# Hidden Structure: Using Network Methods to Map Product Architecture

HARVARD | BUSINESS | SCHOOL

# Hidden Structure: Using Network Methods to Map Product Architecture

**Carliss Y. Baldwin**
**Alan MacCormack**
**John Rusnak**

## Working Paper

**13-093**

**May 1, 2013**

# Hidden Structure: Using Network Methods to Map Product Architecture

Carliss Y. Baldwin, Alan MacCormack and John Rusnak

## Abstract

In this paper, we describe an operational methodology for characterising the architecture of technical systems and demonstrate its application to a large sample of software releases. Our methodology is based upon network graphs, and allows us to identify define three fundamental architectural patterns, which we call core-periphery, multi-core, and hierarchical. We apply our methodology to a sample of 1,286 software releases from 17 applications, and find that 70 – 80% of these systems possess a "core-periphery" architecture under our classification scheme. This type of architecture is characterized by having a single dominant cyclic group (the Core) that is large relative to other cyclic groups and above a threshold with respect to system size. We find that the size of 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 tend to develop systems with smaller Cores, while closed collocated organizations tend to develop systems with larger Cores. Our findings represent a first step in establishing some "stylized facts" about the fine-grained structure of large, real-world technical systems.

**JEL Classification:** D23, L22, L23, M11, O31, O34, P13

## 1. Introduction

All complex systems can be described in terms of their architecture, that is, as a nested hierarchy of subsystems (Simon, 1962). Critically, however, not all subsystems in an architecture are of equal importance. In particular, some subsystems are "core" to system performance, whereas others are only "peripheral" (Tushman and Rosenkopf, 1992). Core subsystems have been defined as those that are tightly coupled to other subsystems, whereas peripheral subsystems tend to possess only loose connections to other subsystems (Tushman and Murmann, 1998). Studies of technological innovation consistently show that major changes in core subsystems as well as their linkages to other parts of the system can have a significant impact on firm performance as well as industry structure (Henderson and Clark, 1990; Christensen, 1997, Baldwin and Clark, 2000). Despite a wealth of research highlighting the importance of understanding system architecture however, there is little empirical evidence on the actual architectural patterns observed across large numbers of real world systems.

In this paper, we propose an operational methodology for analyzing the design of complex technical systems and apply it to a large (though non-random) sample of systems in the software industry. Our objective is to understand the extent to which such systems possess a "core-periphery" structure, as well as the degree of heterogeneity within and across their architectures. We also seek to explore how systems evolve over time, since prior work has shown that significant changes in product architecture can create major challenges for firms and precipitate changes in industry structure (Henderson and Clark, 1990; Tushman and Rosenkopf, 1992; Tushman and Murmann, 1998; Baldwin and Clark, 2000; Fixson and Park, 2008).

The paper makes a distinct contribution to the literatures of both technology management and system design and analysis. In particular, we first describe an operational methodology based on network graphs that can be used to characterize the architecture of large technical systems.[1] We then demonstrate the application of this methodology to a sample of 1,286 software releases from

[1] We define a large system as one having in excess of 300 interacting elements or components.

17 distinct applications. We find that 70-80% of releases possess a core-periphery structure under our classification scheme (described below). However, the size of the Core (defined as the percentage of components in the largest cyclic group), varies widely, even for systems that perform the same function. These differences appear to be associated with different models of development – open, distributed organizations tend to develop systems with smaller Cores, while closed, collocated organizations tend to develop systems with larger Cores. We find the Core components in these systems are often dispersed across different "modules" rather than being concentrated in one or two, making their detection and management difficult for the system architect. Finally, we show that these systems evolve in different ways: some undergo continuous change, while others display discrete jumps. Our findings represent a first step in establishing some "stylized facts" about the fine-grained structure of large, real-world technical systems.

The paper is organized as follows. In the next section, we briefly review the relevant literature on dominant designs, core-periphery systems, product architecture, and network methods for characterizing different architectures. The section following describes our methodology for analyzing and classifying architectures based upon the level of direct and indirect coupling between elements. Next, we describe the results of our empirical investigation of software systems. We conclude by describing the limitations of our method, discussing the implications of our findings for scholars and managers, and identifying questions that merit further investigation.

## 2. Literature Review

In his seminal paper "The Architecture of Complexity," Herbert Simon argued that the architecture of a system, that is, the way the components fit together and interact, is the primary determinant of the system's ability to adapt to environmental shocks and to evolve toward higher levels of functionality (Simon, 1962). However, Simon and others presumed that the architecture of a complex system would be easily discernible. Unfortunately this is not always the case.

Especially in non-physical systems, such as software and services, the structure that appears on the surface and the "hidden" structure that affects adaptation and evolvability may be very different.

The design of a complex technological system (a product or process) has been shown to comprise a nested hierarchy of design *decisions* (Marple, 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 higher-level decisions influence many subsequent design choices, hence are referred to as "core concepts." For example, in developing a new automobile, the choice between an internal combustion engine and electric propulsion represents a core concept that will influence many subsequent decisions about the design. In contrast, the choice of leather versus upholstered seats typically has little bearing on important system-level choices, hence can be viewed as peripheral.

A variety of studies show that a particular set of core concepts can become embedded in an industry, forming a "dominant design" that sets the agenda for subsequent technical progress (Utterback, 1996; Utterback and Suarez, 1991; Suarez and Utterback, 1995). Dominant designs have been observed in 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 of the dominant design experience increased market share and profits.

Much scholarly work has focused on understanding what constitutes a dominant design and why specific designs become dominant (see Murmann and Frenken, 2006, for a review). Despite the wealth of studies however, the concept has proved difficult to pin down empirically. Scholars often differ on what constitutes a dominant design and whether this phenomenon is an antecedent or a consequence of changing industry structure (Klepper, 1996; Tushman and Murmann, 1998; Murmann and Frenken, 2006). For example, Klepper (1996) argues that the concept is tautological: dominant designs are those that succeed and cannot be identified *ex-ante*.

Tushman and Murmann (1998) show that advances in airplane design are not easily classified in terms of a dominant design, but rather were shaped by design changes at the subsystem level. Finally, Murmann and Frenken (2006) find that the past literature has been inconsistent with regard to the unit of analysis used to define a dominant design.

Murmann and Frenken (2006) suggest that the concept of dominant design can be made more concrete by classifying components (and decisions) according to their "pleiotropy." The pleiotropy of a component is the number of functions affected by it, that is, "the number of service characteristics that will change their value when this component in the system is changed" (p. 941). High-pleiotropy components affect many things that the product does, while low-pleitropy components affect only a few. A dominant design, they argue, is an interdependent set of high-pleiotropy components. By definition, these components cannot be changed without inducing widespread changes throughout the system, some of which are likely to hamper performance or even cause the system to fail. For this reason, the authors argue, the designs of high-pleiotropy components are likely to remain unchanged for long periods of time: such stability is the defining property of a dominant design. The authors define the high-pleiotropy components as the "core" of the system, and the remainder as the "periphery."

In sum, dominant design theory argues that the *hierarchy* of decisions (and components that embody those decisions) is an important dimension of product architecture. At the top of the design hierarchy are components whose properties cannot change without requiring changes to many other parts of the system; at the bottom are components that do not trigger widespread or cascading changes. Thus any methodology for discovering the hidden structure of a complex system must reveal something about the hierarchy of components and related design decisions.

In contrast to dominant design theory, where design decisions are hierarchically ordered, , some design decisions may, in fact, be mutually interdependent. For example, if components *A*, *B*, *C*, and *D* must all fit into a limited physical space, then any increase in the dimensions of one will reduce the space available to the others. The designers of such components face what is called

reciprocal interdependence (Thompson, 1967). If they make their initial choices independently, then those decisions must be communicated to the other designers, who may need to change their own original choices accordingly. This second-round of decisions in turn may trigger a third set of changes, with the process continuing until the designers converge on a set of decisions that satisfies the global constraint (in this case, space). Reciprocal interdependency thus gives rise to feedback and "cycling" in a design process. Such cycles are a major cause of rework, delay, and cost overruns (Steward, 1981; Eppinger et al, 1994; Sosa, Mihm and Browning, forthcoming). Thus any methodology for discovering the hidden structure of a complex system must reveal not only something about the hierarchy of components and related design decisions but also the presence of reciprocal interdependence and cycles between components/decisions.

Studies that attempt to characterize the architecture of complex systems often employ network representations and metrics (Holland, 1992, Kaufman, 1993, Rivkin, 2000, Braha et. al., 2006, Rivkin and Siggelkow, 2007 Barabasi, 2009). Specifically, they focus on identifying the linkages that exist between different elements (nodes) in a system (Simon, 1962; Alexander, 1964). A key concept in this field is modularity, which refers to the way that a system's architecture is decomposed into different parts or modules. While there are many definitions of modularity, authors tend to agree on the features that lie at its heart: the interdependence of design decisions within modules, the independence of design decisions between modules, and the hierarchical dependence of modules on components embodying standards and design rules (Mead and Conway, 1980; Baldwin and Clark, 2000; Schilling, 2000). The costs and benefits of modularity have been studied in a stream of research that explores its impact on product line architecture (Sanderson and Uzumeri, 1995), manufacturing (Ulrich, 1995), process design (MacCormack, 2001) process improvement (Spear and Bowen, 1999) and industry evolution (Langlois and Robertson, 1992; Baldwin and Clark, 2000, Fixson and Park, 2008) among other areas.

Studies that use network methods to measure modularity typically focus on capturing the

level of coupling (i.e., dependency or linkage) that exists between different parts of a system. In this respect, one of the most widely adopted techniques is the Design Structure Matrix or DSM. A DSM displays the network structure of a complex system in terms of a square matrix (Steward, 1981; Eppinger et al, 1994; Sharman, Yassine and Carlile, 2002; Sosa et al, 2004, 2007; MacCormack et al, 2006, 2012), where rows and columns represent components (nodes in the network) and off-diagonal elements represent dependencies (links) between components. Metrics that capture the level of coupling for each component can be calculated from a DSM and used to analyze and understand system structure. For example, MacCormack, Rusnak and Baldwin (2006) and LaMantia et. al. (2006) use DSMs and the metric "propagation cost" (described below) to compare software architectures before and after architectural redesigns. 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. Wilkie and Kitchenham (2000) and Sosa et. al. (forthcoming) show that higher levels of component coupling are associated with more frequent changes and higher defect levels. And MacCormack et al (2012) show that the mean level of coupling varies widely across similar systems, the differences being explained, in part, by differences in the way system development is organized.

These and other studies suggest that network methods can be used to evaluate system architecture, as well as changes aimed at making systems easier to upgrade and maintain. In the next section, we describe a methodology based on DSMs that reveals both the hierarchical ordering of components and the presence of cyclic groups within a large network. We use this methodology to analyze a large sample of software releases. Our analysis reveals both surprising similarities in the high-level architecture of many systems plus heterogeneity in the specific details that suggest a high degree of designer discretion, with high potential impact on performance.

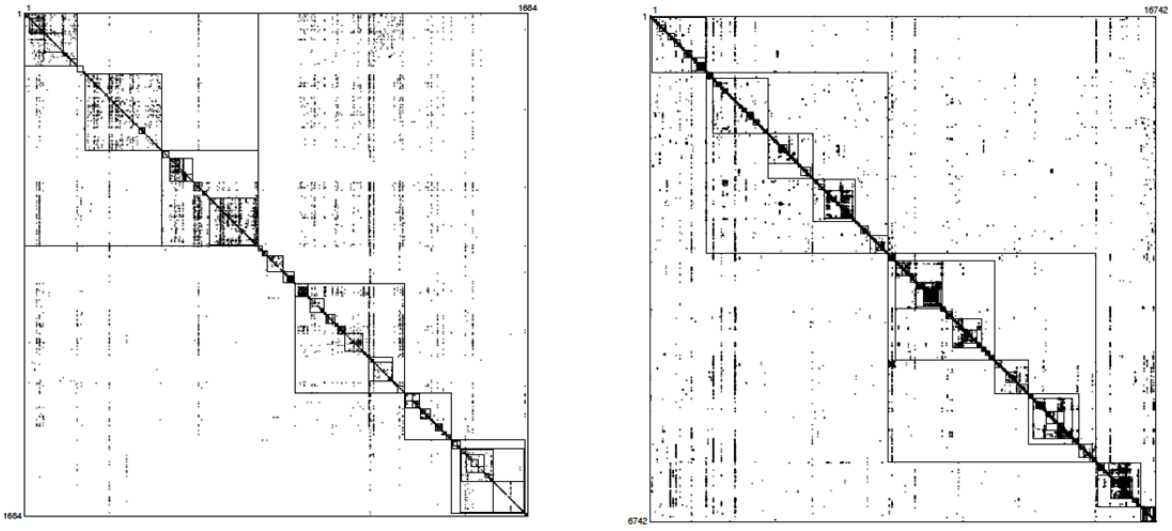
## 3. Methodology

In this section, we describe a systematic approach to determining the hidden structure of large, complex systems. Specifically, after identifying the dependencies between elements, we analyze the system in terms of the hierarchical ordering of elements and the presence of cycles between them. We then classify elements in terms of their position in the resulting network.

Two examples from our dataset serve to motivate the problem and our method of analysis. **Figure 1** shows the structure of two codebases in the form of Design Structure Matrices. Here each diagonal cell represents a component (node), and dependencies between components (links) are recorded in the off-diagonal cells. In this example, the components are software files and the dependencies denote relationships between the functions and procedures in each file (i.e., function calls, class method calls, class method definitions, and subclass definitions). In this example, if file *i* depends on file *j*, a mark is placed in the row of *i* and the column of *j*.

**Figure 1: The Network Structure of Two Codebases—Architect's View**

**Codebase A** **Codebase B**

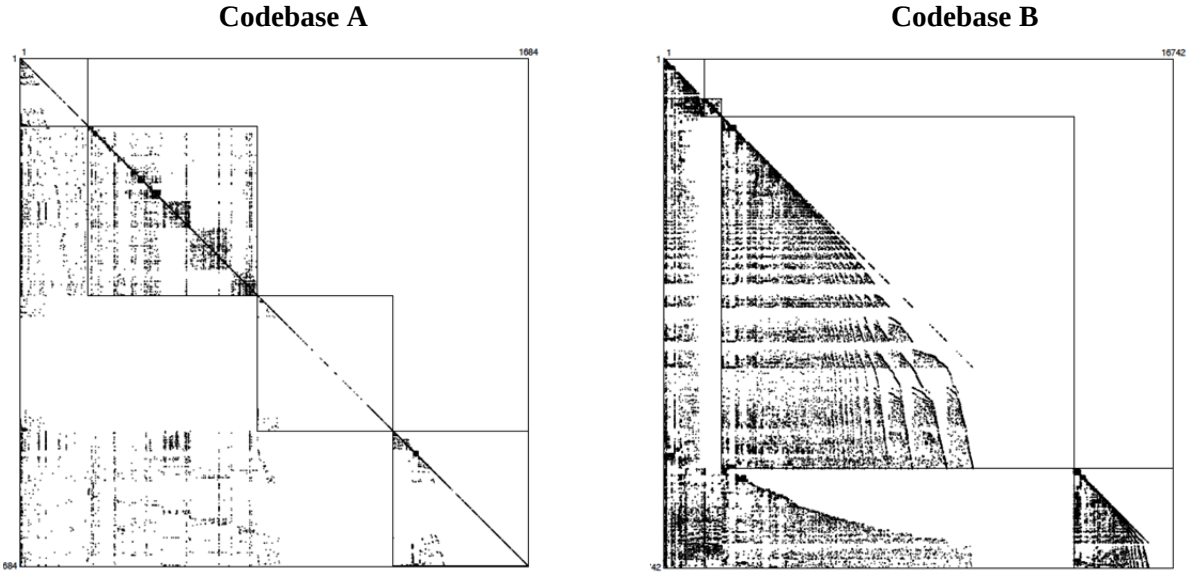



Codebase A is an early version of the Mozilla Application Suite, an early browser

program designed by Marc Andreesen at Netscape. Codebase B is a large commercial system. **Figure 1** shows what we call the "architect's view" of these systems. In software systems, each file has a unique directory path and filename that places it within a set of nested directories. In the figure, the nested directory structure is indicated by the boxes-within-boxes in the matrices. The directory structure is determined by the system's architects and reflects both progarmamming conventions and the designers' intuition as to which functions and files "belong together."

From an architect's view, it is difficult to say whether these codebases exhibit significant differences in terms of system structure. Standard software coupling metrics also do not provide much guidance. For example, according to Chidamber and Kemerer's (1994) coupling metric, a measure often used in software engineering, Codebase A has a coupling level of of 5.39, while Codebase B has a coupling level of 4.86. In contrast, in **Figure 2** we rearrange the components of each system in a way that minimizes the number of dependencies above the diagonal. Dependencies that remain above the diagonal reveal the presence of cyclic interdependencies –A depends on B, and B depends on A – which cannot be reduced to a hierarchical ordering.

**Figure 2: The Network Structure of Two Codebases—Core-Periphery View**



This approach to rearranging components reveals signficant differences in the underlying structure of the two systems. Specifically, Codebase A has a large cyclic group of files, which appear in the second block down the main diagonal. Each component in this group both depends on and is depended on by every other member of this group. These "Core" files account for 33% of the files in the system. Furthermore, the Core, the components depending on it, and those that it depends upon, account for 73% of the system. The remainder of the files in this system are "Peripheral" in that they have few relationships with other files.

Note that we refer to cyclic groups of any size as the "cores" of the system and hence use the terms "cyclic group" and "core" interchangeably. The *largest* cyclic group however, plays a special role in our methodology and so is designated the "Core" (with capitalization). When the Core is large relative to the system as a whole, and in comparison to the size of other cyclic groups, we say that the system has a "core-periphery" architecture.[2]

Returning to our example, we note that the largest cyclic group (i.e., the Core) in Codebase B is much smaller in relation to the system as a whole, accounting for only 3.5% of the files in the system. Almost 70% of the files in this system—shown in the third block down the main diagonal—lie on pathways that have no interdependency, in either direction, with Core files. Systems such as these display a high level of ordering in the dependencies among their components, thus we say that the system has a "hierarchical" architecture.

Critically, the structural relationships revealed in **Figure 2** cannot be inferred from standard measures of coupling nor from DSMs based on the architect's view alone. In the subsections below, we present a methodology to make this "hidden structure" visible and describe metrics that can be used to compare systems and track changes in system structure over time.

[2] Our definition of "Core" components differs from Murmann and Frenken (2006). Their definition is based on hierarchical ordering only and does not take account of cyclic groups.

### 3.1 Overview of Methodology and Rationales

A brief overview of our methodologly is as follows (the technical terms are fully defined in sections below). First, we identify the direct and indirect dependencies between system components in a DSM. We then use these measures to identify the cyclic groups (cores) of the system. Based on the size of the largest cyclic group relative to the system and to other cores, we classify the system architecture as "core-periphery," "multi-core," or "hierarchical." Next we divide the components into four groups based on their hierarchical relationship with the largest cyclic group (Core). Finally, we place the four component groups in order along the main diagonal of a new matrix, and within each group, sort the components to achieve a lower-diagonalized array. **Appendix A** provides a step-by-step description of the methodology.

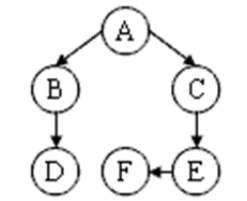
These steps constitute an empirical methodology whose purpose is to reveal both cyclic groups (cores) and hierarchical relationships among the components of a large system. Different parts of this methodology however, are motivated by different concerns. First our concern with hierarchical orderings and cyclic groups is motivated by the theories of dominant designs, and design cycles. Our classification scheme for architectures arose in response to empirical regularities discovered in our dataset. Finally our method of ordering component groups in a new network stems from a desire to represent hidden architectural patterns in pictorial form. Of course, the methodology presented here is not the only way to analyze the architecture of technical systems. Nevertheless, in our empirical work across a range of systems, we have found it a powerful way to discover hidden structure, classify system architectures, and categorize and visualize relationships among system components. We now describe this methodology in detail.

### 3.2 Identify the Direct Dependencies between Components

We represent the architecture of a complex system as a directed network graph made up of components (nodes) and directed dependencies (links) between them. The components are functional units within the architecture, such as software files in a codebase, basic steps in a

production process, or people in an organization. Consistent with both dominant design theory and modularity theory, the links are relationships of the form "A depends upon B" – i.e., if component B changes, component A may have to change as well. Dependencies are defined by the observer and may be things such as function calls, transfers of material, or messages between individuals. **Figure 3** shows an example system as both a directed graph and a Design Structure Matrix (DSM).[3] To distinguish it from the visibility matrix (which is defined in the step below), we call this DSM the "first-order" matrix.

**Figure 3: Example System in Graphical and Design Structure Matrix (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 |

### 3.3 Compute the Visibility Matrix

If we raise the first-order matrix to successive powers, the results show the *direct and indirect* dependencies that exist for successive path lengths. Summing all of these matrices yields the "visibility matrix" *V*, (see **Figure 4**) which shows the direct and indirect dependencies that exist for all possible path lengths. (Sharman, Yassine and Carlile, 2002; Sharman and Yassine, 2004; MacCormack et. al. 2006). When calculating the visibility matrix we choose to include the matrix for *N=0* (i.e., a path length of zero), implying a change to an element will always affect itself.

**Figure 4: The Visibility Matrix for the Example System in Figure 3**

| $V = \Sigma M^n$ ; 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** |

[3] Dependency matrices are also called "adjacency matrices" or "influence matrices."

| | | | | | | |
|---|---|---|---|---|---|---|
| D | 0 | 0 | 0 | **1** | 0 | 0 |
| E | 0 | 0 | 0 | 0 | **1** | **1** |
| F | 0 | 0 | 0 | 0 | 0 | **1** |

The visibility matrix, *V*, is identical to the "transitive closure" of the first-order matrix. That is, it shows all direct and indirect dependencies between components in the system. Transitive closure can be calculated via matrix multiplication or algorithms such as Warshall's algorithm (Stein, Drysdale and Bogart, 2011). Algorithms for matrix multiplication and for calculating transitive closure are widely available and are active areas of mathematical research. Those used in computational programming languages such as Matlab™ or Mathematica™, are heavily optimized and updated as new analytical approaches are discovered. Our methodology takes these algorithms as a given and builds upon them.

### 3.4 Construct Measures from the Visibility Matrix

From the visibility matrix, *V*, we construct several measures. First, for each component (*i*) in the system we define:

- $VFI_i$ (Visibility Fan-In) is the number of components that directly or indirectly depend on *i*. This number can be found by summing the entries in the $i^{th}$ column of *V*.
- $VFO_i$ (Visibility Fan-out) is the number of components that *i* directly or indirectly depends on. This number can be found by summing the entries in the $i^{th}$ row of *V*.

In **Figure 4**, element *A* has *VFI* equal to 1, meaning that no other components depends on it, and *VFO* equal to *6*, meaning that it depends on all other components.

In prior work (MacCormack et. al., 2006, 2012), we defined Propagation Cost as the density of the visibility matrix, and used it to measure visibility at the system level. Intuitively, Propagation Cost equals the fraction of the system affected when a change is made to a randomly selected component. While Propagation Cost is not the focus of this paper, it is an important indicator of a system's overall architectural complexity. We include it here for completeness:

$$\text{Propagation Cost (PC)} \equiv \frac{\sum_{i=1}^{N} VFI_i}{N^2} = \frac{\sum_{i=1}^{N} VFO_i}{N^2}$$

### 3.5 Find and Rank the Size of all Cyclic Groups

The next step is to find all the cyclic groups in the system. By definition, each component within a cyclic group depends directly or indirectly on every other member of the group. Hence these components share the same levels of visibility. Proposition 1, which states this identity in a formal fashion, gives us a way to identify cyclic groups of components within a system.

**Proposition 1.** Every member of a cyclic group has the same *VFI* and *VFO* as every other member. (All proofs of the propositions are given in **Appendix B**.)

#### Submethod 1: Find all Cyclic Groups

(1) Sort the components, first by *VFI* descending, then by *VFO* ascending, to produce an ordered list of components. (Other sort orders are discussed in the **Appendix C**.)

(2) Proceed through the list, comparing the *VFI*s and *VFO*s of adjacent components. If the *VFI* and *VFO* for two successive components are the same, then by Proposition 1, they *might* be members of the same cyclic group. (It could be a coincidence, discussed below)

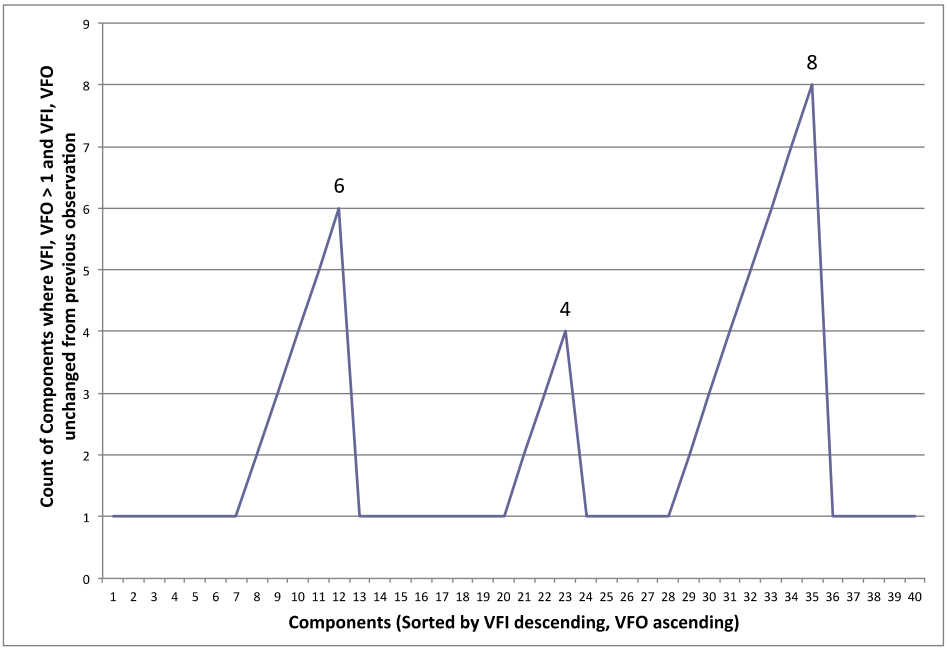
(3) Define a count measure, $m_i$, which will be associated with element *i*:

- If $VFI_i = 1$ or $VFO_i = 1$ or $VFI_i \neq VFI_{i-1}$ or $VFO_i \neq VFO_{i-1}$, set $m_i = 1$ ;
- If $VFI_i > 1$ and $VFO_i > 1$ and $VFI_i = VFI_{i-1}$ and $VFO_i = VFO_{i-1}$, set $m_i = m_{i-1} + 1$ .

The counter, $m_i$, will equal 1 if $VFI_i$ or $VFO_i$ equals one or if *VFI* or *VFO* changes with respect to the previous component in the sorted list. Alternatively, if $VFI_i$ and $VFO_i$ are both greater than one, and neither number changes, then $m_i$ will begin to rise by increments of one. Once *VFI* or *VFO* changes, $m_i$ will drop back to one. Thus, when arrayed against the list of components, the counter, *m*, will display a sawtooth pattern. By Proposition 1, any file on the rising edge of a sawtooth may be a member of a cyclic group. **Figure 5** shows an example of the

results of this procedure for a small system. In this system, there are three groups of components that share the same levels of visibility, with sizes of six, four and eight components.

**Figure 5: Sawtooth Pattern of $m_i$**




The procedure above finds groups of components that share the same VFI and VFO, and hence which might be members of the same cyclic group. The next step is to identify which of these are, in fact, a single cyclic group, and which are composed of multiple smaller cyclic groups that happen to share the same visibility values. To achieve this objective, we introduce Proposition 2, which sets an upper bound on the size of each cyclic group.

**Proposition 2.** Let $A$ be a cyclic group within a DSM. The size of $A$, denoted $N_A$, is bounded as follows:

$$N_A \leq \min(VFI_A, VFO_A, m_A^*) \ ;$$

where $VFI_A$ and $VFO_A$ respectively denote the visibility fan-in and fan-out measures for the group and $m_A^*$ is the maximum value attained by the sawtooth counter, before it drops back to one.

**Submethod 2: Find the Maximum Cycle Size for each Cyclic Group.**

(1) Find the top of each sawtooth blade by identifying the elements $i^*$, such that $m_i^* > m_{i+1}$.

(2) Then for each pair $(i^*, m_i^*)$:

a. Using Proposition 2, calculate the maximum cycle size for the associated group:

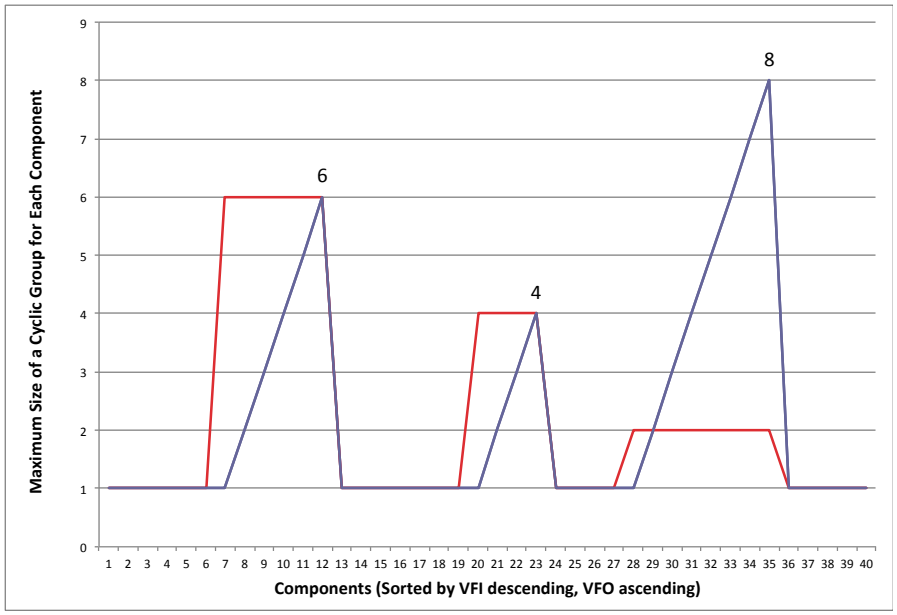
$N_i^* = \min(VFI_{i,}, VFO_i, m_i^*)$ ;

b. For $i = i^*$ to $i^* – m_i^*$, set $n_i = N_i^*$ ;

c. For all others, set $n_i = 1$ .

This method converts the sawtooth pattern of **Figure 5** into a block pattern as shown in **Figure 6**. In this stylized example the third group has $N_i^* < m_i^*$, thus the height of the block is less than the height of the corresponding sawtooth blade. This case arises when, by coincidence, two or more separate cyclic groups have the same *VFI* and *VFO* (see discussion below). Here, four cyclic groups have *VFI =VFO=2*, thus the maximum cycle size for elements in this group is 2. Hence this system, as presented, contains one cyclic group of six components, one cyclic group of four components, and four cyclic groups comprising two components each. We call all of these cyclic groups "cores" of the system. The largest cyclic group however (the "Core") plays a special role in our architectural classification scheme, as described below.

**Figure 6: Block Pattern of $n_i$ Overlaid on Sawtooth Pattern of $m_i$**




**Coincidences.** Elements that have different *VFIs* or *VFOs* cannot be members of the same cyclic group and elements for which $n_i=1$ cannot be part of a cyclic group at all. However,

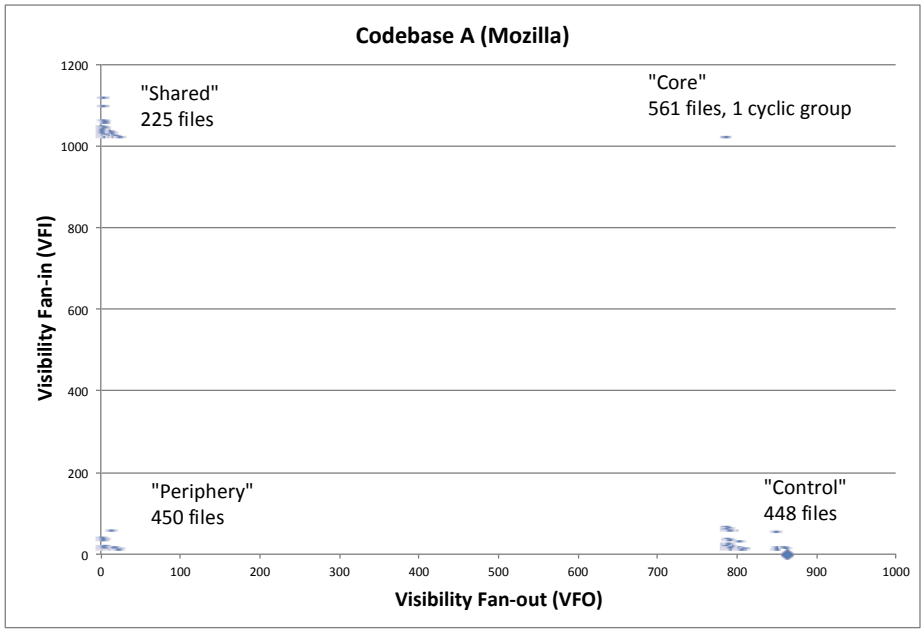
elements with the same *VFI* and *VFO* might be members of different cyclic groups. In other words, disjoint cyclic groups may, by coincidence, have the same Visibility measures. When *VFI*, *VFO* and *n* are large, the probability of coincidences is small and for practical purposes can be ignored. Coincidences are much more likely to arise when *VFI* or *VFO* (hence *n*) is small.

It is easy to determine whether a group of components with the same *VFI* and *VFO* is in fact one cyclic group or several. One simply inspects the subset of the visibility matrix that includes the rows and columns of the group in question and no others. If there are zeros in this submatrix, then the group contains two or more separate cyclic groups. One can identify the subsidiary groups by applying submethods 1 and 2 to the submatrix.

### 3.6 Classify the Architecture according to the size of the Core

Our method of classifying architectures was motivated by the discovery of an empirical regularity in our dataset. As an example, **Figure 7** presents a scatter plot of visibility measures for the components in Codebase A, with *VFI* arrayed on the vertical dimension and *VFO* on the horizontal dimension. The scatter has a "four-square" structure, indicating that there are four basic groups of components, located in the four quadrants of the graph.

**Figure 7: Scatter Plot of Components (Files) for Codebase A (Mozilla)**




First, the largest cyclic group appears in the upper right quadrant with *VFI* (=1009) and *VFO* (=768). This group contains 561 interconnected components, and is larger than any other cyclic group in the system, hence we label it the "Core". The Core contains 33% of the components in this system and is 16 times larger than the next largest cyclic group. In addition to the 561 components in the Core, 448 components depend on it (*VFI* = 1009 = 561+448*)*, and it depends on an additional 225 components (*VFO* = 768 = 561+225).

The 448 components that depend on the Core appear in the lower right quadrant of the graph. All these files depend on the Core, but the Core does not depend on them. We label these "Control" components because they make use of other components in the system but are not themselves used by other components. The 225 components that the Core depends on appear in the top left quadrant of the graph. These components are used—directly or indirectly—by many other components, both in and out of the Core. We label these "Shared" components. Finally, 455 components appear in the lower left quadrant of the graph. None of these files depends on the Core and the Core does not depend on them. We call them "Peripheral" components.

In our empirical work, this "four-square" pattern of *VFI* and *VFO* dependencies is observed frequently. The most salient characteristic of this pattern is the size and centrality of the largest cyclic group, the Core. Dependencies are transmitted from Control components, through Core components, to Shared components. At the same time, there are other components (the Periphery) that remain outside the main flow of dependencies in the system. Thus, in systems with a "four-square" structure (as revealed by this scatter plot), components can be categorized into four types defined by their relationship to the largest cyclic group (the Core).

**Classification.** Based on this empirical regularity, we define a core-periphery architecture as one containing a single cyclic group of components that is dominant in two senses: (1) it is large relative to the system as a whole; and (2) it is substantially larger than any other cyclic group in the system. In systems with a core-periphery architecture, a significant fraction of the system will be linked to the Core via direct or indirect dependencies. For example, in Codebase A, *all* the Shared and Control components are linked to the Core. Thus the total number of components connected to the Core equals (448 + 561 + 225 =) 1,234 or 73% of the system.

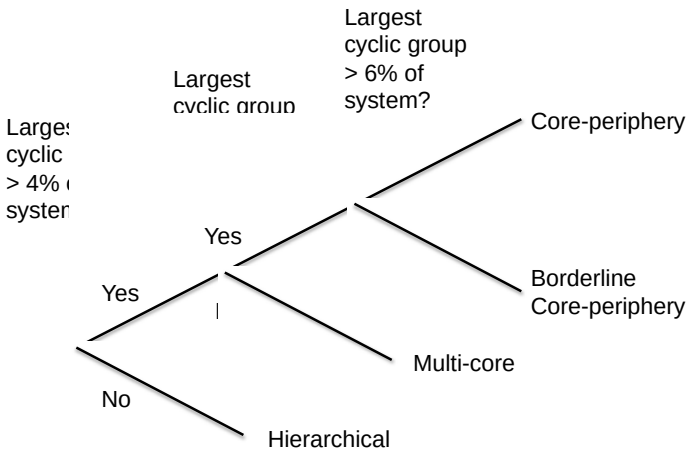
However, our empirical work also revealed systems that did not fit comfortably into the core-periphery classification. Some systems, for example, have several similarly-sized cyclic groups rather than one dominant one. Others like Codebase B have only a few extremely small cyclic groups. In light of this diversity, we sought to establish boundaries between different architectural types. While the precise boundaries are necessarily arbitrary, they give us a way to parse real world systems into different categories for analytic and statistical purposes.

Our first classification boundary is defined as follows: does the Core contain 4% or more of the system's elements? Systems that *do not* meet this test are labeled "hierarchical" systems. Next, within the set of systems that pass this threshold, we assess whether there is a single dominant cyclic group (as in **Figure 7**) or several similarly-sized groups. Our second classification boundary, applied to large-core systems only, is: does the largest cyclic group contain at least 50% more components than the next largest cyclic group? Systems that *do not*

meet this second test we label "multi-core" systems. Finally for systems that meet both the first and second tests, we ask, does the largest cyclic group contain more than 6% of the system? Those meeting this test are labeled "core-periphery" systems, those that do not meet the third test are labeled "borderline core-periphery" systems. **Figure 8** summarizes our classification scheme.

It is important to note that the size of the Core is a continuous variable, and an important parameter in its own right which can be used for analytical purposes regardless of the architectural classification given to a system. The use of any classification scheme based upon the size of the Core will, by definition, generate systems that are similar in nature, but which fall on opposite sides of the threshold and hence will be classified differently. In our classification system therefore, we choose to differentiate between systems that are clearly one type or another, versus those that are "borderline," in terms of the Core being near a threshold of 5% of system size.

**Figure 8: Architectural Classification Scheme**




### 3.7 Classify the Components into Types

For systems identified as having a core-periphery architecture, the components can be divided into four basic groups, corresponding to the quadrants of the "four-square" scatter plot (see **Figure 7**):

- *Core* elements are members of the largest cyclic group. By Proposition 1, all Core elements have the same *VFI* and *VFO*, denoted $VFI_C$ and $VFO_C$ respectively.
- *Shared* elements have $VFI \geq VFI_C$ and $VFO < VFO_C$.
- *Peripheral* elements have $VFI < VFI_C$ and $VFO < VFO_C$.
- *Control* elements have $VFI < VFI_C$ and $VFO \geq VFO_C$.

For hierarchical and multi-core systems, this partitioning can sometimes be problematic. In a multi-core system, the classification of components may not be stable over time: if one cyclic group overtakes another in terms of size, the identity of the "Core" and the resulting partition may change dramatically, even if the overall pattern of dependency changes very little. In hierarchical systems, this partition can lead to an unbalanced number of components in each category, which is problematic for statistical analysis.

To address these problems, we define an alternative way to classify components based on the *median* values of *VFI* and *VFO*. The median partition yields groupings that are more equal in size and more stable over time (assuming dependency patterns do not change significantly as the system evolves). However, in this partition, the high-*VFI* and high-*VFO* components will not in general be members of the same cyclic group, hence we call them "Central" (instead of "Core"). Similarly, the remaining categories are identified as Shared-M, Control-M and Periphery-M.

**3.8 Visualize the Architecture**

Using our component classification scheme, we construct a reorganized DSM that reveals the "hidden structure" of the system. We first:

(1) Place components in the order *Shared, Core (or Central), Periphery, Control* down the main diagonal of the DSM; and then

(2) Sort within each group by *VFI* descending, then *VFO* ascending.

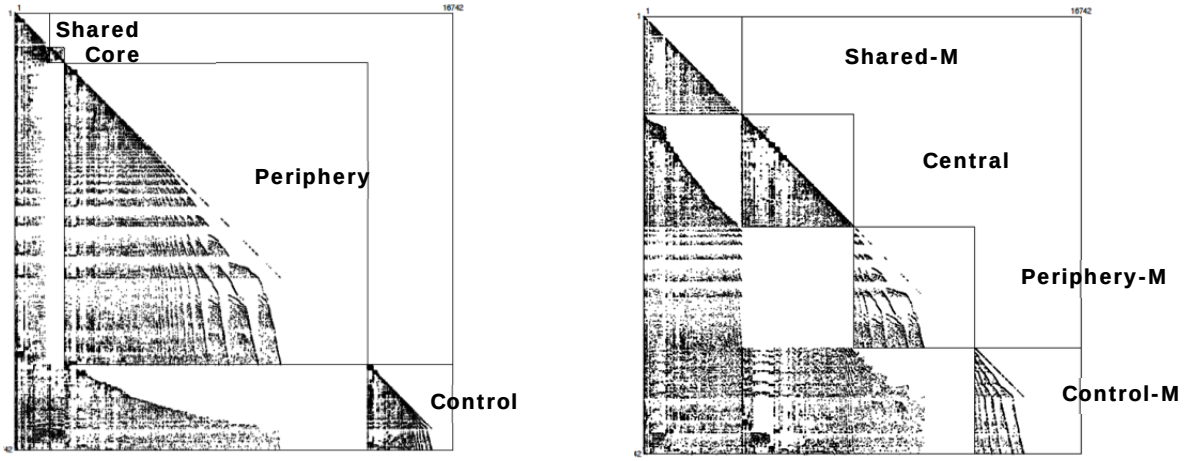
This methodology obtains a reordered DSM with the following properties:

- Cyclic groups are clustered around the main diagonal.
- There are no dependencies *across groups* above the main diagonal.

- There are no dependencies between the *Core* (or *Central)* group and the *Periphery* above or below the main diagonal.
- Except for cyclic groups, each block is lower diagonalized.

If the largest cyclic group is the basis of the partition, we call this the "core-periphery view" of the system. If the median value of *VFI* and *VFO* are the basis of the partition, we call this the "median view." **Figure 9** shows both views of Codebase B.

**Figure 9: Core-Periphery and Median Views of Codebase B (a Hierarchical System)**




**Core-Periphery View** **Median View**

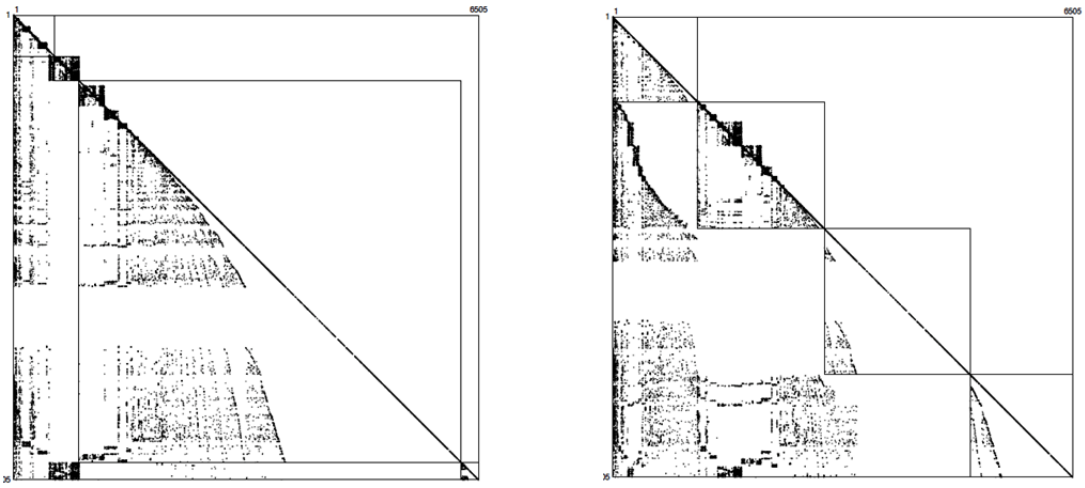
The core-periphery and median views are complementary ways of visualizing the flow of dependencies in a large technical system. In general, the core-periphery view is more informative as the largest cyclic group increases in size relative to the system as whole and other cyclic groups. However, we have found that, especially in borderline cases, both views are helpful.

**Figure 10** shows the core-periphery and median views of Codebase C, a multi-core system. Codebase C is a version of Open Office, an open source suite of applications that includes a word processor, a spreadsheet program, and a presentation manager. The multiple cores in this system correspond to different applications. As anticipated, a core-periphery categorization leads to unbalanced groupings: 82% of the system including the second and third

largest cyclic groups are in the periphery. The median partition, by contrast, results in more balanced groupings and places all signficant cyclic groups in the "Central" region. It also reveals interesting subsidiary structures,(e.g., the three largest cyclic groups appear to be independent).

**Figure 10: Core-periphery and Median Views of Codebase C (a Multi-core System)**

**Core-Periphery** **Median**



This concludes the description of our methodology. In the next section we describe the application of these methods to the analysis of large software systems.

## 4. Empirical Application

In this section, we describe the application of our methodology to a large (non-random) sample of real world software systems. Specifically, we explore the frequency with which different architecturual types are observed, and the variations observed in the size of the Core across systems and releases. We also examine whether the Core components identified by our methods are typically clustered in a few subsystems or distributed across many. Finally, we investigate changes in the size of the Core as systems grow over time to determine if and how discontinuous changes are made. Our main objective is to establish some stylized facts about real world systems, and identify consistent patterns of behavior: not to formally specify or test hypotheses. We view this empirical work as a first step in establishing useful benchmarks that can

inform future studies.

### 4.1 Data

Our dataset comprises 1286 different software releases from 17 different software applications for which we could secure access to source code. (See **Appendix D** for a list of applications, their function and origin, the number of releases and system size as of the last release). Many of these systems are active open source software projects. Some started as commercial products but were later released under an open source license (e.g., the Mozilla browser). Finally, a small number of releases are proprietary systems developed by commercial firms, whose names are disguised.

In assembling this dataset, we focused on large software systems that at some point in their history obtained many users. Hence we do not include in our sample open source projects from repositories such as SourceForge, which are typically small systems with few users. Although some of our systems (e.g., Linux) start small, all have more than three hundred source files as of the last release in our dataset. That said, our sample is not random nor is it representative of the industry in general, thus we do not claim the results are general. Our exploratory research only aims to provide a starting point for subsequent empirical investigation and hypothesis testing.

We obtained the source code for each release in the sample and processed it to identify all major dependencies between source files, including function calls, class method calls, class method definitions, and subclass definitions. We used this data to calculate *VFI* and *VFO* for each file and hence the *Propagation Cost* for each release. Using our methodology, for each release we identified the Core, classified the architecture as core-periphery, borderline, hierarchical, or multi-core, and classified the components into four categories (Shared, Core, Periphery, Control).

**Table 1** summarizes the descriptive data for our sample. The dataset includes releases with a wide spectrum of sizes, from less than 50 components, to over 12,000. The size of the Core varies considerably, from under 10 components to over 3,000 components. As a fraction of the total system, the Core size varies broadly from 1% to 75% of all components. The average

release has 1,724 components, of which 201 (16%) are in the Core.

**Table 1: Descriptive Data for the Sample**

| | MIN | MAX | MEAN | MEDIAN |
|---|---|---|---|---|
| System Size (files) | 45 | 12949 | 1724 | 781 |
| Core Size (files) | 6 | 3310 | 201 | 74 |
| Core Size % of System | 1% | 75% | 16% | 9% |

### 4.2 The Prevalence of Core-periphery Structures

We find that 867 of the 1286 releases (67%) possess a core-periphery architecture according to the definition given in Section 3, while 309 (22%) are "borderline," defined as having a Core greater than 4% but less than 6% of the system. Of the rest, 94 (7%) are hierarchical, and 6 (.5%) are multi-core. (The multi-core systems, belonging to Open Office, contain smaller core-periphery systems such as Word and Calc.) Thus core-periphery architectures dominate the releases in this sample, although the range of architectures and Core sizes is very large. However, the number of borderline systems is surprisingly large (at least as compared to what would be expected if Core size is distributed evenly throughout the range).

We chose to classify *systems* according to the architecture *of the last release* in our sample. The last release is usually the largest, offers the highest degree of functionality, and presumably has been most refined to the state that represents the "optimal" design for the system. The results of this are shown in **Table 2**. Of the 17 systems, 13 (76%) had a core-periphery architecture, three (18%) were borderline, two (12%) hierarchical and one (6%) multi-core. Again we find the core-periphery architecture is the most common, although the high number of borderline systems is again surprising. The low number of hierarchical and multi-core systems is notable.

In conclusion, the majority of systems and releases in our sample possess a core-periphery structure, defined as having the largest cyclic group of components comprise greater than 6% of system components. A significant fraction (around 20%) have Core sizes around this threshold, between 4-6%. Only a small number of systems and releases have hierarchical structures. Finally, only one system – comprising a "suite" of distinct applications – has a multi-core

structure.

**Table 2: Classification of Systems**

| | | Core | | |
|---|---|---|---|---|
| **System Name** | **No. Files (Last Release)** | **No. Files (Last Release)** | **% System (Last Release)** | **Architecture Classification (Last Release)** |
| Mozilla | 5899 | 157 | 2.7% | Hierarchical |
| OpenAFS | 1304 | 51 | 3.9% | Hierarchical |
| GnuCash | 543 | 23 | 4.2% | Borderline |
| Abiword | 1183 | 59 | 5.0% | Borderline |
| Apache | 481 | 25 | 5.2% | Borderline |
| Chrome | 4186 | 260 | 6.2% | Core-periphery |
| Linux (kernel) | 8414 | 621 | 7.4% | Core-periphery |
| MySQL | 1282 | 160 | 12.5% | Core-periphery |
| Ghostscript | 653 | 90 | 13.8% | Core-periphery |
| Darwin | 5685 | 939 | 16.5% | Core-periphery |
| Open Solaris | 12949 | 3310 | 25.6% | Core-periphery |
| MyBooks | 2434 | 675 | 27.7% | Core-periphery |
| PostGres | 703 | 282 | 40.1% | Core-periphery |
| XNU | 781 | 351 | 44.9% | Core-periphery |
| GnuMeric | 314 | 148 | 47.1% | Core-periphery |
| Berkeley DB | 299 | 146 | 48.8% | Core-periphery |
| | | | | |
| Open Office | 7360 | 346 | 4.7% | Multi-core |
| Write (Open Office) | 814 | 372 | 45.7% | Core-periphery |
| Calc (Open Office) | 545 | 328 | 60.2% | Core-periphery |

**Detecting Core-periphery Architectures**

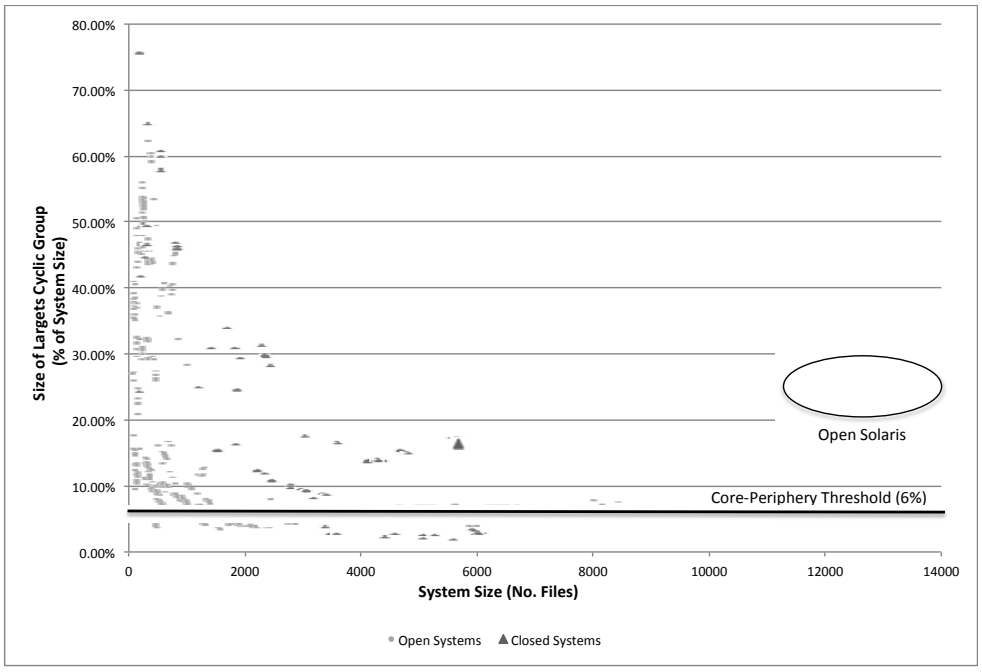
It is natural to ask whether the presence of a core-periphery architecture (or the lack thereof) can be detected from the summary statistics for a system (e.g., number of files, directories or lines of code, average number of dependencies per file) or from inspection of the first-order dependency matrix. To explore this question, we compared systems that possessed a core-periphery architecture with those that did not, focusing on differences in both the quantitative data and the visual plots of DSMs using the architect's view (i.e. sorting files by directory as in **Figure 1**). We found no variable that could reliably predict whether a system

possessed a core-periphery structure, and no consistent pattern of direct dependencies in the architectural view of a DSM that would signal the presence of dominant cyclic group. Thus detecting the presence of a core-periphery architecture cannot be achieved solely by examining the direct dependencies for a system, but requires an assessment of the paths by which these dependencies propagate.

### 4.3 The Size of the Core across Different Systems

We next compare the size of the Core across systems and releases. **Figure 11** plots Core size (as a percent of system size) against the system size for all releases in our sample. The graph differentiates between systems that began as open source projects (light circles), and those that originated as or continue to be commercial products (dark triangles).

**Figure 11: The Size of the Core (Largest Cyclic Group) versus Total System Size**

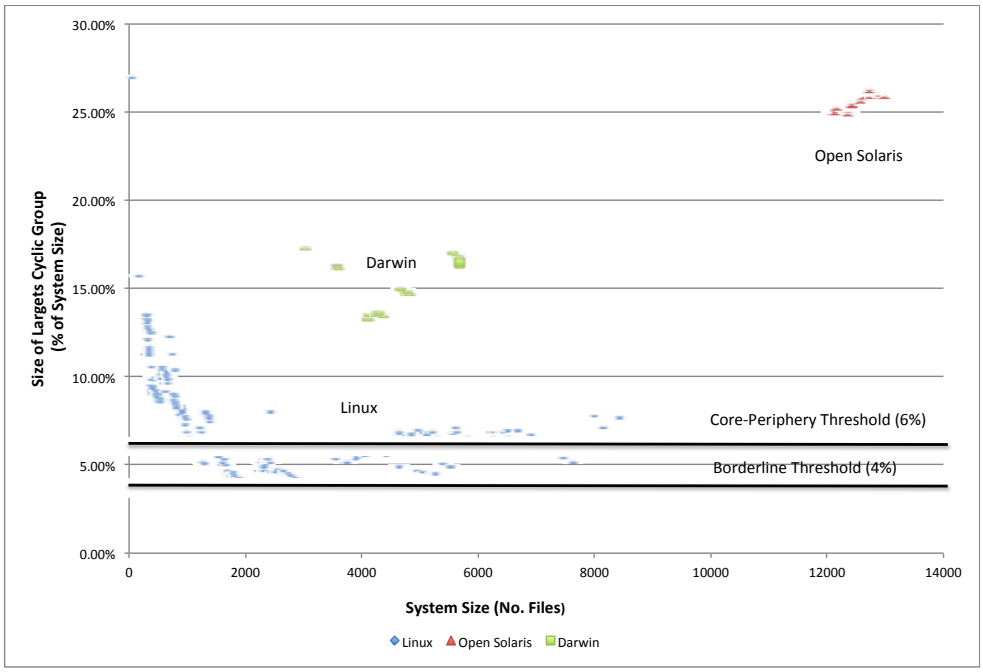



In this chart, we observe that for very small systems, the size of the Core varies substantially, from less than 5% to a maximum of 75% of the system. For larger systems however, the Core declines as a proportion of the system. Indeed there appears to be a negative exponential relationship between Core size and system size. With the exception of Open Solaris

(shown at the far right of the graph), in systems that exceed 3,000 source files, the Core never exceeds 20% of the system. Intuitively, this pattern makes sense. For small systems, a relatively large Core is still small in absolute terms, and thus architects and developers can still comprehend its internal structure. In larger systems however, even a moderately large Core creates cognitive and coordination challenges, given that architects and developers must understand and communicate with each other about many possible direct and indirect interdependencies. For larger systems, there is therefore a disproportionate benefit from having relatively smaller Cores.

Next we examine Core sizes for systems performing similar functions. **Figure 12** focuses on three operating systems in our sample: Linux, Open Solaris, and Darwin, the platform upon which Apple's OS X software is based. The contrasts are striking. With Linux, relative Core size declines and then flattens as the system has grown. In contrast, Open Solaris has a large Core in both absolute and relative terms. Darwin falls between the two: as it grew from 3017 files (Darwin 1.3.1) to 5685 files (Darwin 8.9), its Core grew from 512 files to 939 files components, averaging 15% of the system. Clearly there is wide variation in Cores size across systems of similar function.

**Figure 12: The Size of the Core for Systems of Similar Function**




*Explaining Different Core Sizes: Different Organizational Types*

We sought to explore one possible driver of differences in Core size – the type of organization that develops a system. Here we built on prior theoretical work which argues that product designs tend to reflect the structure of the organizations in which they are conceived, an effect known as Conway's Law or the "mirroring hypothesis" (Conway, 1968; Henderson and Clark, 1992; Sosa et al, 2004; Colfer and Baldwin, 2010; MacCormack et al, 2012). This theory suggests that organizations with co-located developers in close communication (as is typical within corporations) will produce relatively tightly coupled systems. In contrast, organizations with geographically distributed developers not in close communication (as is typical of open source projects) will produce relatively loosely coupled systems. A relatively large (or small) Core is in turn evidence of tighter (or looser) coupling among the components in the system.

To conduct this exploration, we compare the design of systems with similar functions that emerge from different types of organizations, specifically, open source versus commercial

development. We use a matched-pair design, comparing the size of the Core between systems of similar size and function. Our sample was based on a prior study that explored differences in the propagation cost between open source and commercial systems (See MacCormack et al, 2012 for details on how the matched pairs were selected.) **Table 3** shows the size of the Core (relative to system size) for the five matched pairs. In every case, the systems that originated as open source projects have smaller Cores than systems originating as commercial products. Indeed in one case (financial management software) the open source system has a hierarchical architecture, while the commercial system of similar size has a Core that accounts for 70% of the system. Although many factors influence the choice and design of system architectures, this comparison, building upon the prior investigation, provides evidence that differences in system architecture and particularly Core size may be driven in part by differences in the structure of the developing organization.

**Table 3: The Size of the Core for a Sample of Matched-Pair Products**

| Application Category | Open Source Product | | Closed (Commercial) Product | |
|---|---|---|---|---|
| | System Size | Core Size | System Size | Core Size |
| Financial Mgmt | 466 | 3.4% | 471 | 69.60% |
| Word Processor | 841 | 6.10% | 790 | 46.10% |
| Spreadsheet | 450 | 25.80% | 532 | 57.30% |
| Operating System | 1032 | 6.30% | 994 | 28.00% |
| Database | 465 | 7.70% | 344 | 48.80% |

### 4.4 The Location of Core Components in a System

We next explore whether Core components tend to be located in a few subsystems or are distributed throughout a system. Given the information-based nature of software, there is no ex-ante need for the Core components in a system to be physically co-located. They can be distributed throughout a system and still function as intended. However, from the perspective of the system architect (or maintainer) there are cognitive and coordination benefits to locating Core components in a small number of directories, thereby allowing them to be managed together.

We find somewhat surprisingly that Core components are often *not* located in a small number of directories, but instead are distributed throughout a system. **Table 4** provides an example, showing the distribution of Core files in the top-level directories of Linux 2.3.39. This system possesses 118 Core components out of a total of 2419 (4.9%). However, rather than the Core components being concentrated in one or two subsystems (i.e., directories), 8 of the 10 top-level directories contain at least one Core component.

**Table 4: Distribution of Core Files across Directories (Linux 2.3.39)**

| Directory | Total Files in Directory | Core Files in Directory | Core Files as a Percent of Directory |
|---|---|---|---|
| '~arch | 689 | 53 | 8% |
| '~drivers | 1051 | 18 | 2% |
| '~fs | 334 | 20 | 6% |
| '~init | 2 | 1 | 50% |
| '~ipc | 4 | 2 | 50% |
| '~kernel | 23 | 10 | 43% |
| '~lib | 5 | 0 | 0% |
| '~mm | 18 | 10 | 56% |
| '~net | 279 | 4 | 1% |
| '~scripts | 14 | 0 | 0% |
| | | | |
| Total | 2419 | 118 | 4.9% |

Our data suggests that the main flow of dependencies (from Control to Core to Shared) may not be apparent from the visible structure of the system. That is, the system has *Hidden Structure*. Simply inspecting the directory structure in a system will in general *not* be sufficient to reveal where Core components are located. Hence changes to one Core component may propagate to other Core components in seemingly remote parts of the system. This issue is especially pertinent when a legacy system must be maintained or adapted where only limited documentation exists on its original design. Only through a detailed analysis of the chains of direct and indirect dependencies can the "hidden structure" of the system be made visible.

### 4.5 The Evolution of System Structure

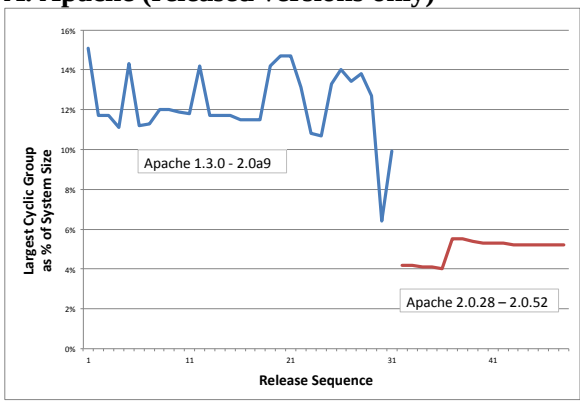
In the final application of our methodology, we explore how the Cores of all 17 systems evolve over time. This is accomplished by plotting Core size as a function of release, and observing whether the trend is continuous or discontinuous, and whether Core size declines, remains flat, or increases, over time. The data revealed that relative Core size declined consistently in three cases; in 8 cases it remained flat; and in two cases it increased.[4] The four

[4] In one case (Chrome), we had only one release, hence insufficient data.

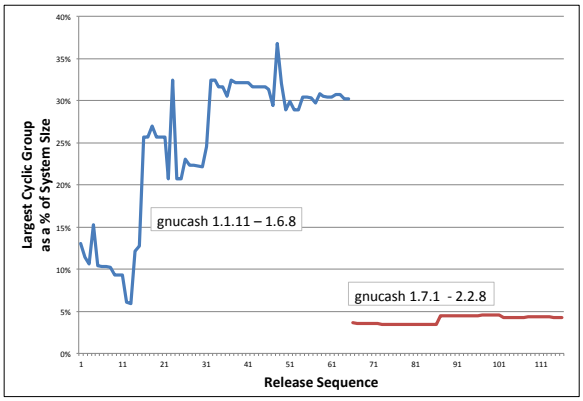
remaining systems (Apache, Gnucash, Linux and Mozilla) exhibited discontinuous breaks in Core size. These are the most interesting cases, and they are shown in **Figure 13**.

**Figure 13: Systems Exhibiting Discontinuous Changes in Core Size**

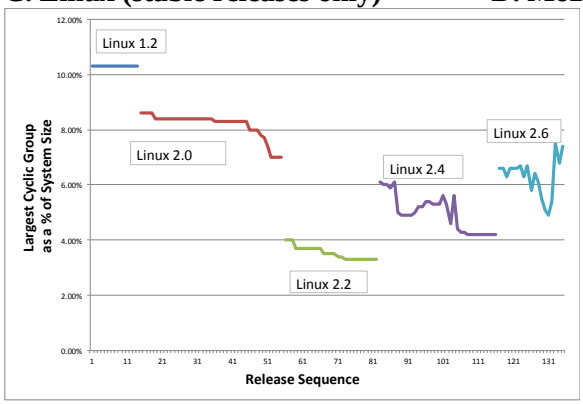
**A. Apache (released versions only)**



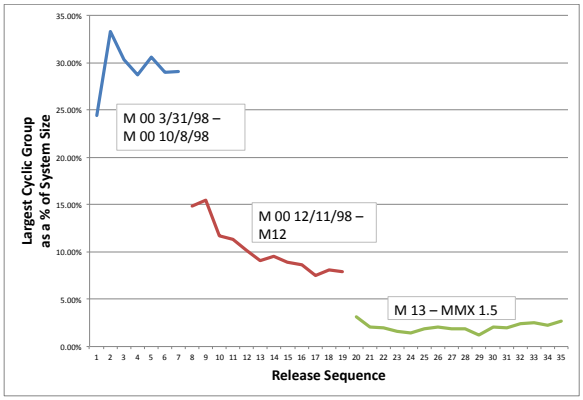

**B. Gnucash**




**C. Linux (stable releases only)**




**D. Mozilla**




Apache began with a core-periphery architecture, with a Core in the range of 12% to 14%. A significant redesign of the system took place between version 2.0.a9 and 2.0.28. In version 2.0.28, the Core size dropped to 4% of the system, rising to just over 5% in subsequent releases. The case of Gnucash is more dramatic. Early on, the Core grew significantly from 13 to 70 files or approximately 30% of the system. With release 1.7.1, however, system size almost doubled (232 to 449 files), but the Core dropped from 70 to 16 files (3.6% of the system). In later releases, the Core has consistently accounted for just 4-5% of the system.

Note that Apache and Gnucash are relatively small systems.[5] In their size range (below 500 files), Core size relative to system size varies considerably (see **Figure 9**). In small systems, Core interdependencies can be directly inspected and understood by architects and developers, thus architectural changes aimed at reducing the size of the Core may have low priority. In contrast, our next two examples, Linux and Mozilla are large systems, which have grown significantly over time, and which present greater challenges in terms of understanding the whole.

In the case of Linux, discontinuous changes in the size of the Core have coincided with major releases.[6] **Figure 13 C** shows that Linux started out as a core-periphery system with the Core initially accounting for just over 10% of the system. This figure dropped to around 8% for Linux 2.0 and to just over 4% with Linux 2.2. However, we observe small discontinuous jumps in Core size associated with the release of Linux 2.4 and 2.6. The Mozilla Application Suite exhibited two discontinuous changes in Core size, although here the trend is consistently downward. (See **Figure 13 D**.) The first discontinuity occurred in December 1998: the Core dropped from 680 files (29% of the system) to 223 files (15%). (System size also dropped but not as much.) Subsequently, the system grew significantly (from 1508 to 3405 files) while the Core grew only slightly (from 223 to 269 files or 7.9% of the system). We know from prior work that the change in Mozilla's design in December 1998 was the result of a purposeful redesign effort, which had the explicit objective of making the codebase more modular, hence easier for contributors to work within (MacCormack et al, 2006). As **Table 5** shows, achieving this goal led to substantially smaller Core and Shared groups and larger Periphery and Control groups. (Note, we do not know the reasons behind the second discontinuous change in the architecture of this codebase.)

[5] The last releases in our dataset contained 481 files and 543 files respectively.

[6] During the period of our sample, the Linux kernel used an "even-odd" version numbering scheme. Even numbers in the second place of the release number (e.g., 2.4.19) denoted "stable" releases that were appropriate for wide deployment; odd numbers (e.g., 2.5.19) denoted "development" releases that were the focus of ongoing experimentation. Work on the even and odd numbered releases would go on simultaneously, hence release numbers are in temporal sequence only within two sets. http://www.linfo.org/kernel_version_numbering.html. The even-odd numbering practice was discontinued with the release of version 2.6.0.

**Table 5: Components in Each Category before and after Mozilla Redesign**

| Type of Component | % before Redesign (4/8/98 Release) | % after Redesign (12/11/98 Release) |
|---|---|---|
| | | |
| Shared | 13% | 3% |
| Core | 33% | 15% |
| Periphery | 27% | 36% |
| Control | 27% | 46% |
| Total | 100% | 100% |

To summarize, we found no single pattern to characterize the way the Core of a system evolves over time. Changes in relative Core size often appear continuous (i.e. display no sharp breaks), while the Core may increase, stay the same or decrease in relation to the system as a whole. In the majority of cases in our sample (76%), the Core did not seem to be a focus of major redesign effort. In a few cases, however, we saw discontinuous changes that seemed to be the result of architectural intervention, rather than incremental change. The most dramatic of these resulted in a reduction of the size of the Core. In one case (Mozilla, December 1998), we know from interviews with the architects involved that the purpose of the redesign was to reduce system complexity. These findings are consistent with the conjecture (from design theory) that cyclical dependencies are problematic because they increase cognitive complexity and the number of iterations needed to arrive at an acceptable design. Note however, our earlier finding that Core files are dispersed through the system means that it may be hard to identify the components in the system that give rise to these problematic cyclical dependencies. A positive feature of our methodology therefore is that it identifies the Core and its members, potentially aiding managers in the process of architectural redesign.

## 5. Discussion

In this paper, we developed robust and reliable methods to detect the core components in a complex system, to establish whether these systems possess a core-periphery structure, and to 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 – 67% to 76% – possess a core-periphery structure. Another ~20% are borderline core-periphery. However, it is important to note that a significant number of systems *lack* such a structure. This implies 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 differences in design are not driven solely by system requirements. Instead, these differences appear to be driven by the characteristics of the organization in which system development occurs. Specifically, we find evidence that variations in system structure are explained, in part, by the different models of development used to build systems. That is, product structures "mirror" the structure of their parent organizations (Henderson and Clark, 1990, Sosa et al, 2004; Colfer and Baldwin, 2010). 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, Jain and Kumaraswamy, 2002).

Our findings highlight the difficulties that face a system architect. In particular, we find no discernible pattern of *direct* dependencies that can reliably predict 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. 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 these 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 decide where to focus attention. It is not simply a matter of concentrating on subsystems that contain most of the core components. Important relationships may exist between these components and others within subsystems that, on the surface, appear insignificant. This highlights the need to understand patterns of coupling at the component level, and not to assume that all key relationships in a complex system are located in a few subsystems.

These issues are especially pertinent in 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 – obligations that are increasingly referred to as a system's "technical debt" (Brown et al, 2010; Kruchten, 2012; Nord, 2012). 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 available when important design choices need revisiting. This difficulty is compounded by the fact that designers rarely document their design choices well, requiring the hidden structure to be recovered by inspection of the source code.

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

results. First, our work was conducted in the software industry, a unique context given that designs exist purely as information, and are not bounded by physical limits. Whether the results could be replicated for physical products remains an important empirical question. Second, given the difficulty in obtaining proprietary software, we adopted a non-random sample of systems for which we had access to the source code. Although we limited 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 are clearly sensitive to the thresholds used in determining what represents a core-periphery versus a hierarchical or multi-core structure. Indeed, it was this sensitivity that led us to define a category for borderline systems near the thresholds chosen.

Our work opens up a number of avenues for future study, especially given that 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 the firms that develop those offerings (e.g., see MacCormack and Iansiti, 2009). Other 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 identify whether the organizational network as a whole predicts future product structure.

Another avenue of research is the use of our methodology to predict the location of product defects, developer productivity, and even developer turnover. In separate case studies, Akaikine (2009) and Sturdevant (2013) have applied our methodology to two large commercial codebases in different firms. Both studies found significant differences in performance measures, including defect resolution times and developer productivity, across different component categories (Core,

Shared, Peripheral, Control). However further work is needed to generalize these observations, both within single systems comprising many thousands of components, and across larger samples of systems, serving different purposes and emanating from different organizations

Software is a natural venue in which to develop and test our methodology, because dependencies between software components can be automatically extracted from source code using widely available tools. However, our methods can be applied to any technical system whose architecture can be represented as a network graph with directed links. Corporate IT systems and enterprise architectures can be represented in this fashion, and automated tools to extract dependencies (e.g., between applications and tasks) are now being developed. The extension of our methods to IT systems and enterprise architectures is a promising avenue for future research.

All in all, our methods may be helpful in locating and measuring the *technical debts* in a system that is, the costs of making and verifying future changes in a complex technical system. 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: A Methodology for Analyzing, Classifying and Viewing the Architecture of a Complex System

1) Represent the system in terms of a Design Structure Matrix (DSM). If element *j* depends on element *i*, place a "1" in the column of *i* and the row of *j*. Call this matrix *A* (the first-order matrix).

2) Compute the visibility matrix for *A* using matrix multiplication or an algorithm (such as Warshall's) for computing transitive closure. Call this matrix *V*.

3) For each element *i*, compute $VFI_i$ as the column sum of *V* for that element and $VFO_i$ as the row sum of *V* for that element.

4) Identify the cyclic groups of the system and identify the largest. (Other cycle-finding algorithms may be used here.)

   a) Sort the elements, first by *VFI* descending, then by *VFO* ascending.
   b) Proceed through the sorted list, comparing the *VFI*s and *VFO*s of adjacent elements.
   c) Define a count measure, $m_i$, for each element *i*:

      - If $VFI_i = 1$ or $VFO_i = 1$ or $VFI_i \neq VFI_{i-1}$ or $VFO_i \neq VFO_{i-1}$, set $m_i = 1$ ;
      - If $VFI_i > 1$ and $VFO_i > 1$ and $VFI_i = VFI_{i-1}$ and $VFO_i = VFO_{i-1}$, set $m_i = m_{i-1} + 1$ .

   The counter, $m_i$, will equal 1 if $VFI_i$ or $VFO_i$ equals one or if *VFI* or *VFO* changes with respect to the previous component in the sorted list. Alternatively, if $VFI_i$ and $VFO_i$ are both greater than one, and neither number changes, then $m_i$ will begin to rise by increments of one. Once *VFI* or *VFO* changes, $m_i$ will drop back to one.

   d) Identify the elements $i^*$, such that $m_i^* > m_{i+1}$.
   e) Then for each pair $(i^*, m_i^*)$:
      i) Use Proposition 2 (in **Appendix B**) to calculate the maximum size, $M_i^*$, for the associated cyclic group;
      ii) For $i = i^*$ to $i^* - m_i^*$, set $n_i = M_i^*$ ;
      iii) For all others, set $n_i = 1$ .
   f) Find the set of elements, *C*, for which $n_C > n_{\sim C}$. (If there is a tie, the system has either a multi-core or a hierarchical architecture.)
   g) Check that *C* contains only one cyclic group. If so, these elements form the largest cyclic group of the system.

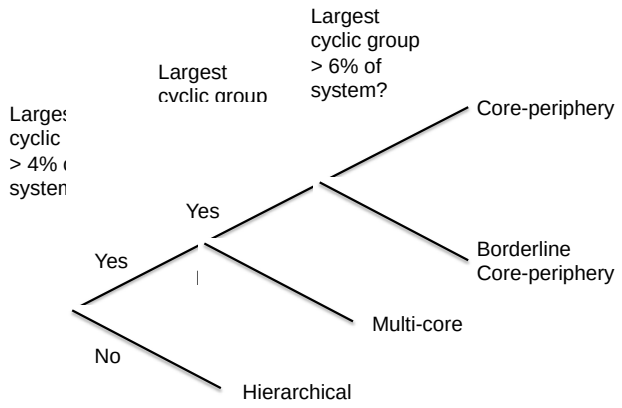
5) Classify the architecture using the following tests:

   a) Is $n_C \geq .04\,N$? (Largest cyclic group accounts for at least 4% of the system.)
   b) Is $n_C \geq 1.5 \max n_{\sim C}$? (Largest cyclic group is at least 50% larger than next largest.)
   c) Is $n_C \geq .06\,N$? (Largest cyclic group accounts for at least 6% of the system.)

   If answer to all three questions is "yes", classify the system as having a core-periphery architecture. If the answer to (a) and (b) is "yes", and (c) is "no", classify the system as borderline core-periphery. If the answer to (a) is "yes" and (b) is "no", classify the system as multi-core. Finally, if the answer to (a) is "no", classify the system as a hierarchical

architecture.

**Classification of Architectures:**




6) Classify the components of the system into four groups according to the core-periphery partition or the median partition:

**Core-periphery Partition.** Define the largest cyclic group as the "Core" of the system. Let $VFI_C$ and $VFO_C$. respectively denote the *VFI* and *VFO* of elements in the Core. Allocate the non-Core elements to three groups as follows:

a) "Shared" elements have $VFI \geq VFI_C$ and $VFO < VFO_C$.
b) "Peripheral "elements have $VFI < VFI_C$ and $VFO < VFO_C$.
c) "Control" elements have $VFI < VFI_C$ and $VFO \geq VFO_C$.

**Median Partition.** Compute the medians, $VFI_M$ and $VFO_M$. Allocate elements to four groups as follows:

d) "Shared" elements have $VFI \geq VFI_M$ and $VFO < VFO_M$.
e) "Central" elements have $VFI \geq VFI_M$ and $VFO \geq VFO_M$.
f) "Peripheral "elements have $VFI < VFI_M$ and $VFO < VFO_M$.
g) "Control" elements have $VFI < VFI_M$ and $VFO \geq VFO_M$.

7) Create a reordered DSM to visualize the system based on the core-periphery or median partitions:

a) Order the elements by group as follows: Shared, Core (or Central), Peripheral, Control.
b) Within each group, sort the elements by *VFI* descending, *VFO* ascending.

## Appendix B: Proofs of the Propositions

**Proposition 1.** Every member of a cyclic group has the same *VFI* and *VFO* as every other member.

**Proof.** Members of a cyclic group all directly or indirectly depend on one another. This means that if element *x* outside the group depends on *a* in the group, then *x* will indirectly depend on all other members of the group. As this applies to any *x* and any *a*, the *VFI* of all members of the group will be the same. Conversely if *a* in the group depends on *y* out of the group, then all members of the group will indirectly depend on *y*. This applies to any *y* and *a*, thus the *VFO* of all members of the group will be the same. *QED*

**Proposition 2.** Let *A* be a cyclic group within a DSM. The size of *A*, denoted $N_A$, is bounded as follows:

$$N_A \leq \min(VFI_A, VFO_A, m_A^*) \ ;$$

where $VFI_A$ and $VFO_A$ respectively denote the visibility fan-in and fan-out measures for the group and $m_A^*$ is the maximum value attained by the sawtooth counter, before it drops back to one.

**Proof**. All members of a cyclic group fan into and out of each other, thus $N_A \leq VFI_A$ and $N_A \leq VFO_A$. $m_A^*$ counts the number of elements with the same *VFI* and *VFO*: a cyclic group with these properties cannot be larger than this count thus $N_A \leq m_A^*$ . The size of the group, $N_A$, is subject to all three constraints, hence the minimum number is the binding constraint. *QED*

**Proposition 3.** Sorting members of a sequence (with no embedded cycles) by *VFI* descending causes all dependencies to fall below the main diagonal of the DSM.

**Proof.** Let the sort result in a particular ordering of elements: *1, 2, ..., i, j, ... N*, where *j* is below *i*. Now suppose a dependency from element *i* to *j* appears in the row of *i* and the column of *j*, which, by definition, lies to the right of the main diagonal. The presence of a link from *i* to *j* implies that *i* must depend on all elements that *j* depends on. If *i* already depends on *j* then *i* and *j* are part of cycle which contradicts the premise of no embedded cycles. If *i* and *j* are not part of a cycle, then

all the elements that depend on *i* must depend on *j*. Also *i* itself must depend on *j*. Therefore:

$$VFI_j = VFI_i + 1 .$$

But this contradicts the sorting algorithm, which stipulates that:

$$VFI_i \geq VFI_j .$$

*QED*

**Proposition 4.** In a "core-periphery" or "median" DSM, there are no dependencies between groups above the main diagonal.

**Proof.** Consider the core-periphery view first. The proof follows the same logic as Proposition 3. First suppose a dependency exists from a Shared element *i* to a Core element *j*. (By definition, *j* lies below *i* in the DSM.) Then either element *i* is part of the Core cyclic group or the Core has $VFI_C = VFI_i+1$ by transitive dependency. But, according to the definition of Shared elements, $VFI_C \leq VFI_i < VFI_i+1$. Thus a dependency from a Shared element to a Core element leads to a contradiction. Similar reasoning applies to dependencies from Shared to Periphery and Control elements, from the Core to Periphery and Control elements, and from the Periphery to Control elements.

The proof is identical for the median view. *QED*

**Proposition 5.** In a "core-periphery" or "median" DSM, there are no dependencies between the Core or Cental group and the Periphery *above or below* the main diagonal.

**Proof.** Proposition 4 says there are no dependencies from the Core or Central group to the Periphery. But suppose there is a dependency from element *j* in the Periphery to element *i* in the Core or Central group. By definition, *i* lies above *j* in the DSM, thus the dependency would appear below the main diagonal. By transitive dependency, $VFO_j \geq VFO_C + 1$. But by definition, $VFO_j < VFO_C$, hence we have a contradiction. *QED*

## Appendix C: Different Sort Orders

The sort order *VFI* descending, *VFO* ascending is not unique in its ability to lower diagonalize and identify cyclic groups. Table 2 shows which combinations achieve both goals.

**Table 2**
**Properties of Different Sort Orders**

| First Sort | Second Sort | Finds Cycles | Lower-Diagonalizes |
|---|---|---|---|
| | | | |
| *VFI* descending | *VFO* ascending | √ | √ |
| | descending | √ | √ |
| *VFI* ascending | *VFO* ascending | √ | no |
| | descending | √ | no |
| *VFO* descending | *VFI* descending | √ | no |
| | ascending | √ | no |
| *VFO* ascending | *VFI* descending | √ | √ |
| | ascending | √ | √ |

Of the four sort orders that work, we use *VFI descending, VFO ascending* for the following reasons. A first sort by *VFI descending* places elements with many incoming dependencies at the top of the matrix. In contrast, a first sort by *VFO ascending* places elements with few dependencies, e.g., *VFI = VFO = 1* near the top.[7] Given a first sort by *VFI descending*, a second sort by *VFO ascending* places elements with many outgoing dependencies near or at the bottom of each *VFI* layer.[8] This reinforces the concept of dependencies flowing from lower to upper parts of the matrix.

[7] With a *VFI* descending first sort, such elements appear near the bottom.

[8] Note: A "layer" is a group of elements with the same *VFI*, but possibly different *VFOs*. By Proposition 3, elements within a layer cannot depend on each other unless they are part of a cycle. See Wong (2010) for another method of computing layers.

## Appendix D: List of Systems Analyzed

| | System Name | Function | Number of Versions | Origin | No. Files (Last Release) |
|---|---|---|---|---|---|
| 1 | Mozilla | Web Browser | 35 | Commercial | 5899 |
| 2 | OpenAFS | File Sharing | 106 | Open source | 1304 |
| 3 | GnuCash | Financial Management | 116 | Open source | 543 |
| 4 | Abiword | Word Processor | 29 | Open source | 1183 |
| 5 | Apache | Web Server | 52 | Open source | 481 |
| 6 | Chrome | Web Browser | 1 | Open source | 4186 |
| 7 | Linux (kernel) | Operating System | 544 | Open source | 8414 |
| 8 | MySQL | Database | 18 | Open source | 1282 |
| 9 | Ghostscript | Image Display and Conversion | 35 | Open source | 653 |
| 10 | Darwin | Operating System | 36 | Commercial | 5685 |
| 11 | Open Solaris | Operating System | 28 | Commercial | 12949 |
| 12 | MyBooks | Financial Management | 5 | Commercial | 2434 |
| 13 | PostGres | Database | 46 | Open source | 703 |
| 14 | XNU | Operating System | 43 | Open source | 781 |
| 15 | GnuMeric | Spreadsheet | 162 | Open source | 314 |
| 16 | Berkeley DB | Database | 12 | Commercial | 299 |
| | | | | | |
| 17 | Open Office | Productivity Suite | 6 | Commercial | 7360 |
| 18 | Write (Open Office) | Word Processor | 6 | Commercial | 814 |
| 19 | Calc (Open Office) | Spreadsheet | 6 | Commercial | 545 |
| | | | 1286 | | |

## References

Akaikine, Andrei (2010) "The Impact of Software Design Structure on Product Maintenance Costs and Measurement of Economic Benefits of Product Redesign," S.M. thesis, System Design and Management Program, Massachusetts Institute of Technology.

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.

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

N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka, **Managing Technical Debt in Software-Reliant Systems**, *FoSeR '10: Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research*, 47-52, Nov 2010.

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

Chidamber, S.R. and C.F. Kemerer (1994) "A metrics suite for object oriented design," *IEEE Transaction on Software Engineering*, 20(6): 476-493.

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

Cataldo, M., A. Mockus, J.A. Roberts and J.D. Herbsleb (2009) "Software Dependencies, Work Dependencies, and Their Impact on Failures," *IEEE Transactions on Software Engineering*, 35(6): 864-878.

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.

Colfer, Lyra J. and Carliss Y. Baldwin (2010) "The Mirroring Hypothesis: Theory, Evidence and Exceptions," Harvard Business School Working Paper No. 10-058, January 2010 (revised, June 2010), available at http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1539592 .

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.

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

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

Fixson, Sebastian K. and Jin-Kyu Park (2008). "The Power of Integrality: Linkages between Product Architecture, Innovation and Industry Structure," *Research Policy* 37(8):1296-1316.

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.

Philippe Kruchten, Robert L. Nord, Ipek Ozkaya: Technical Debt: From Metaphor to Theory and Practice. IEEE Software 29(6): 18-21 (2012)

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. Available at: http://www.people.hbs.edu/cbaldwin/DR2/LaMantia-Cai-MacCormack-Rusnak WICSA2008.pdf , viewed 4/16/13.

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

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

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

MacCormack, Alan, Carliss Baldwin and John Rusnak (2012) "Exploring the Duality Between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis," *Research Policy,* 41(8): 1309-1324.

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.*

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

Mead, Carver and Lynn Conway (1980) *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA.

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.

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

Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan: Making Architecture Visible to Improve Flow Management in Lean Software Development. IEEE Software 29(5): 33-39 (2012)

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.

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).

Sharman, David, Ali Yassine and Paul Carlile (2002). "Characterizing Modular Architectures," *Proceedings of the ASME 14th International Conference on Design Theory & Methodology*, DTM-34024, Montreal, Canada (September).

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, Jurgen Mihm and Tyson Browning (forthcoming) "Linking Cyclicality and Product Quality," *Manufacturing & Service Operations Mangement*.

Sosa, Manuel, Steven Eppinger and Craig Rowles (2004) "The Misalignment of Product Architecture and Organizational Structure in Complex Product Development," *Management Science*, 50(12):1674-1689

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

Stein, Clifford, Robert L. Drysdale and Kenneth Bogart (2011) *Discrete Mathematics for Computer Scientists*, Boston, MA: Addison-Wesley.

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).

Sturdevant, Daniel J. (2013) "System Design and the Cost of Architectural Complexity," Ph.D thesis, Engineering Systems Division, Massachusetts Institute of Technology.

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

Thompson, James D. (1967) *Organizations in Action: Social Science Bases of Administrative Theory*, New York, NY: McGraw-Hill.

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*

Wilkie, F. G. and B. A. Kitchanham (1999) "Coupling Measures and Change Ripples in C++ Application Software," *The Journal of Systems and Software*, 52(2000): 157-164.