place in files with McCabe scores of *high* or *very high*. Three quarters of activity occurs in core files.

Table 13: Developers and Activity in Each Release

Release	1	2	3	4	5	6	7	8
Developers in sample	35	46	59	67	64	67	69	71
number of managers	8	9	15	13	12	13	12	8
number of ind. contributors	27	37	44	54	52	54	57	63
Mean time with company	4.5	4.5	5.2	4.8	4.7	5.5	5.9	5.5
Changes produced per developer	69	79	87	70	86	82	69	68
Lines produced per developer	3357	4214	4443	3681	4967	4406	3361	3676
for features & tasks	1233	2168	2279	2231	3226	2867	1838	2260
for bug fixes	2118	2038	2154	1440	1735	1531	1517	1410

Table 14: Activity For the Average Developer by Task and Location in Codebase For Each Release

Release	1	2	3	4	5	6	7	8
Developers in sample	35	46	59	67	64	67	69	71
Lines produced per developer	3357	4214	4443	3681	4967	4406	3361	3676
C++ lines produced per developer	2471	3030	3160	2784	3622	3261	2582	2763
Type of work								
for features & tasks	1233	2168	2279	2231	3226	2867	1838	2260
for bug fixes	2118	2038	2154	1440	1735	1531	1517	1410
% lines for bug fixes	63%	48%	48%	39%	35%	35%	45%	38%
Age of file								
old file (>= 2 years)	2006	2235	2104	1704	2420	2204	1575	1791
new file (< 2 years)	1319	1882	2274	1951	2518	2162	1771	1828
% lines in new files	39%	45%	51%	53%	51%	49%	53%	50%
Component complexity								
low McCabe (< 21)	1901	2550	2742	2428	2737	2729	2166	2198
high McCabe (>= 21)	1354	1509	1614	1215	2171	1608	1148	1407
% lines high McCabe file	40%	36%	36%	33%	44%	37%	34%	38%
Architectural complexity								
peripheral file	241	47	142	112	102	147	130	20
utility file	33	12	8	58	86	73	21	28
control file	609	402	737	733	761	614	435	736
core file	1511	2510	2242	1856	2635	2392	1971	1976
% lines in core file	61%	83%	71%	67%	73%	73%	76%	72%

The histograms in Figure 42 and Figure 43 show the distribution of developer contributions to files with high McCabe complexity scores (those >20) and files with high architectural complexity (those in the core). Note that over 200 developer-releases make above 90% of their contributions to core files.

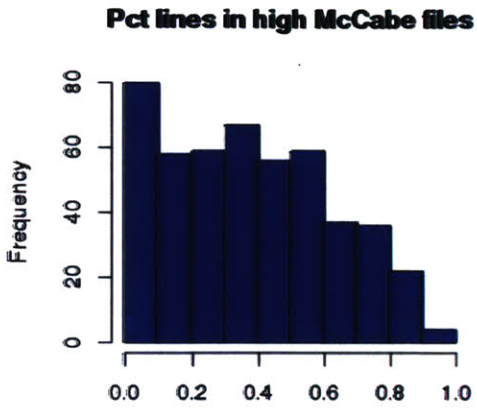

Figure 42: Histogram of Activity in High McCabe Files

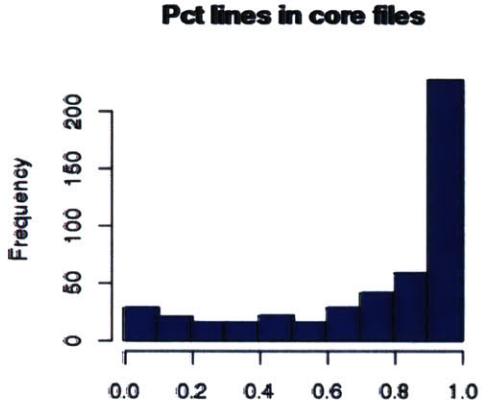

Figure 43: Histogram of Activity in Core

8.2 Modeling Architectural Complexity and Developer Productivity

Our second proposition is that architectural complexity negatively impacts productivity. In order to analyze the determinants of developer productivity, we construct three statistical models using the software developer as the unit of analysis. In the first model, the dependent variable is the total number of lines produced by an individual to implement features or do other non bug-related tasks (the number of bug-fix lines is included as a control). In the second model, the dependent variable is the number of lines of code produced by that individual to fix defects (the number of lines that person produced for purposes other than to fix bugs is included as a control). In the third model, the dependent variable is the total number of lines of code produced by an individual during a given release window for features, bug fixes, and other tasks (with the percentage of lines dedicated to bug-fixes is included as a control). The independent variable under study in all three sets of models is the percentage of lines a person submitted to "core" files. This measure is designed to estimate the amount of work the individual does in files with high levels of architectural complexity.

The purpose of the first model is to determine how productive individuals were when implementing features or doing other non-bug related tasks in files with varying levels of architectural complexity. The second model is used to estimate how productive individuals were while fixing bugs in files with different levels of architectural complexity. The purpose of the third model is to determine the impact of architectural complexity on the overall productivity (in terms of total lines produced) of the individual.

In each of these models, we use a panel-data approach that aims to control for individual differences in developer productivity. Dummy variables are included for each of the 8 releases and each of the individual developers. By including these dummy variables, we construct regressions that capture changes in productivity due to complexity *within* the individual rather than *between* them. Put another way, these regressions are designed to determine if individuals were less productive during releases during which they worked in more complex code rather than to determine if a group of people working in more complex code is less productive than a group working in less complex code. (Even if the former statement is true, the latter may not be the case if highly skilled developers are disproportionately allocated to the core.)

A variety of controls were included for the individual including length of employment, managerial status, the amount of work done in new (rather than legacy) files and amount of work done in files with high levels of McCabe cyclomatic complexity.[12]

Parameters for all models were estimated using a Negative Binomial regression due to the count nature of the dependent variable and the fact that the conditional data is overdispersed, invalidating the assumptions of the simpler Poisson model. The Zelig framework was used to run regressions and subsequent simulations to estimate parameter values. [198-201]

---

[12] Control variables representing the proportion of lines submitted to files with "high" direct fan-in and direct-fan out were not included due to the fact that they are highly correlated with visibility scores and because the extreme levels of skew in their distributions (the distribution of fan-in scores fit a power-law distribution for instance) make it difficult to obtain or interpret results. Due to the highly skewed nature of the data, the sample of files with "high" direct visibility scores is insufficiently small.

The following table lists variables used in the three sets of models:

Table 15: Variables Included In Statistical Models Predicting Developer Productivity

Variable	Purpose	Type	Description
Lines of code produced to implement features or perform other non-bug related tasks	Dependent Variable	Count	The number of lines of code produced by a developer to implement features or do some other non-bug related task. If a change was associated with multiple change requests, some of which were to fix bugs, then only a portion of the change will count as a bug fix, and the rest will be considered a feature or task. The number of lines of code in a change will be allocated proportionally based on the proportion allocated to bugs and non-bugs.
Lines of code produced to fix bugs	Dependent Variable	Count	The number of lines of code produced by a developer to fix bugs during a release window. If a change was associated with multiple change requests, only some of which were to fix bugs, then only a portion of the change will count as a bug fix. The number of lines of code in a change will be allocated proportionally based on the proportion allocated to bugs and non-bugs.
Lines of code produced to fix bugs, implement features, or perform other tasks	Dependent Variable	Count	The number of lines of code produced by a developer during a release window. All changes submitted by the developer during the release window to fix bugs, implement features, or do other tasks are considered and the lines added plus the lines deleted in each of those changes are totaled.
Years employed	Control	Float	The time employed (in years) of the developer on the date of the software release. Computed by subtracting the developer's hire date from the release date.
Is manager?	Control	Boolean	Boolean variable indicating whether a developer is a manager on the release date.
Percent of lines submitted to new files	Control	Percent	A file is considered to be a "new file" if it is less than two years old. File age is computed by subtracting the date of the file's first change from the release date. The percentage of lines submitted to new files is computed by determining the proportion of lines produced by a developer during a release that modified new files.

Percent of lines submitted to fix bugs	Control	Percent	The percentage of lines of code that were produced by a developer to fix bugs. If a change was associated with multiple change requests, only some of which were to fix bugs, then only a portion of the change will count as a bug fix. The number of lines of code in a change will be allocated proportionally based on the proportion allocated to bugs and non-bugs.
Percent of lines submitted into files with "high" or "very high" McCabe classifications	Control	Percent	A file is considered to have a "high" or "very high" McCabe score if the Modified cyclomatic complexity of the most complex function/method is above 20. The percentage of lines submitted to files with "high" or "very high" McCabe scores is computed by determining the proportion of lines produced by a developer during a release that modified those files. [112]
Release index	Control	Categorical	Each file observation has dummy variables indicating which of the 8 development windows the observation was made for.
Login	Panel	Categorical	Each developer login is used as a dummy variable. This variable is used in fixed-effects panel-data models.
Percent of lines submitted to core files	Independent Variable	Percent	Determined by finding the proportion of lines produced that were submitted to files given the architectural complexity classification of "core" using the transitive closure based techniques developed by MacCormack, Baldwin, and Rusnak [8, 9]

8.3 Regression Models

The results for regressions predicting the productivity of an individual during a release window are shown in Table 16, Table 17, and Table 18. Note that while each of these regressions contained dummy variables for the release and the individual, these dummies were omitted from tables.

Table 16 shows results for regressions in which the productivity of individuals implementing features and doing other non-bug tasks is predicted. (Each model contained the lines produced to fix bugs as a control.) Developers are much more productive when implementing features and working in new (rather than legacy) files. They are less productive when implementing features and working in files with high

McCabe cyclomatic complexity. After all controls are included in the model, developers are also found to be less productive when developing features and working in core files. This result is significant at the 5% level.

Table 17 shows results for regression in which the productivity of individuals correcting defects during a release is predicted. (Each model contained the lines produced for feature work & other non-bug related tasks as a control.) Developers are shown to be much more productive when implementing bug fixes if they are working in new (rather than legacy files). Developers with more experience (those with longer tenures at the firm) are more productive when fixing bugs than less experienced developers. The ability to effectively fix bugs appears to grow with experience more than feature-development productivity. Developers are also found to be much less productive when fixing bugs in the core than when fixing bugs elsewhere. This result is significant at the 0.1% level. Working in the core appears to have a stronger negative impact on the productivity of those fixing bugs than those implementing features.

Table 18 shows results for regressions in which total developer productivity (features and bug-fixes combined) during a release is predicted. Employees with more years of experience were more productive. While this is not surprising, it is interesting to note that the strength of the effect grew as other controls were added, suggesting that as employees gain experience, they are moved into more complex regions of the codebase, work more on legacy code, or work on harder bug fixes, thereby suppressing the productivity gains they would have if left in more approachable regions of the codebase. When developers work in new files (those less than 2 years old) they are much more productive. This suggests that new feature development is easier than maintaining legacy code. As might be expected, developers are much less productive when they are working on bug fixes than when they are implementing features. Surprisingly, McCabe cyclomatic complexity had no statistically significant impact on overall developer productivity, however.

Note that our second proposition holds. During time periods in which an individual worked in core files, the number of lines of code they produced declined. Architectural complexity has a significant negative impact on a developer's overall productivity. This result is significant at the 1% level.

Table 16: Predicting LOC Produced per Developer to Implement Features For One Release (Neg Binomial Panel Data Model)

Parameter	Model 1: developer attributes		Model 2: type of work		Model 3: cyclomatic complexity		Model 4: all controls		Model 5: architectural complexity		Model 6: combined	
Lines for bug fixes	-7.12E-05		-6.84E-05		-0.00005961		-6.74E-05		-0.00007681	.	-7.84E-05	.
Log(years employed)	2.80E-01						4.93E-01				4.84E-01	
Is manager?	-2.83E-01						-2.52E-01				-2.93E-01	
Pct lines in new files			1.80E+00	***			1.70E+00	***			1.71E+00	***
Pct lines high cyclomatic					-1.16601056	***	-6.48E-01	.			-6.13E-01	.
Pct lines in core									-0.61094326	.	-6.19E-01	*
Residual Deviance	560.7696		558.4638		560.5962		558.324		560.7079		558.1296	
Degrees of Freedom	290		291		291		288		291		287	
AIC	8170.656		8135.143		8162.143		8136.784		8166.867		8135.753	
Theta	0.8512584		0.902979		0.8614868		0.910243		0.8540307		0.915163	
Std-err	0.05032488		0.05380377		0.05103293		0.0543059		0.05051371		0.05464003	
2 x log-lik	-7792.656		-7759.143		-7786.143		-7754.784		-7790.867		-7751.753	

N = 478 developer/releases

Dummy variables for each of 8 releases omitted. Dummy variables for each of 178 developers omitted.

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

Table 17: Predicting LOC Produced per Developer to Fix Defects For One Release (Neg Binomial Panel Data Model)

Parameter	Model 1: developer attributes		Model 2: type of work		Model 3: cyclomatic complexity		Model 4: all controls		Model 5: architectural complexity		Model 6: combined	
Lines for features & tasks	-0.00002894	.	-0.00003436	*	-0.00002287		-0.00003286	.	-0.00003183	.	-0.00003869	*
Log(years employed)	0.41418368	*					0.47664084	**			0.51248987	**
Is manager?	-0.00925084						0.00234582				-0.05832787	
Pct lines in new files			0.21861235				0.31967026	*			0.35717162	*
Pct lines high cyclomatic					0.33677149	.	0.44466236	*			0.49647843	**
Pct lines in core									-0.48544331	**	-0.56740321	***
Residual Deviance	509.5084		509.5916		509.5686		509.208		509.4193		508.8542	
Degrees of Freedom	290		291		291		288		291		287	
AIC	7934.951		7935.576		7934.91		7931.536		7930.616		7923.786	
Theta	2.916188		2.901444		2.905165		2.957875		2.929278		3.013898	
Std-err	0.1808552		0.1798761		0.1801246		0.183591		0.1817136		0.1872798	
2 x log-lik	-7556.951		-7559.576		-7558.91		-7549.536		-7554.616		-7539.786	

N = 478 developer/releases

Dummy variables for each of 8 releases omitted. Dummy variables for each of 178 developers omitted.

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

Table 18: Predicting LOC Produced per Developer For One Release. (Neg Binomial Panel Data Model)

Parameter	Model 1: developer attributes	Model 2: type of work	Model 3: cyclomatic complexity	Model 4: all controls	Model 5: architectural complexity	Model 6: combined
Log(years employed)	0.233711			0.32335 *		0.336831 *
Is manager?	-0.12336			-0.0397		-0.081573
Pct lines in new files		0.524365 ***		0.56379 ***		0.578597 ***
Pct lines for bugs		-1.075852 ***		-1.08704 ***		-1.076668 ***
Pct lines high cyclomatic			-0.312413 .	0.14775		0.171612
Pct lines in core					-0.417167 **	-0.399158 **
Residual Deviance	500.67	495.95	500.63	495.81	500.51	495.6
Degrees of Freedom	291	291	292	288	292	287
AIC	8752.2	8624.5	8749.3	8625.3	8745.8	8619.4
Theta	3.521	4.51	3.527	4.557	3.551	4.628
Std-err	0.218	0.283	0.219	0.286	0.22	0.29
2 x log-lik	-8376.187	-8248.529	-8375.327	-8243.285	-8371.818	-8235.376

N = 478 developer/releases

Dummy variables for each of 8 releases omitted. Dummy variables for each of 178 developers omitted.

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

8.4 *Interpretation of Results*

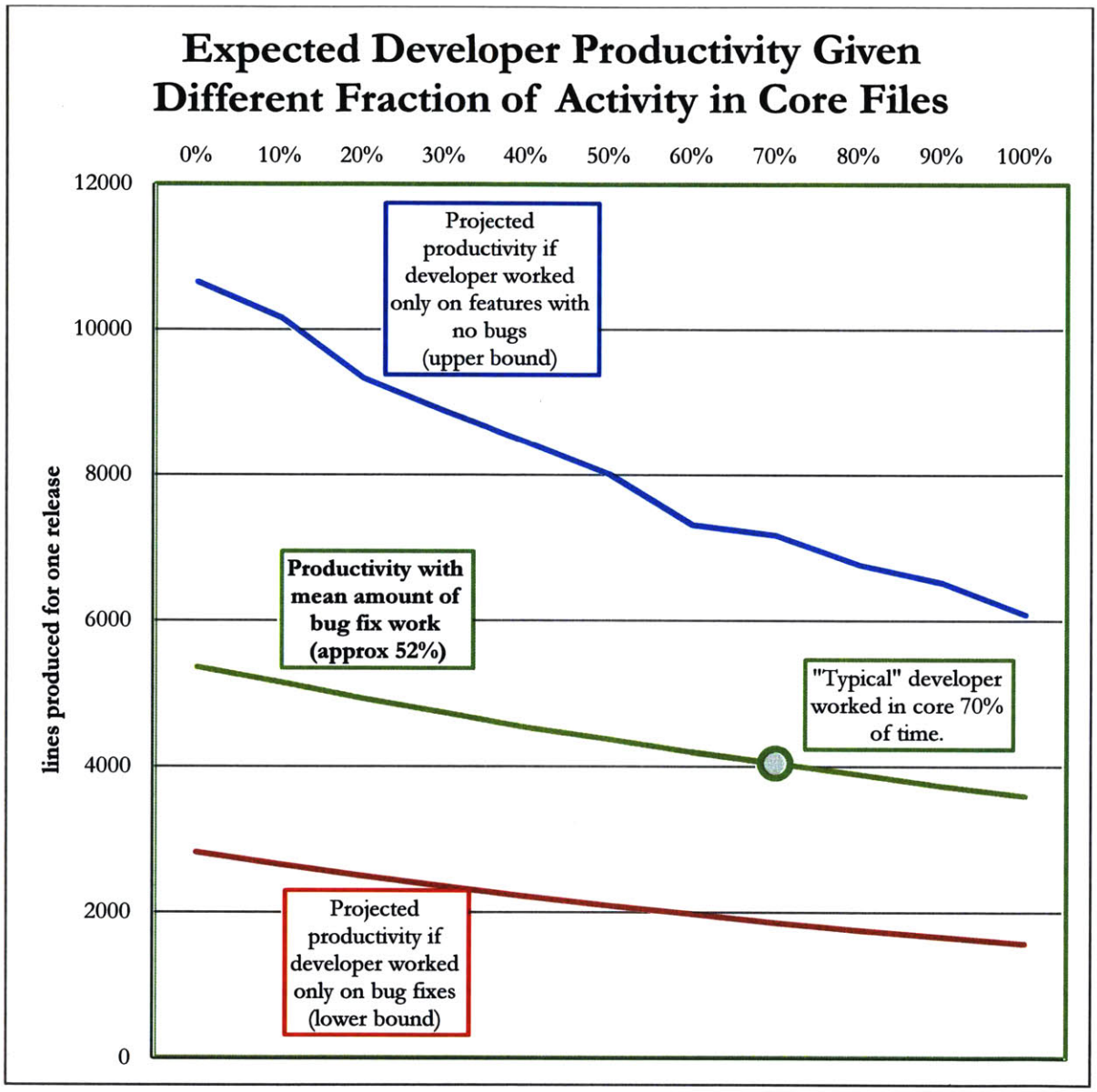

Figure 44: Developer Productivity and Architectural Complexity

Three sets of simulations were run to determine the response of the outcome variables (the number of lines that the typical developer would produce during a release) to changes in a developer's percentage of activity in the core. In these simulations, most control variables were set to their mean values. The "typical" developer was selected by choosing the individual owning the median-valued person-specific dummy variable

coefficient. Managerial status was set to *false*. Length of employment was set to the mean value of 5.1 years. The percent of lines contributed by this prototypical developer to new files (those under 2 years of age) was set to 44%. The percent of lines contributed to files with high McCabe cyclomatic complexity (with scores above 20) was set to 38%.

A simulation was run to predict the expected productivity that would be achieved if 100% of a developer's effort could be dedicated to implementing new features or doing other non-bug related tasks and no bug-fixing were necessary. This simulation used the full version of the regression model shown in Table 16. (In addition to setting controls to the values just described, the control variable *lines for bug fixes* was set to 0.) The blue line shown in Figure 44 shows the result of varying the percent of lines submitted to core files on feature productivity for this hypothetical (and blessed) individual. All else being equal, the developer working only on features in the periphery would produce 10655 lines of changes during a release. This same individual would only produce 6083 lines for features when positioned in the core.

A second simulation was run to predict the expected productivity that would be achieved if a developer was forced to dedicate 100% of his effort to fixing bugs. This simulation used the full version of the regression model shown in Table 17. (In addition to setting controls to the values previously described, the control variable *lines for features and tasks* was set to 0.) The red line in Figure 44 shows the response of bug-fix productivity when the percent of lines submitted to core is varied. All else being equal, if our unlucky developer working only on bug fixes is in the periphery, 2815 lines of changes would be produced. This same individual would produce only 1567 lines if positioned in the core.

Our third (and final) simulation was run to predict the expected productivity that would be achieved if a developer spent the typical proportion of time split between feature work and bug fixes. This simulation used the full version of the regression model shown in Table 18. (In addition to setting controls to the values previously described, the control variable *pct lines for bugs* was set to the mean value of 52%.) The green line in Figure 44 shows the impact of varying the percent of lines submitted to core files on overall productivity. All

else being equal, the typical developer working in the periphery will produce 5359 lines of changes during a release while this same individual would only produce 3594 lines if positioned in the core.

Our results suggest that the effect that architectural complexity has on developer productivity is quite strong. All else being equal, architectural complexity accounts for a near halving of the lines of code that can be produced by an individual in any given release as one moves from the periphery to the core. At Iron Bridge, approximately 70% of lines produced go into core files. Based on the contents of Figure 44, one might speculate that a refactoring that shank the core such that only 50% of average developer's lines produced went into core files would yield a productivity increase of 10%. In addition, one should remember that "all else" is not actually equal. The strong relationship between defects and complexity found in the previous chapter tells us that developers in the periphery will spend more time than average developing features on the "blue curve" in Figure 44 while developers in the core will spend more time on the "red curve" contending with bugs. The productivity loss that results from moving form the periphery to the core may therefore be substantially greater than 50%. If a refactoring effort successfully shrank the core it would move the average reduce the effort the average developer dedicates to development in the core (moving them further towards the left on the green line) and reduce the number of bugs a developer had to contend with (increasing the amount of time spent on the blue curve rather than the red curve). The combined effect could lead to significant productivity gains.

Table 19: Predicted Number of Lines Produced by Average Developer at Various Architectural Complexity Levels if Only Implementing Features or Doing Other Non-Bug Related Tasks

Percent of lines submitted to core	Expected number of lines produced	Standard deviation	CI lower bound (2.5%)	CI upper bound (97.5%)
0%	10655	55875	157	59604
10%	10160	39304	155	57549
20%	9333	39720	143	51680
30%	8879	34418	135	49307
40%	8453	32600	130	47562
50%	8013	34744	124	44698
60%	7317	28836	116	40804
70%	7174	31413	110	39910
80%	6767	28421	103	37766
90%	6520	28799	99	36694
100%	6083	23883	93	34388

Table 20: Predicted Number of Lines Produced by Average Developer at Various Architectural Complexity Levels if Only Fixing Bugs

Percent of lines submitted to core	Expected number of lines produced	Standard deviation	CI lower bound (2.5%)	CI upper bound (97.5%)
0%	2815	1740	833	7322
10%	2647	1602	799	6799
20%	2490	1503	759	6395
30%	2349	1402	725	5979
40%	2212	1305	689	5610
50%	2088	1226	655	5269
60%	1974	1158	624	4968
70%	1854	1082	587	4636
80%	1749	1021	557	4393
90%	1657	970	528	4171
100%	1567	912	499	3931

Table 21: Predicted Number of Lines Produced by Average Developer at Various Architectural Complexity Levels

Percent of lines submitted to core	Expected number of lines produced	Standard deviation	CI lower bound (2.5%)	CI upper bound (97.5%)
0%	5359	2870	1748	12662
10%	5148	2764	1674	12200
20%	4928	2650	1609	11646
30%	4734	2552	1543	11217
40%	4534	2430	1473	10689
50%	4373	2376	1411	10395
60%	4197	2279	1355	9990
70%	4043	2211	1299	9681
80%	3892	2133	1246	9309
90%	3730	2055	1183	8955
100%	3594	1998	1136	8694

9 Result 3: Link Between Architectural Complexity and Staff Turnover

In our third analysis we explore the proposition that developers working in architecturally complex files have a greater likelihood of leaving the firm (either voluntarily or involuntarily). We explore the relationship between the fraction of lines of code an individual contributes to "core" files (relative to peers) and whether that person left the firm during the 8 release windows studied or during the subsequent 4 years. In these models we control for a variety of factors, each of which could be considered an alternative explanation for why a developer might have left the firm. Controls tested include a developer's prior length of employment with the firm, managerial status, fraction of activity working in new (rather than legacy) code, faction of activity spent fixing bugs, and fraction of activity working in files with high McCabe cyclomatic complexity. The goal of these models is to determine if architectural complexity has a significant impact on staff turnover, even when weighed against these other viable explanations for attrition.

9.1 Descriptive Statistics on Developer Turnover

The sample of software developers used to explore turnover included 108 people who wrote code in the C++ portion of Iron Bridge's codebase during the 8 releases studied. Each person appears in the dataset only once. In order to be included in the sample, a developer must have contributed to the product for at least one of those 8 releases. The sample included developers who were already employed at the beginning of the window and people who joined the firm at some point during the eight development periods under study. Of the 108 developers in the sample, 62 were present during the first release while the rest joined later. Developer-specific data from multiple releases was pooled. Developers were then divided into groups of *stayers* and *leavers*. Stayers were those who remained employed for up to 4 years beyond the last release measured. Leavers were those who left the firm (either voluntarily or involuntarily) during the 8 development windows or during the subsequent 4 years. There were 91 stayers and 17 leavers in the sample.

Table 22: Comparing the Population of Stayers and Leavers

	Mean		Median		Standard deviation	
	stay	*leave*	*stay*	*leave*	*stay*	*leave*
Years employed (window start)	3.88	2.68	1.92	0.92	3.90	3.59
Lines produced per release	4114	2962	3343	2284	2630	2224
Percent lines to fix bugs	43.1%	48.7%	38.4%	42.2%	22.5%	21.3%
Percent lines in new files	47.8%	45.1%	44.3%	41.3%	23.1%	25.6%
Percent lines in high McCabe files	35.4%	35.1%	35.6%	34.0%	19.2%	19.6%
Percent lines in core files	71.1%	87.3%	80.1%	93.8%	27.4%	17.4%
Years employed rank	51.1%	42.4%	52.8%	41.7%	30.0%	27.1%
Lines produced per release rank	52.6%	39.1%	52.8%	36.1%	28.8%	28.3%
Bug fraction rank	49.2%	57.3%	49.1%	55.6%	29.2%	27.7%
New file fraction rank	51.1%	46.9%	51.9%	43.5%	28.6%	31.5%
High McCabe rank	50.5%	50.1%	50.9%	47.2%	29.1%	29.6%
Core file fraction rank	47.7%	65.2%	45.4%	71.3%	29.3%	23.0%

Stay: N=91, Leave: N=17

Table 22 shows differences between these two populations. Note that those who stayed had been employed for longer. The mean stayer had been employed for 3.88 years prior to her first sampled release cycle. The mean leaver had only been employed for 2.7 years. The mean stayer produced more than 4000 lines of code per release on average, while the mean leaver produced less than 3000. Stayers were slightly more likely to work on features rather than bug fixes, work in new rather than legacy code, and did not spend as much time working in "core" files.

9.2 Modeling Architectural Complexity and Staff Turnover

Our third proposition is that developers working in regions of the codebase with higher levels of architectural complexity will have higher levels of turnover. In order to analyze the determinants of developer turnover, we constructed a set of statistical models using a Boolean dependent variable indicating whether the developer was a *stayer* or a *leaver*.

A variety of controls were included for each individual including an indicator of whether that person was ever a manager during the 8 releases studied. People were also ranked based on their length of employment, the number of lines they produced per release cycle, the fraction of their activity that went into fixing bugs, the fraction of their activity that went into developing new files, and the faction of their activity in files with high McCabe cyclomatic complexity.[13] Rankings between developers were used in this analysis (rather than raw percentages) because we are testing relative propensity to leave the firm (vs. absolute productivity.)

Each control used in our models offers an alternative explanation for why a person might stay or leave the firm (both voluntarily or involuntarily). Employees with longer tenures should be more likely to stay, both because they have already demonstrated a desire to stay with the firm and because younger individuals tend to be more mobile. Managers should be more likely to stay for similar reasons. Individuals who are more productive should be more likely to stay for a variety of reasons if the culture rewards productivity. Individuals fixing bugs may feel that their jobs are less rewarding. Individuals working on new features (more likely in new files) may feel more rewarded than those working to maintain legacy functionality (more likely in older files). Finally, individuals working in files with high McCabe cyclomatic complexity may leave because they do not find it rewarding to work in code that is more fragile or error prone.

Each developer in the sample was given a ranking based on the fraction of their code that went into core files. This ranking was used as the independent variable under study. Logistic regressions were used here due to the binary nature of the dependent variable. Variables used in those models are presented in Table 23.

---

[13] Control variables representing the proportion of lines submitted to files with "high" direct fan-in and direct-fan out were not included due to the fact that they are highly correlated with visibility scores and because the extreme levels of skew in their distributions (the distribution of fan-in scores fit a power-law distribution for instance) make it difficult to obtain or interpret results. Due to the highly skewed nature of the data, the sample of files with "high" direct visibility scores is insufficiently small.

Table 23: Variables Included in Statistical Model Predicting Developer Turnover

Variable	Purpose	Type	Description
Developer left the firm?	Dependent Variable	Boolean	If a developer left the firm (either voluntarily or involuntarily) during the 8 development windows under study or within the subsequent 4 years, the developer is defined as a *leaver*. If still employed, the developer is defined as a *stayer*.
Was developer ever a manager?	Control	Boolean	Boolean variable indicating whether a developer was a manager at any point in the 8 development windows under observation.
Employment length rank	Control	Rank	The percentile rank of a developer's length of employment during the first release in which the developer appears in the data set.
Lines produced rank	Control	Rank	The percentile rank of a developer's total number of lines produced divided by the number of releases in which they were observed.
Bug fraction rank	Control	Rank	The percentile rank of a developer's fraction of lines submitted to fix bugs over all development windows in which they were observed.
New file fraction rank	Control	Rank	The percentile rank of a developer's fraction of lines submitted into "new" files over all the releases in which they were observed. A file is considered to be a "new file" if it is less than two years old. File age is computed by subtracting the date of the file's first change from the release date. The percentage of lines submitted to new files is computed by determining the proportion of lines produced by a developer during a release that modified new files.
High McCabe rank	Control	Rank	The percentile rank of a developer's fraction of lines submitted to modify files containing any functions/methods with Modified Cyclomatic Complexity scores above 20 over all releases in which they were observed. [112]
Core fraction rank	Independent Variable	Rank	The percentile rank of a developer's proportion of lines submitted to modify files given the architectural complexity classification of "core" using the transitive closure based techniques developed by MacCormack, Baldwin, and Rusnak over all the releases in which they were observed. [8, 9]

9.3 Regression Models

Regression results for models predicting staff turnover are shown in Table 24. Developer productivity (lines of code produced per unit time) was negatively associated with turnover and was significant at the 10% level. More productive developers were more likely to remain with the firm. Whether an employee was a manager was almost statistically significant, with a P value of 11.06%, suggesting that a slightly larger sample size might lead us to conclude that managerial status also decreases turnover. No other variable had a P value below 50%. Although most controls in the model were not significant predictors all pointed in the expected direction.

Note that our third proposition holds. Developers working in more architecturally complex regions of the code (those files considered "core") were much more likely to leave the firm (voluntarily or involuntarily). The coefficient of interest became stronger rather than weaker as controls were added. (Statistical significance improved as well, with a P value of 1.35% in the full model.) Of all variables included in these models, architectural complexity had the strongest impact on turnover. Although the main set of models uses ranks in calculations, similarly significant results are obtained by running regressions on an alternative set of models using raw percentages rather than ranks as predictors (shown in Table 25).

Table 24: Predicting Turnover Among Developers Based on Rankings (Logistic Model)

Parameter	Model 1: developer attributes	Model 2: developer productivity	Model 3: type of work	Model 4: cyclomatic complexity	Model 5: all controls	Model 6: architectural complexity	Model 7: full
Years employed rank	-0.5926				-0.8768		-0.7582
Is manager?	-0.8718				-1.2221		-1.5216
Lines produced per release rank		-1.6825 .			-1.7528		-2.2323 .
Fraction work to fix bugs rank			0.9146		0.6502		0.2045
Fraction work in new file rank			-0.2214		-0.4287		-0.8333
Fraction work high cyclomatic rank				-0.0504	0.0402		-0.7483
Fraction work in core rank						2.2519 *	2.9558 *
Residual Deviance	91.541	90.851	92.843	94.032	86.171	88.586	78.952
Degrees of Freedom	105	106	105	106	101	106	100
AIC	97.541	94.851	98.843	98.032	100.17	92.586	94.952

N = 108 software developers

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

Table 25: Predicting Turnover Among Developers (Logistic Model)

Parameter	Model 1: developer attributes	Model 2: developer productivity	Model 3: type of work	Model 4: cyclomatic complexity	Model 5: all controls	Model 6: architectural complexity	Model 7: full
Years employed	-0.0535				-0.0784		-0.0786
Is manager?	-0.8123				-1.0545		-1.1398
Lines produced per release		-0.0002 .			-0.0002		-0.0003 .
Fraction of lines to fix bugs			1.0526		0.6694		0.0579
Fraction of lines in new files			-0.1638		-0.6652		-1.3219
Fraction lines in high McCabe files				-0.0954	-0.2562		-1.4194
Fraction of lines in core files						3.5440 *	4.1114 *
Residual Deviance	91.525	90.884	93.112	94.03	86.656	87.181	78.632
Degrees of Freedom	105	106	105	106	101	106	100
AIC	97.525	94.884	99.112	98.03	100.66	91.181	94.632

N = 108 software developers

*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

9.4 Interpretation of Results

Two sets of simulations were run to determine the expected probability that a developer would leave the firm (voluntarily or involuntarily) as a result of the complexity of the code they are working in. The first simulation employed the full version rank-based model shown in Table 24. Control variables for all ranks were simply set to the median, or 0.5. The second simulation employed the full version of the turnover model that used percentages rather than percentile ranks (shown in Table 25). In this second simulation, control variables were set to their means. The number of years a developer was employed (prior to their first release in the window) was set to 3.7 years. *Lines produced per release* was set to 3932. The *percentage of lines submitted to fix bugs* was set to 44%. The *percentage of lines submitted into new files* (those younger than 2 years old) was set to 47%. The *percent of lines submitted to high McCabe* files was set to 35%. Managerial status was set to *false* in both simulation runs. The response of the *probability of leaving* variable to changes in the architectural complexity of the code being worked on was determined by running simulations for both relative and absolute levels of architectural complexity.

Table 26 shows the impact of relative rank on the probability of turnover. The developer with the smallest fraction of lines in the core had a 5% chance of leaving the firm. The developer with the largest fraction has a 44% chance. Moving from the 25th to the 75th percentile more than quadrupled the probability of leaving. Table 27 looks at the probability of turnover in absolute rather than relative terms. A developer working entirely in the periphery has a 2% chance of leaving the firm, while a developer working entirely in the core has a 31% chance of leaving the firm.

Table 26: Predicted Probability of Leaving the Firm For Developers Based On Their Relative Amount of Work in Core

Percentile rank for fraction of work in core	Expected probability of leaving	Standard deviation	CI lower bound (2.5%)	CI upper bound (97.5%)
0th	0.0503	0.0418	0.0079	0.1625
10th	0.0621	0.0433	0.0134	0.1757
20th	0.0772	0.0444	0.0215	0.1914
30th	0.0971	0.0455	0.0343	0.2090
40th	0.1220	0.0464	0.0526	0.2324
50th	0.1545	0.0487	0.0776	0.2660
60th	0.1953	0.0542	0.1056	0.3164
70th	0.2450	0.0661	0.1340	0.3903
80th	0.3037	0.0848	0.1575	0.4863
90th	0.3693	0.1084	0.1792	0.5962
100th	0.4376	0.1325	0.1965	0.7022

Table 27: Predicted Probability of Leaving the Firm at Various Architectural Complexity Levels

Percentage of lines contributed to core	Expected probability of leaving	Standard deviation	CI lower bound (2.5%)	CI upper bound (97.5%)
0%	0.0219	0.0461	0.0003	0.1421
10%	0.0262	0.0471	0.0006	0.1525
20%	0.0320	0.0472	0.0013	0.1647
30%	0.0398	0.0481	0.0029	0.1733
40%	0.0505	0.0484	0.0060	0.1823
50%	0.0660	0.0493	0.0124	0.1967
60%	0.0887	0.0498	0.0254	0.2148
70%	0.1211	0.0506	0.0476	0.2426
80%	0.1671	0.0533	0.0824	0.2888
90%	0.2300	0.0652	0.1218	0.3743
100%	0.3108	0.0922	0.1530	0.5100

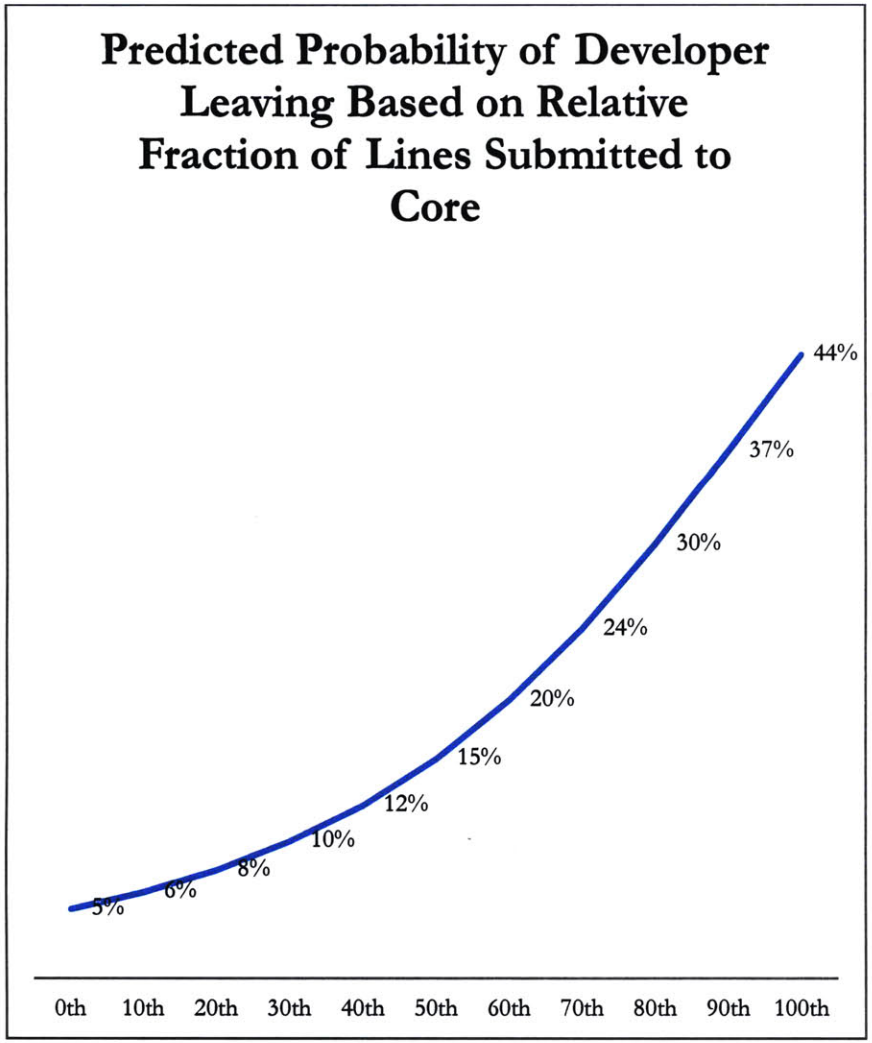

Figure 45: Predicted Developer Turnover (1)

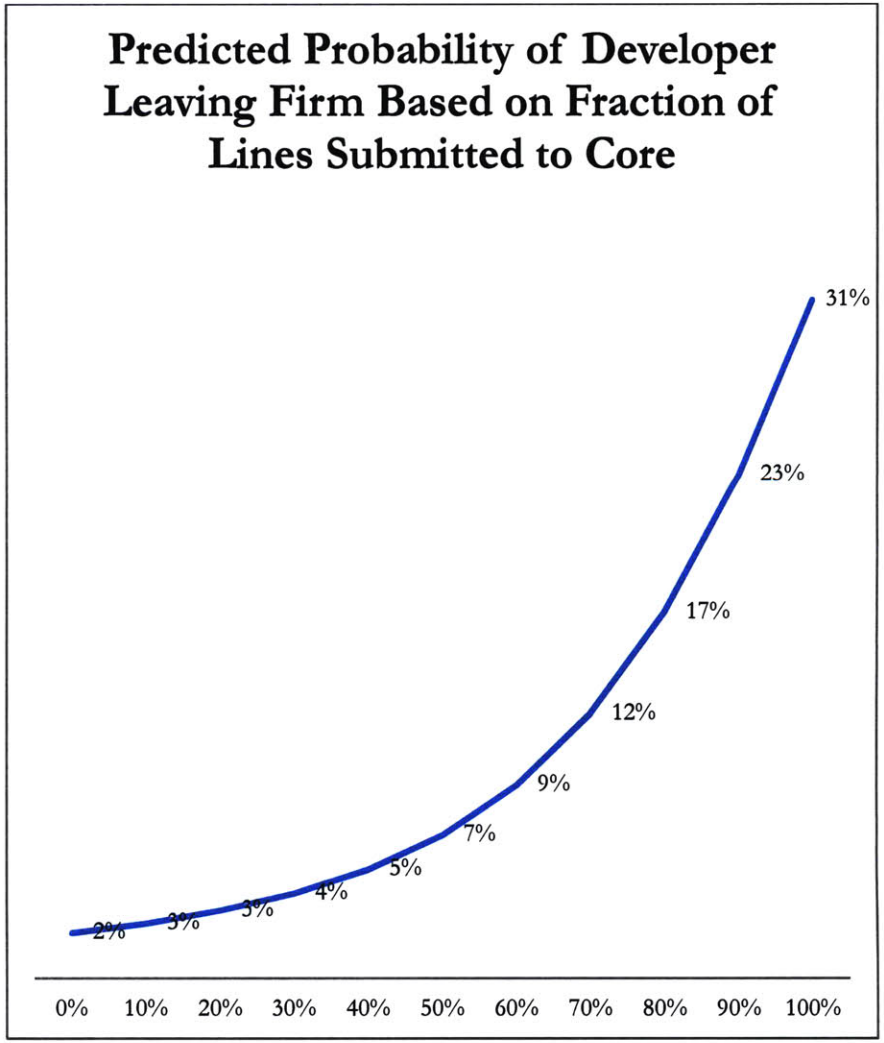

Figure 46: Predicted Developer Turnover (2)

The link between architectural complexity and turnover was surprisingly strong. A variety of plausible controls were included in this analysis, each representing a sound alternative hypothesis for why a developer might leave the firm. None yielded a stronger effect on developer turnover than architectural complexity.

10 Discussion & Conclusions

Software systems today can be composed of millions of entities (such as functions, classes, methods, data structures, etc.) that are connected in countless ways. Designing and maintaining these systems is hard; keeping all of the complexity in the system under control is of the utmost importance. When complexity causes different elements of a system to interact in unanticipated ways, or when parts of a system are so complex that they move beyond the bounds of human cognitive capacities, a host of interconnected problems begin to occur. When we lose control of complexity in a system's design, it can lead to project failure, business failure, and/or man-made disaster. Even systems of high quality with a sustainable level of overall complexity may have some sub-systems and cross-cutting concerns that are unmanageable.

In order to control complexity in large systems, architects often employ certain well-known patterns in their designs to keep *architectural complexity* in check: hierarchies, modules, and layering schemes, among others. When carefully applied, these patterns can aid developer comprehension and enable independence of action. They can also endow systems with a variety of beneficial properties including reliability, evolvability, scalability, and flexibility, just to name a few.

In this research we set out to explore the costs that architectural complexity within a product imposes on the firm that develops and maintains it. A study was conducted at Iron Bridge Software, Inc., the developer of a mature commercial software product under active development. Measures of architectural complexity were taken for source-code files in 8 successive releases of their software. In order to operationalize the notion of architectural complexity, we used procedures and metrics devised by MacCormack, Baldwin, and Rusnak [8, 9] Three important cost drivers were explored: defect density, software developer productivity, and development staff turnover. To explore the relationship between each of these three cost drivers and architectural complexity within the codebase, three regression-based analyses were conducted.

The primary findings of this study were that within Iron Bridge's codebase and development organization:

- Architecturally complex source code files have a much higher defect density. The most complex code was found to have triple the defect density of the least complex code.
- Architectural complexity impairs the productivity of software engineers working with it. If a hypothetical group of engineers working in the least architecturally complex regions of the codebase were to be moved into the most complex regions, their productivity would decline (conservatively) by 50%.
- Architectural complexity causes staff turnover. Software engineers working in the most complex regions of the code had a probability of leaving the firm that was ten times greater than their peers working in least architecturally complex code.

10.1 Contributions to Academic Literature

This dissertation makes a number of contributions to academic literature in the areas of system design, software design, engineering management, and complexity theory. While some previous studies have explored the impact of network-based metrics on defect density [158], none have explored the impact of network based complexity metrics on defects in a commercially produced and mature software system over multiple releases. We are aware of no studies that have systematically looked at the relationship between a system's architecture and the productivity of technical professionals that work on different parts within that structure. We are also unaware of any past studies exploring the link between a system's architecture and staff turnover. We therefore believe that this study makes a number of novel contributions to the academic literature.

This work empirically demonstrated that architectural complexity is an important driver of defects. This result was not entirely unexpected. It was unexpected, however, that MacCormack's architectural complexity metric would predict defects as well as (if not somewhat better than) the widely accepted McCabe cyclomatic complexity metric. Most traditional software complexity metrics (such as McCabe's) are reductionist in nature. They capture properties of individual software components considered in isolation while ignoring the architectural patterns that link them. Architectural complexity, on the other hand, is a holistic concept that largely ignores the contents of individual files and focuses only on the

patterns binding the parts to the whole. For this reason, architectural complexity may capture a fundamentally different concept of quality that is equally important to more traditional measures.

While theoretical and descriptive work done over many decades has led us to implicitly trust the notion that the architecture of a complex system should influence productivity of engineers working within it, this work is the first to establish the link in an empirical setting and the first to provide quantitative estimates for the strength of that relationship. In order to establish this link we used a fixed effects panel-data approach to look at productivity differences within the same individual across multiple time periods. The relationship that was found was statistically significant and quite strong. We were able to estimate the effect of architectural complexity in the code that a developer worked in on overall productivity, productivity while implementing product enhancements, and productivity while fixing defects.

This work is the first to explore the influence of architecture on technical staff turnover. We had no idea if a relationship between architectural complexity and staff turnover could be established. The fact that the impact of architectural complexity was found not only to be substantial and statistically significant, but also to be of greater importance than a developer's tenure, productivity, fraction of effort in new (vs. legacy) code, fraction of effort working on bugs, and fraction of effort working in files with high McCabe complexity was very surprising. It should be noted that this analysis was performed after we realized the strength of the effect architectural complexity had on both defect density and individual productivity. The rationale for exploring the link between complexity and turnover rested on the premise that developers in architecturally complex code likely had other problems stemming from (or causing) their decreased productivity and higher defect-introduction rate. They might also have more trouble making reliable estimates, more trouble delivering on schedule, more failed attempts to solve problems, more unanticipated side effects, more sleepless nights, more anxiety, and more stress. Exacerbating this situation is the fact that architectural complexity is not directly observable. Architecturally complex code might appear well constructed and appropriately commented upon isolated inspection. Appreciating the true reason that such code may have too many defects or produce too many side effects might require a person to mentally traverse *indirect* links to discover cyclical dependency chains that span the organization. This task might be impossible given human cognitive constraints. As a result, managers and peers might

evaluate developers in architecturally complex regions of a large system more harshly than their abilities and contributions actually warrant. The hypothesis that all of these interconnected factors would cause detectably higher rates of staff turnover turned out to be correct. While turnover can be healthy for an organization, it is hard to see turnover that results from architectural complexity as anything other than a negative. Ultimately, some new developer will be placed in the same position as the last while the root-cause may go unaddressed.

A surprising negative result found in this study was that no statistically significant relationship between McCabe complexity and development staff productivity or turnover could be established. While it is possible that a different measurement or analysis approach might have establish this link, it is also possible that architectural complexity and component-centric measures of complexity behave in different ways, and therefore impact software engineers differently. Architectural complexity can slow progress by causing rework, subtle side effects, and deadlock across organizational boundaries in ways that component-specific complexity does not. Component-complexity can be directly perceived, is contained, can be avoided, and can be corrected by a single engineer acting unilaterally. Architectural complexity, on the other hand, is invisible, results from dependencies that span the system, cannot be avoided, and requires coordinated action across organizational boundaries to mitigate. For these reasons, it is entirely possible that architectural complexity truly has a much stronger impact on productivity, morale, and staff turnover. Future work should be done to explore this possibility.

This work provides support for the validity and utility of the MacCormack, Baldwin, and Rusnak approach. The MacCormack approach provides a repeatable means of extracting architectures from software and a quantifiable means of measuring complexity within that software design. Much of the prior work connecting these architectural metrics to outcome variables was qualitative or descriptive, however. By using the MacCormack approach to measure complexity on a large scale in a commercial setting and then relating that complexity to quantities of obvious managerial interest, this dissertation lends support to the validity and practical utility of MacCormack's methods.

Finally, this work confirms many of the intuitive beliefs held within the system design and design structure matrix community, but is the first to offer empirical support for some of those beliefs. This

work also reemphasizes the importance of hierarchy and modularity as high-level design principles by demonstrating how costly deviations from them can be.

10.2 Contributions to Managerial Practice:

A number of insights gained over the course of this research have the potential to contribute to managerial practice. By conducting a case study within a representative commercially successful software firm, exploring a large codebase that is both mature and growing rapidly, and studying large community of paid software developers in all career stages, we greatly increase the possibility that results gleaned here will be applicable in other firms or organizations responsible for developing and maintaining software codebases or other complex systems.

The first contribution of this work is simply to demonstrate how costly architectural complexity can actually be and to suggest an approach to computing the financial value of successful redesign efforts. We found that differences in architectural complexity could account for 50% drops in productivity, three-fold increases in defect density, and order-of-magnitude increases in staff turnover. This is not the whole story however. When considering the cost of additional defects and lower productivity in combination, the picture becomes more dramatic because the 50% productivity loss we calculated assumes that a developer's ratio of feature development to bug-correction work is held constant as he moves from the periphery to the core. Because complex code has more bugs, and because bug fixes requires much more time (per line of code) to implement than comparably sized features, productivity (as measured by LOC produced per unit time) will slip further than 50%. One should also consider the fact that defect correction is a necessary but non-value-add activity. The *value* that a customer derives from a developer's productive output does not include work done to fix bugs (unless those bugs were released into the market place.) This suggests a possible alternative productivity measure that excludes defect-correction LOC written from consideration altogether, further amplifying the effect. When considering the cost of increased staff turnover among developers in architecturally complex code, one must consider the cost of recruiting and training replacements and the cost of bugs that rookie developers will introduce. Such a calculation must also account for the fact that developers in the core are likely the hardest to replace

because of the need for higher skill levels and steeper learning curves. Using the techniques developed in this thesis, it should be possible for firms to estimate the financial cost of their complexity by assigning a monetary value to the decreased productivity, increased defect density, and increased turnover it causes. As a result, it should be possible for firms to more accurately estimate the potential dollar-value of refactoring efforts aimed at improving architecture. While we have not gone through the exercise of converting the above factors into dollar figure estimates of cost, we are now quite convinced that refactoring efforts that successfully reduce architectural complexity have the potential to create enormous financial value for the firm.

A second contribution of this work to managerial practice is to point toward a means of managing refactoring efforts. By utilizing tooling similar to the programming interface, database infrastructure, mathematical analysis code, and DSM software that we created for this work (shown in Figure 32), managers would have a means of tracking progress towards complexity reduction. They would have the ability to visualize the structure of code as it changed and could track resulting cost reductions by monitoring the movement of KPIs in subsequent time periods. These managers would feel more confident when moving forward with larger refactoring efforts because they would have a key feedback mechanism allowing their organization to move, learn, and adjust as needed. Without the ability monitor architecture and the cost it imposes, managing a system overhaul is a much more uncertain proposition, often leading to "death marches" and costly failures.

This thesis also has other practical applications. It demonstrates that the MacCormack, Baldwin, and Rusnak metrics can be added to the list of metrics that successfully identify defect-prone files or predict future defects. By combining architectural metrics with other previously validated component-based defect predictors, we should gain accuracy. Better predictive capabilities give organizations better ability to proactively clean problematic areas of their codebases and appropriately allocate testing resources.

This work also helps address some of the problems with software cost and schedule estimation models. Kemerer said that the Achilles heel of estimation models is their lack of a sound underlying theory of developer productivity. [159] Schedule and cost models base their estimates for required effort on LOC, function point, or other counts that capture code volume but ignore the interdependence between

elements in a system resulting from its architectural or structural properties. Because this research has demonstrated that architectural complexity is a very strong driver of an individual's productivity, cost estimation techniques that rely on productivity estimates might improve if measures of architectural complexity were taken into account.

Finally, this work suggests that a shift in mindset might be warranted in software firms that attempt to use quality metrics and models (see [108]) derived from experiences gained in the advanced manufacturing world. Many *Lean* or *Six-Sigma* inspired models transplanted into software settings measure bugs, bug introduction rate, bug correction rate, and use a variety of techniques to link bugs to team or individual performance. Unfortunately, these models often have no concept of what a bug is, what causes them, and what can be done to reduce their frequency. In these models, "bugs" are completely disembodied from the code or software architecture in which they rest. The implicit message underlying models that only measure people and bugs is that people are to be blamed for bugs. While it is partially true that people cause bugs, we believe that this research points towards a healthier mental model. We propose that complex architecture causes bugs, impairs productivity, and thwarts understanding. We propose that developers are in some senses its victims. By measuring software structure and architectural complexity we can give development organizations the ability to coordinate actions and address the actual root causes behind defects and project failures. We can better understand which projects are likely to have false-starts, offer additional support for those working in entangled parts of the code, and give combat pay to those attempting to refactor cross-cutting concerns. In general, an understanding of architectural complexity and the costs that it imposes should help an organization more appropriately set expectations and allocate resources.

10.3 Limitations of This Work

This work has a few important limitations that should be understood.

Because this dissertation has focused primarily on the *cost* of complexity, it must be said that complexity is not inherently bad, even if it leads to a variety of increased costs. Managers must focus on *value* – benefit minus cost – to make rational decisions. *Complexity adds value.* No system will be free of complexity, and

a system with more complexity may have other benefits, such as increased performance, that offset the costs. Even if a system is undesirably complex, it might not be worth addressing if the cost of refactoring would be greater than the expected cost reduction. This dissertation does not consider the full spectrum of factors that must go into a cost-benefit analysis. We also do not compute values for many benefits modularity is known to provide, such as increased "option value" [5]. We do not consider the fact that design modularity is sometimes incompatible with physics or other hard requirements in the problem domain. [7] Finally, because we are only taking measurements within a successful firm that ships popular software products, we can only report on the cost of complexity in a situation where that complexity is controlled to an acceptable degree. In this research, we have no means of estimating the cost of *uncontrolled* design complexity, which has lead to the failure of firms. [24, 59] (This firm-level data may, however, capture the impact of isolated project failures.) This research also affords us no means of studying the risk posed to public safety by uncontrolled complexity in technical systems during their operation. Although we make no probabilistic estimates for complexity spiraling out of control, or for the damage done should that occur, these possibilities should always be weighted during system design, tipping the scales towards complexity control to some extent.

Although many steps were taken to ensure a high-degree of internal validity, a single firm study suggests some threats to external validity. It is possible that Iron Bridge is unrepresentative and that our conclusions therefore have limited applicability. We believe this to be unlikely. Although Iron Bridge has some unique attributes, it is a reasonably representative large software development firm. Over the years, Iron Bridge adopted several industry standard languages, development tools, and techniques. Its design and project-management practices are similar to those of other large commercially oriented software firms as well. This adoption was the result of both deliberate organizational-learning efforts and cross-pollination due to hiring. Software professionals have migrated between Iron Bridge and other firms, carrying their knowledge, experiences, and practices with them.

A third important point to make is that files and software developers are the units of analysis in this study; the firm and the codebase are not. The study design employed only permitted us to evaluate the cost of more architecturally complex vs. less architecturally complex regions within the same codebase. A

variety of setting-specific factors, such as unique tooling, processes, and measurements would make apples-to-apples comparisons *between* Iron Bridge's system and some other system very hard, even if that were the intent of this work.

One potential barrier to broad generalizability stems from the choice of domain. An analysis of the effect of *software* complexity on quality within a *software* architecture and the productivity of *software developers* might not be generalizable to engineering systems of different sorts. This concern is real, and the reader should understand that lessons learned here might not all be directly applicable to systems of a different type. There are fundamental distinctions between information and physics resulting in differences between the nature of software and electro-mechanical objects. These necessarily lead to differences between the practice of software engineering and engineering in other domains. However, there are enough similarities that we believe many of the results established in this work could be generalized outside the field of software. One reason is that we are looking at how fallible and boundedly rational designers cope with complicated and complex system designs. Large organizations made up of those humans will face many of the same challenges and experience many of the same pitfalls no matter what they are building. A second reason is that we are not focusing much on properties unique to software or information. Instead, we are focusing on interconnection patterns between coupled elements that come together to form hierarchies and modules. These common patterns are found in large systems of all types, man-made and natural, and have been shown to relate to their ability to survive, scale, and evolve. It is therefore reasonable to suspect that the influence of these universal properties associated with complex system architectures will influence cost drivers in a similar manner regardless of system type.

10.4 Directions for Future Work

This work raises many interesting questions that could be explored in future work.

Due to the fact that this case study only explored the cost of complexity in a single firm, an obvious extension would be to find other suitable firms in which to replicate the analyses presented here. Replication could be done under similar conditions or with slight variations. For instance, because this report presented results gleaned from a C++ codebase developed by a mature commercial firm, future

research could look at younger firms or firms using shorter release cycles. It would also be good to validate these results in codebases written in other languages or in other application domains.

A second extension would be to do pre- and post- analyses of refactoring efforts within a codebase such as the one developed by Iron Bridge. While this work provides evidence that portions of a codebase with lower complexity also have lower costs, it does not look for evidence that specific refactoring both reduced architectural complexity and lowered costs. Investigators could work with multiple teams that have previously conducted a refactoring to look for evidence that their effort resulted in improvements. In so doing, we could gain confidence in, and improve, complexity and cost measures. These pre- and post-comparisons may also allow us to determine the extent to which refactoring was worthwhile from a financial standpoint. We could then use the tools devised over the course of this research to work with teams initiating new redesign efforts to determine if structure-related information and past cost information can help in their planning.

Thirdly, it should be noted that the turnover study left some questions unanswered and deserves to be revisited in the future. While the previous discussion explored the means by which complexity could increase turnover, an alternative explanation that may not have been adequately explored is the possibility that the "best" developers both work in the core and have the most promising opportunities outside the firm. If this were the case, then activity in the core would simply be a proxy for valuable engineering competence, and therefore mobility. In order to more fully explore this alternative explanation salary levels and performance ratings might be included in regression analysis. Future work following up on the turnover analysis should also attempt include a larger sample of *leavers* by profiling more development periods and should employ a *hazard model* approach. We find the current results linking complexity to turnover to be very intriguing and believe further work should be done to gain additional confidence in the surprisingly strong result that was obtained here.

Fourthly, while this work measures a variety of costs, there are many that have gone unexplored. A valuable extension to this work would be to round out the cost analysis with a study of learning curves and career growth patterns. It is likely that complexity in a codebase affects rookies and veterans differently. It is possible that rookies who are slowly moved into the core thrive while rookies thrown

into the core immediately suffer. It is also possible that rookies have a harder time diagnosing bugs in the core than veterans do. More fully understanding the dynamics of new-hire ramp-up, migration patterns around the codebase, the role that complexity has in performance evaluation and morale, and the resulting impact on turnover would help round out our understanding of the dynamic interplay between complexity in the codebase and the employees who work in it.

Another area for future work would be to use these findings and other insights gained about the cost of complexity to improve upon the system dynamics or agent-based models that simulate the dynamics of large projects as they unfold through time. A large body of system dynamics work has been done to look at the dynamics of project management. [24, 27-29] These models have explored the interplay between productivity, cost, quality, turnover, morale, and learning curves and have been used for planning on many large programs. One limitation of these models is that they tend to treat all tasks equally. To my knowledge none of those models have incorporated information about architecture or the cost of architectural complexity and none have attempted to explore the relationship between the growth patterns of a complex system and the dynamics of the development effort that produces it.

Finally, future work should investigate the link between architecture, morale, and turnover. The results presented in this thesis demonstrating that architectural complexity can significantly increase turnover raises a variety of questions about the underlying causal mechanism. We previously posited that high architectural complexity might affect morale through a variety of means. We suggested that because complexity leads to lower productivity and more defects, it also likely leads to overwork, burnout, stress, and other things that harm morale and lead to higher voluntary turnover. On the other hand, if managers evaluate engineers negatively as a result of lower productivity or higher defect introduction rates, then higher involuntary turnover may result. Further investigation to explore these links between complexity, morale, and turnover would be valuable.

10.5 Concluding Remarks

It should be noted that the study presented in this dissertation analyzed eight *old* releases. Our data therefore fails to capture the impact of recent architectural changes. In the intervening time period between 'release 8' and today, a strategic initiative – *a modularization program* – was begun with the goal of splitting Iron Bridge's traditionally monolithic codebase into a hierarchy of modular units with well defined interfaces and explicitly declared dependencies. The database infrastructure that was constructed so that we could conduct this study using historical releases has also been used by "Clark Kent," the senior system architect driving Iron Bridge's overall modularization effort, to explore the structure of the current codebase for the past few years. Clark (and now a growing group of development managers) uses this tool to explore the codebase's dependency structure, cross-module coupling, and module interfaces. He also uses it to identify places where design rules were being violated by exploring undeclared dependencies and the use of non-public interfaces. In the future it may be possible to study the benefits that Iron Bridge reaps from its significant investment in architectural improvement today.

To conclude this work we will summarize points made by "David Parker," Iron Bridge's CEO, during a recent interview. Dave told us that Iron Bridge is taking serious steps to address complexity in software architecture because he and other company leaders are convinced that modularization is the singe most important thing that can be done to improve quality, improve development team productivity, make the firm more agile, and give Iron Bridge a good future market position. He noted that teams that have historically suffered from too much complexity have had a harder time developing code and have been slower delivering features. He also held out the experiences of many teams that had been "broken free" and had become much more effective as a result.

Dave says that ongoing architectural improvement is being prioritized today and will continue to receive significant investment into Iron Bridge's foreseeable future. He plans to ensure that progress continues in two ways. Firstly, he plans to invest more in measurement systems and metric development so that complexity can be tracked, appropriately managed, and appropriately dealt with by teams working

individually and collectively. Secondly, he says that development teams must explicitly prioritize some refactoring as part of their release schedule to have a balanced budget of activities.

When asked why he believes architecture and modularity are so important, Dave spoke about a new concept he came across in the past few years: *technical debt.* Dave viewed architectural problems and complexity in Iron Bridge's code as a debt to be paid off because "if it gets too large, just like financial debt, it starts to weigh you down." He believes that this new analogy has helped many people in the organization better understand the importance of refactoring and making appropriate short- vs. long-term tradeoffs in the code. Iron Bridge's leadership is convinced that architecture is very important and that more must be done to address the cost of complexity.

11 Bibliography

[1] E. Crawley, "Lecture notes for ESD.34 - System Architecture (2007), Massachusetts Institute of Technology," unpublished.

[2] J. M. Sussman, "The new transportation faculty: The evolution to engineering systems," 1999.

[3] J. M. Sussman, "Ideas on complexity in systems-twenty views," ed: MIT Engineering Systems Division, Working Paper Series, 2000.

[4] S. K. Fixson and J. K. Park, "The power of integrality: Linkages between product architecture, innovation, and industry structure," *Research Policy,* vol. 37, pp. 1296-1316, 2008.

[5] C. Y. Baldwin and K. B. Clark, *Design rules: The power of modularity, Volume 1* vol. 1: The MIT Press, 1999.

[6] S. Shapiro, "Splitting the difference: The historical necessity of synthesis in software engineering," *Annals of the History of Computing, IEEE,* vol. 19, pp. 20-54, 1997.

[7] D. E. Whitney, "Why mechanical design will never be like VLSI design," *Research in Engineering Design,* vol. 8, pp. 125-138, 1996. Revised and updated as "Physical limits to modularity," MIT Engineering Systems Symposium, working paper (2004), see: http://esd.mit.edu/symposium/pdfs/papers/whitney.pdf

[8] A. D. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science,* pp. 1015-1030, 2006.

[9] A. D. MacCormack, J. Rusnak, and C. Y. Baldwin, "The impact of component modularity on design evolution: Evidence from the software industry," Harvard Business School, Harvard Business School Technology and Operations Management Unit Research Papers 08-038, 2007.

[10] O. L. de Weck, D. Roos, and C. L. Magee, *Engineering systems: Meeting human needs in a complex technological world*: The MIT Press, 2011.

[11] M. R. Smith, *Military enterprise and technological change: Perspectives on the American experience*: The MIT Press, 1985.

[12] R. H. Coase, "The nature of the firm," *Economica,* vol. 4, pp. 386-405, 1937.

[13] A. Smith, "An inquiry into the nature and causes of the wealth of nations," 1776.

[14] F. W. Taylor, *The principles of scientific management.* New York and London: Harper & Brothers, 1911.

[15] M. D. Fagen, A. E. Joel, and G. Schindler, *A history of engineering and science in the Bell system.* New York: Bell Telephone Laboratories, inc., 1975.

[16] D. A. Mindell, "Automation's finest hour: radar and system integration in World War II," *Systems, Experts, and Computers,* pp. 27-56, 2000.

[17] RAE, "The challenges of complex IT projects," The Royal Academy of Engineering, BCS Working Group, 2004.

[18] N. Leveson, "A new accident model for engineering safer systems," *Safety Science,* vol. 42, pp. 237-270, 2004.

[19] N. G. Leveson, "A systems-theoretic approach to safety in software-intensive systems," *Dependable and Secure Computing, IEEE Transactions on,* vol. 1, pp. 66-86, 2004.

[20] F. P. Brooks, "No silver bullet: Essence and accidents of software engineering," *IEEE computer,* vol. 20, pp. 10-19, 1987.

[21] J. C. Munson and T. M. Khoshgoftaar, "Measuring dynamic program complexity," *Software, IEEE,* vol. 9, pp. 48-55, 1992.

[22] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Transactions on software Engineering,* pp. 96-104, 1979.

[23] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM,* vol. 27, pp. 42-52, 1984.

[24] J. M. Lyneis, K. G. Cooper, and S. A. Els, "Strategic management of complex projects: a case study using system dynamics," *System Dynamics Review,* vol. 17, pp. 237-260, 2001.

[25] M. Lewyn, "Flying in place: The FAA's air-control fiasco," *Business Week,* vol. 90, 1993.

[26] R. De Neufville, "The baggage system at Denver: prospects and lessons," *Journal of Air Transport Management,* vol. 1, pp. 229-236, 1994.

[27] T. Abdel-Hamid and S. E. Madnick, *Software project dynamics: An integrated approach*: Prentice-Hall, Inc., 1991.
[28] T. K. Abdel-Hamid, "The dynamics of software project staffing: A system dynamics based simulation approach," *Software Engineering, IEEE Transactions on,* vol. 15, pp. 109-119, 1989.
[29] R. J. Madachy, *Software process dynamics*: Wiley-IEEE Press, 2008.
[30] D. V. Steward, "Design structure system: A method for managing the design of complex systems," *Engineering Management, IEEE Transactions on,* vol. 28, pp. 71-74, 1981.
[31] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, "A model-based method for organizing tasks in product development," *Research in Engineering Design,* vol. 6, pp. 1-13, 1994.
[32] R. P. Smith and S. D. Eppinger, "A predictive model of sequential iteration in engineering design," *Management Science,* pp. 1104-1120, 1997.
[33] R. P. Smith and S. D. Eppinger, "Deciding between sequential and concurrent tasks in engineering design," *Concurrent Engineering,* vol. 6, pp. 15-25, 1998.
[34] S. D. Eppinger, M. V. Nukala, and D. E. Whitney, "Generalised models of design interation using signal flow graphs," *Research in Engineering Design,* vol. 9, pp. 112-123, 1997.
[35] A. Tripathy and S. D. Eppinger, "Organizing Global Product Development for Complex Engineered Systems," *Engineering Management, IEEE Transactions on,* pp. 1-20, 2011.
[36] C. Y. Baldwin and K. B. Clark, "Managing in an age of modularity," *Harvard Business Review,* vol. 75, pp. 84-93, 1997.
[37] N. Leveson, M. Daouk, N. Dulac, and K. Marais, "A systems theoretic approach to safety engineering," *Dept. of Aeronautics and Astronautics, Massachusetts Inst. of Technology, Cambridge,* 2003.
[38] D. Braha and Y. Bar-Yam, "The statistical mechanics of complex product development: Empirical and analytical results," *Management Science,* vol. 53, pp. 1127-1145, 2007.
[39] E. S. Raymond, *The cathedral and the bazaar*: O'Reilly Media, 1999.
[40] E. Von Hippel, "Lead users: a source of novel product concepts," *Management Science,* pp. 791-805, 1986.
[41] E. Von Hippel, *Democratizing innovation*: the MIT Press, 2005.
[42] J. H. Holland, *Adaptation in natural and artificial systems*: University of Michigan press, 1975.
[43] J. H. Holland, "Complex adaptive systems," *Daedalus,* vol. 121, pp. 17-30, 1992.
[44] A. A. Mina, D. Braha, and Y. Bar-Yam, "Complex engineered systems: A new paradigm," *Complex Engineered Systems,* pp. 1-21, 2006.
[45] W. B. Rouse, "Complex engineered, organizational and natural systems," *Systems Engineering,* vol. 10, pp. 260-271, 2007.
[46] H. A. Simon, "The architecture of complexity," *Proceedings of the American Philosophical Society,* vol. 106, pp. 467-482, 1962.
[47] H. A. Simon, "Prediction and prescription in systems modeling," *Operations Research,* pp. 7-14, 1990.
[48] H. A. Simon, *The sciences of the artificial,* third ed.: The MIT Press, 1996.
[49] H. A. Simon, "Can there be a science of complex systems," 2000, pp. 3-14.
[50] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, and R. Jeffries, "Manifesto for agile software development," *The Agile Alliance,* pp. 2002-04, 2001.
[51] H. A. Simon, *The sciences of the artificial,* first ed.: The MIT Press, 1969.
[52] J. W. Forrester, *Principles of systems: text and workbook chapters 1 through 10*: Wright-Allen Press, 1969.
[53] J. Sterman, *Business dynamics*: Irwin, 2000.
[54] J. Yardley and G. Harris, "2nd day of power failures cripples wide swath of India," in *The New York Times*, ed: The New York Times Company, 2012.
[55] D. Whitney, E. Crawley, O. de Weck, S. Eppinger, C. Magee, J. Moses, W. Seering, J. Schindall, and D. Wallace, "The influence of architecture in engineering systems," *Engineering Systems Monograph,* 2004.
[56] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns," 1999.
[57] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems: software technologies, engineering processes, and business practices*: Addison-Wesley Professional, 2003.
[58] T. U. Pimmler and S. D. Eppinger, "Integration analysis of product decompositions," presented at the ASME Design Theory and Methodology Conference, Minneapolis, MN, 1994.

[59] F. P. Brooks, *The mythical man-month: essays on software engineering (anniversary ed.)*: Addison-Wesley Longman Publishing Co., Inc., 1995.
[60] F. P. Brooks, *The design of design: Essays from a computer scientist*: Addison-Wesley Professional, 2010.
[61] K. T. Ulrich and S. D. Eppinger, *Product design and development*, first ed.: McGraw-Hill, 1995.
[62] K. Ulrich, "The role of product architecture in the manufacturing firm," *Research policy*, vol. 24, pp. 419-440, 1995.
[63] J. Moses, "Foundational issues in engineering systems: A framing paper," *Engineering Systems Monograph*, 2004.
[64] P. Checkland, *Systems thinking, systems practice*: Wiley & Sons, 1981.
[65] M. Minsky, *The society of mind*: Simon and Schuster, 1988.
[66] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," *Software Engineering, IEEE Transactions on*, pp. 259-266, 1985.
[67] P. Naur, B. Randell, and F. L. Bauer, *Software Engineering: Report on a conference sponsored by the NATO science committee, Garmisch, Germany, 7th to 11th October 1968*: Scientific Affairs Division, NATO, 1969.
[68] N. Wirth, "Program development by stepwise refinement," *Communications of the ACM*, vol. 14, pp. 221-227, 1971.
[69] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, 1972.
[70] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Transactions on software Engineering*, pp. 128-138, 1979.
[71] G. Booch, "Object-oriented development," *IEEE Trans. Software Eng.*, vol. 12, pp. 211-221, 1986.
[72] P. H. Loy, "A comparison of object-oriented and structured development methods," *ACM SIGSOFT Software Engineering Notes*, vol. 15, pp. 44-48, 1990.
[73] R. D. Banker and S. A. Slaughter, "The moderating effects of structure on volatility and complexity in software enhancement," *Information Systems Research*, vol. 11, pp. 219-240, 2000.
[74] J. Moses, "Flexibility and Its Relation to Complexity and Architecture," in *Complex Systems Design & Management*, M. Aiguier, F. Bretaudeau, and D. Krob, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 197-206.
[75] C. Y. Baldwin and J. Woodard, "The architecture of platforms: A unified view," *Harvard Business School Finance Working Paper No. 09-034*, 2008.
[76] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, pp. 131-183, 1992.
[77] D. Jackson, *Software abstractions: logic, language and analysis*: The MIT Press, 2006.
[78] S. D. Eppinger and T. R. Browning, *Design structure matrix methods and applications*: The MIT Press, 2012.
[79] A. D. MacCormack, C. Y. Baldwin, and J. Rusnak, "The architecture of complex systems: do core-periphery structures dominate?," Harvard Business School, 2010.
[80] S. H. Strogatz, "Exploring complex networks," *Nature*, vol. 410, pp. 268-276, 2001.
[81] P. Mahadevan, D. Krioukov, M. Fomenkov, X. Dimitropoulos, and A. Vahdat, "The Internet AS-level topology: Three data sources and one definitive metric," *ACM SIGCOMM Computer Communication Review*, vol. 36, pp. 17-26, 2006.
[82] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," 1999, pp. 251-262.
[83] M. D. Morelli, S. D. Eppinger, and R. K. Gulati, "Predicting technical communication in product development organizations," *Engineering Management, IEEE Transactions on*, vol. 42, pp. 215-222, 1995.
[84] G. D. Bergland, "A guided tour of program design methodologies," *Computer Magazine*, 1981.
[85] T. R. Browning, "Applying the design structure matrix to system decomposition and integration problems: a review and new directions," *Engineering Management, IEEE Transactions on*, vol. 48, pp. 292-306, 2001.
[86] C. M. Rowles, "System integration analysis of a large commercial aircraft engine," Massachusetts Institute of Technology, 1999.
[87] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, "Understanding the effects of product architecture on technical communication in product develoment organizations," 2000.
[88] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, "A network approach to define modularity of components in complex products," *Journal of Mechanical Design*, vol. 129, p. 1118, 2007.

[89] E. S. Suh, M. R. Furst, K. J. Mihalyov, and O. Weck, "Technology infusion for complex systems: A framework and case study," *Systems Engineering,* vol. 13, pp. 186-203, 2010.

[90] F. K. Levy, G. L. Thompson, J. D. Wiest, and H. U. G. S. o. B. Administration, *The ABCs of the critical path method*: Harvard University, Graduate School of Business Administration, 1963.

[91] G. A. Blaauw and F. P. Brooks Jr, *Computer architecture: concepts and evolution*: Addison-Wesley Longman Publishing Co., Inc., 1997.

[92] D. M. Sharman and A. A. Yassine, "Architectural valuation using the design structure matrix and real options theory," *Concurrent Engineering,* vol. 15, pp. 157-173, 2007.

[93] J. McNerney, J. D. Farmer, S. Redner, and J. E. Trancik, "Role of design complexity in technology improvement," *Proc Natl Acad Sci U S A,* vol. 108, pp. 9008-13, May 31 2011.

[94] S. Novak and S. D. Eppinger, "Sourcing by design: Product complexity and the supply chain," *Management Science,* pp. 189-204, 2001.

[95] S. Valverde, R. F. Cancho, and R. V. Sole, "Scale-free networks from optimal design," *EPL (Europhysics Letters),* vol. 60, p. 512, 2002.

[96] S. Valverde and R. Solé, "On the nature of design," *Complex Engineered Systems,* pp. 72-100, 2006.

[97] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," 2001, pp. 99-108.

[98] D. Settas and I. Stamelos, "Resolving complexity and interdependence in software project management antipatterns using the dependency structure matrix," *Software Engineering Research, Management and Applications,* pp. 205-217, 2008.

[99] A. D. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the duality between product and organizational architectures: A test of the mirroring hypothesis," *Harvard Business School Working Paper No. 08-039,* 2008.

[100] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," 2005, pp. 167-176.

[101] J. Rusnak and A. D. MacCormack (advisor), "The design structure analysis system: a tool to analyze software architecture," Doctoral Thesis, Harvard Business School, Harvard University, 2005.

[102] M. J. LaMantia and J. M. Utterback (advisor), "Dependency models as a basis for analyzing software product platform modularity: A case study in strategic software design rationalization," S.M. Engineering & Management Master's Thesis, System Design and Management Program, Massachusetts Institute of Technology, 2006.

[103] M. J. LaMantia, Y. Cai, A. MacCormack, and J. Rusnak, "Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases," 2008, pp. 83-92.

[104] A. Akaikine and A. D. MacCormack (advisor), "The impact of software design structure on product maintenance costs and measurement of economic benefits of product redesign," S.M. Engineering and Management Master's Thesis, System Design & Management Program, Massachusetts Institute of Technology, 2010.

[105] M. E. Sosa, T. Browning, and J. Mihm, "Studying the dynamics of the architecture of software products," 2007, pp. 4-7.

[106] J. D. Summers and J. J. Shah, "Mechanical engineering design complexity metrics: size, coupling, and solvability," *Journal of Mechanical Design,* vol. 132, 2010.

[107] W. W. Agresti and W. M. Evanco, "Projecting software defects from analyzing Ada designs," *Software Engineering, IEEE Transactions on,* vol. 18, pp. 988-997, 1992.

[108] S. H. Kan, *Metrics and models in software quality engineering*: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[109] M. H. Halstead, *Elements of software science*: Elsevier New York, 1977.

[110] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on,* pp. 308-320, 1976.

[111] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *Software Engineering, IEEE Transactions on,* vol. 17, pp. 1284-1288, 1991.

[112] M. Bray, K. Brune, D. A. Fisher, J. Foreman, and M. Gerken, "C4 software technology reference guide - a prototype," Software Engineering Institute, Carnegie Melon Institute, 1997.

[113] J. Troster, "Assessing design-quality metrics on legacy software," in *CASCON '92*, 1992, pp. 113-131.

[114] L. L. Craddock, "Analyzing cost-of-quality, complexity, and defect metrics for software inspections," IBM, Rochester, Minn. Technical Report TR07.844, 1987.
[115] C. F. Kemerer and S. A. Slaughter, "Determinants of software maintenance profiles: an empirical investigation," *Journal of Software Maintenance,* vol. 9, pp. 235-252, 1997.
[116] A. Kuntzmann-Combelles, P. Comer, J. Holdsworth, and S. Shirlaw, "Handbook of the application of metrics in industry: A quantitative approach to software management," ed: AMI ESPRIT project, University of the Southbank, London, 1992.
[117] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on,* vol. 29, pp. 297-310, 2003.
[118] D. N. Card, "Designing software for producibility," *Journal of Systems and Software,* vol. 17, pp. 219-225, 1992.
[119] D. N. Card and R. L. Glass, *Measuring software design quality*: Prentice-Hall, Inc., 1990.
[120] C. F. Kemerer, "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering,* vol. 1, pp. 1-22, 1995.
[121] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Management Science,* pp. 433-450, 1998.
[122] D. N. Card and W. W. Agresti, "Measuring software design complexity," *Journal of Systems and Software,* vol. 8, pp. 185-197, 1988.
[123] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer,* vol. 27, pp. 44-49, 1994.
[124] E. Yourdon and L. L. Constantine, *Structured design: fundamentals of a discipline of computer program and systems design*: Prentice-Hall, Inc., 1979.
[125] G. J. Myers, "Composite/structured design," 1978.
[126] S. Henry and D. Kafura, "Software structure metrics based on information flow," *Software Engineering, IEEE Transactions on,* pp. 510-518, 1981.
[127] J. P. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler, "Software complexity measurement," *Communications of the ACM,* vol. 29, pp. 1044-1050, 1986.
[128] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM,* vol. 41, pp. 67-73, 1998.
[129] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on,* vol. 27, pp. 1-12, 2001.
[130] M. Giffin, O. de Weck, G. Bounova, R. Keller, C. Eckert, and P. J. Clarkson, "Change propagation analysis in complex technical systems," *Journal of Mechanical Design,* vol. 131, p. 081001, 2009.
[131] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*: Addison-Wesley Professional, 1999.
[132] R. Henderson, "Breaking the chains of embedded knowledge: Architectural innovation as a source of competitive advantage," *Design Management Journal (Former Series),* vol. 2, pp. 43-47, 1991.
[133] N. P. Repenning, "Understanding fire fighting in new product development□," *Journal of Product Innovation Management,* vol. 18, pp. 285-300, 2001.
[134] N. P. Repenning and J. D. Sterman, *Nobody ever gets credit for fixing problems that never happened.* Harvard Bus Pub, 2001.
[135] N. P. Repenning and J. D. Sterman, "Capability traps and self-confirming attribution errors in the dynamics of process improvement," *Administrative science quarterly,* vol. 47, pp. 265-295, 2002.
[136] D. Jackson, "Dependable software by design," *Scientific American,* vol. 294, pp. 68-75, 2006.
[137] W. W. Gibbs, "Software's chronic crisis," *Scientific American,* vol. 271, pp. 72-81, 1994.
[138] N. Leveson, "Medical devices: The therac-25," *Appendix of: Safeware: System Safety and Computers,* 1995.
[139] N. G. Leveson, "Role of software in spacecraft accidents," *Journal of spacecraft and Rockets,* vol. 41, pp. 564-575, 2004.
[140] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an N-version software experiment," *IEEE Transactions on software Engineering,* pp. 238-247, 1990.
[141] E. Rechtin and M. W. Maier, "The art of systems architecting," 1997.

[142] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," 1976, pp. 592-605.
[143] N. G. Leveson, "Complexity and Safety," in *Complex Systems Design & Management*, O. Hammami, D. Krob, and J.-L. Voirin, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 27-39.
[144] E. W. Dijkstra, *A discipline of programming* vol. 1: prentice-hall, 1976.
[145] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576-580, 1969.
[146] E. J. Weyuker, "Testing component-based software: A cautionary tale," *Software, IEEE*, vol. 15, pp. 54-59, 1998.
[147] T. Yamaura, "How to design practical test cases," *Software, IEEE*, vol. 15, pp. 30-36, 1998.
[148] T. Yamaura, "Why Johnny can't test [software]," *Software, IEEE*, vol. 15, pp. 113-115, 1998.
[149] K. J. Britton and D. M. Schaible, "Testing in NASA Human-Rated Spacecraft Programs: How much is just enough?," *Signature*, 2003.
[150] M. S. Phadke, *Quality engineering using robust design*: Prentice Hall PTR, 1995.
[151] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *Software Engineering, IEEE Transactions on*, vol. 35, pp. 864-878, 2009.
[152] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, pp. 653-661, 2000.
[153] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, pp. 169-180, 2000.
[154] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?," *Software Engineering, IEEE Transactions on*, vol. 34, pp. 497-515, 2008.
[155] R. W. Selby and V. R. Basili, "Error localization during software maintenance: Generating hierarchical system descriptions from the source code alone," in *Software Maintenance, 1988., Proceedings of the Conference on*, 1988, pp. 192-197.
[156] J. B. Lohse and S. H. Zweben, "Experimental evaluation of software design principles: an investigation into the effect of module coupling on system modifiability," *Journal of Systems and Software*, vol. 4, pp. 301-308, 1984.
[157] D. A. Troy and S. H. Zweben, "Measuring the quality of structured designs," *Journal of Systems and Software*, vol. 2, pp. 113-120, 1981.
[158] M. E. Sosa, J. Mihm, and T. Browning, "Linking cyclicality and product quality," INSEAD, INSEAD Faculty & Research Working Paper 2012/137/TOM, 2012.
[159] C. F. Kemerer, "An empirical validation of software cost estimation models," *Communications of the ACM*, vol. 30, pp. 416-429, 1987.
[160] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *Software Engineering, IEEE Transactions on*, vol. 33, pp. 33-53, 2007.
[161] B. Boehm, C. Abts, and S. Chulani, "Software development cost estimation approaches—A survey," *Annals of Software Engineering*, vol. 10, pp. 177-205, 2000.
[162] B. W. Boehm, R. Madachy, and B. Steece, *Software Cost Estimation with Cocomo II with Cdrom*: Prentice Hall PTR, 2000.
[163] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *Software Engineering, IEEE Transactions on*, pp. 345-361, 1978.
[164] M. Cohn, *Agile estimating and planning*: Prentice Hall, 2006.
[165] L. M. Laird and M. C. Brennan, *Software measurement and estimation: a practical approach* vol. 2: Wiley-IEEE Computer Society Press, 2006.
[166] R. Lagerström, L. Marcks von Würtemberg, H. Holm, and O. Luczak, "Identifying factors affecting software development cost," in *Proc. of the Fourth International Workshop on Software Quality and Maintainability (SQM)*, 2010.
[167] J. Printz, "A Natural Measure for Denoting Software System Complexity," in *Complex Systems Design & Management*, M. Aiguier, F. Bretaudeau, and D. Krob, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 173-195.
[168] C. Y. Baldwin and K. B. Clark, "Modularity in the design of complex engineering systems," *Complex Engineered Systems*, pp. 175-205, 2006.

[169] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, "Identifying modular and integrative systems and their impact on design team interactions," *Journal of Mechanical Design,* vol. 125, p. 240, 2003.
[170] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," *ACM SIGSOFT Software Engineering Notes,* vol. 26, pp. 99-108, 2001.
[171] A. Yassine, N. Joglekar, D. Braha, S. Eppinger, and D. Whitney, "Information hiding in product development: The design churn effect," *Research in Engineering Design,* vol. 14, pp. 145-161, 2003.
[172] M. Carrascosa, S. D. Eppinger, and D. E. Whitney, "Using the design structure matrix to estimate product development time," presented at the 1998 ASME Design Engineering Technical Conferences, Atlanta, Georgia, USA, 1998.
[173] J. D. Blackburn, G. D. Scudder, and L. N. Van Wassenhove, "Improving speed and productivity of software development: a global survey of software developers," *Software Engineering, IEEE Transactions on,* vol. 22, pp. 875-885, 1996.
[174] A. D. MacCormack, C. F. Kemerer, M. Cusumano, and B. Crandall, "Trade-offs between productivity and quality in selecting software development practices," *Software, IEEE,* vol. 20, pp. 78-85, 2003.
[175] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Communications of the ACM,* vol. 36, pp. 81-94, 1993.
[176] S. McConnell, "What does 10x mean? Measuring variations in programmer productivity," in *Making Software: What Really Works, and Why We Believe It, A. Oram and G. Wilson, Eds., O'Reilly Series. O'Reilly Media,* O. a. Wilson, Ed., ed, 2011, pp. 567-575.
[177] B. Curtis, "Substantiating programmer variability," *Proceedings of the IEEE,* vol. 69, pp. 846-846, 1981.
[178] H. D. Mills, "Software productivity," 1988.
[179] T. DeMarco and T. Lister, "Programmer performance and the effects of the workplace," in *Proceedings of the 8th international conference on Software engineering,* 1985, pp. 268-272.
[180] B. Curtis, E. M. Soloway, R. E. Brooks, J. B. Black, K. Ehrlich, and H. R. Ramsey, "Software psychology: The need for an interdisciplinary program," *Proceedings of the IEEE,* vol. 74, pp. 1092-1106, 1986.
[181] D. N. Card, "A software technology evaluation program," *Information and Software Technology,* vol. 29, pp. 291-300, 1987.
[182] B. W. Boehm and P. N. Papaccio, "Understanding and controlling software costs," *Software Engineering, IEEE Transactions on,* vol. 14, pp. 1462-1477, 1988.
[183] J. D. Valett and F. E. McGarry, "A summary of software measurement experiences in the software engineering laboratory," *Journal of Systems and Software,* vol. 9, pp. 137-148, 1989.
[184] W. F. Boh, S. A. Slaughter, and J. A. Espinosa, "Learning from experience in software development: A multilevel analysis," *Management Science,* vol. 53, pp. 1315-1331, 2007.
[185] P. C. Pendharkar and G. H. Subramanian, "An empirical study of ICASE learning curves and probability bounds for software development effort," *European Journal of Operational Research,* vol. 183, pp. 1086-1096, 2007.
[186] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Journal of Systems and Software,* vol. 7, pp. 341-355, 1987.
[187] E. T. Chen, "Program complexity and programmer productivity," *Software Engineering, IEEE Transactions on,* pp. 187-194, 1978.
[188] R. Katz, *The human side of managing technological innovation: A collection of readings*, second ed.: Oxford University Press, 2004.
[189] S. G. Westlund and J. C. Hannon, "Retaining talent: Assessing job satisfaction facets most significantly related to software developer turnover intentions," *Journal of Information Technology Management,* vol. 19, pp. 1-15, 2008.
[190] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp, "Motivation in software engineering: A systematic literature review," *Information and Software Technology,* vol. 50, pp. 860-878, 2008.
[191] F. R. Tanner, "On motivating engineers," in *Engineering Management Conference, 2003. IEMC'03. Managing Technologically Driven Organizations: The Human Side of Innovation and Change*, 2003, pp. 214-218.
[192] S. Ramachandran and S. V. Rao, "An effort towards identifying occupational culture among information systems professionals," in *Proceedings of the 2006 ACM SIGMIS CPR conference on*

computer personnel research: Forty four years of computer personnel research: achievements, challenges & the future, 2006, pp. 198-204.

[193] D. M. Sharman and A. A. Yassine, "Characterizing complex product architectures," *Systems Engineering*, vol. 7, pp. 35-60, 2004.

[194] J. N. Warfield, "Binary matrices in system modeling," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 3, pp. 441-449, 1973.

[195] J. Luo and C. L. Magee, "Detecting evolving patterns of self□organizing networks by flow hierarchy measurement," *Complexity*, vol. 16, pp. 53-61, 2011.

[196] J. Luo and C. L. Magee (advisor), "Hierarchy in industry architecture: Transaction strategy under technological constraints," PhD in Engineering Systems Doctoral Dissertation, Engineering Systems Division, Massachusetts Institute of Technology, 2010.

[197] K. Hölttä-Otto and O. de Weck, "Degree of modularity in engineering systems and products with technical and business constraints," *Concurrent Engineering*, vol. 15, pp. 113-126, 2007.

[198] K. Imai, G. King, and O. Lau, "Negbin: Negative binomial regression for event count dependent variables," *Zelig: Everyone's Statistical Software*, 2007.

[199] K. Imai, G. King, and O. Lau, "Toward a common framework for statistical analysis and development," *Journal of Computational and Graphical Statistics*, vol. 17, pp. 892-913, 2008.

[200] K. Imai, G. King, and O. Lau, "Zelig: Everyone's statistical software," *R package version*, pp. 3.4-5, 2009.

[201] W. N. Venables and B. D. Ripley, *Modern applied statistics with S*: Springer, 2002.