Submission ID: 16544

Modular Design and Option Value: The Impact of System Architecture on Developer Productivity

Abstract

How do system design decisions affect the ability of a system to adapt to uncertain future demands? While a variety of studies explore the link between a system's design and its *technical* performance (e.g., the number of defects experienced in future) few empirical studies provide robust quantitative data on how design decisions create (or destroy) value through their impact on a systems ability to adapt to *future* needs. Despite strong theoretical and descriptive evidence that modular (i.e., loosely-coupled) systems are easier to adapt than tightly-coupled systems, the lack of empirical confirmation of the size and direction of such differences represents a serious gap in our knowledge with respect to how modular designs can create "option" value.

We address this gap by analyzing the relationship between design decisions and *developer productivity* in a large, successful commercial software system. Our analysis relies upon measuring the level of coupling of 14,000 components in the system, and using these measures to predict the productivity of developers over 8 successive six-month periods. Critically, we adopt a panel-data approach, to control for differences in developer skills, which are likely to dwarf other predictors of productivity. We show that a developer contributing to the most modular parts of the system is 75% more productive than a developer contributing to the least modular parts. Our findings are a critical first step in highlighting the option value of modularity, in that adapting to future demands (i.e., creating new features) takes less time and effort for more modular parts of a system.

1. Introduction

How do system design decisions affect the ability of a system to respond to future demands? A variety of studies have examined the link between system design and performance, with a view to developing insights into how design decisions should be made during the development of new and complex technological systems (Banker et al, 1993; Banker and Slaughter, 2000). This work reveals the critical impact of architectural choices in resolving potential trade-offs between, on the one hand, system performance (e.g., in terms of speed, capacity, flexibility, etc.) and on the other, equally desirable attributes such as reliability and maintainability, which may be associated with designs of a rather different nature (e.g., those possessing greater modularity). Few empirical studies however, explore how design decisions create options to improve *in future*. While existing theory predicts that modular systems will be more adaptable than integral systems (Baldwin and Clark, 2000) no work empirically confirms this prediction, nor indicates the magnitude of differences in the degree of adaptation between designs.

This topic is especially relevant in the software industry, given the dynamics of how software is developed. In particular, software systems rarely die. Instead, each new version forms a platform upon which subsequent versions are built. With this approach, today's developers bear the consequences of system design decisions made long ago (MacCormack et al, 2007a). Unfortunately, the first designers of a system often have different objectives from those that follow, especially if the system is successful and long lasting (something that may be quite uncertain at the time of its design). While early designers prioritize speed and functionality, later designers may place greater value on reliability and adaptability. Rarely are all these objectives met by the same design.

Complexity across systems, and the complexity of different regions within the same system, varies widely. In the battle to channel the behavior of a large system so that complexity is managed, the principal weapon in the designer's arsenal is *architecture*. 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 architecture endow systems with inherently beneficial properties, and also address basic human limitations in dealing with complexity. Design is not easy or straightforward, however. Weighing the costs and benefits of alternative choices is difficult. Designers must choose between competing ways to decompose a system into hierarchical structures and competing criteria for determining which functionality should be clustered in each module 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. As a result, a single system may have regions with widely varying levels of modularity, associated costs and consequent abilities to adapt.

In this study, we evaluate the relationship between system design decisions and developer productivity in a large, mature, commercial software system. We characterize the system's design using a network analysis technique called Design Structure Matrices (DSMs) (Steward, 1981; Eppinger et al, 1994). Our analysis allows us to determine the level of coupling between each component, and thereby to evaluate which are "Core" (tightly-coupled to others) and which are "Peripheral" (loosely-coupled to others). Our objective is to understand the extent to which these different levels of component modularity drive differences in the productivity of developers when developing new features. This measure provides a proxy for the broader "option value" associated with different parts of the system's design. If developing a new feature involves less time and effort in one part of the design versus another, that part of the design, by definition, has higher option value (all else being equal) for responding to future demands.

Software is an ideal context in which to study these issues given the information-based nature of the product. Software code can be analyzed automatically to identify the level of coupling between components, and hence determine which are highly interdependent, versus those that are peripheral (MacCormack et al, 2012). Furthermore, using software version control systems, we can directly trace the work products of individual developers, to the parts of the design that they work within.

Our findings make an important contribution to the literature exploring the design and management of complex technological systems in general, and software in particular. We find significant differences in developer productivity across this large, commercial software system comprising 14,000 components. Specifically, developers working in the most modular (i.e., most loosely-coupled) parts of the system are 75% more productive in developing new features than developers working in the least modular parts of the system. These differences are all the more dramatic given we use a panel data approach which controls for individual differences in skill. In particular, we exploit variations in the proportion of work developers do in different parts of the system in 8 different time periods, adopting a "differences within developer" approach that allows us to tease out the true impact of architecture.

The paper proceeds as follows. In the next section, we review the prior literature on system design, focusing on work that explores the degree to which measures of system architecture have been shown to predict performance. We then describe our research methods, which make use of a technique called Design Structure Matrices (DSMs) to understand the structure of a system by measuring the level of coupling between components. Next, we introduce the context for our study and describe the large, commercial software system that we analyze. Finally, we report our empirical results, and discuss their implications for the academy and for managers.

2. Literature Review

A large number of studies contribute to our understanding of the design of complex systems (Holland, 1992; Kaufman, 1993; Rivkin, 2000; Rivkin and Siggelkow, 2007). Many of these studies are situated in the field of technology management, exploring factors that influence the design of physical or information-based products (Braha et al, 2006). Products are complex systems in that they comprise a large number of components with many interactions between them. The scheme by which a product's functions are allocated to these components is called its "architecture" (Ulrich, 1995; Whitney et al, 2004). Understanding how architectures are chosen, how they perform and how they can be changed are critical topics in the study of complex system design.

Modularity is a concept that helps us to characterize different designs. It refers to the way that a product's architecture is decomposed into different parts or modules. While there are many definitions of modularity, authors tend to agree on the concepts that lie at its heart; the notion of interdependence within modules and independence between modules (Ulrich, 1995). The latter concept is often referred to as "loose-coupling." Modular designs are loosely-coupled in that changes made to one module have little impact on others. Just as there are degrees of coupling, there are degrees of modularity.

The costs and benefits of modularity have been discussed in a stream of research that has sought to examine its impact on the management of complexity (Simon, 1962), product line architecture (Sanderson and Uzumeri, 1995), manufacturing (Ulrich, 1995), process design (MacCormack, 2001) process improvement (Spear and Bowen, 1999) and industry evolution (Baldwin and Clark, 2000). Despite the appeal of this work however, few studies have used robust empirical data to examine the relationship between measures of modularity and the outcomes that it is thought to impact (Schilling, 2000; Fleming and Sorenson, 2004). Most studies are conceptual or descriptive in nature.

Studies that attempt to measure modularity typically focus on capturing the level of coupling that exists between different parts of a system. In this respect, the most promising technique comes from the field of engineering, in the form of the Design Structure Matrix (DSM). A DSM highlights the inherent structure of a design by examining the dependencies that exist between its constituent elements in a square matrix (Steward, 1981; Eppinger et al, 1994; Sosa et al, 2003). These elements can represent design tasks, design parameters or the actual components. Metrics that capture the degree of coupling between elements have been calculated from a DSM, and used to compare different architectures (Sosa et al, 2007). DSMs have also been used to explore the degree of alignment between task dependencies and project team communications (Sosa et al, 2004). Recent work extends this methodology to show how design dependencies can be automatically extracted from software code and used to understand architectural differences (MacCormack et al, 2006). We use this approach in this paper.

2.1: System Design, Maintenance and Adaptation

The most significant empirical studies exploring the link between system design, modularity and the cost of maintenance and adaptation have come from the field of software. This topic is of particular importance given how software is developed. Rarely do software projects start from scratch. Instead, the prior version is used as a platform upon which new functionality is built. In many projects, "legacy" code exceeds newly developed code, so significant efforts must be devoted to maintenance. Understanding how software systems should be designed, and how design decisions drive subsequent costs to maintain and adapt a system over time, is a crucial area for attention.

The formal study of software modularity began with Parnas (1972) who proposed the concept of "information hiding" as a mechanism for dividing code into modular units. This required designers to separate a module's internal details from its external interfaces, reducing the coordination costs involved in system development and facilitating changes to modules without affecting other parts of the design. Subsequent authors built on this work, proposing metrics to capture the level of *coupling* between modules and *cohesion* within modules (e.g., Selby and Basili, 1988; Dhama, 1995). Modular designs were asserted to have both low coupling and high cohesion. This work complemented studies that sought to measure the complexity of the design for the purposes of predicting the productivity of system development (e.g., McCabe 1976;

Halstead, 1976). Whereas measures of complexity focus on the *number and nature* of the elements in a system, measures of modularity focus on linkages *between* these elements.

Studies seeking to link measures of system design with the costs of maintenance focus on predicting the cost and frequency of changes across systems. Banker et al (1993) examine 65 maintenance projects across 17 systems and find that project costs increase with system complexity, as measured by the average “procedure” size and the number of “non-local” branching statements (i.e., component interdependency). Kemerer and Slaughter (1997) examine modification histories for 621 software modules and find that enhancement and repair frequency increase with module complexity, as measured by the number of module decision paths (McCabe, 1976) normalized by size. Banker and Slaughter (2000) examine three years of modification data from 61 software applications and find that total modification costs increase with application complexity, as measured by the number of input/output data elements per unit of functionality. Finally, Barry et al (2006) examine the evolution of 23 applications over a 20-year period and find that an increase in the use of standard components (a proxy for modularity) is associated with a decline in the frequency and magnitude of modifications.

The studies above make major contributions to our understanding of the characteristics that drive productivity and quality in software system development. However, they don’t address several critical issues that must be resolved in order to assess the option value that stems from greater modularity in a software system. First, most of these studies measure the mean complexity of *components* in a system, but fail to capture data on the linkages *between* components – the key driver of modularity. Second, most studies use a cross-sectional research design where the primary unit of analysis is the system. They do not explore the relative differences in performance between the components with different levels of modularity located *within* the same system. Finally, these studies typically use the source file as the level of analysis, and not the individual developer. Hence we do not know how differences in modularity impact an *organization's* ability to adapt, by developing new features in response to new demands.

To address the first concern, we characterize a system's design in terms of the coupling *between* components, as opposed to the complexity of the components themselves. To address the second concern, we adopt a research design that captures data at the component level, allowing us to determine if there are systematic differences in performance that are explained by levels of coupling. To address the third concern, we a research design that has the individual developer as the unit of analysis, exploring how the productivity of each developer is influenced by where in the system design he/she is asked to work. Hence our research hypothesis can be stated as follows:

H1: Developers working in more modular (i.e., more loosely-coupled) parts of the system will be more productive than developers working in less modular (i.e., more tightly-coupled) parts of the system.

3. Research Methods

Below, we describe how we apply DSMs to analyze a large, commercial software system, allowing us to determine the level of coupling between system components.

3.1 Applying DSMs to the Analysis of Software Systems[1]

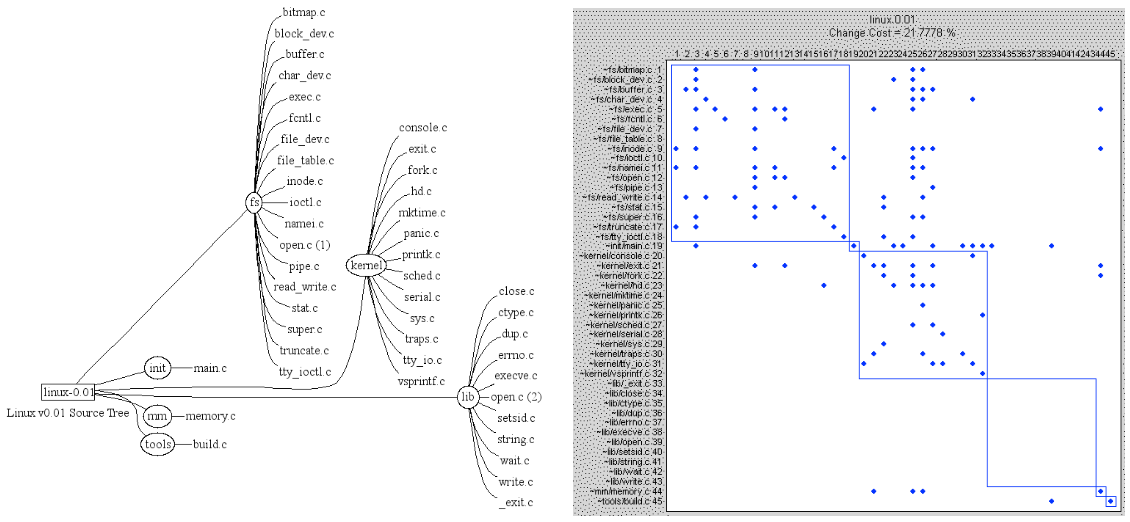
There are two choices to make when applying DSMs to a software product: The level of analysis and the type of dependency to analyze. With regard to the former, there are several levels at which a DSM can be built: The *directory* level, which corresponds to a group of source files that all relate to the same subsystem; the *source file* level, which corresponds to a collection of linked processes and functions; and the *function* level, which corresponds to a set of instructions that perform a very specific task. We analyze designs at the source file level for a number of reasons. First, source files are the level most directly equivalent to the components of a physical product. Second, most prior work on software design uses the source file as the primary level of analysis (e.g., Eick et all, 1999; Rusovan et all, 2005; Cataldo et al, 2006). Third, tasks and responsibilities are typically allocated to programmers at the source file level. Finally, software development tools use the source file as the unit of analysis for updating and evolving the design.

There are many types of dependency between source files in a software product.[2] We focus on several important dependency types used in prior work on system design (Banker and Slaughter, 2000; Rusovan et al, 2005) specifically; function calls, class method calls, class method definitions, and subclass definitions. Dependencies are captured between source files, in a specific direction. For example, if FunctionA in SourceFile1 calls FunctionB in SourceFile2, then we note that SourceFile1 depends upon (or "uses") SourceFile2. This dependency is marked in location (1, 2) in the DSM. Critically, this does *not* imply that SourceFile2 depends upon SourceFile1; the dependency is not symmetric unless SourceFile2 also calls a function in SourceFile1.

[1] The methods described here build on prior work in this field (MacCormack et al, 2006; Sosa et al, 2009).
[2] For a discussion of different dependency types, see Shaw and Garlan (1996) and Dellarocas (1996).

To capture the dependencies, we use a commercial tool called a "Call Graph Extractor" (Murphy et al, 1998), which takes software code as input, and outputs the dependencies between each source file.[3] We display this data in a DSM using the *Architectural View*. This view groups each source file into a series of nested clusters defined by the directory structure, with boxes drawn around each layer in the hierarchy. To illustrate, we show the Directory Structure and Architectural View for Linux v0.01 in **Figure 1**. This system comprises six subsystems, three of which contain only one component and three of which contain between 11-18 components. In the Architectural view, each "dot" represents a dependency between two components (i.e., source files).

Figure 1: The Directory Structure and Architectural View of Linux version v0.01.



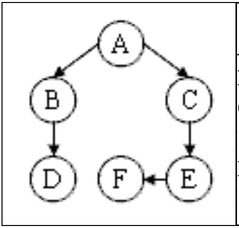
[3] Dependencies can be extracted statically (from the source code) or dynamically (when the code is run). We use a static call extractor because it uses source code as input, does not rely on program state (i.e., what the system is doing at a point in time) and captures the system structure from the designer's perspective.

3.2 Measuring the Level of Component Coupling

In order to assess system structure, we develop measures of the degree to which components are coupled to each other. To achieve this, we capture all the *direct and indirect* dependencies a component possesses with other components, a concept known as "Visibility" (Sharmine and Yassine 2004; Warfield 1973). To account for the fact that software dependencies are asymmetric we develop separate measures for dependencies that flow into a component ("Fan-In") versus those that flow out from it ("Fan-Out").

To illustrate, consider the system depicted in **Figure 2** in graphical and DSM form. Element A depends upon elements B and C. In turn, element C depends upon element E, hence a change to element E may have a *direct* impact on element C, and an *indirect* impact on element A, with a "path length" of two. Similarly, a change to element F may have a direct impact on element E, and an indirect impact on elements C and A, with a path length of two and three, respectively. Element A therefore has a Fan-Out Visibility of five, given it is connected to all other elements, either directly or indirectly.

Figure 2: Example System in Graphical and DSM Form



$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ B & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ C & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ D & 0 & 0 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ F & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

To calculate the visibility of each element, we use matrix multiplication. By raising the DSM to successive powers of n, we obtain the direct and indirect dependencies that exist for successive path lengths n. Summing these matrices yields the visibility matrix, which shows the direct and indirect dependencies between elements *for all possible path lengths* up to the maximum, defined by the size of the DSM.[4] **Figure 3** illustrates the derivation of this matrix for the example above.

Figure 3: The Derivation of the Visibility Matrix

M^0

	A	B	C	D	E	F
A	**1**	0	0	0	0	0
B	0	**1**	0	0	0	0
C	0	0	**1**	0	0	0
D	0	0	0	**1**	0	0
E	0	0	0	0	**1**	0
F	0	0	0	0	0	**1**

M^1

	A	B	C	D	E	F
A	0	**1**	**1**	0	0	0
B	0	0	0	**1**	0	0
C	0	0	0	0	**1**	0
D	0	0	0	0	0	0
E	0	0	0	0	0	**1**
F	0	0	0	0	0	0

M^2

	A	B	C	D	E	F
A	0	0	0	**1**	**1**	0
B	0	0	0	0	0	0
C	0	0	0	0	0	**1**
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

M^3

	A	B	C	D	E	F
A	0	0	0	0	0	**1**
B	0	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

M^4

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	0	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

$V = \Sigma M^n ; n = [0,4]$

	A	B	C	D	E	F
A	**1**	**1**	**1**	**1**	**1**	**1**
B	0	**1**	0	**1**	0	0
C	0	0	**1**	0	**1**	**1**
D	0	0	0	**1**	0	0
E	0	0	0	0	**1**	**1**
F	0	0	0	0	0	**1**

The measures of component visibility are derived from this matrix. Fan-In Visibility (VFI) is obtained by summing down the columns; Fan-Out Visibility (VFO) is obtained by summing along the rows. For comparisons between systems of different sizes, VFI and VFO can be expressed as a percentage of the number of components in a system.

Once computed, VFI and VFO scores for components across a system can be rank-ordered and plotted to see their distributions. Figure 4 shows the distribution of visibility scores for one of the releases in the system we analyze. When these distributions contain large steps demarcating the boundary between files that are loosely- and tightly-connected, as is the case here, it indicates the network has a "core-periphery" structure (MacCormack, 2010). In particular, this system has a large core of files that are interdependent, sharing the same levels of Visibility Fan-In and Visibility Fan-Out. This

[4] We choose to include the matrix for n=0, implying that an element will always depend upon itself.

is the largest cycle in the system. We define it as the "Core". Other types of files are defined by their visibility levels relative to the core, as noted in Table 1.

Figure 4: Distribution of Visibility Measures by Value reveals Bipolar Distribution

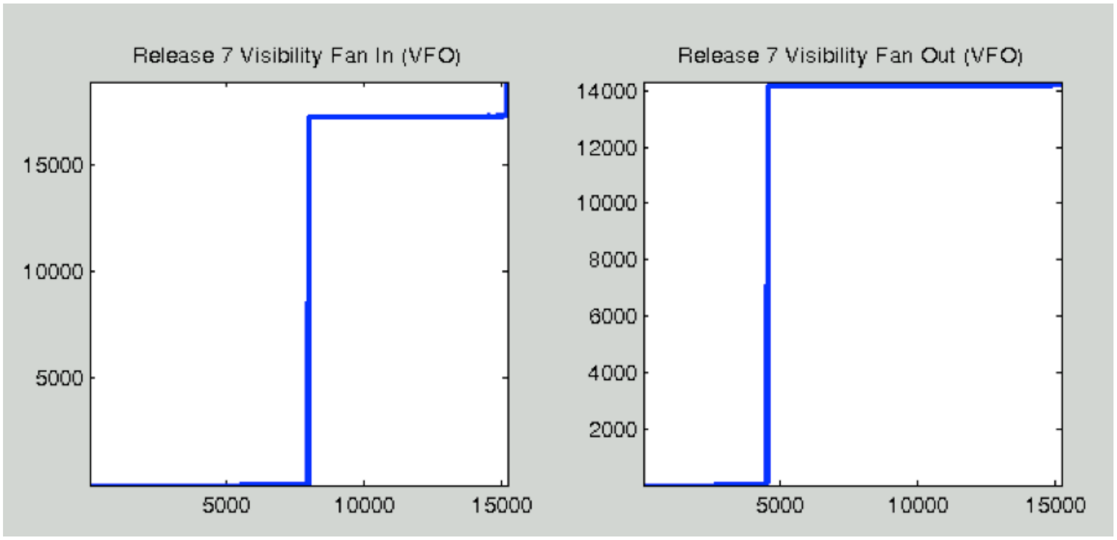


Table 1: Mapping Visibility Scores to File Type

VFI	VFO	File Type	Description
High	High	**Core**	Core regions form highly integral clusters, containing large cycles in which components are directly or indirectly co-dependent. They regions are hard to decompose into smaller parts and may become unmanageable if they become too large.
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.
Low	Low	**Peripheral**	Peripheral components do not influence and are not influenced by much of the rest of the system.

In this research, we use each file's classification as core, utility, control, or periphery, as our indicator of the level of modularity. Core files are the least modular because their high levels of connectedness indicate that they are in regions of the network that are coupled by large cycles. Peripheral files are the most modular, because they are only loosely-connected to other parts of the system.

4. Empirical Data and Analytical Approach

The software under examination in this study is a portion of a very large code-base owned by a commercial firm with many years of market success. Over time, thousands of professionals wrote software consisting of hundreds of thousands of files and millions of lines of code in several 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 six-month cadence. Within this cadence, teams have coordinated periods for planning, feature development, and quality control. Each development cycle concludes with the release of a new version of the software customers. Information was extracted from software source code for eight successive shipped versions of the software and information about periods of development activity leading up to each release. Architecture metrics were extracted from source code for each version of the software. Information about development costs the organization incurred were extracted from version control systems, change tracking systems, and human resource databases.

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 share 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 suite. Teams use a shared change tracking system, source code version control system, shared code validation tools, and a common project management processes.

Iron Bridge was chosen for investigation because it represents a natural experiment. Because teams at Iron Bridge have independent control over software but centralized calendars and tools, the company has done some a number of things that enable this research. First, the effect of process, tools, and schedule are controlled. The impact of the architecture on costs incurred by the organization when developing within 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 can be established across teams. Thirdly, Iron Bridge's history of data-collection and 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 so cannot explain productivity differences because they do not know the true efforts applied to development. 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. In addition, access to human-resource databases allows us to control for time with the company and managerial status. Finally, Iron Bridge maintains an integrated change tracking and version control system. Policy dictates that developers include the identification number of specific features or bugs being tracked through the development pipeline when submitting software patches 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 intact a substantial portion of the time.

Iron Bridge's codebase consists of code written in C++, Java, and a scripting language similar to Perl. The C++ portion of this codebase was chosen for this study to make our analysis tractable. This set of files originally began as C language code, and evolved to contain a mix of procedural C and object-oriented C++ language constructs over time. 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. Second, the C++ portion of the codebase contains some of the oldest code, and therefore 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. Fourth, C++ code is the heart of the overall system. It implements many of the most important functionality and algorithms. This portion of the code forms a platform on top of which the Java and scripting code rest.[5]

[5] The primary purpose of the Java portion of the codebase is to implement graphical user interfaces (GUIs) on top of functionality provided by C++ code and the scripting language's interpreter is implemented in

Files were removed from the sample for a variety of reasons. Steps were taken to clean the sample of C++ studied. 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.
- Header files were removed because their contents consist of interface descriptions rather than implementation details, and because they are much smaller than other files.

Figure 5 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 core, utility, control, or peripheral. 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 process of lower diagonalization congregates the four distinct DSM file types into distinct regions in the picture. The small box in the upper left contains utility files, followed by core, peripheral, and control files. The bottom two panels plot the visibility scores for files in a sorted order. When this is done, the bimodal nature of the 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 file type classifications. The prominent step in the middle of each graph is the demarcation line between "low" and "high" scores for purposes of defining categories. Once files are assigned to "low" or "high" regions on both visibility dimensions, classification is straightforward. Table 2 shows the number of files for each

C++, meaning that each line of scripted code is ultimately interpreted and executed by C++ code in the codebase under examination.

release, and how these split across the different categories of file type. Note the growth of the codebase and of the size of the core through time.

Figure 5: Release 7 DSMs and Visibility Plots

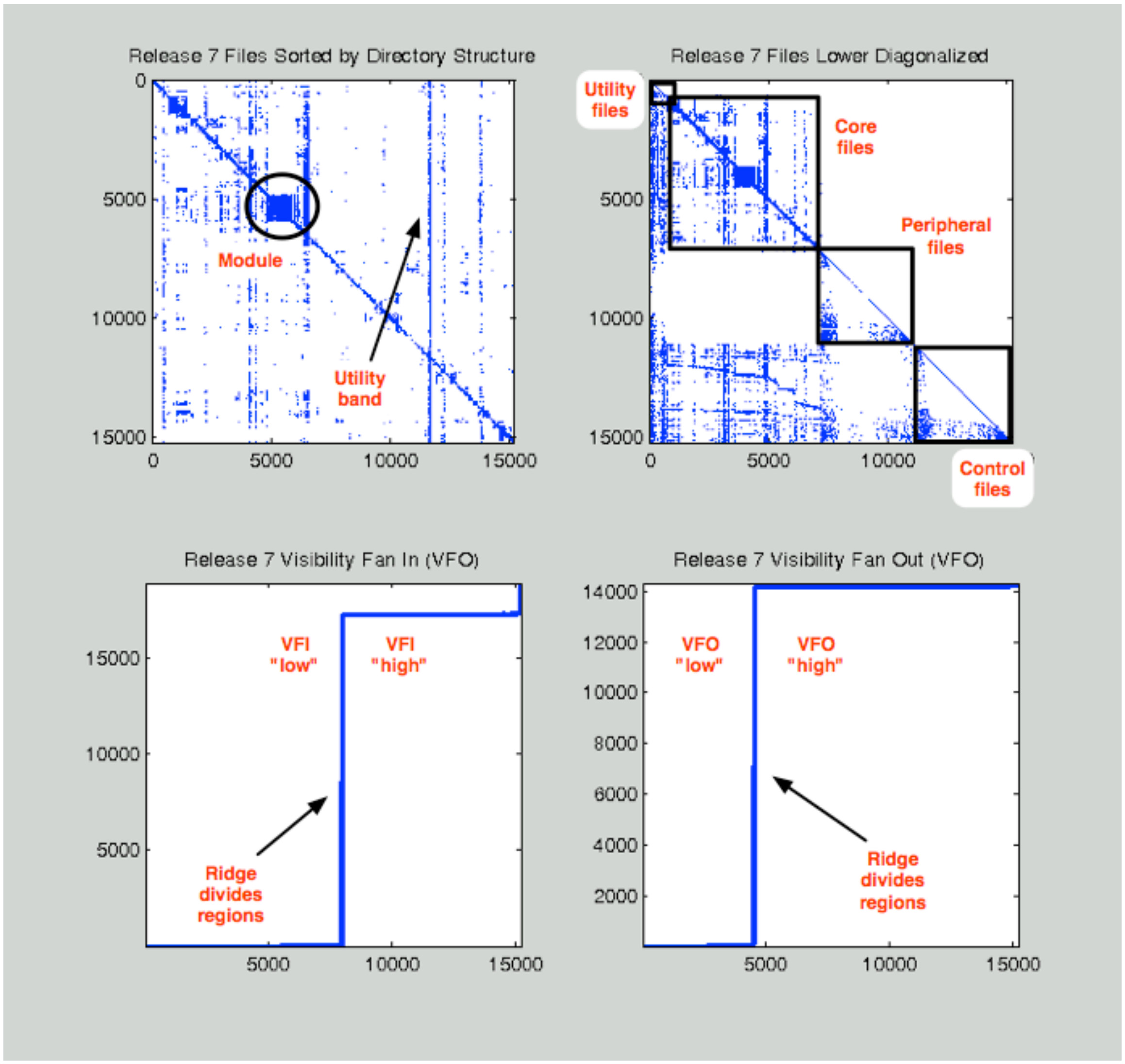


Table 2: File Count Broken Down by Type of File

Release	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Total number of files	9937	10447	10671	11576	12186	12311	13295	13941
Architectural 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

5. Empirical Results

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 an alternative explanation for why a developer's productivity may vary. Controls tested include a developer's tenure with the firm, managerial status, fraction of activity working in new (rather than legacy) code, fraction of activity spent fixing bugs, and fraction of activity working in files with high McCabe Cylomatic complexity. The latter is a common measure of complexity in software, which focuses on the *internal* complexity of components, rather than the position of components in the network of dependencies. The goal of our analysis was to explore whether our measures of modularity predict developer productivity after controlling for other factors.

The sample of developers used to explore this question included 178 people who wrote 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 patches. Of these 2 million lines produced, 1.1 million were created to implement features or perform some other non-bug related tasks and 800,000 were produced to fix bugs. Our empirical measures are described in detail in Table 3.

Table 3: Variables Included In Models Predicting Developer Productivity

Variable	Purpose	Type	Description
Lines of code produced to implement features or 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 patch was associated with multiple change requests, some of which were to fix bugs, then only a portion of the patch will count as a bug fix, and the rest will be considered a feature or task. The number of lines of code in a patch 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 patch was associated with multiple change requests, only some of which were to fix bugs, then only a portion of the patch will count as a bug fix. The number of lines of code in a patch will be allocated proportionally based on the proportion allocated to bugs and non-bugs.
Lines of code produced to fix bugs, implement features, or do other tasks	Dependent Variable	Count	The number of lines of code produced by a developer during a release window. All patches 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 patches 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 patch 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 patch was associated with multiple change requests, only some of which were to fix bugs, then only a portion of the patch will count as a bug fix. The number of lines of code in a patch 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. [114]
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 [1, 2]

Table 4 below shows descriptive data on 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, 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 observation is striking, but is a generally understood phenomenon.)

Table 4: 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
Patches 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 5 below breaks down development by type of activity being performed (feature work vs. bug fix) and the location of work. Approximately half 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 5: Activity For Average Developer by Task and File Type by 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
Lines produced in DSM	2471	3030	3160	2784	3622	3261	2582	2763
Type of patch								
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%

In order to analyze the determinants of developer productivity, we construct three 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, tasks, and bug fixes (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 (i.e., low levels of modularity).

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 *within* individuals rather than *between* them. That is, these regressions are designed to determine if individuals are less productive during releases in which they worked in more modular parts of the design, rather than to determine if a group of people working in the more modular parts are more productive than a group working in less modular parts.

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. 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 built into the R statistical software suite was used to run regressions and subsequent simulations to estimate the value of parameters.

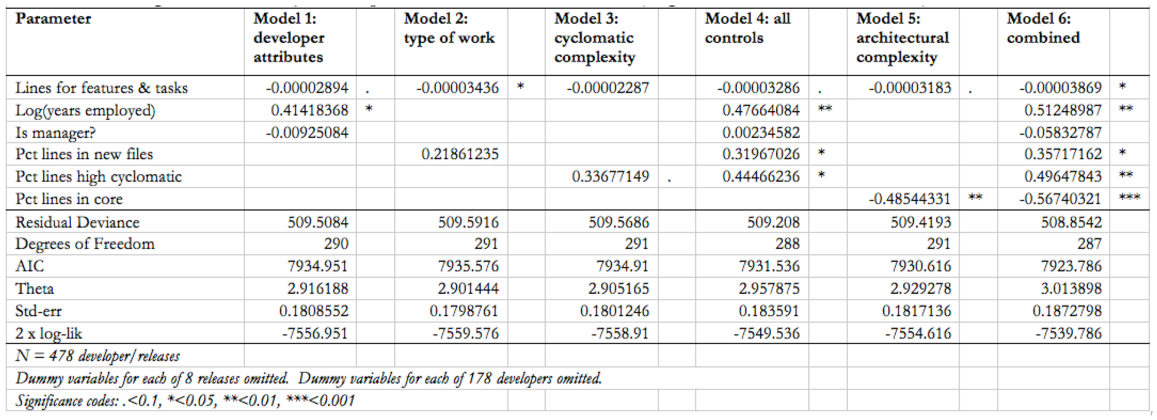
The results for regressions predicting the productivity of an individual during a release window are shown below. Note that while each of these regressions contained dummy variables for the release and the individual, these dummies were omitted from tables. Table 6 shows results for regressions in which 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 developing features and working in new (rather than legacy) files. They are less productive when developing features and working in files with high McCabe cyclomatic complexity. After all the controls are included in the model, developers are found to be less productive when developing features and working in core files. This result is significant at the 5% level.

Table 6: Predicting Developer Lines of Code for New Features (one Release)

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 7 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 as a control.) Developers are 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 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 has a stronger negative impact on the productivity of those fixing bugs than those implementing features.

Table 7: Predicting Developer Lines of Code to Fix Defects (one Release)



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 8 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 are 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 developer productivity. Finally, during time periods in which an individual worked more in core files, the number of total 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.

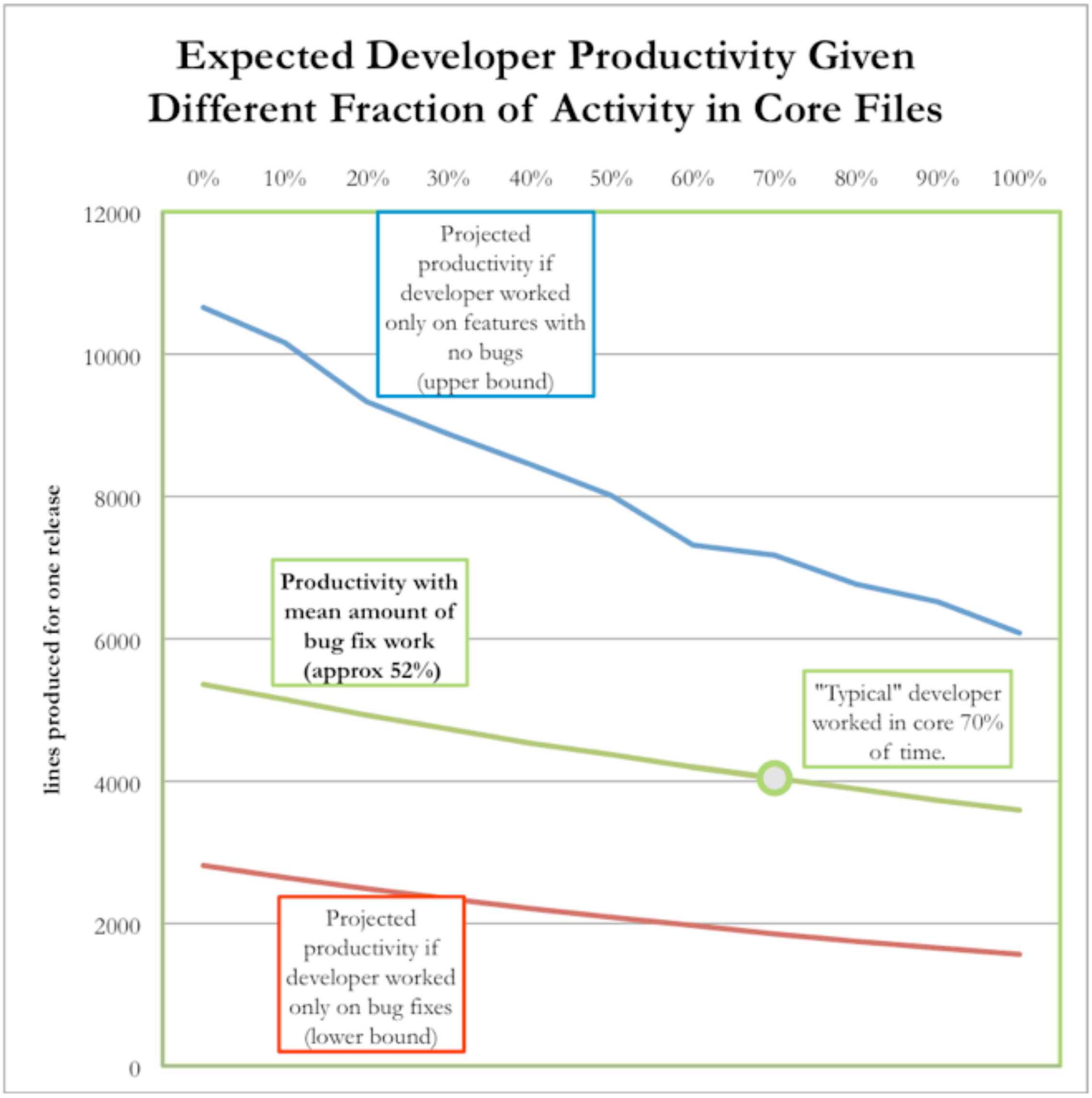
Table 8: Predicting Total Lines of Code for a Developer (one Release)

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*												

5.1 Interpreting the Results

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 person-specific dummy variable coefficient with the median value. 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%. The results of these simulations are shown in Figure 6.

Figure 6: Simulations of Developer Productivity



First, a set of simulations was run to predict the expected productivity that would be achieved if 100% of a developer's effort could be dedicated 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 6. (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 6 shows the result of varying the percent of lines submitted to core files on feature productivity for this prototypical individual. All else equal, the developer working only on features in the periphery would produce 10655 lines of changes during a release. This same individual only produced 6083 lines for features when working in the core. A second set of simulations 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 7. (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 6 shows the response of bug-fix productivity when the percent of lines submitted to the core is varied. All else equal, if a developer works only on bug fixes in the periphery, 2815 lines of changes would be produced. This same individual would produce only 1567 lines if working 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 8. (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 6 shows the impact of varying the percent of lines submitted to core files on overall productivity. All else 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 working in the core.

Our results suggest that the effect that modularity has on developer productivity is strong. All else equal, system design 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 6, one might speculate that a refactoring which shrunk 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 previous work (Sosa et al, 2012) tells us that developers in the core will spend more of their time contending with bugs, thereby magnifying the impact presented on the green line in Figure 6. If shrinking the core reduced the number of bugs a developer had to contend with, it would increase the amount of time spent on the blue curve rather than the red curve, resulting in significant productivity gains.

6. Discussion

Our work makes an important contribution to the academy and to the practice of managers. In particular, we show that components with higher levels of interdependency are associated with lower levels of productivity in a large, commercial software system. The magnitude of the differences between components is surprisingly large. Specifically, we find that modular components are associated with an increase in new feature productivity of 75% as compared to tightly-coupled components.

Our paper makes an important contribution to academic study, in that we highlight the potential for architectural change to create value in mature technological systems. While prior studies have developed many insights into the process of system design, fewer have focused on the potential for these early decisions to become "misaligned" with the mission of a system over the longer-term, especially if priorities change in terms of valuing performance attributes such as cost and efficiency. These insights have particular relevance in contexts where the life of a system is uncertain, and where there are distinct trade-offs in design associated with differing performance characteristics. For example, a new start-up developing an Internet software application would not prioritize maintainability over speed, given there is only a low probability the firms or the system will be long-lasting. Should the start-up succeed however, the design decisions embedded in the system design may become increasingly misaligned with requirements, consuming greater amounts of cost as the system grows and evolves. Such a dynamic suggests the need for a periodic review of critical system design decisions, especially when it is clear that the life of a system may be extended versus early expectations.

Our results also have important implications for managers. Above all, they highlight the importance of design decisions made early in the life of a complex system. Choices about levels of component coupling are typically founded upon the trade-offs faced within the *current* version of a design, for example, in terms of superior performance versus increased reliability. Yet our results reveal the long-lasting nature of these choices. Tightly-coupled components cost significantly more to maintain many years after a system has been introduced. The challenge for a decision-maker is that these longer-term costs are neither easy to calculate nor as salient as the near-term benefits that may stem from a design that is more tightly-coupled. Given these factors, it is likely that managers systematically under-invest in modularity when developing complex systems. Our work highlights a set of methods that can help managers to justify architectural changes in such systems, even at the later stages of a system's life.

Several limitations of our study must be considered in assessing the generalizability of our results. First, our work is conducted in the software industry, a unique context given that designs exist purely as information, and are not bounded by physical limits. Whether the results would be replicated in industries based upon electronic hardware or physical systems remains an important empirical question. Second, we examine a single product in this industry, hence cannot be sure that the findings apply to other software products. While we examine the costs of system design across a sample of 14,000 components, this sample comes from a single product, hence will reflect idiosyncratic practices and design choices associated with the parent organization that developed it. Finally, while we speculate on the potential value that could be released via a redesign, these actions would have costs and other (perhaps unintended) impacts on system performance. Decisions on the benefits of architectural change must therefore carefully assess these other costs and impacts before it is known if this would be optimal.

This work generates a number of promising avenues for future study. First, we need to understand the extent to which design choices vary, for example, across products that perform similar functions. If designs are, to a large degree, dictated by function, the ability to improve on the dynamics observed here might be limited (i.e., some designs may need to be more tightly-coupled than others). Second, work is needed to expose the broader organizational influences on a system's design, which steer firms away from what appear to be optimal choices. Given prior work suggests that products "mirror" the organizations that develop them, the misalignment of system design and long-term requirements may stem more from the nature of the firm, than any functional need (Conway, 1968; Henderson and Clark, 1990; MacCormack et al, 2007b). Finally, the methods described here can be used to assess the degree to which regular patterns are found in system design and evolution. Prior work has shown that systems comprise a central core around which are arranged peripheral components (Tushman and Murmann, 1998). Future research could explore the prevalence of such patterns and identify the factors that explain differences between them. This agenda promises to help us understand the choices available to a designer, as well as the potential to change these choices and thereby release additional value later in a systems life.

REFERENCES

Alexander, Christopher (1964) *Notes on the Synthesis of Form,* Cambridge, MA: Harvard University Press.

Baldwin, Carliss Y. and Kim B. Clark (2000). *Design Rules, Volume 1, The Power of Modularity*, Cambridge MA: MIT Press.

Banker, Rajiv D. and Sandra A. Slaughter (2000) "The Moderating Effect of Structure on Volatility and Complexity in Software Enhancement," *Information Systems Research*, 11(3):219-240.

Barabasi, A. Scale-Free Networks: A Decade and Beyond, *Science,* Vol 325: 412-413

Braha, Dan., A.A. Minai and Y. Bar-Yam (2006) "Complex Engineered Systems: Science meets technology," Springer: New England Complex Systems Institute, Cambridge, MA.

Cataldo, Marcelo, Patrick A. Wagstrom, James D. Herbsleb and Kathleen M. Carley (2006) "Identification of Coordination Requirements: Implications for the design of Collaboration and Awareness Tools," *Proc. ACM Conf. on Computer-Supported Work,* Banff Canada, pp. 353-362

Christensen, Clayton M. (1997) *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*, Boston MA: Harvard Business School Press.

Clark, Kim B. (1985) "The Interaction of Design Hierarchies and Market Concepts in Technological Evolution," *Research Policy* 14 (5): 235-51.

Conway, M.E. (1968) "How do Committee's Invent," *Datamation,* 14 (5): 28-31.

Dellarocas, C.D. (1996) "A Coordination Perspective on Software Architecture: Towards a design Handbook for Integrating Software Components," *Unpublished Doctoral Dissertation*, M.I.T.

Dosi, Giovanni (1982) "Technological paradigms and technological trajectories," *Research Policy,* 11: 147-162

Eick, Stephen G., Todd L. Graves, Alan F. Karr, J.S. Marron and Audric Mockus (1999) "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions of Software Engineering,* 27(1):1-12.

Eppinger, S. D., D.E. Whitney, R.P. Smith, and D.A. Gebala, (1994). "A Model-Based Method for Organizing Tasks in Product Development," *Research in Engineering Design* 6(1):1-13

Fleming, L. and O. Sorenson, "Science and the Diffusion of Knowledge." Research Policy 33, no. 10 (December 2004): 1615-1634

Gokpinar, B., W. Hopp and S.M.R. Iravani (2007) "The Impact of Product Architecture and Organization Structure on Efficiency and Quality of Complex Product Development," Northwestern Univ. Working Paper.

Henderson, R., and K.B. Clark (1990) "Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms," *Administrative Sciences Quarterly*, 35(1): 9-30.

Holland, John H. (1992) *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, 2nd Ed. Cambridge, MA:* MIT Press.

Kauffman, Stuart A. (1993) *The Origins of Order*, New York: Oxford University Press

Klepper, Steven (1996) "Entry, Exit, Growth and Innovation over the Product Life Cycle, *American Economic Review*, 86(30):562-583.

Landes, D. (1983) *Revolution in Time,* Harvard University Press, Cambridge, MA

Langlois, Richard N. and Paul L. Robertson (1992). "Networks and Innovation in a Modular System: Lessons from the Microcomputer and Stereo Component Industries," *Research Policy,* 21: 297-313, reprinted in *Managing in the Modular Age: Architectures, Networks, and Organizations,* (G. Raghu, A. Kumaraswamy, and R.N. Langlois, eds.) Blackwell, Oxford/Malden, MA.

MacCormack, Alan and M. Iansiti, (2009) "Intellectual Property, Architecture and the Management of Technological Transitions: Evidence from Microsoft Corporation," Journal of Product Innovation Management, 26: 248-263

MacCormack, Alan D. (2001). "Product-Development Practices That Work: How Internet Companies Build Software," *Sloan Management Review* 42(2): 75-84.

MacCormack, Alan, John Rusnak and Carliss Baldwin (2006) "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science, 52(7): 1015-1030.*

MacCormack, Alan, John Rusnak and Carliss Baldwin (2007a) "The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry," Harvard Business School Working Paper, 08-038.

MacCormack, Alan, John Rusnak and Carliss Baldwin (2007b) "Exploring the Duality between Product and Organizational Architectures," Harvard Business School Working Paper, 08-039.

Marple, D. (1961), "The decisions of engineering design," *IEEE Transactions of Engineering Management*, 2: 55-71.

Murmann, Johann Peter and Koen Frenken (2006) "Toward a Systematic Framework for Research on Dominant Designs, Technological Innovations, and Industrial Change," *Research Policy* 35:925-952.

Murphy, G. C., D. Notkin, W. G. Griswold, and E. S. Lan. (1998) An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology,* 7(2):158--191

Rivkin, Jan W. (2000) "Imitation of Complex Strategies" *Management Science* 46:824-844.

Rivkin, Jan W. and Nicolaj Siggelkow (2007) "Patterned Interactions in Complex Systems: Implications for Exploration," *Management Science*, 53(7):1068-1085.

Rusovan, Srdjan, Mark Lawford and David Lorge Parnas (2005) "Open Source Software Development: Future or Fad?" *Perspectives on Free and Open Source Software,* ed. Joseph Feller et al., Cambridge, MA: MIT Press.

Sanderson, S. and M. Uzumeri (1995) "Managing Product Families: The Case of the Sony Walkman," *Research Policy*, 24(5):761-782.

Schilling, Melissa A. (2000). "Toward a General Systems Theory and its Application to Interfirm Product Modularity," *Academy of Management Review* 25(2):312-334, reprinted in *Managing in the Modular Age: Architectures, Networks, and Organizations* (G. Raghu, A. Kumaraswamy, and R.N. Langlois, eds.), Blackwell, Oxford/Malden, MA.

Sharman, D. and A. Yassine (2004) "Characterizing Complex Product Architectures," *Systems Engineering Journal*, 7(1).

Shaw, Mary and David Garlan (1996). *Software Architecture: An Emerging Discipline*, Upper Saddle River, NJ: Prentice-Hall.

Simon, Herbert A. (1962) "The Architecture of Complexity," *Proceedings of the American Philosophical Society* 106: 467-482, repinted in *idem.* (1981) *The Sciences of the Artificial*, *2nd ed.* MIT Press, Cambridge, MA, 193-229.

Sosa, M., J. Mihm, and T. Browning. (2009). "Can we predict the generation of bugs? Software architecture and quality in open-Source development." INSEAD working paper 2009/45/TOM

Sosa, Manuel, Steven Eppinger and Craig Rowles (2007) "A Network Approach to Define Modularity of Components in Complex Products," *Transactions of the ASME* Vol 129: 1118-1129

Spear, S. and K.H. Bowen (1999) "Decoding the DNA of the Toyota Production System," *Harvard Business Review*, September-October.

Steward, Donald V. (1981) "The Design Structure System: A Method for Managing the Design of Complex Systems," *IEEE Transactions on Engineering Management* EM-28(3): 71-74 (August).

Suarez, F and J.M. Utterback, (1995) Dominant Designs and the Survival of Firms, *Strategic Management Journal, Vol. 16: 415-430*

Tushman, Michael L. and Lori Rosenkopf (1992) "Organizational Determinants of Technological Change: Toward a Sociology of Technological Evolution," *Research in Oragnizational Behavior* Vol 14: 311-347

Tushman, Michael L. and Murmann, J. Peter (1998) "Dominant designs, technological cycles and organizational outcomes" in Staw, B. and Cummings, L.L. (eds.) *Research in Organizational Behavior*, JAI Press, Vol. 20.

Ulrich, Karl (1995) "The Role of Product Architecture in the Manufacturing Firm," *Research Policy,* 24:419-440, reprinted in *Managing in the Modular Age: Architectures, Networks, and Organizations,* (G. Raghu, A. Kumaraswamy, and R.N. Langlois, eds.) Blackwell, Oxford/Malden, MA.

Utterback, J. M. (1996) *Mastering the Dynamics of Innovation,* Harvard Business School Press, Boston, MA.

Utterback, J. M. and F. Suarez (1991) Innovation, Competition and Industry Structure, *Research Policy, 22: 1-21*

von Krogh, G. M. Stuermer, M. Geipel, S. Spaeth, S. Haefliger (2009) "How Component Dependencies Predict Change in Complex Technologies," ETH Zurich Working Paper

Warfield, J. N. (1973) "Binary Matricies in System Modeling," *IEEE Transactions on Systems, Management, and Cybernetics*, Vol. 3.

Whitney, Daniel E. (Chair) and the ESD Architecture Committee (2004) "The Influence of Architecture in engineering Systems," Engineering Systems Monograph, http://esd.mit.edu/symposium/pdfs/monograph/architecture-b.pdf, accessed December 10th 2007