HARVARD | BUSINESS | SCHOOL

The Architecture of Complex Systems: Do Core-periphery Structures Dominate?

Alan MacCormack
Carliss Baldwin
John Rusnak

Working Paper

10-059

The Architecture of Complex Systems: Do Core-periphery Structures Dominate?

Alan MacCormack[a]
MIT Sloan School of Management
50 Memorial Drive, E52-538
Cambridge, MA
alanmac@mit.edu

Carliss Baldwin, John Rusnak
Harvard Business School
Soldiers Field Park
Boston, MA
cbaldwin@hbs.edu; jrusnak@hbs.edu

MIT Sloan School of Management Working Paper Number 4770-10

Draft Version 4.0 Dated: January 19th, 2010

[a] Corresponding Author

Abstract

Any complex technological system can be decomposed into a number of subsystems and associated components, some of which are core to system function while others are only peripheral. The dynamics of how such "core-periphery" structures evolve and become embedded in a firm's innovation routines has been shown to be a major factor in predicting survival, especially in turbulent technology-based industries. To date however, there has been little empirical evidence on the propensity with which core-periphery structures are observed in practice, the factors that explain differences in the design of such structures, or the manner in which these structures evolve over time.

We address this gap by analyzing a large number of systems in the software industry. Our sample includes 1,286 software releases taken from 19 distinct applications. We find that 75-80% of systems possess a core-periphery structure. However, the number of components in the core varies widely, even for systems that perform the same function. These differences appear to be associated with different models of development – open, distributed organizations developing systems with smaller cores. We find that core components are often dispersed throughout a system, making their detection and management difficult for a system architect. And we show that systems evolve in different ways – in some, the core is stable, whereas in others, it grows in proportion to the system, challenging the ability of an architect to understand all possible component interactions. Our findings represent a first step in establishing some stylized facts about the structure of real world systems.

1. Introduction

All complex systems can be decomposed into a nested hierarchy of subsystems (Simon, 1962). Critically however, not all these subsystems are of equal importance (i.e., centrality). In particular, some subsystems are "core" to system performance, whereas others are only "peripheral" (Tushman and Rosenkopf, 1992). Core subsystems are defined as those that are tightly coupled to other subsystems, whereas peripheral subsystems possess only loose connections to other subsystems (Tushman and Murmann, 1998). Studies of technological innovation consistently show that major changes in core subsystems, or the mechanisms by which these subsystems link together, can have a significant impact on firm performance (Henderson and Clark, 1990; Christensen, 1997). Despite the wealth of research highlighting the importance of understanding how such systems are structured however, there is little empirical evidence on the patterns that are observed across large samples of real world systems.

In this paper, we adopt network representations of a design to analyze a large sample of systems in the software industry. Our methods allow us to determine the level of coupling between different components in a system, and thereby to determine which are core and which are peripheral to system functions. Our objective was to understand the extent to which complex technological systems possess a "core-periphery" structure, and to explore the factors that explain differences between these structures (e.g., differences in the number of core components across systems). We also sought to examine how systems evolve over time, given prior work finds that major changes to the core components in a design can precipitate significant changes in industry structure.

Software is an ideal context in which to study these questions given the information-based nature of the product. Software code can be analyzed automatically to identify the level of coupling between components, and hence to determine unambiguously which are core, and which are peripheral. Furthermore, we can track the evolution of a design over time, comparing each new version to its predecessor to reveal how components evolve. Finally, different models of organization are observed in this industry (i.e., open source versus commercial development) providing an opportunity to assess whether these different organizing principles tend to result in differing product structures.

Our findings make an important contribution to the literature exploring the design of complex technological systems. We show that core-periphery structures dominate, with 75-80% of systems in our sample possessing such a structure. However, we find tremendous variation in the number of core components across systems, even when controlling for system size and function. Critically, these differences appear to be driven by differences in the structure of the developing organization. We also find that core components tend to be distributed throughout a system, rather than being concentrated within a few subsystems. And the number of core components is not always stable, but often increases as a system grows over time. These general patterns highlight the difficulties a system architect faces in designing and managing such systems.

The paper proceeds as follows. In the next section, we review the prior literature on the design of complex technological systems, focusing on how these systems can be characterized in terms of their core and peripheral elements. We then describe our research methods, which make use of a technique called Design Structure Matrices to understand the structure of systems by measuring the levels of coupling between components. Next, we introduce the context for our study and describe the sample of software systems that we analyze. Finally, we report our empirical results, and discuss the implications of our findings for both the academy and for practicing managers.

2. Literature Review

The design of a complex technological system has been shown to comprise a nested hierarchy of design decisions (Marples, 1961; Alexander, 1964; Clark, 1985). Decisions made at higher levels of the hierarchy set the agenda (or technical trajectory) for problems that must be solved at lower levels of the hierarchy (Dosi, 1982). These decisions influence many subsequent design choices, hence are referred to as "core concepts." For example, in developing a new automobile, the choice between Internal Combustion Engine and Electric Propulsion represents a core conceptual decision that will influence many subsequent design decisions. In contrast, the choice of leather versus upholstered seats typically has little bearing on other important system choices.

A variety of studies show that a particular set of core concepts can become embedded in an industry, representing a "dominant design" (DD) that sets the agenda for subsequent technical change (Utterback, 1996; Utterback and Suarez, 1991; Suarez and Utterback, 1995). DDs have been observed across many different industries, including typewriters, automobiles and televisions (Utterback and Suarez, 1991). Their emergence is associated with periods of industry consolidation, in which firms pursuing non-dominant designs fail, while those producing superior variants experience increased market share and profits. As a result, understanding what constitutes a DD, why specific designs come to dominate and how this process can be influenced are topics of considerable importance.

Unfortunately, scholars differ on what constitutes a DD and whether this phenomenon is an antecedent or a consequence of changing industry structures (Klepper, 1996; Tushman and Murmann, 1998; Murmann and Frenken, 2006). For example, Klepper argues the concept is not sufficiently distinct to identify DDs ex-ante, and further, that DDs are a consequence (not a cause) of competitive dynamics. Murmann and Frenken show that the past literature has been inconsistent in identifying DDs, especially with regard to the level of analysis. And Tushman and Murmann describe how advances in airplane design are not easily classified in terms of a DD, but rather are shaped by design evolutions at the *subsystem* level, both breakthrough and incremental in nature.

Subsystems analysis provides a somewhat different approach to defining the core of a system. All complex systems can be decomposed into a nested hierarchy of subsystems (Simon, 1962). Yet this hierarchy is conceptually different from the hierarchy of design decisions that is the basis of a dominant design. The design hierarchy specifies the temporal ordering of design decisions, whereas the subsystem hierarchy indicates the *ex post* physical and logical relationships between design components. The first is a "flow" hierarchy, while the second is a "containment" hierarchy (Luo et. al, 2009).

Critically, not all subsystems are of equal importance (i.e., centrality). Core subsystems are defined as those that are tightly coupled to other subsystems, whereas peripheral subsystems possess only loose connections to others (Tushman and Rosenkopf, 1992, Tushman and Murmann, 1998). For example, in an automobile, the engine represents only one subsystem of the many required for a vehicle to function effectively. However, it is a core subsystem in that the changes to the engine design have a large potential impact on the designs of other subsystems.

Studies of technological innovation consistently show that major changes in core subsystems, or the mechanisms by which these subsystems link together, have a significant impact on firm performance. For example, Henderson and Clark (1990) and Christensen (1997) describe how leading firms in the photolithography and disc-drive industries failed when faced with changes in the way subsystems were connected. Furthermore, Langlois and Robertson (1985) and Sanderson and Uzermi (1990) show that design changes at the subsystem level help to explain differences in firm performance in the markets for home stereo equipment and portable music players.

Recent work has sought to resolve the tensions between the concept of a dominant design and the notion that a complex system comprises core and peripheral elements. Murmann and Frenken (2006) argue that a DD is best defined as a consistent set of choices with respect to the design of core (or "high-pleiotropy") subsystems. In essence, the centrality of core subsystems implies that major changes in these subsystems yield a new DD. For example, the shift to quartz oscillation technology in watch construction in the mid 1970's was associated with a new DD, given that the oscillation subsystem was tightly connected to other subsystems (Landes, 1983; Tushman and Rosenkopf, 1992). Conversely, changes to peripheral subsystems such as the bracelet had little impact on other subsystems, and did not affect the DD. When all core subsystems are in "eras of incremental change," the DD is stable (Tushman and Murmann, 1998).

While complex systems comprise a number of subsystems, these subsystems, in turn, encompass many components that work together to deliver needed functionality. A decision must therefore be made as to the unit of analysis appropriate for determining core and peripheral elements in a system. For example, in an automobile, the engine is a core subsystem, in that the design decisions made within have a great potential impact on other subsystems. However, not all components of the engine possess this characteristic; some have little or no impact on design decisions made elsewhere. Defining core and peripheral elements at the component level therefore allows a more precise evaluation of the core elements in a system. As a consequence, we adopt the following definitions:

Definition 1: *A complex technological system comprises a number of different subsystems and associated components. Components differ in their degree of centrality. Core components are those that are tightly coupled to other components. Peripheral components are those that are only loosely coupled to other components.*

Definition 2: *Core subsystems are those that contain many core components. Peripheral subsystems are those that contain only a few (or no) core components.*

Definition 3: *A Dominant Design is defined by a consistent pattern of choices with respect to the selection and design of the core components in a system.*

The preceding discussion raises a number of important questions for work that seeks to deepen our understanding of the structure of complex technological systems. First, is it possible to determine unambiguously the core and peripheral components in a system? Second, do real world systems always possess a core-periphery structure? Third, what factors help explain the *differences* between such structures (e.g., differences in the size of the core across systems?). And fourth, how do such structures evolve – are the core components stable, or do they tend to change significantly over time? To answer these questions requires that we understand how to evaluate system design structure.

Evaluating System Design Structure

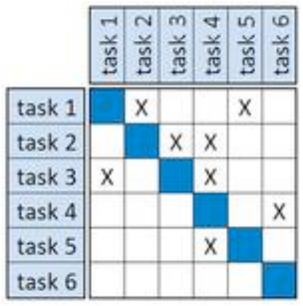
A large number of studies contribute to our understanding of the design of complex systems (Holland, 1992; Kaufman, 1993; Rivkin, 2000; Rivkin and Siggelkow, 2007). Many of these studies are situated in the field of technology management, exploring factors that influence the design of physical or information-based 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 adapted are critical topics in the study of complex systems.

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 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.

The costs and benefits of modularity have been discussed in a stream of research that has sought to examine its impact on a range of activities including 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, many studies are conceptual or descriptive in nature, offering little insight into how modularity can be measured in a robust and repeatable fashion (notable exceptions include Schilling (2000) and Fleming and Sorenson (2004)).

Studies that attempt to measure modularity typically employ network representations of a design (Barabasi, 2009). Their objective is to identify the level of coupling that exists between different elements (Simon, 1962; Alexander, 1964). One of the most widely adopted techniques comes from the field of engineering, in the form of the Design Structure Matrix (DSM). A DSM highlights the structure of a system by examining the dependencies that exist between its constituent elements in a square matrix (Steward, 1981; Eppinger et al, 1994; Sosa et al, 2007). These elements can represent design tasks, design parameters or the actual components in a system (see **Figure 1**).

Figure 1: A Task-based Design Structure Matrix



	task 1	task 2	task 3	task 4	task 5	task 6
task 1	🟦	X			X	
task 2		🟦	X	X		
task 3	X		🟦	X		
task 4				🟦		X
task 5				X	🟦	
task 6						🟦

Metrics that capture the level of coupling for each element can be calculated from a DSM and used to analyze system structure. For example, MacCormack et al (2007a) show that tightly coupled ("core") components tend to survive longer and are more costly to maintain, as compared to loosely coupled equivalents. Similarly, Von Krogh et al (2009) and Sosa et al (2009) find higher levels of component coupling are associated with more frequent changes and higher defect levels. Cataldo et al (2006) and Gokpinar et al (2007) show that teams developing components with higher levels of coupling require increased amounts of communication to achieve a given level of quality. Finally, MacCormack et al (2007b) show that the mean level of coupling varies widely across systems, the differences being explained, in part, by different models of development.

In this paper, we use DSMs to analyze a large sample of complex technological systems, thereby revealing the underlying design structure for these systems. Our objective is to understand the extent to which systems possess a core-periphery structure, and explore the factors that explain differences between these structures. We also examine how these systems evolve over time, given prior work finds that major changes in the core components of a design can precipitate significant changes in industry structure. The context for our investigation is the software industry. We choose this industry because software can be analyzed automatically to identify the level of coupling between components, allowing us to determine which are core and which are peripheral. Using such an approach allows us to assess the design structure for a large number of systems, and thereby make inferences about the nature of real world systems.

3. Research Methods

Below, we describe how we apply DSMs to analyze software systems, allowing us to establish whether these systems possess a core-periphery structure, to measure important attributes of this structure, and to assess how system structure changes over time.

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.

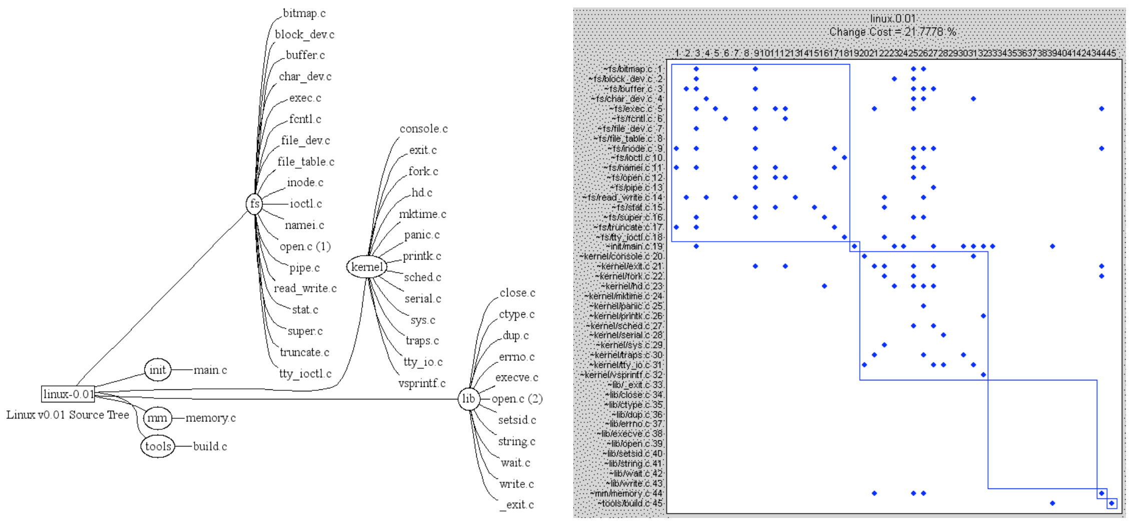
There are many types of dependency between source files in a software product.[2] We focus on one important dependency type – the "Function Call" – used in prior work on system design (Banker and Slaughter, 2000; Rusovan et al, 2005). A Function Call is an instruction that requests a specific task to 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 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.

To capture function calls, 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.[34] 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

[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).
[3] 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.
[4] Rather than develop a call-graph extractor, we tested several commercial products that could process source code written in both different 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++, distributed by Scientific Toolworks, was ultimately selected.

the Directory Structure and Architectural View for the first version of Linux v0.01 in **Figure 2**. 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 2: The Directory Structure and Architectural View of Linux version v0.01.

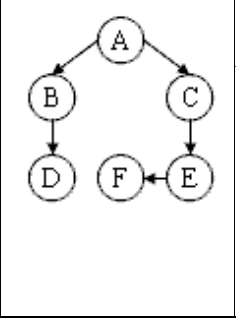


3.2 Measuring the Level of Component Coupling

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" (Sharman 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 3** 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 3: 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

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.[5] **Figure 4** illustrates the derivation of this matrix for the example above. The measures of component visibility are derived directly from this matrix. Fan-In Visibility (FIV) is obtained by summing down the columns; Fan-Out Visibility (FOV) is obtained by summing along the rows. For comparisons between systems of different sizes, FIV and FOV can be expressed as a percentage of the total number of components in a system.

Figure 4: 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**

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

3.3 Defining Core and Peripheral Components

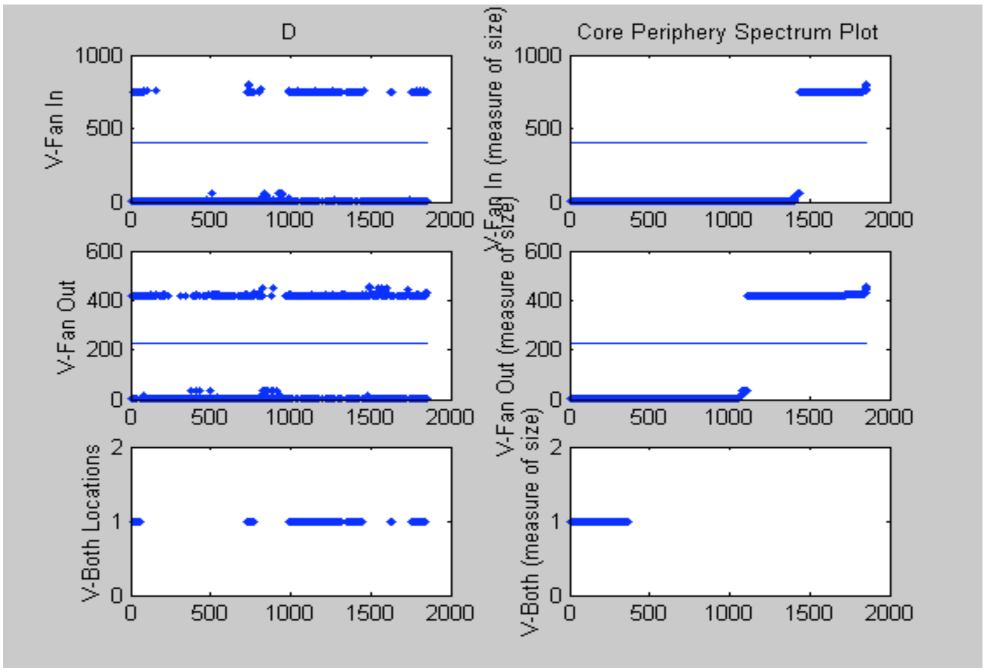
We define a "Core" component as one that is tightly coupled to many other components in both directions. That is, many other components depend upon it (i.e., it has a high FIV) *and* it depends upon many other components (i.e., it has a high FOV). To evaluate whether a component meets this criterion, we observe the maximum level of visibility for each measure, and define those that exceed 50% of this value as "high." Given we capture two measures of visibility, each of which that can be high or low, every component can be classified into one of four canonical types (see **Table 1**).

Table 1: Four Canonical Types of Component

Core Components: High FIV, High FOV	Files with high visibility on both measures are "Core" files. They are "seen by" many files and "see" many files. They are often linked directly or indirectly to all other core files.
Shared Components: High FIV, Low FOV	Files with high FIV are "Shared" files. They provide shared functionality to many different parts of the system. These files are "seen by" many files, but do not "see" many files.
Peripheral Components: Low FIV, Low FOV	Files with low visibility on both measures are "Peripheral" files. They are neither "seen by" many files nor "see" many files. They typically execute independently of other files.
Control Components: Low FIV, High FOV	Files with high VFO are "Control" files. They direct the flow of program control to different parts of the system. These files "see" many other files, but are not "seen by" many files.

To illustrate this analysis, we provide the "Spectrum Plot" for a sample system in **Figure 5**. This plot displays data on the two measures of visibility, one above the other, organized alphabetically by directory (on the left) and by increasing value (on the right). In each graph, we identify the 50% threshold above which visibility is defined to be high. A third pair of graphs ("V-both") identifies the core components of the system, defined as those components that possess a high level of visibility on both measures. The left hand graph reveals *where* in the system the core components are located; the right hand graph reveals *how many* core components the system has in total.

Figure 5: The Spectrum Plot for a System of ~ 2,000 Components.

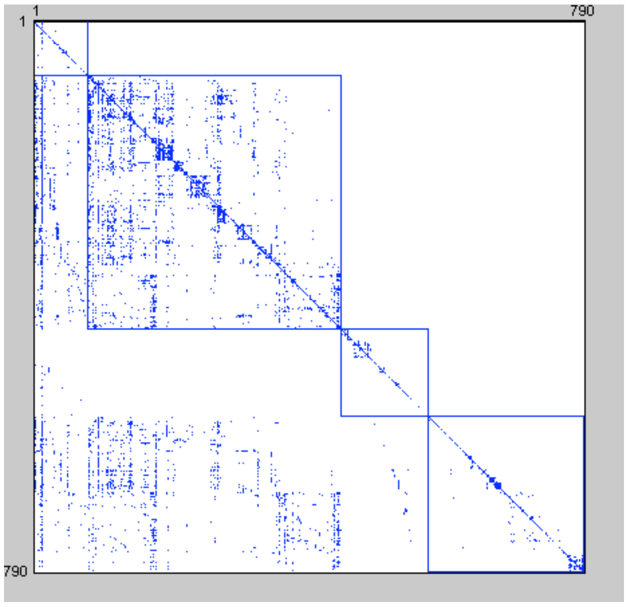


Note that in this example, the measures of visibility are not normally distributed. Rather they appear to possess a bi-polar distribution, in that most components have an FIV of zero or ~ 800 and an FOV of zero or ~ 400. This pattern is observed in many sample systems, and can be traced to the nature of the visibility measure, which captures all direct and indirect connections for a component. A single connection to a component with many dependencies will result in a different component sharing these same dependencies. As a consequence, many files possess similar levels of visibility. This pattern, where it exists, increases the ease and robustness with which components can be divided into those that possess high and low levels of visibility.

3.4 Displaying System Design Structure

To display the results of this analysis in a DSM, we can group components into the categories defined above. We order these categories according to whether components *depend upon* others, or are *depended upon* by others. Shared components are placed highest in the DSM given many others depend upon them. Control components are placed lowest in the DSM given they depend upon many others. Core components have both characteristics, and are placed in the upper-middle part of the DSM. Peripheral components have neither characteristic, and are placed in the lower-middle part of the DSM.[6] In **Figure 6** below, we show the *Core-Periphery View* for a sample system. This view, by the nature of its construction, creates "white space" in the upper-right hand part of the DSM. That is, components placed higher in the DSM do not depend upon components below them, unless they are both in the same category.

Figure 6: The Core-Periphery View of System Structure



4. Empirical Data and Analytical Approach

4.1 Empirical Data

The dataset comprises 1286 different software releases from 19 different software applications for which we could secure access to the source code (see **Appendix A**). Many of these systems are active open source software projects, given it is often possible to access historical data on all past releases in such systems. Some systems were once commercial products that at some point were released under an open source license (e.g., the Mozilla web browser, and the Open Office productivity suite). Finally, a small

[6] This process mirrors prior work in which the DSM elements are tasks, and not components. Tasks that must be completed before others begin are placed earlier in the sequence (i.e., at the top of the DSM). As a result, the DSM has a "lower triangular" form, meaning that most dependencies are below the diagonal. Above diagonal (cyclical) dependencies imply that a dependency cannot be resolved by sequencing.

number of releases come from proprietary systems developed by commercial firms, for which we have disguised any data that we show.

We focus only on analyzing large software systems that reach a minimum level of complexity and that have proven relatively successful in the industry (as measured by having a large number of end-user deployments). Hence we do not include in our sample open source projects from repositories such as SourceForge, which are typically extremely small systems developed by a handful of developers, and are not widely deployed with end-users. All of the systems in our sample evolve to contain over three hundred source files. That said however, our sample is not random in nature or representative of the industry as a whole. This will limit the generality of our results.

We obtained the source code for each release in the sample, processed it to identify dependencies between source files, and used this data to calculate the measures of visibility (FIV and FOV) for each file. We then allocate each file to one of the four categories described above. We define a release as possessing a core-periphery structure if it has one or more core components. **Table 2** contains descriptive data for the sample. Our sample includes a wide spectrum of system sizes, from less than 50 components, to over 12,000. The number of core components varies considerably, from a minimum of zero to a maximum of over 3,000 components. As a fraction of the system, the core varies from zero to 75% of all components. The average release has 1742 components, of which 188 (14.9%) are found in the core.

Table 2: Descriptive Data for the Sample

	MIN	MAX	MEAN	MEDIAN
System Size	45	12949	1724	781
Core Components	0	3310	188	72
Core % of System	0.0%	74.9%	14.9%	8.3%

4.2 Analytical Approach

Given the exploratory nature of this study, our objective was not to formally test hypotheses about system structure, but to examine patterns in the data. We were interested in the extent to which core-periphery structures were observed, and the factors that appeared to predict whether a system possessed such a structure. We also wanted to examine the characteristics of such structures, focusing on variations in the size of the core, and the factors that best explained these variations. Next, we wished to explore where the core components were located, focusing on whether they were clustered in a few core subsystems or distributed throughout many of them. Finally, we wanted to evaluate patterns in the evolution of systems, focusing on changes in the size of the core as a system grew. Below, we describe our observations in each of these areas.

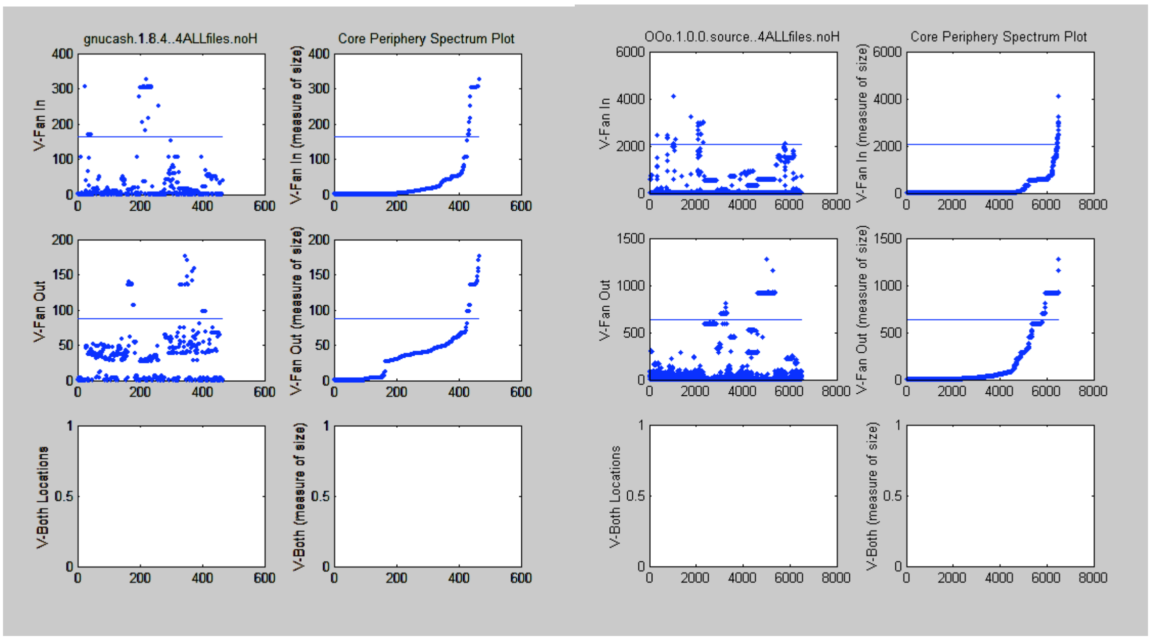
5. Empirical Results

5.1 The Dominance of Core-periphery Structures

We find that 1027 of the 1286 releases (80%) possess a core-periphery structure; meaning they contain components with both high levels of Fan-In and Fan-Out Visibility. Focusing only on the distinct applications in our sample, we find that 14 of the 19 systems tend to exhibit a core-periphery structure (74%). We conclude that core-periphery structures dominate this sample. However, the fact that there are exceptions to this pattern is important. Software systems do not have to possess such a structure. Choice plays a critical role in determining the outcome of system design processes.

Below, we explore the characteristics of core-periphery systems in greater detail. First however, we consider the small number of systems that do not possess such a structure, to evaluate whether consistent patterns are observed in their structure. In this respect, we note two different types of system: those that appear to have "no core;" and those that appear to have "multiple cores." To illustrate the difference, spectrum plots from two sample systems are shown below (see **Figure 7**).

Figure 7: Spectrum Plots from Two Systems with no Core

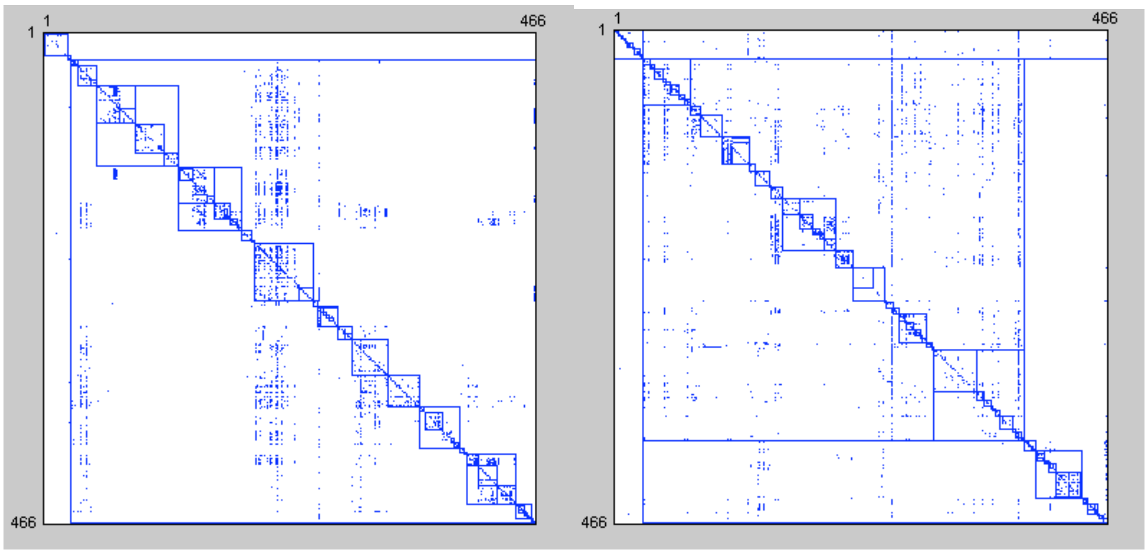


In the first system (Gnucash 1.8.4) we observe components with high levels of FIV and components with high levels of FOV, but none that possess both attributes. Furthermore, there are no "trapping states" visible for either measure (i.e., flat lines which indicate that many components share the same level of visibility). By contrast, in the second system (Open Office 1.0.0) we note the presence of multiple trapping states for both measures and observe that several of these states are aligned across the two. This indicates the presence of several distinct groups of components, within which each has the same level of visibility. In Open Office, four distinct groupings are found, each of which relates to a specific subsystem in the design (the components are located in the same directory, as indicated by their proximity in the spectrum plot). We note that this system is the only one in the sample that consists of a collection of relatively independent applications (e.g., a spreadsheet and word processor) each of which that represent systems in their own right. This may be a factor in explaining why we observe a multi-core structure in this system, but not in any others in our sample.

The Detection of Core-periphery Structures

It is natural to ask whether the presence of a core-periphery structure (or the lack thereof) can be detected unambiguously from the summary statistics for a system, or from inspection of the direct dependencies in the Design Structure Matrix. To answer this question, we compared those systems that possessed a Core-periphery structure with those that did not, focusing on differences in both the quantitative data that describes them as well as the visual plots of their system architectures. We found no variable that could reliably predict whether a system possessed a Core-periphery structure, and no consistent pattern of dependencies in the Design Structure Matrix that would signal the presence of such a structure. To illustrate, **Figure 8** below shows the Architectural View of two systems, chosen because they are of similar size and dependency density, and possess similar Design Structure Matrices. Analysis reveals that the system on the left has no core, while the one on the right has a large core comprising 37% of the system. This indicates the role played by *indirect* dependencies, which dictate the differences in structure in these systems. We conclude that detecting the presence of a Core-periphery structure cannot be achieved solely by examining the direct dependencies for a system, but requires an assessment of all potential paths through which dependencies propagate.

Figure 8: Comparison of two Similar Systems with Different Design Structures



5.2 The Size of the Core across Different Systems

Our second topic focuses on the size of the core across systems, in terms of both the number of core components, as well as the proportion of the system that this represents. In particular, we conduct four different types of analysis: first, comparing core sizes across systems of similar size; second, comparing core sizes across systems of similar function; third, comparing core sizes across systems of similar size *and* function; and fourth, examining the relationship between core size and system size for all systems. We find that core size varies widely across systems. However, for large systems there appears to be a limit, with respect to the proportion of a system the core represents.

Our first analysis focuses on differences in core size across systems of similar size. **Table 3** below shows the number of core components for all distinct applications in our sample with a total size between 300 and 400 source files.[7] The size of the core for the six resulting systems varies between 38 and 209 components. On a proportionate basis, this represents from 12% to 64% of a system. This indicates that there are substantial variations in the number of core components across systems of similar size.

Table 3: The Size of the Core for Systems of Similar Size

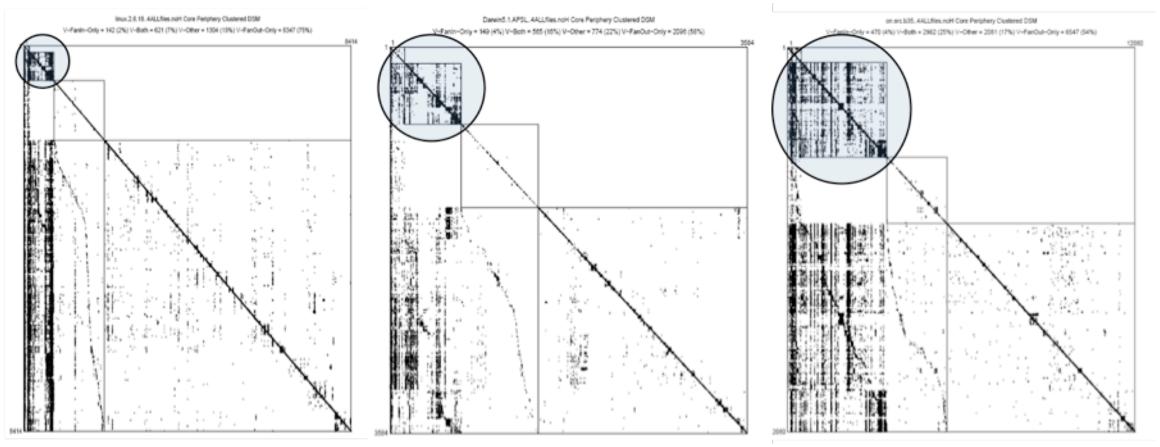
Software	System Size	Number of Core Files	Core as a % of System
MyBooks	322	207	64.30%
PostGresql	333	207	62.2%
Berekeley DB	305	140	45.9%
GnuMeric	311	141	45.3%
Linux	301	39	13.0%
Apache	300	38	12.7%

Our second analysis focuses on the differences in core size across systems that perform similar functions. **Figure 9** below shows the core-periphery view of three different operating systems: Linux, Open Solaris, and Darwin (the platform upon which Apple's OSX software is based). The contrasts are striking. Linux has the smallest core, with 621 components, representing 7% of the system. By contrast, Open Solaris has an extremely large core, with 2982 components, accounting for 25% of the system. Darwin

[7] Where multiple versions of a product exist in this range, we choose the version closest to 350 source files.

falls between these observations, with a core of 565 components, representing 16% of the system. This indicates there are substantial variations in the size of the core across systems of similar size, both on an absolute and a relative basis.

Figure 9: The Size of the Core for Systems of Similar Function



Linux Core = 7%   Darwin Core = 16%   Solaris Core = 25%

Our third analysis focuses on exploring one of the possible drivers of differences in core size – the type of organization that develops a system. In particular, we build on prior work that argues product designs tend to reflect the design of the organizations in which they are conceived, an effect known both as Conway's Law and the "Mirroring Phenomenon" (Conway, 1968; Henderson and Clark, 1992). This theory suggests that organizations with a large core of tightly coupled developers would produce systems with a larger core. In contrast, organizations in which most developers were only loosely coupled or "peripheral" contributors would produce systems with a smaller core.

To evaluate this argument, we compare designs that emerge from different types of organization, specifically, open source (distributed) versus closed source (collocated) development. Given the need to control for differences in design that are driven by the differing functions a system must perform, we use a matched-pair design, comparing the size of the core only between systems of similar size and function. **Table 4** contains a comparison of the size of the core for five matched-pairs, each of which consists of one "open" and one "closed" system. The sample is drawn from a prior study exploring differences in the mean level of coupling between products (MacCormack et al, 2007b).

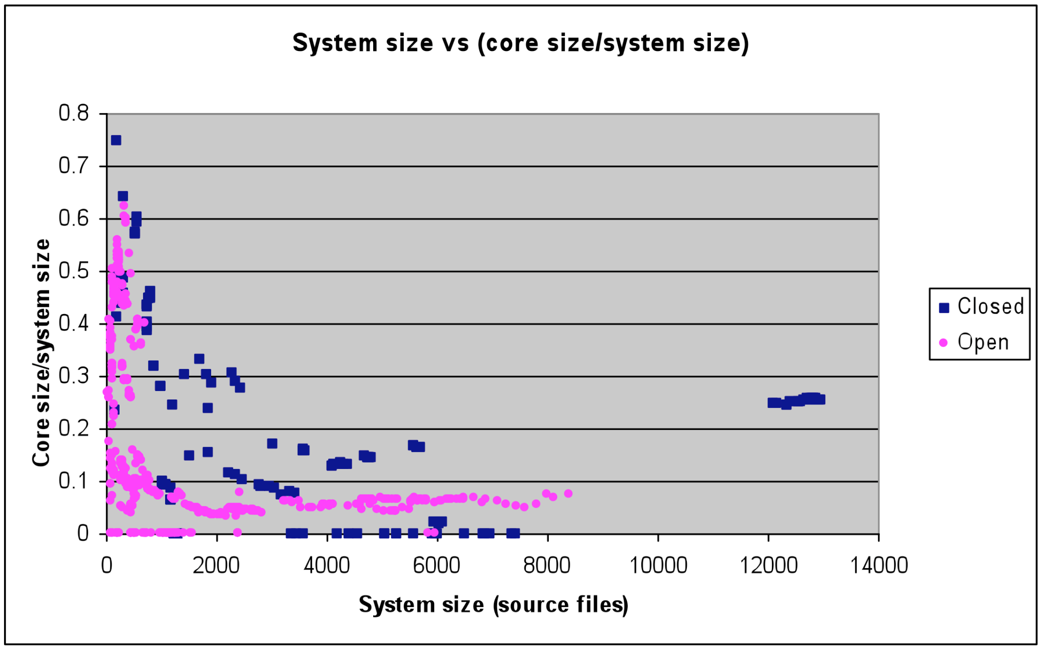
We find that in every case, the open system has a smaller core than the closed system. Remarkably, in three of the five comparisons, the open source system has no core components whatsoever. We conclude that differences in core size may be driven, in part, by differences in the structure of the developing organization.

Table 4: The Size of the Core for a Sample of Matched-Pair Products

Application Category	**Open Source Product**		**Closed (Commercial) Product**	
	System Size	**Core Components**	**System Size**	**Core Components**
Financial Mgmt	466	0.0%	471	69.6%
Word Processor	841	0.0%	790	46.1%
Spreadsheet	450	25.8%	532	57.3%
Operating System	984	7.3%	992	24.7%
Database	465	0.0%	344	48.8%

Our fourth analysis explores the relationship between core size and system size. Given the size of the core is constrained by the size of the system, the analysis focuses on the size of the core in relative terms – that is, the percentage of the system it represents. **Figure 10** below shows a plot of core size against system size measured in source files. The graph differentiates between systems that originate in open source projects, and those that originate from commercial efforts, given we know from the results described above that different organizational forms appear to yield designs with different structures.

Figure 10: The Size of the Core versus Total System Size



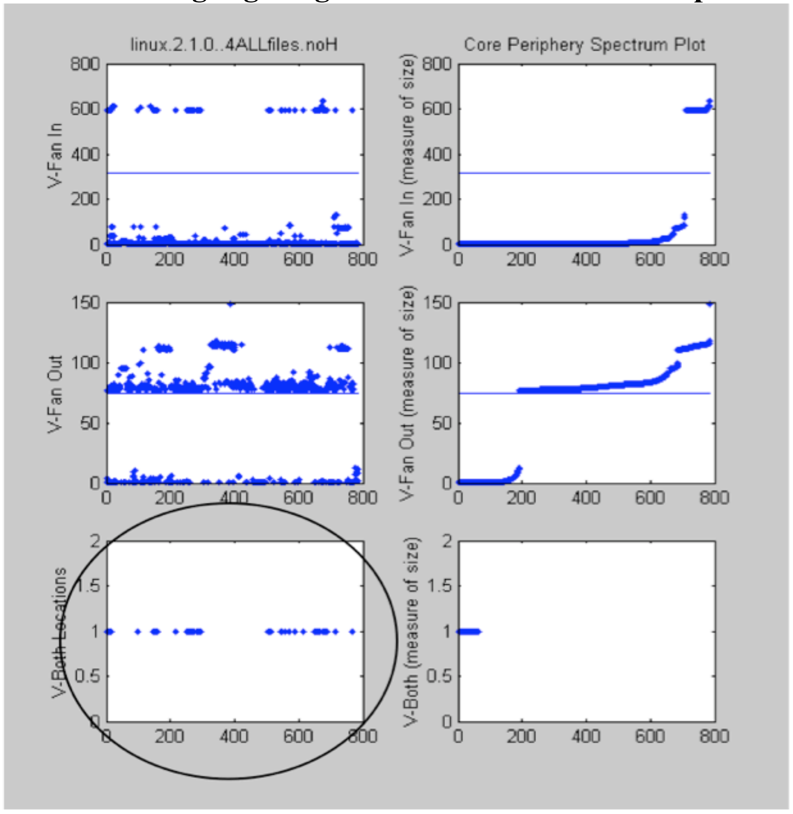
The graph reveals that for very small systems, the size of the core varies substantially, from a minimum of zero to a maximum of 75% of the system. For larger systems however, the maximum observed core size declines. In essence, there appears to be a negative exponential relationship between maximum core size and total system size. For systems that exceed 3,000 source files, there are no cores that exceed 30% of the system. Intuitively, this pattern makes sense. For small systems, having a proportionately large core is not a problem, given it is small enough in absolute terms that the architect can still understand all possible component interactions. In larger systems however, even a moderately large core will quickly bring cognitive challenges, given the need to understand many possible interactions (e.g., in a system of 3,000 source files, a core of 10% would result in the need to understand the interactions between 300 components). In such systems, there is a need to partition the system into smaller pieces, allowing work to proceed in parallel, while minimizing the risk of problematic interdependencies.

5.2 The Location of Core Components in a System

Our third topic focuses on the location of core components. In particular, we examine whether these components tend to be located in a few core subsystems or conversely, are distributed throughout a system. Given the information-based nature of software, there is no need for core components to be "physically" connected to each other, raising the possibility that they can be distributed throughout a system without penalty. However, from the perspective of the system architect (or maintainer) there are benefits to locating core components in a small number of subsystems. In software, this would imply core components are located in a small number of directories.

To explore this issue, we examine the spectrum plots for sample systems, to evaluate the location of core components. Surprisingly, we find that in the majority of systems, core components are *not* located in a small number of distinct directories, but instead are distributed throughout the system. To illustrate, **Figure 11** shows the spectrum plot for Linux 2.1.0. This system possesses 65 core components. However, rather than being concentrated in one directory, they are spread across a number of different directories. In fact, 7 of the 10 top-level directories in this system contain at least one core component.

Figure 11: Spectrum Plot Highlighting the Location of Core Components



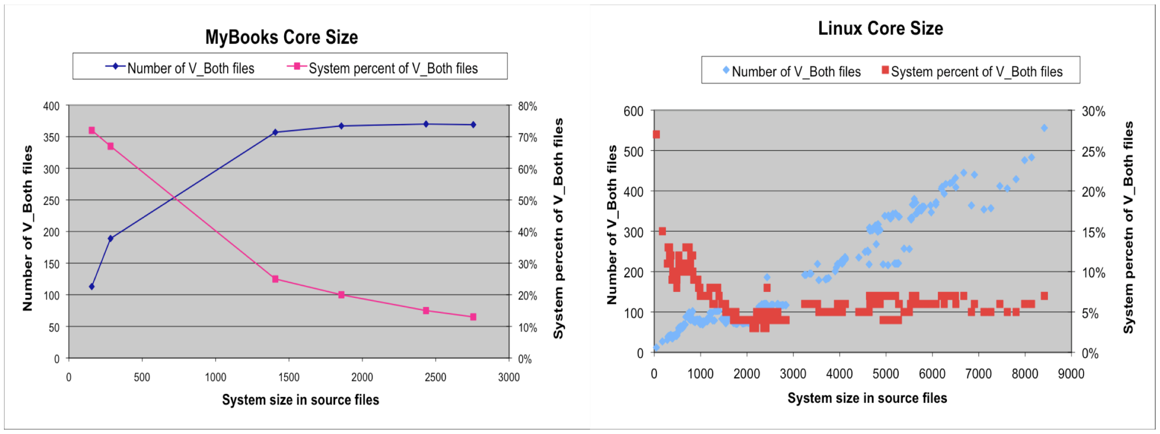
The implications of this finding are important, in that for many systems, it is not clear which components are core, and which are peripheral. Combined with earlier results, which highlight the difficulty of detecting whether a system possesses a core-periphery structure from the first-order dependency structure, they illustrate the cognitive challenge facing a system architect. This issue is especially pertinent to situations in which a legacy system must be maintained or adapted with limited access to documentation or to staff familiar with its design. In such circumstances, an inspection of the subsystem structure will *not* be sufficient to reveal where core components are located. It is only through a detailed analysis of the chain of direct and indirect dependencies with the potential to propagate throughout the system that this structure can be made visible.

5.4 The Evolution of System Structure

Our fourth analysis focuses on the evolution of a system's structure over time. In contrast to the analyses above, which focuses on the relationship between core size and system size *across* systems, the focus here is on the relationship between core size and system size *within* a specific system over time. In particular, we examine the evolution of the 13 distinct applications for which we had access to enough versions to detect a trend. We find that these systems obey one of two patterns with respect to how the core evolves. The first is a "stable" core, in which the core remains constant in size regardless of how large the system becomes. The second is a "growing" core, in which the core increases at a similar rate to the system, meaning that its size relative to the system remains constant.

We illustrate these dynamics in **Figure 12** below, using data from two systems. In the first, note the size of the core increases over the first three versions of the software, then plateaus at around 360 components. As the system grows thereafter, few if any core components are added, hence the core declines as a proportion of the whole system. By comparison, the second chart shows a system in which the core grows over time, mirroring increases in the size of the system as a whole. As a consequence, the fraction of the system represented by the core remains stable at between 5 to 6 percent.

Figure 12: Two Different Patterns in the Evolution of the Core



All the systems in our sample that possess a core-periphery structure evolve in a way that falls between the patterns described above, with 11 of the 13 tending towards growth. It is notable that in general, as systems evolve, there are few departures from the trend once established. However, we do note several outliers in terms of behavior, which yield additional information about the development paths that a system can take, as well as the impact of managerial actions that seek to influence this path. We discuss these below.

Discontinuities in the Evolution of Core Components

It is notable that all of the systems that possess no core components in their earliest versions continue to have no core components in their latest versions. Conversely, we only observe two examples of systems that initially possess a core-periphery structure, but evolve in a way that this structure subsequently disappears. We conclude that the initial decisions that lead to the existence (or absence) of a core-periphery structure are "sticky" in nature, and difficult to change without substantial redesign efforts.

In two of the systems in our sample we note discontinuities in the size of the core, relative to the pattern that would be expected as a system grows. The first is illustrated in **Table 5**, which shows the size of the core in Linux across all several major releases. We see that in release 2.4.0, the core jumps in size to over 200 components from 75 in the prior version. This is explained in part, by the increase in size of the system, from 1946 to 3243 components. However, the increase in core size (181%) is greater than that of the system (67%) meaning the core grows proportionately from 4% to 7% of the whole.

We conclude that the changes introduced in version 2.4.0 had the effect of substantially increasing the number of tightly coupled components in the system.[8]

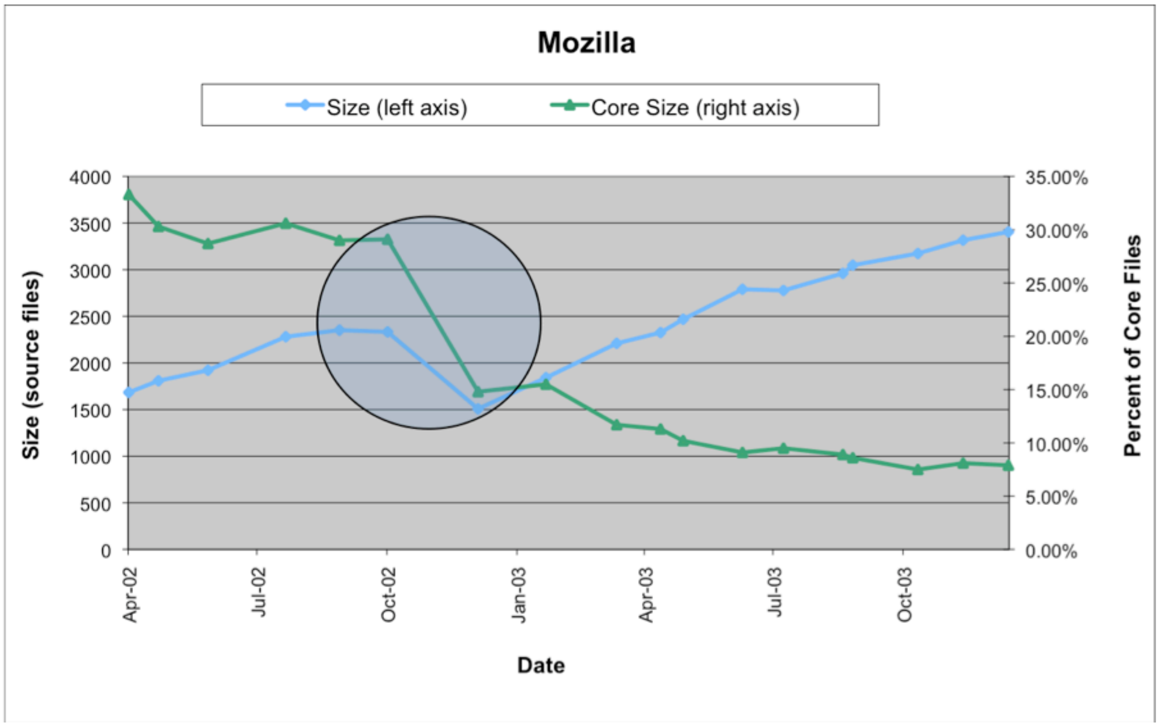
Table 5: The Size of the Core in Linux across all Major Versions

Version	System Size	Core Size	Core as a % of System
1.0	282	31	11.0%
1.1.0	282	31	11.0%
1.2.0	400	41	10.3%
1.3.0	431	42	9.7%
2.0	779	67	8.6%
2.1.0	785	65	8.3%
2.2.0	1891	76	4.0%
2.3.0	*1946*	*72*	*3.7%*
2.4.0	*3243*	*197*	*6.1%*
2.5.0	4047	216	5.3%
2.6.0	6194	407	6.6%

The second discontinuity noted with respect to how the size of the core changes as a system grows is from the evolution of the Mozilla web browser. **Figure 13** shows a plot of Mozilla's core size over time, expressed as a proportion of the system. It reveals a significant decline in the size of the core in the fall of 1998. Prior to this discontinuity, the core comprised 33% of the entire system; subsequently, it consumed less than 15%. Clearly, there was a significant change in the design between these two versions, which had the effect of dramatically reducing the number of tightly coupled components.

[8] One explanation relates to the increasing amount of code contributed by major corporations such as IBM, as Linux gained support in the software industry. Given this code was developed via a different model of

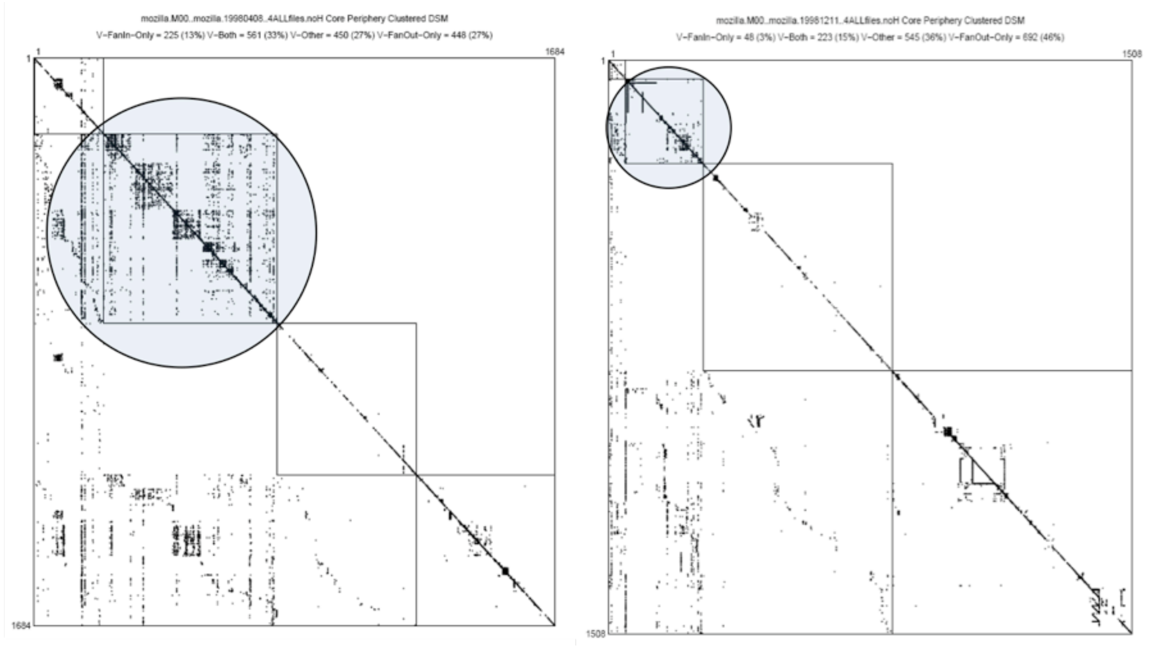
Figure 13: The Size of the Core in Mozilla over Time



We know from prior work that the change in Mozilla's design observed in the fall of 1998 was the result of a *purposeful* redesign effort, which had the explicit objective of making the codebase more modular, and hence easier for contributors to work with (MacCormack et al, 2006). This was important given the software had just been released into the open source domain, and was not originally structured to support the type of distributed, parallel development that is the norm in open source projects. Achieving this objective required decreasing the level of coupling between components, thereby facilitating the move of some components from the core to the periphery of the system. **Figure 14** shows Mozilla's design prior to and after this process, displayed in using the Core-periphery view. The transformation is striking. Prior to the redesign, the core contains 561 components; after the redesign, it contains only 223 components.

organization, it may have been different in nature from that which was typical of prior contributions.

Figure 14: Mozilla's Core-periphery Structure before and after a Redesign


mozilla.M00..mozilla.19980408..4ALLfiles.noH Core Periphery Clustered DSM
V-FanIn-Only = 225 (13%) V-Both = 561 (33%) V-Other = 450 (27%) V-FanOut-Only = 448 (27%)
1
1684
1
1684
mozilla.M00..mozilla.19981211..4ALLfiles.noH Core Periphery Clustered DSM
V-FanIn-Only = 48 (3%) V-Both = 223 (15%) V-Other = 545 (36%) V-FanOut-Only = 692 (46%)
1
1508
1
1508

6. Discussion

In this paper, we developed methods to detect the core components in a complex system, establish whether these systems possess a core-periphery structure, and measure important elements of these structures. Our results complement the wealth of theoretical papers published on system design and architecture. The findings represent a first step in establishing some stylized facts about the structure of real world systems.

We find that the majority of systems in our sample – 75% to 80% – possess a core-periphery structure. However, it is significant that a substantial number of systems *lack* such a structure. This implies that a considerable amount of managerial discretion exists when choosing the "best" architecture for a system. Such a conclusion is supported by the large variations we observe with respect to the characteristics of such systems. In particular, there are major differences in the number of core components across a range of systems of similar size and function, indicating that the differences in design are not driven solely by system requirements. These differences appear to be driven instead, by the characteristics of the organization within which system development occurs.

The large variation in the number of core components across systems in this industry, suggests that at this level, there is no Dominant Design of software products. Given the great diversity of functions that software performs however, this is hardly a big surprise. A more relevant comparison is the variation within the same category of application. Yet here again, we find tremendous variety in the structure of systems, from those that possess no core components, to those in which half the system is contained in the core.

This lack of Dominant Designs (as defined by a consistent pattern of choices with respect to the selection and design of core components) may be driven by several factors. First, software is an information-based product with no physical manifestation, so the structure of systems may be more "malleable" than that for other types of product. Second, the software industry is relatively young in comparison to sectors such as automobiles and airplanes; hence it is possible standardized designs have not yet emerged. Finally, the source code in commercial software products is typically unobservable and protected by patent and copyright laws.[9] Given the difficulty in reverse-engineering the underlying design for software systems, it may therefore be more difficult for standards to evolve and become widely adopted by many industry players. [10]

These are all good arguments for why no Dominant Designs are observed in the software industry, even within the same category of applications. However, our findings also support a different and potentially more important conclusion. Specifically, we find evidence that variations in system structure can be explained, in part, by the different models of development used to develop systems. That is, product structures "mirror" the structure of their parent organizations (Henderson and Clark, 1990). This result is consistent with work that argues designs (including Dominant Designs) are not necessarily optimal technical solutions to customer requirements, but rather are driven more by social and political processes operating within firms and across industries (Noble, 1984; David, 1985; Tushman and Rosenkopf, 1992; Tushman and Murmann, 1998; Garud et al, 2002).

Our findings highlight the cognitive difficulties that face a system architect. In particular, we find no discernible pattern of direct dependencies that can reliably predict

[9] Most software is distributed in "binary" form, meaning that the actual code is not visible to the user.

whether a system has a core, and if it does, how large it is. In essence, system structure is driven to a large extent by the *indirect* dependencies between components, which are much harder for an analyst to understand. Consider that most developers have a good grasp of the dependencies they must manage between their component(s) and others, but only a limited knowledge of the ways in which other components, in turn, are connected. This challenge is magnified by the fact that many development tools highlight only direct dependencies, providing no way to analyze the propagation of changes via indirect paths.

This problem is compounded by the fact that in many systems, the core components are not located in a small number of subsystems, but are distributed throughout the system. A system architect therefore has to identify where to focus attention. It is not simply a matter of concentrating on subsystems that appear to contain most of the core components. Important relationships may exist between these components and others within subsystems that, on the surface, appear relatively insignificant. This highlights the need to understand patterns of coupling at the component level, and not to assume that all the key relationships in a complex system are located in a few key subsystems.

These issues are especially pertinent to the context of software, given that legacy code is rarely re-written, but instead forms a platform upon which new systems are built. With such an approach, today's developers bear the consequences of design decisions made long ago. Unfortunately, the first designers of a system often have different objectives from those that follow, especially if the system is successful and therefore long lasting. While early designers may place a premium on speed and performance, later designers may value reliability and maintainability. Rarely can all these objectives be met by the same design. A different problem stems from the fact that the early designers of a system may no longer be around when important structural design choices need revisiting. This difficulty is compounded by the fact that designers rarely document design choices well, often requiring the structure to be recovered merely from inspection of the source code.

Several limitations of our study must be considered in assessing the generalizability of 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

[10] While many of the systems in our sample are in the open source domain, a significant number started life as commercial products, hence their design was initially proprietary and unobservable.

the results would be replicated for physical products remains an important empirical question. Second, given the difficulty in obtaining proprietary software, we adopt a non-random sample of systems for which we have access to the source code. While we limit our enquiry to successful systems with thousands of user deployments, we cannot be sure that the overall results are representative of the industry. Finally, our findings may be sensitive to the thresholds used in determining the core and peripheral components in a system. We note however, that tests of different assumptions with respect to these thresholds shows that the overall patterns of variation in the data are consistent and robust, even as the specific results of each analysis may change.

Our work opens up a number of avenues for future study, given we have developed methods to identify and track the core components in a system over time. For example, prior work suggests that exogenous technological "shocks" in an industry can cause major dislocations in the design of systems and change the competitive dynamics. This assertion could be tested, by examining the impact of major technological transitions in this industry (e.g., the rise of object-oriented programming languages and the World-Wide-Web) on the design and survival of both software products and their producing firms (e.g., see MacCormack and Iansiti, 2009). Further work might explore, in greater detail, the association we find between product and organizational designs. Such work is facilitated by the fact that software development tools typically assign an author to each component in the design. As a consequence, it is possible to understand who is developing core components, to analyze their social networks and to indentify whether the organizational network as a whole predicts future product structure. Ultimately, this agenda promises to deepen our knowledge of the structures underlying complex technological systems. It will also improve our ability to understand the ways in which a manager can shape and influence the future evolution of these systems.

APPENDIX A: LIST OF SYSTEMS ANALYZED

System Name	**Number of Versions**
Abiword	29
Apache	52
Berkeley DB	12
Chrome	1
Calc (Open Office)	6
Darwin	36
Ghostscript	35
GnuCash	106
GnuMeric	162
Linux	544
Mozilla	35
MyBooks	5
MySQL	18
OpenAFS	106
Open Office	6
Open Solaris	28
PostGres	46
Write (Open Office)	6
XNU	43

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.

David, Paul A. (1985) "Clio and the Economics of QWERTY," American Economic Review 75(2):332-337

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

Garud, Raghu, Sanjay Jain and Arun Kumaraswamy (2002) "Institutional Entrepreneurship in the Sponsorship of Technological Standards: The Case of Sun Microsystems and Java," Academy of Management Journal, 45(1):196-214

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 University 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.

Luo, Jianxi, Daniel E. Whitney, Carliss Y. Baldwin and Christopher L. Magee (2009) "Measuring and Understanding Hierarchy as an Architectural Element in Industry Sectors," Harvard Business School Working Paper 09-144.

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

Noble, David F. (1984) Forces of Production: A Social History of Industrial Automation, Oxford: Oxford University Press

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

Utterback, James 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