08-038

The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry

Alan MacCormack
John Rusnak
Carliss Baldwin

Abstract

Much academic work asserts a relationship between the design of a complex system and the manner in which this system evolves over time. In particular, designs which are modular in nature are argued to be more "evolvable," in that these designs facilitate making future adaptations, the nature of which do not have to be specified in advance. In essence, modularity creates "option value" with respect to new and improved designs, which is particularly important when a system must meet uncertain future demands.

Despite the conceptual appeal of this research, empirical work exploring the relationship between modularity and evolution has had limited success. Three major challenges persist: first, it is difficult to measure modularity in a robust and repeatable fashion; second, modularity is a property of individual components, not systems as a whole, hence we must examine these dynamics at the microstructure level; and third, evolution is a *temporal* phenomenon, in that the conditions at time t affect the nature of the design at time t+1, hence exploring this phenomenon requires longitudinal data.

In this paper, we tackle these challenges by analyzing the evolution of a successful commercial software product over its entire lifetime, comprising six major "releases." In particular, we develop measures of modularity at the component level, and use these to predict patterns of evolution between successive versions of the design. We find that modularity has a strong and unambiguous impact on design evolution. Specifically, we show that i) tightly-coupled components are "harder to kill," in that they have a greater likelihood of survival in subsequent versions of a design; ii) tightly-coupled components are "harder to maintain," in that they experience more surprise changes to their dependency relationships that are *not* associated with new functionality; and iii) tightly-coupled components are "harder to augment," in that the mix of new components added in each version is significantly more modular than the legacy design.

- HARD TO MEASURE
- HARD TO TRACK

THIS STUFF MATTERS
(BUT OFTEN NOT VALUED)

1. Introduction

What factors should influence the design of a complex system? A variety of academic work has tackled this question, highlighting the critical importance of system design decisions to the creation and capture of value at multiple organizational levels. Design decisions have been shown to influence the structure of industries and the value created within (Baldwin and Clark, 2000); the likelihood of firm failure in the face of radical technological change (Henderson and Clark, 1990); the optimal level of product variety (Sanderson and Uzumeri, 1995); and the performance of products themselves (Ulrich, 1995). A consistent theme in these studies is the notion that the design of a complex system involves a series of choices. The output of these choices, in turn, dictates a system's performance along multiple (often competing) dimensions.

An important stream of research within this broader literature focuses on the link between system design decisions and system *evolution* (Simon, 1962). Studies in this area argue that some designs are more "adaptable" than others, in that they facilitate the process of modifying or updating the system's components to reflect changing conditions or circumstances. This feature is valuable to the degree that a system must meet a variety of future requirements, the specifics of which cannot be predicted ex-ante (MacCormack et al, 2001). In such situations, one must "design for uncertainty."

How do systems that are more evolvable differ from those which are not? Prior work argues that this characteristic stems from designs that are "modular" in nature. Modular designs are "loosely-coupled" in that the system's functions are decomposed into relatively independent parts separated by well defined interfaces. Loose-coupling allows each part to be modified, substituted or deleted with minimal impact on the rest of the system. In essence, modularity creates "options" to adapt a design to meet unforeseen future requirements. Given the widespread theoretical support for a link between modularity and evolution, one might imagine a wealth of empirical evidence confirms this association. Yet a robust test of this relationship has proven elusive, due to the challenges in measuring modularity and assessing its impact on a design over time.

This paper reports empirical data from a study that examines the relationship between modularity and design evolution. In contrast to prior work which explores this topic at the system level, we focus instead on the microstructure of a design; the individual components from which a system is built and the dependencies that exist between them. This is an important distinction given that complex systems comprise a mix of components, some of which are tightly-coupled to others and some of which are relatively independent. Our work explores whether these different dependency structures explain differences in evolution. In particular, we examine three different aspects of evolution; component *survival*, an indicator of the degree to which components can be removed or substituted over time; component *maintainability*, a measure of the stability of legacy components in a design; and component *augmentation*, a measure of the ease with which new components can be added to a design.

Our research is situated in the software industry, an ideal context in which to study issues of design structure given the information-based nature of the product. Software can be processed automatically to identify the constituent components of a design and the dependencies that exist between them, a technique that is not possible with a physical product. Furthermore, we can track the evolution of a design over time, comparing each new version to its predecessor to reveal how components evolve. We use these properties to help analyze the evolution of a successful commercial software product from first release to its current design. The dataset encompasses six major versions released at varying intervals over a 15 year period.

TIGHT COUPLING IS WORSE
BUT PERSISTS LONGER

Our results make an important contribution to literature that explores the design of complex systems, in that we find strong support for the existence of a relationship between component modularity and design evolution. Specifically, we show that tightly-coupled components have a higher probability of survival as a design evolves; in essence, they are "harder-to-kill." We also find that tightly-coupled components are harder to maintain, in that they are more likely to experience surprise design changes unrelated to newly added or removed functionality. Finally, we show that tightly-coupled components are harder to augment, in that the mix of new components added in each version is significantly more modular than the legacy design. These results have important implications for managers, highlighting the impact of design decisions made today on both the evolution and maintainability of a design in subsequent years.

The paper proceeds as follows. We first review the prior literature that explores the relationship between modularity and design evolution. We then outline our research methods, which employs a technique called the Design Structure Matrix to analyze the evolution of a commercial software product. Using this technique, we develop several hypotheses for the relationships that exist between component modularity and design evolution. Next, we describe our empirical data and the methods we use to prepare this data for analysis. Finally, we provide the results of our hypothesis tests and discuss their implications for both academic researchers and designers of complex systems.

2. Literature Review

A growing number of studies contribute to our understanding of the design and management of complex systems (Holland, 1992; Kaufman, 1993; Rivkin, 2000; Rivkin and Siggelkow, 2007). Many 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 its components is called its "architecture" (Ulrich, 1995; Whitney et al, 2004). Understanding how architectures are chosen, how they perform and how they can be adapted are critical topics in the design of complex systems.

CONWAY'S LAW LEADS TO MODULARITY

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

Modularity yields three types of benefit in a design process (Baldwin and Clark, 2000).[1] First, it increases the range of "manageable" complexity by decomposing the functions of a complex system into parts that can be developed independently. Second, modularity allows designers to work in parallel, assuming that they adhere to the "design rules" that define the role of components in the system. Finally, modularity

[1] Here we focus on the benefits of modularity to a designer. Modularity brings important yet different benefits to the manufacturers and users of a product (see Ulrich, 1995; Baldwin and Clark, 2000).

accommodates uncertainty in that changes to one part of the design have little impact on others. This latter benefit is valuable both during a design process and *after* it is complete, given modules can be improved, substituted or removed as technical possibilities and market needs evolve. In essence, modularity facilitates design *evolution*.

The link between modularity and evolution was first made explicitly by Simon (1962) who argued that "nearly-decomposable" systems facilitate experimentation and problem-solving. A variety of researchers built on this foundation, articulating the dynamics of this relationship across both organizational and technical systems (Weick, 1976; Langlois and Robertson, 1992; Ulrich, 1995; Garud and Kumaraswamy, 1995; Sanchez and Mahoney, 1996; Schilling, 2000). Recent work formalizes this reasoning by showing that modularity creates design "options" (Baldwin and Clark, 2000). Within a system, modules are free to evolve in a decentralized manner; hence greater modularity is associated with an increase in the number of possible paths for future adaptations.

While many studies make significant theoretical contributions to our understanding of the link between modularity and evolution, fewer studies explore this link empirically. The most important works are based upon descriptive case studies that illustrate how this relationship is manifested, but do not constitute a "test" of its existence. For example, Langlois and Robertson (1992) highlight the role of modularity in shaping the evolution of the stereo component and microcomputer industries; and Sanderson and Uzumeri (1995) show how the success of Sony's Walkman was enabled by the adoption of modular subsystems that could be reused across products and updated independently. Studies like these suffer from two problems: First, they do not measure modularity or design evolution in a systematic or repeatable fashion; and second, they view modularity as a characteristic of a whole system, rather than of its constituent components. Hence we do not know if components with differing levels of modularity evolve differently.

The most promising technique for measuring modularity has come from the field of engineering, in the form of the Design Structure Matrix (DSM). A DSM highlights a design's structure by examining the dependencies that exist between its constituent elements in a square matrix (Steward, 1981; Eppinger et al, 1994). These elements can represent tasks to be performed, parameters to be defined or actual components in a design. A key contribution of the DSM literature has been in showing that modularity depends not only on the *number* of dependencies between elements, but also on their *pattern* of distribution (Sosa et al, 2003; Sharman and Yasine, 2004; Rivkin and Siggelkow, 2007). Complex systems comprise a mix of elements with different levels of dependency; DSMs can be used to reveal design differences at the microstructure level. In recent work, metrics which capture the degree of coupling in a system have been developed and used to compare different designs (MacCormack et al, 2006).

2.1 Studies of Software Design and Evolution

The most significant empirical studies exploring issues of design structure and evolution have come in the field of software. The 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, the amount of "legacy code" exceeds new code, hence significant efforts must be devoted to maintenance. Indeed, mature products often contain significant amounts of code from their earliest versions, even if major evolutions in design have since been made (e.g., MacCormack and Herman, 2000). This dynamic creates unique challenges, in that today's developers must bear the consequences of design decisions made years earlier. Understanding how designs evolve, how they can be made more "evolvable," and the role that modularity plays in this process are critical areas 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 other studies which sought to measure product *complexity* for the purposes of predicting development productivity and quality (e.g., McCabe 1976; Halstead, 1976). While measures of complexity focus on the number of elements in a design, measures of modularity focus on the pattern of dependencies *between* elements.

COMPLEXITY → N(ELEMENTS)
MODULARITY → N(DEPENDENCIES)

SLOWER CODING WHEN TEAM CHAT AND CODE STRUCTURE MISALIGNED

Efforts to measure software modularity empirically typically center on capturing the level of coupling between different parts of a design. Two broad methods are employed. The first analyzes specific types of dependency between components, using these to assess a design's structure. For example, recent critiques of the Linux operating system have examined both the use of global variables (variables used by many parts of a design; Schach et al, 2002) and the use of function calls (calls between different parts of a design; Rusovan et al, 2005). The second infers the presence of dependency by assessing which components must be changed in order to fulfill a modification request (MR). For example, Eick et al (1999) show that code decays over time as measured by the number of files changed to complete a MR; while Cataldo et al (2006) show that the time required to complete a MR depends on the degree of alignment between team communication and the component dependencies implied by patterns of MR changes.

The formal study of software evolution has its roots in empirical work by Lehman and Balady (1976; 1985) resulting in what are called the "laws of program evolution." These laws build from rich observations of real world systems combined with theoretical insights from computer science to predict general patterns of system growth. Subsequent studies to verify these laws have produced mixed results (see Barry and Kemerer, 2007 for a comprehensive review). Much of the problem may stem from the deterministic nature of these laws, which aim to describe "central tendencies" in system evolution. In practice, many critical contingencies exist (such as the level of modularity) that might lead one to observe (or induce) different evolutionary dynamics.

Studies seeking to link measures of modularity with system evolution have tended to focus on predicting the cost and frequency of changes across different systems. Banker et al (1993) examine 65 maintenance projects across 17 systems and find that project costs increase with system complexity, as measured by average "procedure" size and number of "non-local" branching statements (a type of dependency). 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 business 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. This relationship is mediated by the use of greater structure, as captured by the number of "calls" 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 subsequent modifications. This dynamic is attributed to the localization of changes within modular components, which reduces the level of change for the system as a whole.

Our review of this literature suggests several criteria must be met to explore the relationship between modularity and design evolution in a robust fashion. First, the study should use longitudinal data, given the need to relate structural attributes of a design at time t to subsequent patterns of change and the resultant design at t+1. Second, the study should explore these dynamics at the microstructure level, given that modularity is a property of individual components rather than the design as a whole.[2] And third, the study must account for different aspects of evolution, given that prior studies show modularity may influence each in different ways. Consider that if modular components are more easily adapted than others we might expect them to be substituted, removed or augmented more frequently over time. However, prior work also shows that modular components are easier to maintain, hence may be *less* likely to experience design changes associated with "corrective" actions (i.e., "churn"). A study must differentiate between these different types of change in order to explain observed patterns of evolution.

These criteria can be met by applying the technique of Design Structure Matrices (DSMs) to analyze the relationship between modularity and design evolution. DSMs provide a robust and repeatable way to analyze and measure the characteristics of a design at the component level, and to track evolution over time. Recent work explores the use of DSMs to both model alternative software design choices ex-ante and examine the impact of intentional re-design efforts ex-post (Sullivan et al, 2001; Lopes and Bajracharya, 2005; MacCormack et al, 2006; La Mantia et al, 2008). We apply this technique to analyze the evolution of a successful commercial software product.

[2] To illustrate, consider two designs with eight components: in one, each component is tightly-coupled to just one other, forming four two-component modules; in the other, four of the components form a single tightly-coupled module, whereas the others have no dependencies with any component. Both these designs

3. Research Methods and Hypotheses

Below, we describe how we apply DSM methods to software and develop several measures of component modularity that can be derived from a DSM. We then formalize our approach to analyzing design evolution and develop hypotheses for the relationships that exist between component modularity and three different aspects of evolution: component survival, component maintainability and component augmentation.

3.1 Applying DSMs to Software[3]

There are two choices to make when applying DSMs to a software product: The level of analysis and the type of dependency. 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 pertain to a specific subsystem; the *source file* level, which corresponds to a collection of related processes and functions; and the *function* level, which corresponds to a set of instructions that perform a specific task. We analyze designs at the source file level for a number of reasons. First, source files tend to contain functions with a similar focus. Second, tasks and responsibilities are allocated to programmers at the source file level, allowing them to maintain control over all the functions that perform related tasks. Third, software development tools use the source file as the unit of analysis for version control. And finally, prior work on 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).

B-IT DATED

There are many types of dependency between source files in a software product.[4] We focus on one important dependency type – the "Function Call" – used in prior work on design structure (Banker and Slaughter, 2000; Rusovan et al, 2005). A Function Call is an instruction that requests a specific task to be executed. The function called may or may not be located within the source file originating the request. When it is not, this creates a dependency between two 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

have the same number of dependency relationships, but in one, each component has the same level of coupling whereas in the other, there are two completely different levels of coupling.

[3] The methods we describe here build on prior work in this field (see MacCormack et al, 2006 for details).

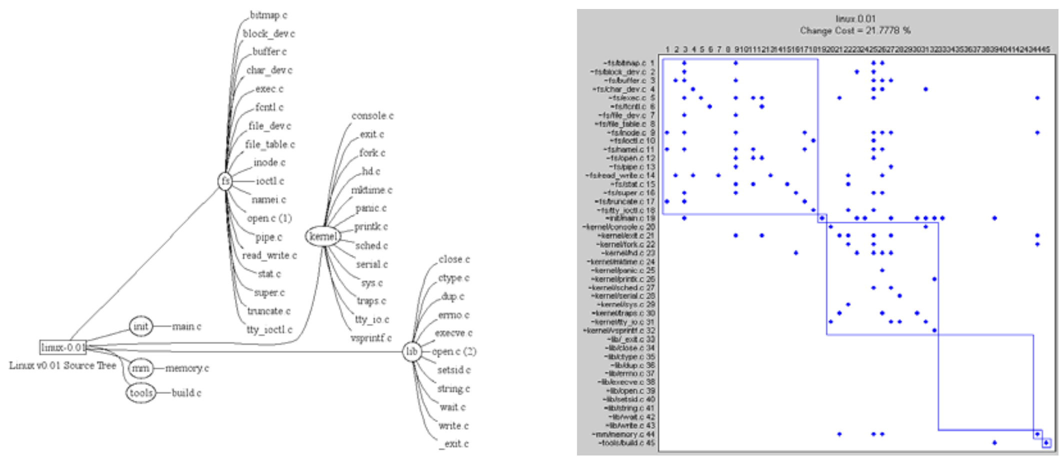
ANT I CAN USE?

the DSM. Note this does not imply that SourceFile2 depends upon SourceFile1; the dependency is not symmetric unless SourceFile2 also calls a function in SourceFile1.

To capture function calls, we input a product's source code into a tool called a "Call Graph Extractor" (Murphy et al, 1998). This tool is used to obtain a better understanding of system structure and interactions between parts of the code.[5] Rather than develop our own extractor, we tested several commercial products that could process source code written in both procedural and object oriented languages (e.g., C and C++), capture indirect calls (dependencies that flow through intermediate files), run in an automated fashion and output data in a format that could be input to a DSM. A product called Understand C++[6] was selected given it best met all these criteria.

The DSM of a software product can be displayed using the *Architectural View*. This groups each source file into a series of nested clusters defined by the directory structure, with boxes drawn around each successive layer in the hierarchy. The result is a map of dependencies, organized by the programmer's perception of the design. To illustrate, the Directory Structure and Architectural View for Linux v0.01 are shown in **Figure 1**. Each "dot" represents a dependency between two particular components (i.e., source files).

Figure 1: The Directory Structure and Architectural View of Linux version 0.01.



---

[4] Several authors have developed comprehensive categorizations of dependency types (e.g., Shaw and Garlan, 1996; Dellarocas, 1996). Our work focuses on one important type of dependency.
[5] Function calls 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.
[6] Understand C++ is distributed by Scientific Toolworks, Inc. see <www.scitools.com> for details.

OTHER LANGS??
↳ HAS MANY
NEED TO TRY

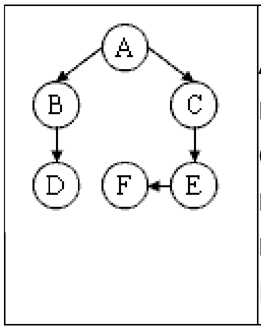
PAID WITH WEIRD TRIAL

3.2 Measuring Component Modularity

In order to assess the impact of component modularity on design evolution, we develop two measures of the degree to which components are coupled to each other. First, we assess the number of *direct* dependencies that a component possesses, a measure we call "Direct Connectivity." Second, we assess the number of both *direct and indirect* dependencies that a component possesses, a measure known as "Visibility" (Sharmine and Yassine 2004; Warfield 1973). In both cases, we use separate measures for dependencies which flow into a component (called "Fan-In") from those which flow out of it (called "Fan-Out") reflecting the asymmetric nature of dependency relationships.

To illustrate, consider the example system depicted in **Figure 2** in both graphical and DSM form. We see that element A depends upon (or "calls functions within") elements B and C, so a change to element C may have a *direct* impact on element A. In turn, element C depends upon element E, so 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 path lengths of two and three, respectively. There are no indirect dependencies between elements for path lengths of four or more.

Figure 2: Example System in Graphical and DSM Form



	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

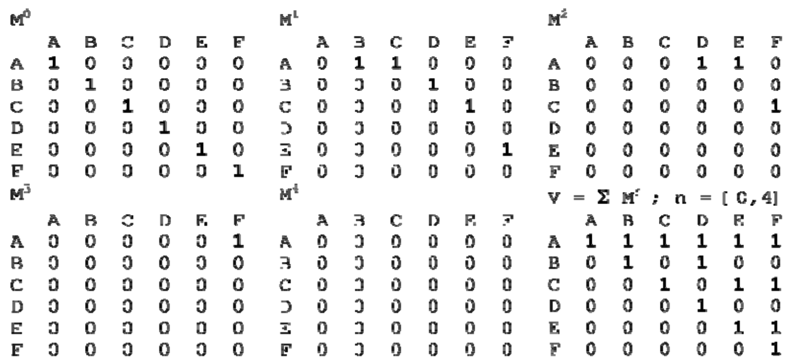
CHANGE TO F CAN MEAN CHANGE TO A INDIRECTLY VIA E, C

The measures of Direct Connectivity (DC) are derived directly from the DSM. For example, element A has a DC Fan-Out of two, given it depends upon elements B and C; and it has a DC Fan-In of zero given that no elements depend upon it. To identify the Visibility of each element, we use the technique of matrix multiplication. Specifically, by raising the DSM to successive powers of n, we obtain the direct and indirect dependencies that exist for each successive path length n. Summing these matrices yields the visibility matrix V, 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.[7] **Figure 3** illustrates the derivation of this matrix.

THIS IS NEAT AUTOMATABLE TOO

Figure 3: The Derivation of the Visibility Matrix

VISIBILITY IS HOW MANY ELEMENTS CAN CAUSE CHANGES



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 Visibility are derived directly from the visibility matrix. Visibility Fan-Out (VFO) is obtained by summing along the rows. For example, element A has a VFO of six, meaning it depends upon all other elements, directly or indirectly. Visibility Fan-In (VFI) is obtained by summing down the columns. For example, element A has a VFI of one meaning it is visible only to itself.[8] For comparative purposes, VFO and VFI can be expressed as a percentage of the number of elements in the system.

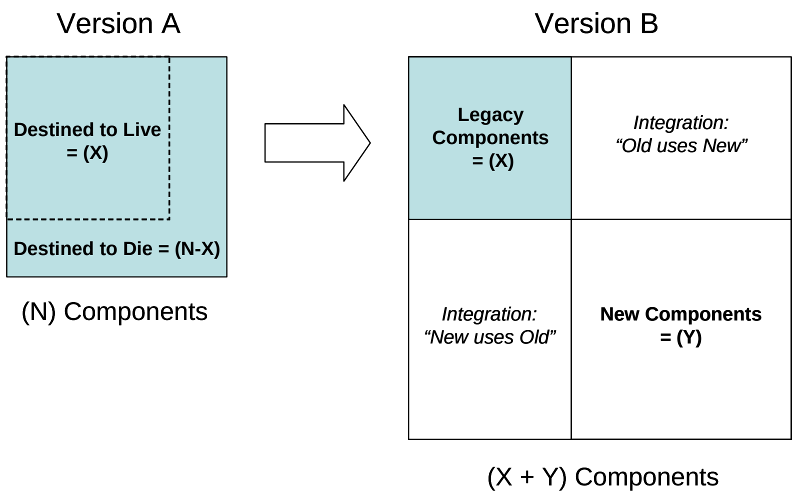
Note that our two measures represent the opposite ends of a continuum along which levels of coupling can be measured. The first captures only direct links between components. The second captures all direct and indirect links between components, giving them equal weight regardless of path length. In this respect, our research design sheds light on a variety of potential measures that lie in-between these two extremes.

[7] Note that we choose to include the matrix for n=0 (i.e., a path length of zero) when deriving the visibility matrix, implying that an element will always depend upon itself.
[8] The mean visibility of all components provides an indication of the coupling for the system as a whole. This measure is referred to as *Propagation Cost* in prior work (MacCormack et al 2006).

3.3 Analyzing Design Evolution using DSMs

Our analysis approach involves using DSMs to track the evolution of a design at the component level, assessing how these dynamics are affected by the levels of coupling between components. In order to formalize this approach, we consider the DSMs of two successive versions of a design, as depicted in **Figure 4**. Version A contains N components, each of which may have dependencies with other components in the design. Version B, its successor, inherits X "legacy" components while N-X components "die." In addition, Y new components are added. These new components may have dependencies with each other as well as with the legacy components. The dependencies among the legacy components may also have changed in version B.

Figure 4: Analyzing Design Evolution Using DSMs



Our first hypothesis concerns the nature of the components that "die" in the transition from one version to the next. This dynamic occurs for two reasons: The functionality a component provides is superseded by a new component and it is *substituted;* or the functionality a component provides is no longer needed and it is *excluded* (Baldwin and Clark, 2000). These actions represent major adaptations, in that the older component no longer exists in the new version. Such adaptations are likely to be more difficult to the degree that a component is tightly-coupled to other components in the design. In essence, tightly-coupled components will be "harder to kill." Hence our first hypothesis:

H1: The likelihood of survival between two versions is positively associated with a component's level of Direct Connectivity and Visibility in the earlier version.[9]

Our second hypothesis concerns changes to the dependency relationships between legacy components, a dynamic we refer to as "churn." Consider that in each new version of a design there are many changes to the dependency relationships between components. These can be divided into those which are "expected" (i.e., they are associated with newly added or removed components) and those which are "surprises" (i.e., they are associated with legacy components).[10] Prior work has shown that tightly-coupled components are harder to maintain, in that they generate more frequent and costly design modifications associated with correction or repair. These components are therefore more likely to be a source of "surprise" dependency changes. Hence our second hypothesis:

H2: The likelihood of "churn" between two versions is positively associated with a component's level of Direct Connectivity and Visibility in the earlier version.[11]

Our third hypothesis concerns the nature of the components that are added to a design as it evolves. Prior work argues that modular components are easier to develop and add to a design, given they can be built and tested independently and integrated with existing components more easily. Of course, in any new version it is likely that a mix of components with different levels of coupling will be added to the design. However, the proportions of each are likely to differ, as compared to the legacy components inherited from the prior version. Hence our third hypothesis:

H3: New components added to a design are likely to have lower levels of Direct Connectivity and Visibility than legacy components inherited from the prior version.

[9] The null hypothesis is that the likelihood of survival is independent of the level of coupling. If all components in a design are equally likely to be substituted or excluded, this would be true.
[10] Note that with this definition, surprises can only occur among the X legacy components.

4. Description of the Data

The dataset comprises six major releases of a successful commercial software product. The six releases date from the early 1990s to 2006, each representing a new version of the product that was marketed and sold to consumers. Our objective was to capture new "platform" versions, not intermediate releases or updates. While the design continued to evolve in-between each version, the intermediate states were not fully tested or sold as a whole, hence do not represent stable or complete designs. We obtained the source code for each release from the vendor, and processed the code to extract the function call dependencies between each source file in each version. We used this data to calculate measures of modularity for each source file in each version. **Table 1** gives descriptive data for each version (the DSMs are shown in **Appendix A**).

Table 1: Descriptive Data for Each Version[12]

Version (V)	A	B	C	D	E	F
Source Files	156	284	1407	1857	2434	2756
Dependencies	1409	2806	7025	8727	10424	11128
Density of DSM	5.57%	3.49%	0.36%	0.25%	0.18%	0.15%
Survivors (in V+1)	116	170	1261	1296	2330	n/a
Survival Rate	74.4%	59.9%	89.6%	69.8%	95.7%	n/a

→ wow

Two observations are apparent from this data. First, the early versions of the product are much smaller than later versions and have a higher density of dependencies between files. Second, the extent to which the design is changed varies significantly from version to version. For example, only 60% of the files in Version B survive to Version C. By contrast, almost 90% of the files in Version C survive to Version D. These differences are driven by variations in the release cycle (the time between versions); the maturing of the product; and the particular goals for each release. Our empirical approach involves testing our hypotheses within each version, as well as for an aggregate model which pools all observations, controlling for inter-version differences by the use of dummies.

Table 2 and Table 3 display descriptive data for measures of component modularity. The data reveals that the distributions of these measures are non-normal. In particular,

[11] The null hypothesis is that the likelihood of a "surprise" dependency change is independent of the level of coupling. If all legacy components are equally likely to experience "churn," this would be true.
[12] There is no "successor" to version F, which is the current design sold in the market.

standard deviations often exceed the mean and median values are often significantly lower than the mean. The skewed distributions are illustrated in **Appendix B**, which shows the histograms of values from version C. We observe that Direct Connectivity has an exponential distribution, while Visibility has a Bi-Polar distribution. The latter pattern can be traced to a feature of this design that is visible in the DSM. Specifically, it appears to contain a dense "core" of files that are highly visible to one another. If a file has a dependency with one file in the core, it is, in turn, visible to all others.

Table 2: Descriptive Data for Measures of Direct Connectivity by Version[13]

	Mean	StdDev DCFO	StdDev DCFI	Median DCFO	Median DCFI
Version A	8.9	7.2	13.7	7	4
Version B	9.9	9.0	18.6	7	4
Version C	5.9	8.8	17.5	1	1
Version D	4.7	8.7	19.6	1	1
Version E	4.3	8.0	23.4	1	1
Version F	4.0	7.8	23.9	1	1

Table 3: Descriptive Data for Measures of Visibility by Version[14]

	Mean	StdDev VFO	StdDev VFI	Median VFO	Median VFI
Version A	107.7	42.0	54.3	124	135
Version B	176.4	69.0	110.6	203	246
Version C	172.4	197.8	271.3	1	1
Version D	172.2	205.4	315.1	1	1
Version E	172.6	203.7	377.9	1	1
Version F	164.5	202.0	387.9	1	1

To account for these non-normal distributions we apply a transformation to each measure. For Direct Connectivity, we use a log transformation, reflecting the fact that the impact of each added connection is likely to decline as values increase. For Visibility, we transform the measures into a *binary* form by observing whether the values exceed a threshold, defined as the midpoint between zero and the maximum level of visibility (which varies by version). We then allocate each file to one of four different "types" based upon their levels of visibility (see **Table 4**) allowing us to examine whether these different types play different roles in explaining system evolution.

[13] The mean of CFO and CFI are identical, given each "call" out is matched by a corresponding "call" in.
[14] The mean of VFO and VFI are identical, given each "call" out is matched by a corresponding "call" in.

Table 4: Different File “Types” based upon levels of Visibility

High VFI and VFO	Files with high visibility on both dimensions are “Core” files. They are both “seen by” many files and “see” many files. They are often linked directly or indirectly to all other core files.
High VFI only	Files with high VFI are seen by many other files, but do not see other files. They often represent system “Buses,” which provide shared functionality to many different parts of the system.
High VFO only	Files with high VFO see many other files, but are not seen by other files. These often represent system “Brokers,” which direct the flow of program control into the system “Core.”
Low VFO and VFI	Files with low visibility on both dimensions are “Periphery” files. They are neither seen by many files nor see many files. They are often “singletons” which execute independently of other files.

Note that our measures of direct connectivity and visibility are likely to be correlated, given that the visibility matrix includes direct dependencies. The relationship between these two variables however, is subtle. While a zero value for direct connectivity, by definition, leads to low visibility, the opposite is not true. High values of visibility stem not from the number of connections a file has, but the particular files to which it is connected. This can be seen by reflecting upon the “core–periphery” nature of the design. A single connection to the core will yield high visibility; by contrast, many connections to non-core files may yield low visibility.

NOBODY WOULD
USE THIS IRL
WOULD THEY?

To deal with potential correlation we use an empirical approach driven by the data. Given that we make no prediction as to whether Direct Connectivity or Visibility is dominant in explaining evolution, we test the power of each and define the strongest as our primary predictor. We then normalize the second measure to remove that part of its variance explained by the first, making our secondary predictor orthogonal to the first. In the results tables that follow, the secondary predictor is denoted by the use of italics.

5. Empirical Results

5.1 Hypothesis One: Component Survival

We test our first hypothesis by looking at the impact of measures of component modularity on the likelihood of survival in the next version. In particular, we examine whether components that are more tightly coupled to other components are harder to kill (i.e., more likely to survive). **Table 5** contains the results of a logistic regression model predicting component survival for each of the five versions, as well as for all versions considered in aggregate. The aggregate model includes dummies for each version to control for differences in version survival rate related to unobservable factors.[15]

Table 5: Logistic Regression Models Predicting Source File Survival[16]

VERSION	A	B	C	D	E	ALL
Survival Rate	74.4%	59.9%	89.6%	69.8%	95.7%	
Constant	-0.9	-3.1***	1.6****	0.5****	2.8****	2.81****
Version A	n/a	n/a	n/a	n/a	n/a	-3.14****
Version B	n/a	n/a	n/a	n/a	n/a	-3.76****
Version C	n/a	n/a	n/a	n/a	n/a	-1.05****
Version D	n/a	n/a	n/a	n/a	n/a	-2.39****
High VFI Only	3.2**	4.1****	1.5**	1.4****	31.9****	1.74****
High VFO Only	1.1	1.7	1.8****	0.6****	1.0****	0.71****
High VFI and VFO	2.4***	4.4****	3.0**	1.1****	1.2***	1.72****
Connectivity FI	-0.1	0.0	0.3	0.3**	0.5*	0.24***
Connectivity FO	0.8***	0.2	0.7**	0.4****	0.0	0.39****
McFadden R-square	13.5%	27.3%	13.0%	4.5%	See text	18.7%
Estrella R-square	15.2%	34.9%	8.9%	5.5%	See text	16.5%
Sample Size (N)	156	284	1407	1857	2434	6138

**** p<0.1%, *** p<1%, ** p<5%, * p<10%

The results show support for our first hypothesis. While the strength of the effects varies across versions, all four measures of modularity are significant at p=0.001 in the

[15] Note that the aggregate model may include the same source file more than once. Each version represents a *separate* observation of whether a component survives, as predicted by its coupling to other components. This is appropriate given the measures of coupling for each component may change in each version. For example, a component may have high VFO in version A, zero VFO in version B, and die in version C. This would yield two data points: a file with high VFO that survived; and a file with low VFO that died.

FILES THAT OTHERS RELY ON ARE HARDER TO CHANGE

aggregate model. Components which are more tightly-coupled to other components have a higher probability of survival than those which are loosely-coupled. In the aggregate model, measures of modularity predict 16% to 19% of the variation in outcomes.

The pattern of results across versions reveals additional insights. The measures for High VFI Only and High VFI and VFO are always significant. By contrast, the measure for High VFO Only is only significant in later versions. This suggests that fan-in visibility is more dominant in explaining survival. Intuitively, adapting a file when others depend upon it is difficult, whereas adapting a file when it depends upon others may be easier. This interpretation is supported by the coefficient on the measure for High VFI and VFO, which is similar in magnitude to that for High VFI Only. That is, a file that is highly visible on both dimensions has a similar probability of survival to a file with only a high fan-in visibility. There is no added impact for high fan-out visibility.

The pattern of results for the normalized measures of direct connectivity is less clear. The fan-in measure is significant (with p=0.05) in one version as well as the final model. The fan-out measure is significant in three versions as well as the final model. While the results suggest that direct connectivity *may* have an effect on survival that is distinct from its association with visibility, this conclusion needs further testing.

To further illustrate these results, **Appendix C** provides data on component survival rate, split by file type as identified by levels of visibility. We note that files with low visibility have a uniformly lower survival rate than others. The effect is most notable in versions with high turnover. For example, only one of the 24 files with low visibility survives the transition from version B to version C. By contrast, the survival rate for files with high visibility is noticeably higher than the mean. For example, only four of the 357 files with high visibility die in the transition from version C to version D.

To understand the size of effects in our aggregate model, we substitute values and observe the differences in predicted survival rate. **Table 6** displays the actual survival rates for each version as well as predicted survival rates for i) a file with low visibility on both dimensions and ii) a file with high visibility on both dimensions (i.e., a "core" file).

[16] We report two measures of R-square to assess goodness of fit (McFadden, 1984; Estrella, 1998).

Table 6: Actual and Predicted Survival Rates for Files with Different Visibility

VERSION	Actual Survival Rate: All Files	Predicted Survival Rate: Low Visibility	Predicted Survival Rate: High Visibility
A	74%	42.1%	80.0%
B	60%	28.1%	68.6%
C	90%	85.4%	97.0%
D	70%	60.6%	89.6%
E	96%	94.4%	98.9%

Two observations can be made from this table. First, the differences in survival rate are large in magnitude, especially when there is a high rate of turnover in a specific version (i.e., the survival rate is low). For example, in versions A and B, files with high visibility have around twice the likelihood of survival as those with low visibility; in version E, by contrast, the likelihood of survival is similar given that most files survive (note that there are large differences in the likelihood of "dying"). Second, the likelihood of survival for high visibility files is uniformly high. In only one version does the figure drop below 80%. Tightly-coupled files are truly hard to kill.

We note again that our aim was *not* to explain the large variation in survival rate between versions, except to the degree that these are associated with the mix of files with differing levels of modularity. Version survival rates are affected by a variety of other factors, including differing release intervals, the maturing of the product and specific market and technological challenges at the time. Our sample spans a 15-year period over which fundamental changes were occurring in software; from the rise of object-oriented programming to the advent of the Internet. How and when the firm chose to tackle such challenges would have had a major impact on the survival rate for specific versions.

5.2 Hypothesis Two: Component Maintainability

We test our second hypothesis by looking at the impact of measures of modularity on component maintainability. In particular, we examine whether components that are more tightly coupled to other components are more likely to experience "churn" ? surprise dependency changes that are unrelated to newly added or removed functionality. The sample for testing this hypothesis differs to that for hypothesis one. Specifically, we consider only the legacy components that survive from one version to another, given these represent the set of components which may experience maintenance.

COUPLING ⇒ CHURN?

CHURN IS A CHANGE
THAT DOESNT BRING FUNCTIONALITY

DEP REMOVE → GOOD
DEP ADD → BAD

Changes in the dependency relationships of legacy components come in two forms: Those which remove dependencies; and those which add dependencies. While both these actions represent types of churn, they have very different effects. Removing dependencies tends to increase system modularity whereas adding dependencies has the reverse effect. All else being equal, increasing the modularity of the legacy functions in a system is often seen as desirable, hence this type of churn may not be problematic, and may even be planned. We therefore focus on dependency *additions*, given these represent the most problematic type of change to legacy components.

Table 7 gives descriptive data relevant to testing hypothesis two for each release.

Table 7: Descriptive Data for Each Release

VERSION (V)	B	C	D	E	F
Source Files	284	1407	1857	2434	2756
Legacy Source Files (LSF)	116	170	1261	1296	2330
LSF with Dependency Additions	101	149	391	520	341
Churn Rate	87.1%	87.6%	31.0%	40.1%	14.6%
Dependencies	2806	7025	8727	10424	11128
Dependency Changes (from V-1)	2415	8079	4652	8795	1706
of which "Surprise" Changes	348	1184	1937	2386	640
of which, Dependency Additions	220	336	720	940	364
Surprises as % All Changes	14.4%	14.7%	41.6%	27.1%	37.5%
Additions as % All Surprises	63.2%	28.4%	37.1%	39.4%	56.9%

Three observations can be made from this data. First, the churn rate declines over time, indicating that legacy components are increasingly stable. While the trend is not uniform, it is marked in nature; from 87% in early versions to less than 15% in the final version. Second, the number of surprise dependency changes is a significant portion of the total number of changes in each new version, ranging from just under 15% to over 40%. To the degree that development effort is correlated with dependency changes, the data suggests that significant resources must be devoted to maintaining legacy functions. Finally, while the proportion of surprises associated with dependency additions varies from 28.4% to 68.2%, the mean is 39.4%.[17] This suggests that maintenance efforts, considered individually, tended to *increase* the modularity of the legacy functions.

[17] Of the 6,495 surprise dependency changes since version A, there were 3,915 reductions and 2,580 additions. On balance therefore, these efforts have tended to *increase* the modularity of the legacy design.

Table 8 contains the results of a logistic regression model predicting the churn rate of legacy components in each version, as well as for all versions considered in aggregate. The aggregate model includes dummies for each version to control for differences in churn rate that are related to unobservable factors.

Table 8: Logistic Regression Models Predicting Source File Churn[18]

VERSION[19]	B†	C†	D	E	F	ALL
Churn Rate	87.1%	87.6%	31.0%	40.1%	14.6%	
Constant	-38.1	49.0****	-2.87****	-2.46****	-3.94****	-4.10****
Version B	n/a	n/a	n/a	n/a	n/a	2.71****
Version C	n/a	n/a	n/a	n/a	n/a	2.50****
Version D	n/a	n/a	n/a	n/a	n/a	1.01****
Version E	n/a	n/a	n/a	n/a	n/a	1.85****
High VFI Only	40.42	-48.11****	3.23****	2.80****	2.48****	2.88****
High VFO Only	39.38	-47.14****	2.31****	3.06****	2.10****	2.51****
High VFI and VFO	42.28	-46.50****	3.75****	4.27****	4.33****	4.17****
Connectivity FI	2.75**	-0.18	0.85****	0.31***	0.91****	0.68****
Connectivity FO	1.37*	1.43****	1.17****	1.03****	0.54****	0.84****
McFadden R-square	45.1%	14.5%	37.0%	40.6%	36.6%	43.8%
Estrella R-square	37.0%	11.0%	43.6%	50.4%	31.6%	50.1%
Sample Size (N)	116	170	1261	1296	2330	5173

**** $p<0.1\%$, *** $p<1\%$, ** $p<5\%$, * $p<10\%$ † See discussion of model robustness in the text

The results show strong support for our second hypothesis. While the strength of the effects varies across versions, all four measures of modularity are significant at p=0.001 in the aggregate model. Legacy components that are more tightly-coupled have a higher likelihood of experiencing dependency additions. In the aggregate model, measures of modularity predict 40% to 50% of the variation in outcomes.

[18] We report two measures of R-square to assess goodness of fit (McFadden, 1984; Estrella, 1998).

HIGH VISIBILITY FILES CHURN DESPITE MATURITY
LOW VISIBILITY FILES BECOME "FINISHED"

We note that the models for versions B and C behave abnormally with respect to statistical significance and the size of the coefficients. To understand why, we examine data on churn rate split by file type, as identified by levels of visibility (see **Appendix D**). It is instantly apparent why these versions present problems: low visibility files are rare. In version B, only three of the 116 legacy files have low visibility, while in version C, only one of the 170 legacy files has low visibility. These models are therefore not robust. Only in versions D and later are there a large number of legacy components with low visibility. In these versions, the churn rate for these files is extremely low, between 2% and 9%. By comparison, the churn rate for high visibility files is always above 50%.

To understand the size of effects in our aggregate model, we substitute values and observe the differences in predicted churn rate. **Table 9** displays the actual churn rates for each version as well as predicted churn rates for a file with low visibility on both dimensions and a file with high visibility on both dimensions (i.e., a "core" file).

Table 9: Actual and Predicted Churn Rates for Files with Different Visibility

VERSION	Actual Churn Rate: All Files	Predicted Churn Rate: Low Visibility	Predicted Churn Rate: High Visibility
B	87.1%	18.8%	93.8%
C	87.6%	16.4%	92.7%
D	31.0%	4.4%	74.8%
E	40.1%	9.6%	87.3%
F	14.6%	1.6%	51.7%

Several observations can be made from this table. First, the differences in predicted churn rate are dramatic. The likelihood of a low visibility file experiencing churn is always less than 20%, and less than 10% in the later versions with more robust data. By contrast, the likelihood of a high visibility file experiencing churn always exceeds 50%. Second, the churn rate declines over time, *independent* of component visibility. While this trend is not uniform, the decrease is distinct for both file types. Finally, the churn rate for high visibility files remains high, even at the end of the period. Despite the product's maturity, these files are still more likely than not to experience dependency additions. By contrast, the churn rate for low visibility files is close to zero.

[19] There is no analysis for version A given this is the first version of the design so all components are new.

5.3 Hypothesis Three: Component Augmentation

Our third hypothesis looks at differences in the levels of modularity between new and legacy components. In particular, we test whether new components are more loosely-coupled than legacy components. For measures of Direct Connectivity, we conduct a Mann-Whitney-U test of population differences.[20] For measures of Visibility, we conduct a Pearson's Chi-square test, which assesses whether new and legacy components differ in the frequency with which they have high visibility. **Table 10** displays the results of our tests by version (non-significant results are shaded).

Table 10: Differences in Modularity between New and Legacy Source Files[21]

VERSION		B		C		D		E		F	
		New	Legacy	New	Legacy	New	Legacy	New	Legacy	New	Legacy
DCFI	Mean	1.46	2.10	0.78	0.93	0.61	0.97	0.52	0.95	0.37	0.76
	Test Stat.	U=6683****		U=38785****		U=440k****		U=899k****		U=625k****	
DCFO	Mean	1.87	2.25	1.0	2.19	1.0	1.15	0.88	1.20	0.65	1.07
	Test Stat.	U=7425****		U=45753****		U=408k ****		U=870k ****		U=617k ****	
High VFI	Frequency	105 of 168	98 of 116	248 of 1237	151 of 170	54 of 596	365 of 1261	91 of 1138	321 of 1296	6 of 426	406 of 2330
	Test Stat.	χ^2=16.3****		χ^2=348.0****		χ^2=91.6****		χ^2=121.2****		χ^2=72.7****	
High VFO	Frequency	138 of 168	108 of 116	455 of 1237	148 of 170	234 of 596	521 of 1261	381 of 1138	628 of 1296	99 of 426	990 of 2330
	Test Stat.	χ^2=7.1***		χ^2=154.3****		χ^2=0.7		χ^2=56.0****		χ^2=55.8****	
Sample N		284		1407		1857		2434		2756	

**** p<0.1%, *** p<1%, ** p<5%, * p<10%

The results show strong support for our third hypothesis. In almost all versions, there are significant differences between new and legacy components on all measures. New components have both lower direct connectivity and visibility than legacy components.

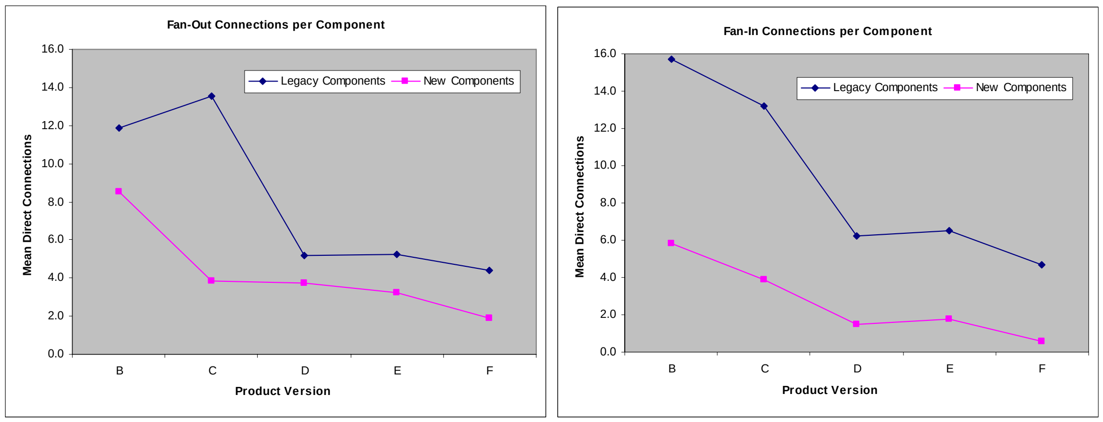
NEW STUFF IS MORE MODULAR 88

[20] The measures of Direct Connectivity are not normally distributed as determined by a Shapiro-Wilk's test. Hence we use a non-parametric test of whether the two samples come from the same distribution.
[21] All components in version A are new and therefore there is no analysis for this version.

ARCHITECTURE BECOMES
MORE MODULAR TOO

To illustrate the nature of this effect, we plot the mean number of direct connections for new and legacy components by version in **Figure 5**.[22] The chart shows that the level of coupling for all components declines over time, suggesting that the design as a whole is becoming more modular. Critically however, the level of coupling for new components is *always* lower than that for legacy components, even as the overall level of coupling declines. We conclude that not only are new components significantly more modular than legacy components, but in addition, the mix of new components becomes increasingly more modular over successive versions of the product.

Figure 5: Differences in Modularity between New and Legacy Components



6. Discussion

Our results confirm the existence of a relationship between component modularity and design evolution that is both statistically significant and large in magnitude. In particular, we show that measures of modularity predict three different aspects of design evolution: component survival, component maintainability and component augmentation. Tightly-coupled components are more likely to survive from one design version to the next, implying that they are less adaptable via the processes of exclusion or substitution; they are more likely to experience "surprise" dependency additions unrelated to new functionality, implying that they demand greater maintenance efforts; and they are harder to augment, in that the mix of new components is more modular than the legacy design.

[22] We focus on the number of direct connections given this is more comparable across versions than the measure of visibility. The value of visibility changes significantly over time as the system grows in size.

CORE DEFINED EARLY
THEN HARD TO CHANGE

In aggregate, our results paint a broad picture of how a design evolves. The core components of a system are defined early in its life. These components are destined to both be long lasting and require greater maintenance efforts. As the system matures, new versions add successively fewer core components, instead placing greater emphasis on peripheral functions. While these may still represent critical additions to the design, they are likely to be restricted in their impact. In essence, the difficulty in adding tightly-coupled components constrains the evolutionary path for a mature system.

It is important to consider alternative explanations for the dynamics we observe. In particular, it might be argued that tightly-coupled components are more important to a system than other components, and it is this that explains their higher rate of survival. Indeed, we believe that tightly-coupled components *are* typically more important than others, in that they often form part of a system's core functions. However, this does not explain why these components are less likely to be adapted through exclusion or substitution. On the contrary, if they are core to the system's function, we might expect them to be adapted *more* frequently, as designers strive to improve performance by deploying technical advances that make some components obsolete and others candidates for replacement. As a result, we believe our results are explained, in the main, by the difficulty in adapting tightly-coupled components.

Our results 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 modularity are typically founded upon the trade-offs this entails within the current version of a design (e.g., in terms of superior performance versus increased reliability). Yet our results reveal the long-lasting and potentially irreversible nature of these choices. Tightly-coupled components are harder to kill hence their choice implies a reduction in future flexibility. And they are harder to maintain, in that they experience more corrective design changes in subsequent versions. The challenge for a decision-maker is that these longer term costs are neither as easy to calculate nor as salient as the near term benefits that may stem from tighter coupling. As a result, managers are likely to systematically under-invest in modularity.

These problems are magnified in the context of software, given that legacy code is rarely re-written, but instead forms a platform upon which new versions are built. In such a system, today's developers bear the consequences of design decisions made long ago. Yet the first designers of a system have different objectives from those that follow, especially if the first of its type in a particular market segment. The emphasis is on product performance and time to market; speed is of the essence. Future "adaptation" is rarely an important consideration when there is no guarantee the product will succeed. These problems are compounded by the fact that designers rarely document design choices well, and may not be employed by the firm when these decisions must be revised.

Our results also make an important contribution to the academy, representing one of the first empirical studies to confirm a link between modularity and design evolution. Our research is distinct from prior work on evolution in two respects. First, we adopt a research design that sheds light on these dynamics at the microstructure level and analyzes the impact of differing choices using longitudinal data. And second, we link measures of modularity to three different aspects of design evolution, all of which must be understood in order to fully describe evolutionary dynamics. The result is a detailed understanding of how designs evolve over time, and the role played in this process by the myriad of individual choices about levels of component modularity.

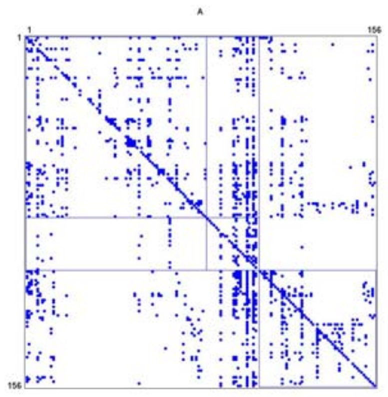
Our study has several limitations which must be considered in generalizing the results. First, we examine a single product in the software industry, hence cannot be sure that the findings apply to other industries or to other products within this industry. With respect to the latter concern, we note that similar analyses on other software to which we have access produces similar results, although the strength of effects differs. Second, our analysis examines only one type of dependency between components, with the assumption that this is a proxy for the overall level of coupling between parts of a system. If different dependencies generate different dynamics, our results may not capture these effects. Finally, our analysis treats each component as a "black box" in that we focus only on its relationships with other boxes, as opposed to what happens within the box. To the extent that these dynamics play a role in explaining patterns of evolution, further work is needed to connect the two levels of analysis.

Our 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 we observe may be limited. Second, work is needed to expose the broader organizational influences on design. Prior work asserts that products tend to mirror the organizations that develop them (Conway, 1968; Henderson and Clark, 1990). This "duality" implies there are *implicit* constraints on design choices which must be better understood. Finally, the methods we develop can be used to assess the degree to which regular patterns occur in system design and evolution. Much work asserts 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 most explain differences between them. Ultimately, this agenda promises to help understand the choices available to a designer, and the impact of their choices on both product and organizational performance.

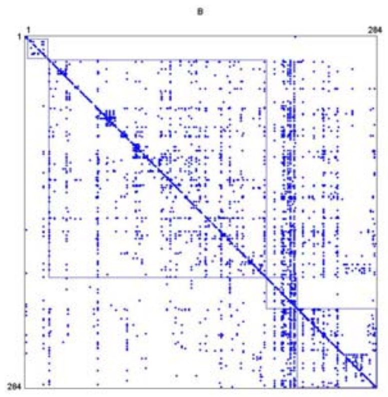
→ THEY DID THIS FOLLOWUP STUDY IN 2013

APPENDIX A: DSMs FOR EACH VERSION OF THE PRODUCT[23]

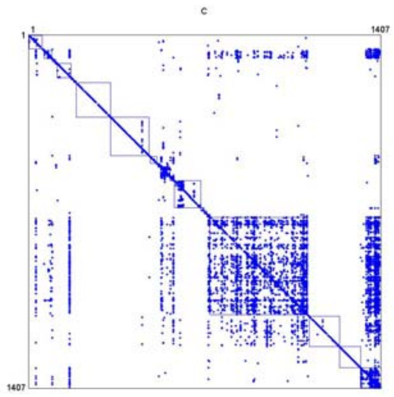
Version A



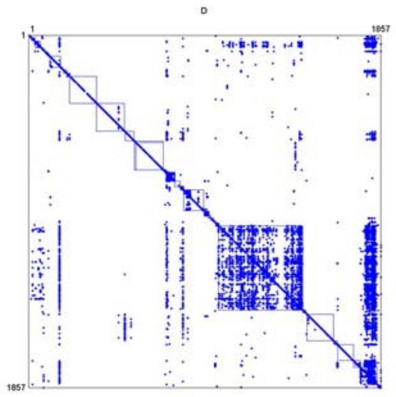
Version B



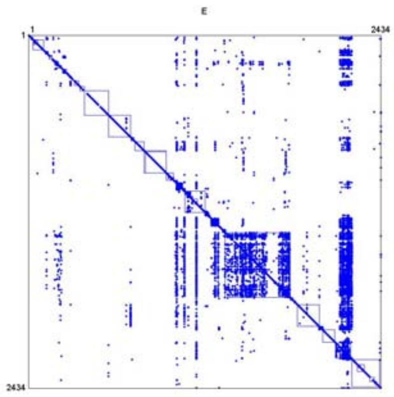
Version C



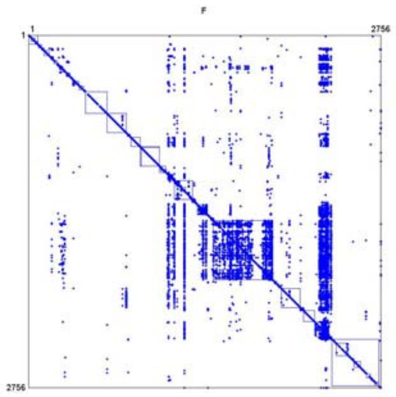
Version D



Version E



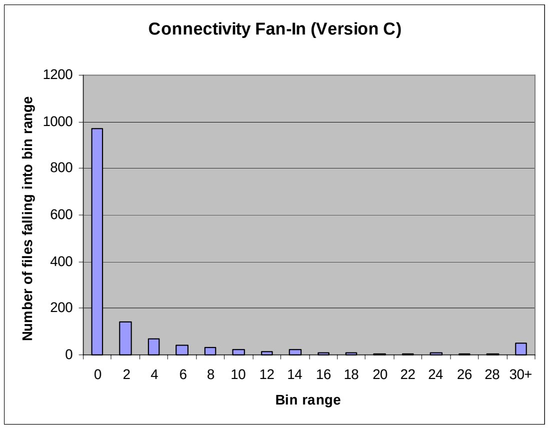
Version F



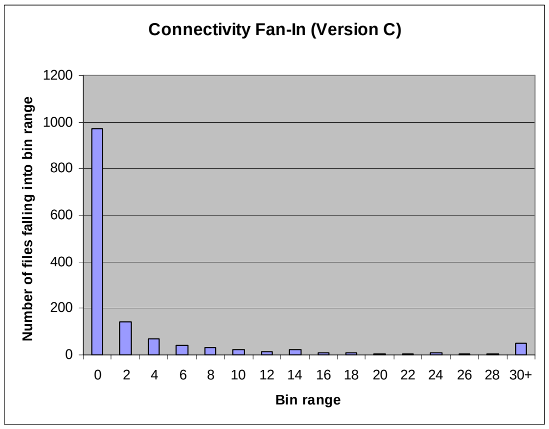
[23] Note that these DSMs are drawn with different scales. The system grows considerably in size over time: the first DSM has 156 elements; the final DSM has 2,756 elements.

APPENDIX B: DISTRIBUTION OF MODULARITY MEASURES (VERSION C)

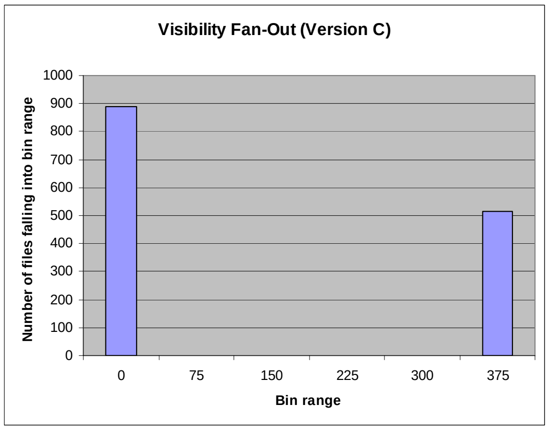
Direct Connectivity Fan-Out



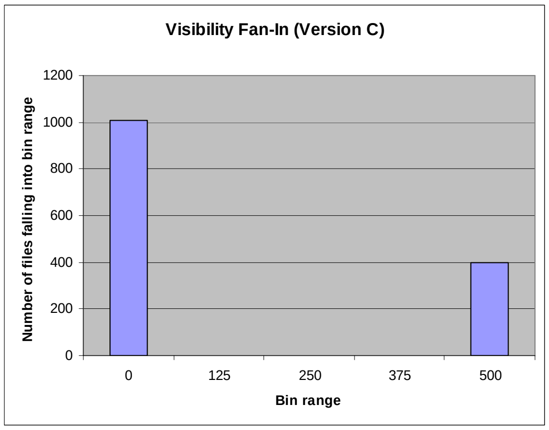
Direct Connectivity Fan-In



Visibility Fan-Out



Visibility Fan-In



APPENDIX C: SOURCE FILE SURVIVAL BY LEVEL OF VISIBILITY

Version	File Type	Files	Survive	Survival Rate	Ratio to Version Mean
A	High VFI and VFO	113	91	80.5%	1.08
	High VFI Only	11	10	90.9%	1.22
	High VFO Only	22	12	54.5%	0.73
	Low VFI and VFO	10	3	30.0%	0.40
	All Files	156	116	74.4%	
B	High VFI and VFO	189	148	78.3%	1.31
	High VFI Only	14	10	71.4%	1.19
	High VFO Only	57	11	19.3%	0.32
	Low VFI and VFO	24	1	4.2%	0.07
	All Files	284	170	59.9%	
C	High VFI and VFO	357	353	98.9%	1.10
	High VFI Only	42	40	95.2%	1.06
	High VFO Only	246	237	96.3%	1.07
	Low VFI and VFO	762	631	82.8%	0.92
	All Files	1407	1261	89.6%	
D	High VFI and VFO	367	302	82.3%	1.18
	High VFI Only	52	45	86.5%	1.24
	High VFO Only	388	291	75.0%	1.07
	Low VFI and VFO	1050	658	62.7%	0.90
	All Files	1857	1296	69.8%	
E	High VFI and VFO	370	362	97.8%	1.02
	High VFI Only	42	42	100.0%	1.04
	High VFO Only	639	625	97.8%	1.02
	Low VFI and VFO	1383	1301	94.1%	0.98
	All Files	2434	2330	95.7%	

APPENDIX D: LEGACY SOURCE FILE CHURN BY LEVEL OF VISIBILITY[24]

Version	File Type	Files	Churn	Churn Rate	Ratio to Version Mean
B	High VFI and VFO	91	85	93.4%	1.07
	High VFI Only	10	7	70.0%	0.80
	High VFO Only	12	9	75.0%	0.86
	Low VFI and VFO	3	0	0.0%	0.00
	All Files	116	101	87.1%	
C	High VFI and VFO	148	132	89.2%	1.02
	High VFI Only	10	7	70.0%	0.80
	High VFO Only	11	9	81.8%	0.93
	Low VFI and VFO	1	1	100.0%	1.14
	All Files	170	149	87.6%	
D	High VFI and VFO	353	234	66.3%	2.14
	High VFI Only	40	23	57.5%	1.85
	High VFO Only	237	91	38.4%	1.24
	Low VFI and VFO	631	43	6.8%	0.22
	All Files	1261	391	31.0%	
E	High VFI and VFO	302	250	82.8%	2.06
	High VFI Only	45	26	57.8%	1.44
	High VFO Only	291	183	62.9%	1.57
	Low VFI and VFO	658	61	9.3%	0.23
	All Files	1296	520	40.1%	
F	High VFI and VFO	362	206	56.9%	3.89
	High VFI Only	42	10	23.8%	1.63
	High VFO Only	625	95	15.2%	1.04
	Low VFI and VFO	1301	30	2.3%	0.16
	All Files	2330	341	14.6%	

[24] This table shows only legacy source files, given these are the only files which experience churn.

REFERENCES

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.

Banker, Rajiv D., Srikant Datar, Chris Kemerer, and Dani Zweig (1993) "Software Complexity and Maintenance Costs," *Communications of the ACM,* 36(11):81-94.

Barry, Evelyn, Chris Kemerer and Sandra Slaughter (2006) "Environmental Volatility, Development Decisions, and Software Volatility: A Longitudinal Analysis," *Management Science,* 52(3): 448-464.

Barry, Evelyn, Chris Kemerer and Sandra Slaughter (2007) "How Software Process Automation Affects Software Evolution: A Longitudinal Empirical Analysis," Unpublished Working Paper, University of Pittsburgh, Katz School of Business.

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.

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.

Dhama, H. (1995) "Quantitative Models of Cohesion and Coupling in Software," *Journal of Systems Software*, 29:65-74.

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.

Estrella, Artuto. (1998). “A New Measure Of Fit For Equations With Dichotomous Dependent Variable", *Journal of Business & Economic Statistics,* Vol. 16, no. 2, pp. 198-205.

Halstead, Maurice H. (1977) *Elements of Software Science, Operating, and Programming Systems Series* Volume 7. New York, NY: Elsevier.

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.

Kemerer, Chris and Sandra Slaughter (1997) "Determinants of Software Maintenance Profiles: An Empirical Investigation," *Software Maintenance: Research and Practice*, 9: 235-251.

Kemerer, Chris and Sandra Slaughter (1999) "An Empirical Approach to Studying Software Evolution," *IEEE Transactions on Software Engineering*, 25(4): 493-509.

LaMantia, Matthew J., Yuanfang Cai, Alan D. MacCormack and John Rusnak (2008) "Analyzing the Evolution of Large-Scale Software Systems using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases," *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architectures (WICSA7)*, Vancouver, BC, Canada, February 18-22.

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.

Lehman, M. M. and L.A. Belady (1976) "A Model of Large Program Development," *IBM Systems Journal,* 15(3): 225-252.

Lehman, M. M. and L.A. Belady (1985) "Program Evolution: Processes of Software Change," A.P.I.C. Studies in Data Processing Volume 27, Academic Press, Orlando, FL.

Lopes, Cristina V. (2005) "On the Nature of Aspects: Principles of Aspect-Oriented Design," *ACM Transactions of Software Engineering*.

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, Roberto Verganti, and Marco Iansiti (2001) "Developing Products on 'Internet Time': The Anatomy of a Flexible Development Process." *Management Science* 47(1):133-150.

MacCormack, Alan., and Kerry Herman (2000) "Microsoft Office 2000," Harvard Business School Multimedia Case Study, HBS Case Number 600-023.

McCabe, T.J. (1976) "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, Jul/Aug, pp. 308-320.

McFadden, Daniel. (1984) "Econometric Analysis of Qualitative Response Models", in Zvi Griliches and Michael D. Intriligator, eds. *Handbook of Econometrics,* Volume 2, (North-Holland: Amsterdam).

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.

Parnas, David L. (1972b) "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM* 15: 1053-58.

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.

Schach, Stephen R., Bo Jin, David R. Wright, Gillian Z. Heller and A. Jefferson Offutt (2002) "Maintainability of the Linux Kernel," *IEE Proc. Software,* Vol. 149. IEE, Washington, D.C. 18-23.

Selby, R. and V. Basili (1988) "Analyzing Error-Prone System Coupling and Cohesion," *University of Maryland Computer Science Technical Report* UMIACS-TR-88-46, CS-TR-2052, June 1988.

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, Manuel, Steven Eppinger and Craig Rowles (2003) "Identifying Modular and Integrative Systems and their Impact on Design Team Interactions", *ASME Journal of Mechanical Design*, 125 (June): 240-252.

Sosa, Manuel, Steven Eppinger and Craig Rowles (2004) "The Misalignment of Product Architecture and Organizational Structure in Complex Product Development," *Management Science*, 50(December):1674-1689.

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).

Sullivan, Kevin, William G. Griswold, Yuanfang Cai and Ben Hallen (2001). “The Structure and Value of Modularity in Software Design,” *SIGSOFT Software Engineering Notes*, 26(5):99-108.

Trujillo, Ortiz, A., F.A. Trujillo-Rodriguez, R. Hernandez-Walls, M.A. Fligner, an S. Perez-Osuna (2003) "FPTest: Non-parametric Fligner-Policello test of two combined random variables with continuous probability distribution," <www.mathworks.com/matlabcentral/fileexchange/>, Accessed December 6th 2007.

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.

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

Weick, Karl E. (1976) “Educational Organizations as Loosely Coupled Systems,” *Administrative Science Quarterly,* Vol. 21, No. 1, March, pp. 1-19.

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.