DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

Citation	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) (October): 1309–1324. doi:10.1016/j.respol.2012.04.011.
Published Version	https://doi.org/10.1016/j.respol.2012.04.011
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:34403525
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Open Access Policy Articles, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#OAP

Exploring the Duality between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis

Corresponding Author:
Alan MacCormack
MIT Sloan School of Management
50 Memorial Drive E52-538
Cambridge MA 02142
alanmac@mit.edu

John Rusnak, Carliss Baldwin
Harvard Business School
Soldiers Field Park
Boston, MA 02163
cbaldwin@hbs.edu; jrusnak@hbs.edu

Abstract

A variety of academic studies argue that a relationship exists between the structure of an organization and the design of the products that this organization produces. Specifically, products tend to "mirror" the architectures of the organizations in which they are developed. This dynamic occurs because the organization's governance structures, problem solving routines and communication patterns constrain the space in which it searches for new solutions. Such a relationship is important, given that product architecture has been shown to be an important predictor of product performance, product variety, process flexibility and even the path of industry evolution.

We explore this relationship in the software industry. Our research takes advantage of a natural experiment, in that we observe products that fulfill the same function being developed by very different organizational forms. At one extreme are commercial software firms, in which the organizational participants are tightly-coupled, with respect to their goals, structure and behavior. At the other, are open source software communities, in which the participants are much more loosely-coupled by comparison. The mirroring hypothesis predicts that these different organizational forms will produce products with distinctly different architectures. Specifically, loosely-coupled organizations will develop more modular designs than tightly-coupled organizations. We test this hypothesis, using a sample of matched-pair products.

We find strong evidence to support the mirroring hypothesis. In all of the pairs we examine, the product developed by the loosely-coupled organization is significantly more modular than the product from the tightly-coupled organization. We measure modularity by capturing the level of coupling between a product's components. The magnitude of the differences is substantial – up to a factor of eight, in terms of the potential for a design change in one component to propagate to others. Our results have significant managerial implications, in highlighting the impact of organizational design decisions on the technical structure of the artifacts that these organizations subsequently develop.

Keywords: Organizational Design, Product Design, Architecture, Modularity, Open-Source Software.

1. Introduction

The architecture of a product can be defined as the scheme by which the functions it performs are allocated to its constituent components (Ulrich, 1995). Much prior work has highlighted the critical role of architecture in the successful development of a firm's new products, the competitiveness of its product portfolio and the evolution of its organizational capabilities (e.g., Eppinger et al, 1994; Ulrich, 1995; Sanderson and Uzumeri, 1995; Sanchez and Mahoney, 1996; Schilling, 2000; Baldwin and Clark, 2000; MacCormack, 2001). For any given set of functional requirements however, a number of different architectures might be considered viable. These designs will possess differing performance characteristics, in terms of important attributes such as cost, quality, reliability and adaptability. Understanding how architectures are chosen, how they are developed and how they evolve are therefore critical topics for academic research.

A variety of studies have examined the link between a product's architecture and the characteristics of the organization that develops it (Conway, 1968; Henderson and Clark, 1990; Brusoni and Prencipe, 2001; Sosa et al, 2004; Cataldo et al, 2006). Most examine a single project, focusing on the need to align team communications to the technical interdependencies in a design. In many situations however, these interdependencies are not predetermined, but are the product of managerial choices. Furthermore, how these choices are made can have a direct bearing on a firm's success. For example, Henderson and Clark (1990) show that leading firms in the photolithography industry stumbled when faced with innovations that required radical changes to the product architecture. They argue that these dynamics occur because designs tend to reflect the organizations that develop them. Given organizations are slow to change, the designs they produce can quickly become obsolete in a changing marketplace. Empirical evidence of such a relationship however, has remained elusive.

In this study, we provide evidence to support the hypothesis that a relationship exists between product and organizational designs. In particular, we use a network analysis technique called the Design Structure Matrix (DSM) to compare the design of products developed by different organizational forms. Our analysis takes advantage of the fact that software is an information-based product, meaning that the design comprises a series of instructions (or "source code") that tell a computer what tasks to perform. Given this

feature, software products can be processed automatically to identify the dependencies that exist between their component elements (something that cannot be done with physical products). These dependencies, in turn, can be used to characterize a product's architecture, by displaying the information visually and by calculating metrics that capture the overall level of coupling between elements in the system.

We chose to analyze software because of a unique opportunity to examine two distinct organizational forms. Specifically, in recent years there has been a growing interest in open source (or "free") software, which is characterized by: a) the distribution of a program's source code along with the binary version of the product[1] and; b) a license that allows a user to make unlimited copies of and modifications to this product (DiBona et al, 1999). Successful open source software projects tend to be characterized by large numbers of volunteer contributors, who possess diverse goals, belong to different organizations, work in different locations and have no formal authority to govern development activities (Raymond, 2001; von Hippel and von Krogh, 2003). In essence, they are "loosely-coupled" organizational systems (Weick, 1976). This form contrasts with the organizational structures of commercial firms, in which smaller, collocated teams of individuals sharing common goals are dedicated to projects full-time, and given formal decision-making authority to govern development. In comparison to open source communities, these organizations are much more "tightly-coupled." The mirroring hypothesis suggests that the architectures of the products developed by these contrasting forms of organization will differ significantly: In particular, open source software products are likely to be more modular than commercial software products. Our research seeks to examine the magnitude and direction of these differences.

Our paper proceeds as follows. In the next section, we describe the motivation for our research and prior work in the field that pertains to understanding the link between product and organizational architectures. We then describe our research design, which involves comparing the level of modularity of different software products by analyzing the coupling between their component elements. Next, we discuss how we construct a sample of matched product pairs, each consisting of one open source and one

[1] Commercial software is distributed in a binary form (i.e., 1's and 0's) that is executed by the computer.

commercially developed product. Finally, we discuss the results of our analysis, and highlight the implications for practitioners and the academy.

2. Research Motivation

The motivation for this research comes from work in organization theory, where it has long been recognized that organizations should be designed to reflect the nature of the tasks that they perform (Lawrence and Lorsch, 1967; Burns and Stalker, 1961). In a similar fashion, transaction cost economics predicts that different organizational forms are required to solve the contractual challenges associated with tasks that possess different levels of interdependency and uncertainty (Williamson, 1985; Teece, 1986). To the degree that different product architectures require different tasks to be performed, it is natural to assume that organizations and architectures must be similarly aligned. To date however, there has been little systematic empirical study of this relationship.

Research seeking to examine this topic has followed one of two approaches. The first explores the need to match patterns of communication within a development project to the interdependencies that exist between different parts of a product's design. For example, Sosa et al (2004) examined a single jet engine project, and found a strong tendency for communications to be aligned with key design interfaces. The likelihood of "misalignment" was shown to be greater when dependencies spanned organizational and system boundaries. Similarly, Cataldo et al (2006) explored the impact of misalignment in a single software development project, and found tasks were completed more rapidly when the patterns of communication between team members were congruent with the patterns of interdependency between components. Finally, Gokpinar et al (2006) explored the impact of misalignment in a single automotive development project, and found subsystems of higher quality were associated with teams that had aligned their communications to the technical interfaces with other subsystems.

The studies above begin with the premise that team communication must be aligned to the technical interdependencies between components in a system, the latter being determined by the system's functionality. A second stream of work however, adopts the reverse perspective. It assumes that an organization's structure is fixed in the short-term, and explores the impact of this structure on the technical designs that emerge. This idea

was first articulated by Conway who stated, "any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure" (Conway, 1968). The dynamics are best illustrated in Henderson and Clark's study of the photolithography industry, in which they show that market leadership changed hands each time a new generation of equipment was introduced (Henderson and Clark, 1990). These observations are traced to the successive failure of leading firms to respond effectively to *architectural* innovations, which involve significant changes in the way that components are linked together. Such innovations challenge existing firms, given they destroy the usefulness of the architectural knowledge embedded in their organizing structures and information-processing routines, which tend to reflect the current "Dominant Design" (Utterback, 1996). When this design is no longer optimal, established firms find it difficult to adapt.

The contrast between the two perspectives can be clarified by considering the dynamics that occur when two distinct organizational forms develop the same product. Assuming the product's functional requirements are identical, the first stream of research would assume that the patterns of communication between participants in each organization should be similar, driven by the nature of the tasks to be performed. In contrast, the second stream of research would predict that the resulting designs would be quite different, each reflecting the architecture of the organization from which it came. We define the latter phenomenon as "mirroring." A test of the mirroring hypothesis can be conducted by comparing the designs of "matched-pair" products – products that fulfill the same function, but that have been developed by different organizational forms. To conduct such a test, we must characterize these different forms, and establish a measure by which to compare the designs of products that they produce.

2.1 Organizational Design and "Loosely-Coupled" Systems

Organizations are complex systems comprising individuals or groups that coordinate actions in pursuit of common goals (March and Simon, 1958). Organization theory describes how the differing preferences, information, knowledge and skills of these organizational actors are integrated to achieve collective action. Early "classical" approaches to organization theory emphasized formal structure, authority, control, and

hierarchy (i.e., the division of labor and specialization of work) as distinguishing features of organizations, building upon work in the fields of scientific management, bureaucracy and administrative theory (Taylor, 1911; Fayol, 1949; Weber, 1947; Simon, 1976). Later scholars however, argued that organizations are best analyzed as *social* systems, given they comprise actors with diverse motives and values that do not always behave in a rational economic manner (Mayo, 1945; McGregor, 1960). As this perspective gained popularity, it was extended to include the link between an organization and the environment in which it operates. With this lens, organizations are seen as *open* systems, comprising "interdependent activities linking shifting coalitions of participants" (Scott, 1981). A key assumption is that organizations can vary significantly in their design; the optimal design for a specific mission is established by assessing the fit between an organization and the nature of the tasks it must accomplish (Lawrence and Lorsch, 1967).

Weick was the first to introduce the concept that organizations can be characterized as complex systems, comprising many elements with different levels of coupling between them (Weick, 1976; Orton and Weick, 1990). Organizational coupling can be analyzed along a variety of dimensions, however the most important of these fall into three broad categories: Goals, structure and behavior (Orton and Weick, 1990). Organizational structure, in turn, can be further decomposed to capture important differences in terms of membership, authority and location. All these dimensions represent a continuum along which organizations vary in the level of coupling between participants. When aligned, they generate two distinct organizational forms, representing opposite ends of this continuum (see **Table 1**). While prior work had assumed that the elements in organizational systems were coupled through dense, tight linkages, Weick argued that some organizations (e.g., educational establishments) were only loosely-coupled. Although real-world organizations typically fall between these "canonical types," they remain useful constructs for characterizing the extent to which organizations resemble one extreme or the other (Brusoni et al, 2001).

Table 1: Characterizing Different Organizational Forms

	Tightly-Coupled	**Loosely-Coupled**
Goals	Shared, Explicit	Diverse, Implicit
Membership	Closed, Contracted	Open, Voluntary
Authority	Formal, Hierarchy	Informal, Meritocracy
Location	Centralized, Collocated	Decentralized, Distributed
Behavior	Planned, Coordinated	Emergent, Independent

The software industry represents an ideal context within which to study these different organizational forms, given the wide variations in structure observed in this industry. At one extreme, we observe commercial software firms, which employ smaller, dedicated (i.e., full-time), collocated development teams to bring new products to the marketplace. These teams share explicit goals, have a closed membership structure, and rely on formal authority to govern their activities. At the other, we observe open source (or "free" software) communities, which rely on the contributions of large numbers of volunteer developers, who work in different organizations and in different locations (von Hippel and von Krogh, 2003). The participants in these communities possess diverse goals and have no formal authority to govern development, instead relying on informal relationships and cultural norms (Dibona et al, 1999). These forms of organization closely parallel the canonical types described above, with respect to the level of coupling between participants. They provide for a rich natural experiment, in that we observe products that perform the same function being developed in each.

2.2 Product Design, Architecture and Modularity

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

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

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

2.3 Software Design

The measurement of modularity has gained most traction in the software industry, given the information-based nature of the product lends itself to analytical techniques that are not possible with physical products. The formal study of software modularity began with Parnas (1972) who proposed the concept of information hiding as a mechanism for dividing code into modular units. Subsequent authors built on this work, proposing metrics to capture the level of “coupling” between modules and “cohesion” within

modules (e.g., Selby and Basili, 1988; Dhama, 1995). This work complemented studies that sought to measure the complexity of software, to examine its effect on development productivity and quality (e.g., McCabe 1976; Halstead, 1976). Whereas measures of software complexity focus on characterizing the number and nature of the elements in a design, measures of modularity focus on the patterns of dependencies *between* these elements. Software can be complex (i.e., have many parts) and modular (i.e., have few dependencies between these parts). In prior work, this distinction is not always clear.[2]

Efforts to measure software modularity generally follow one of two approaches. The first focuses on identifying specific types of dependency between components in a system, for example, the number of non-local branching statements (Banker et al, 1993); global variables (Schach et al, 2002); or function calls (Banker and Slaughter, 2000; Rusovan et al, 2005). The second *infers* the presence of dependencies by assessing which components tend to be changed concurrently. For example, Eick et al (1999) show that code decays over time, by looking at the number of files that must be altered to complete a modification request; while Cataldo et al (2006) show that modifications involving files that tend to change along with others, take longer to complete. While the inference approach avoids the need to specify the type of dependency being examined, it requires access to maintenance data that is not always captured consistently across projects. In multi-project research, dependency extraction from source code is therefore preferred.

With the rise in popularity of open source software, interest in the topic of modularity has received further stimulus. Some authors argue that open source software is inherently more modular than commercial software (O'Reilly, 1999; Raymond, 2001). Others have suggested that modularity is a required property for this method of development to succeed (Torvalds, as quoted in DiBona, 1999). Empirical work to date however, yields mixed results. Some studies criticize the number of dependencies between critical components in systems such as Linux (Schach et al, 2002; Rusovan et al, 2005). Others provide quantitative and qualitative data that open source products are easier to modify (Mockus et al, 2002; Paulsen et al, 2004) or have fewer interdependencies between components (MacCormack et al, 2006). None of these studies however, conducts a

[2] In some fields, complexity is defined to include inter-element interactions (Rivkin and Siggelkow, 2007).

rigorous apples-to-apples comparison between open source and commercially developed software; the results may therefore be driven by idiosyncrasies of the systems examined.

In this paper, we explore whether organizations with distinctly different forms – as captured by the level of coupling between participants – develop products with distinctly different architectures – as captured by the level of coupling between components. Specifically, we conduct a test of the "mirroring" hypothesis, which can be stated as follows: *Loosely-coupled organizations will tend to develop products with more modular architectures than tightly-coupled organizations*. We use a matched-pair design, to control for differences in architecture that are related to differences in product function. We build upon recent work that highlights how DSMs can be used to visualize and measure software architecture (Lopes and Bajracharya, 2005; MacCormack et al, 2006).

3. Research Methods[3]

There are two choices to make when applying DSMs to a software product: The unit of analysis and the type of dependency. With regard to the former, there are several levels at which a DSM can be built: The *directory* level, which corresponds to a group of source files that pertain to a specific subsystem; the *source file* level, which corresponds to a collection of related processes and functions; and the *function* level, which corresponds to a set of instructions that perform a specific task. We analyze designs at the source file level for a number of reasons. First, source files tend to contain functions with a similar focus. Second, tasks and responsibilities are allocated to programmers at the source file level, allowing them to maintain control over all the functions that perform related tasks. Third, software development tools use the source file as the unit of analysis for version control. And finally, prior work on design uses the source file as the primary unit of analysis (e.g., Eick et al, 1999; Rusovan et al, 2005; Cataldo et al, 2006).[4]

[3] The methods we describe here build on prior work in this field (see MacCormack et al, 2006; 2007).
[4] Metaphorically, source files are akin to the physical components in a product; whereas functions are akin to the nuts and bolts that comprise these components.

There are many types of dependency between source files in a software product.[5] We focus on one important dependency type – the "Function Call" – used in prior work on design structure (Banker and Slaughter, 2000; Rusovan et al, 2005). A Function Call is an instruction that requests a specific task to be executed. The function called may or may not be located within the source file originating the request. When it is not, this creates a dependency between two source files, in a specific direction. For example, if FunctionA in SourceFile1 calls FunctionB in SourceFile2, then we note that SourceFile1 depends upon (or "uses") SourceFile2. This dependency is marked in location (1, 2) in the DSM. Note this does not imply that SourceFile2 depends upon SourceFile1; the dependency is not symmetric unless SourceFile2 also calls a function in SourceFile1.

To capture function calls, we input a product's source code into a tool called a "Call Graph Extractor" (Murphy et al, 1998). This tool is used to obtain a better understanding of system structure and interactions between parts of the design.[6] Rather than develop our own extractor, we tested several commercial products that could process source code written in both procedural and object oriented languages (e.g., C and C++), capture indirect calls (dependencies that flow through intermediate files), run in an automated fashion and output data in a format that could be input to a DSM. A product called Understand C++[7] was selected given it best met all these criteria.

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

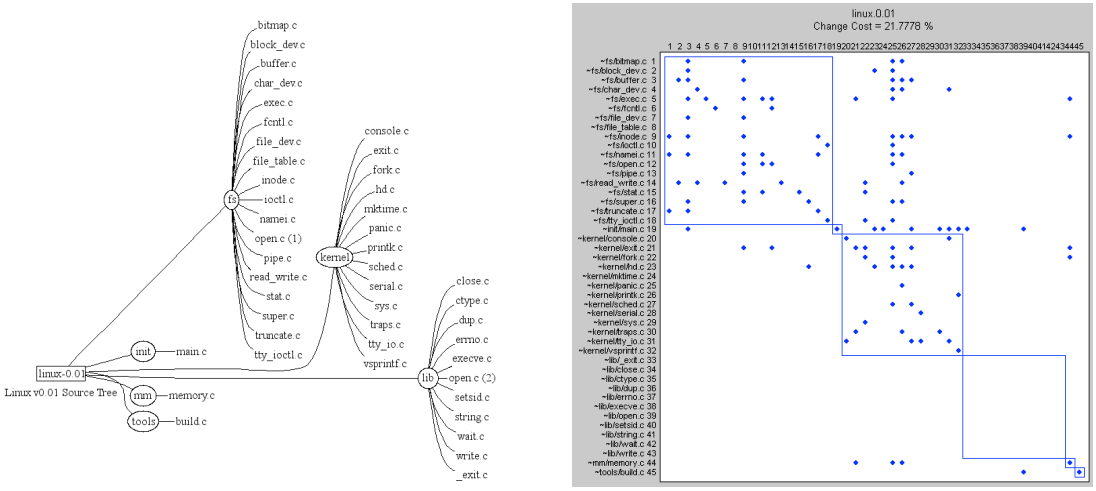
---

[5] Several authors have developed comprehensive categorizations of dependency types (e.g., Shaw and Garlan, 1996; Dellarocas, 1996). Our work focuses on one important type of dependency.

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

[7] Understand C++ is distributed by Scientific Toolworks, Inc. see <www.scitools.com> for details.

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

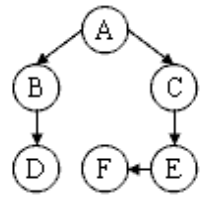


3.1 Measuring Product Modularity

The method by which we characterize the structure of a design is by measuring the level of coupling it exhibits, as captured by the degree to which a change to any single element causes a (potential) change to other elements in the system, either directly or indirectly (i.e., through a chain of dependencies that exist across elements). This work is closely related to and builds upon the concept of visibility (Sharmine and Yassine 2004), which in turn, is based upon the concept of reachability matrices (Warfield 1973).

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

Figure 2: Example System in Graphical and DSM Form



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

We use the technique of matrix multiplication to identify the "visibility" of each element for any given path length (see **Figure 3**). Specifically, by raising the dependency matrix to successive powers of n, the results show the direct and indirect dependencies that exist for successive path lengths of n. By summing these matrices together we derive the visibility matrix V, showing the dependencies that exist between all system elements for all possible path lengths up to the maximum – governed by the size of the DSM itself (denoted by N).[8] To summarize this data for the system as a whole, we compute the density of the visibility matrix, which we refer to as the system's *Propagation Cost*. Intuitively, this metric captures measures the percentage of system elements that can be affected, on average, when a change is made to a randomly chosen element.

Figure 3: The Derivation of the Visibility Matrix

M^0

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

M^1

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

M^2

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

M^3

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

M^4

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

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

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

[8] Note that we choose to include the matrix for n=0 meaning that each element depends upon itself.

4. Sample Construction and Analysis Approach

Our approach is based upon comparing the architectures of products that perform similar functions, which have been developed using two distinct organizational forms. To achieve this, we construct a sample of matched pairs, in which the same product has been developed using these different forms. Our matching process takes into account the function of the software (e.g., a spreadsheet application) as well as its level of sophistication (e.g., the functions that the spreadsheet performs). The latter is achieved by pairing products of similar size, thereby controlling for differences in architecture that are related to the different scope of functionality included in each product.

Developing an ideal sample proves difficult for two reasons. First, many open source software projects are small efforts, involving only one or two developers, hence are not representative of a "loosely-coupled" organization (Howison and Crowston, 2004). To address this problem, we focus only on successful open source software projects that meet a minimum threshold in terms of size and usage.[9] Only a small number of projects meet these criteria. The second challenge is that commercial firms regard source code as a form of intellectual property, hence are reluctant to provide access to it and cautious about research that seeks to compare it with "free" equivalents. Where a commercial product is not available, we therefore adopt two strategies. First, we try to identify a matched product that began its life inside a commercial firm, but that was subsequently released under an open source license. In these cases, we use the *first release* of the open version as a proxy for the architecture developed by the commercial firm. Second, where we have access to information that a small, dedicated organization developed a product, even if the software was released under an open source license, we use this as an observation for a tightly-coupled organization.

Table 2 reports the resulting sample of five paired products, with data on the organizations from which they come.[10] For each open source product, we extract data from the credits file (or feature log) to characterize the organization at the time the

[9] Use was determined by downloads and other data on the number of user installations. Size was measured using the number of source files in the product. After reviewing the universe of potential projects, we defined a threshold of 300 source files as being representative of a successful open source project.

[10] Note that we could not access data that matched several well-known open source products (e.g., Apache). We provide two possible matches for Linux, given the ideal commercial product – the Solaris operating system, developed by Sun – is significantly larger and more sophisticated than the open source product.

product was released. We report the number of unique contributors, the number of unique email domains (a proxy for the number of organizations represented, e.g., ibm.com) and the number of unique high-level email domains (a proxy for the number of countries represented, e.g., xx.xx.uk). The results show open source communities are loosely-coupled, in that they possess many contributors who work in different organizations and different locations. While comparable data for commercial products is not available, archival analysis suggests that these teams are smaller, staffed from within a single organization, and work predominantly at a single location (see **Appendix A** for details**)**. The contrast between these two organizational forms is therefore distinctive.

Table 2: Sample of Matched Pairs

Product Type	**Loosely-Coupled**[11]		**Tightly-Coupled**
1: Financial Mgmt	GnuCash 1.8.4	138 Contributors 76 Organizations 27 Countries	MyBooks
2: Word Processing	Abiword 0.9.1	160 Contributors 68 Organizations 27 Countries	StarWriter
3: Spreadsheet	Gnumeric 1.4.2	19 Contributors 9 Organizations 7 Countries	StarCalc
4a: Operating System	Linux 2.1.32	228 Contributors 64 Organizations 42 Countries	Solaris
4b: Operating System	Linux 2.6.8	515 Contributors 201 Organizations 54 Countries	XNU
5: Database	MySQL 3.20.32a[12]	60 Contributors N/A Organizations 25 Countries	Berkeley DB

Our sample contains only five pairs, however it provides sufficient statistical power to test our hypothesis in two respects. First, each matched pair represents an independent test of the hypothesis, given we can analyze data on visibility at the source file level, and

[11] Data for organizations and countries is based upon the unique email identifier for each contributor.
[12] MySQL does not have a credits file. Source for data: *The Economist*, March 16th, 2006.

conduct a test of differences between the two populations of components (i.e., the "basket" of open source files and the basket of commercially developed files). We can therefore make inferences about the differences between products *within* a pair. Second, our set of five matched pairs provides a test of the hypothesis *across* this population of large, successful open source projects and their commercial equivalents. This second test is conducted by considering the probability that five open source products will have a lower propagation cost than commercial equivalents merely by chance. [13]

We note that differences in the level of coupling between components could be driven by different choices, in terms of how much functionality and code to include in a source file. Hence it is important to check whether there are systematic biases in these measures across. **Table 3** contains data on the products in our sample, focusing on the number of source files, the number of lines of code and functions per source file, and the density of dependencies. The data reveal considerable heterogeneity. For example, in three cases, the commercial product has a significantly greater number of lines of code per source file, yet these differences are not replicated in terms of functions per source file. On balance, while there are notable differences between some product pairs, these differences are not consistent in magnitude or direction. It is therefore unlikely that they will unduly influence the results found across the sample.

[13] Assuming the null hypothesis, the chance of finding that the open source product is more modular than the commercial product in all five matched pairs is $(0.5)^5 = 0.03125$ ($p<0.05$). While our sample is not a true "random draw" from the populations of each type of organization, it is biased only to the degree that we require accessibility to the source code of products, and the existence of a matched-pair for analysis.

Table 3: Quantitative Data on Products

Product Type	Data[14]	Loosely-Coupled	Tightly-Coupled
1. Financial Mgmt	Source Files	466	471
	Lines of Code/File	613	1563
	Functions/File	17.3	19.4
	Dependency Density	1.4%	1.9%
2. Word Processing	Source Files	841	790
	Lines of Code/File	390	936
	Functions/File	12.3	22.7
	Dependency Density	0.5%	1.0%
3. Spreadsheet	Source Files	450	532
	Lines of Code/File	804	797
	Functions/File	22.2	25.9
	Dependency Density	1.6%	1.4%
4a. Operating System	Source Files	6675	12081[15]
	Lines of Code/File	666	572
	Functions/File	14.5	9.9
	Dependency Density	0.1%	0.1%
4b. Operating System	Source Files	1032	994
	Lines of Code/File	726	677
	Functions/File	13.2	14.3
	Dependency Density	0.6%	0.7%
5. Database	Source Files	465	344
	Lines of Code/File	213	540
	Functions/File	4.6	9.6
	Dependency Density	0.9%	1.9%

5. Empirical Results

We report the results of our comparisons in **Table 4**. To evaluate whether the designs in each pair differ significantly, we calculate visibility for each source file in each product, and then compare the two populations using a Mann-Whitney-Wilcoxon (MWW) test of differences in means.[16] The mean level of visibility across all source files in a product is, by definition, the propagation cost of the system and can be read from the table. We report the MWW test statistic (U), z-score and significance level for each pair. Given visibility is not symmetric – in-degree and out-degree measures differ for each file

[14] Note that all the systems in our sample are programmed in C and C++. We report data for C and C++ files only. We do not include "Header files" which are far smaller in terms of code and functionality.
[15] Solaris is significantly bigger than any version of Linux that exists. Hence we report data on a second operating system – XNU – that is comparable to Linux in terms of size.
[16] We use this non-parametric test of differences because the distribution of visibility data among source files is skewed; hence a simple t-test would be inaccurate.

– we report separate tests for each measure. We find statistically significant differences between all pairs. The direction of the differences supports our hypothesis in each case.

Table 4: Propagation Cost Measures for each Matched Pair

Product Type	Loosely-Coupled	Tightly-Coupled	MWW Test Statistic	
			In-Degree	Out-Degree
1: Financial Mgmt	7.74%	**47.14%**	U=194247 z = 12.6 (p<0.1%)	U=189741 Z = 11.6 (p<0.1%)
2: Word Processing	8.25%	**41.77%**	U=410832 z = 8.3 (p<0.1%)	U=549546 Z = 22.9 (p<0.1%)
3: Spreadsheet	23.62%	**54.31%**	U=174030 z = 12.3 (p<0.1%)	U=180024 Z = 13.6 (p<0.1%)
4a: Operating System	7.18%	**22.59%**	U=49.4Mn z = 25.6 (p<0.1%)	U=65.0Mn z = 69.5 (p<0.1%)
4b: Operating System	7.21%	**24.83%**	U=594522 z = 6.2 (p<0.1%)	U=786574 Z = 20.8 (p<0.1%)
5: Database	11.30%	**43.23%**	U=90814 z = 3.3 (p<1.0%)	U=126564 Z = 14.1 (p<0.1%)

The DSMs for each matched pair are shown in **Appendix B**. Below, we use these visual comparisons, in conjunction with the data on propagation cost, to discuss the insights revealed by each comparison. We then examine the third product pair in further detail, given that the propagation cost of the open source product in this pair is significantly higher than any of the other open source products.

In pair number one, we see distinct differences in architecture. The open source product is divided into many smaller modules, with few dependencies between them. The exception is one block of files in the center that are called by much of the rest of the system, a structure we call a "bus," given it delivers common functionality to many components. By comparison, the commercial product has one large central module, within which there are many interdependencies between components. The system's propagation cost is 47.1%, in contrast to the open source product, which is 7.7%.

In pair number two, the visual differences are not as clear. Each product is divided into many smaller modules of a similar size. However, the commercial product has twice

the density of dependencies – 1% versus 0.5% – and these dependencies are spread throughout the system, rather than being concentrated within a few parts. As a result of this pattern, the propagation cost of this system exceeds 41%, in contrast to the open source product, which has a propagation cost of only 8.25%.

In pair number three, the hypothesis is again supported. We note however, that the open source product has a much higher propagation cost – over 23% - than any other open source product in our sample. The open source product has a larger density of dependencies than the closed source product. Many of these dependencies are to a group of files located within the largest module, which surprisingly, have not been grouped within a separate sub-module. By contrast, the closed source product possesses a more hierarchical structure, with a few top-level modules, within which are a number of smaller sub-modules. Despite having a lower dependency density, this product has a very high propagation cost, suggesting it is the *pattern* of dependencies, and not the number of them, that drives the high level of coupling between components.

In our fourth product category, we examine two matched pairs. In the first, which compares Linux with Solaris, the hypothesis is supported. The propagation cost of Solaris is over 22%, a significant number given the system's size. The figure implies that, on average, a change to a source file has the potential to impact over 2,400 other files. By contrast, the figure for Linux is around 7%. While still large in absolute terms, the difference between the two systems is significant, especially with regard to contributors choosing between the two. Our results suggest that contributing to Linux is far easier, all else being equal, than contributing to the "open" version of Solaris.

The comparison above is not ideal in that Solaris is significantly larger than Linux, consisting of twice as many source files. The differences in architecture may therefore be driven, in part, by differences in the functionality these systems provide.[17] To address this issue, we look at a second matched product – XNU – and compare it to a version of Linux of similar size. The result is consistent with that of Solaris. The propagation cost of XNU is just over 24%, in comparison to 7.4% for a version of Linux of similar size.

[17] Note that in every code base we have analyzed, propagation cost tends to remain constant or decline as a system grows in size. This is a product of the fact that the rate of dependency addition is often lower than the rate of growth in component pairs, hence the density of the visibility matrix declines with size. This dynamic biases the test *against* our hypothesis when comparing Linux and Solaris.

Of note, the structure of these products appears visually similar. The density of dependencies is also comparable. Once again, this result suggests it is the pattern of dependencies in XNU that drives its higher propagation cost. This pattern generates a higher number of *indirect* links between system components.

In pair number five, the hypothesis is again supported. This pair is unusual in that the commercial product comprises a large number of very small modules (i.e., it has a "flat" hierarchy). It may therefore appear more modular from the architect's viewpoint. However, the number and pattern of dependencies between source files is such that the product has a very high propagation cost exceeding 43%. By comparison, the open source product contains an additional layer of hierarchy, with several sub-modules nested within a larger module. Combined with its lower density, this design has a propagation cost of only 11.3%.

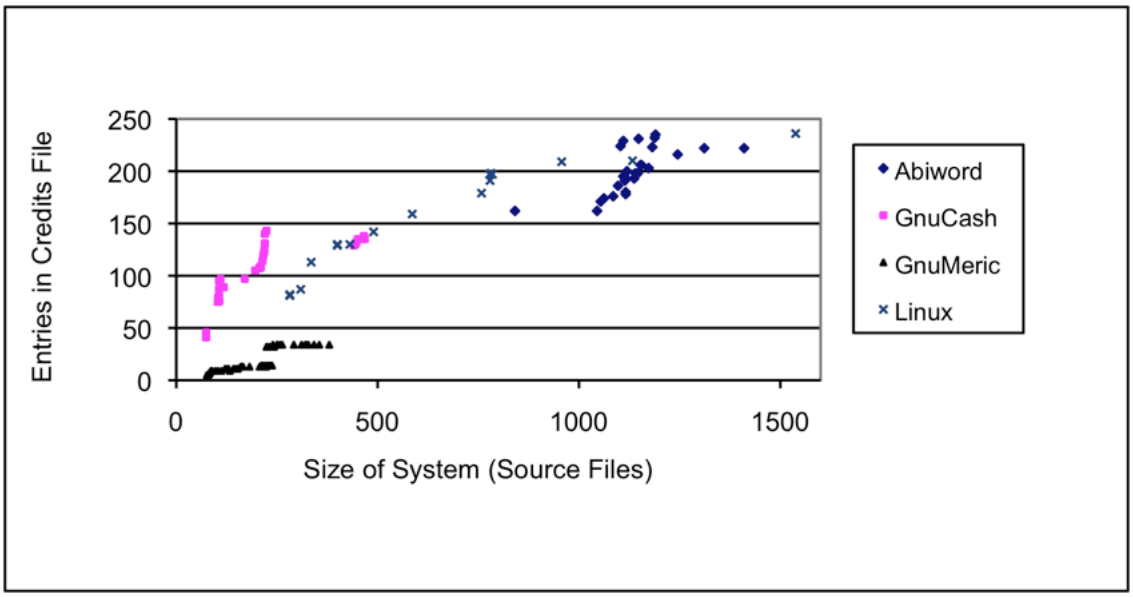
5.1 Exploring the High Propagation Cost in Gnumeric

While the hypothesis is supported in all the pairs we examine, there is one anomaly within the group of open source products. Specifically, Gnumeric has a significantly higher propagation cost than all other open source products. One possible explanation is that spreadsheet applications require more integral architectures, and hence both open and commercially developed products have higher propagation costs relative to other types of product. Alternatively however, Gnumeric may not, in fact, be developed by a loosely-coupled organization. To explore this possibility, we examine the number of contributors for GnuMeric in comparison to other open source projects. We gather data from two sources: the credits file and the change log. The credits file is a list of individuals who have contributed to a system's development. Each individual's name is listed once, and once added is generally never removed. The change log is a detailed listing of each change made to the product in each version. The change log from GnuMeric identifies the individuals who developed the code being added/changed.[18]

[18] We do not use the Concurrent Versioning System (CVS) system for our analysis, a tool that is sometimes used to control submissions of source code in a project. In many projects, contributions are batched together and submitted by a few individuals who have "access control." But these individuals are not always the authors of the code they submit. Using a change log overcomes this limitation.

To capture the *number* of contributors, we developed a script to count how many names appeared in the credit file of each open source product in our study. We captured this data for multiple versions, creating a plot of the size of the credits file as the system grows in size. **Figure 4** displays the result. GnuMeric has a much smaller number of credits file entries than open source products of a similar size. By contrast, Linux, AbiWord and Gnucash all have similar patterns of contributor growth, having three to five times as many credits file entries, adjusted for size.[19]

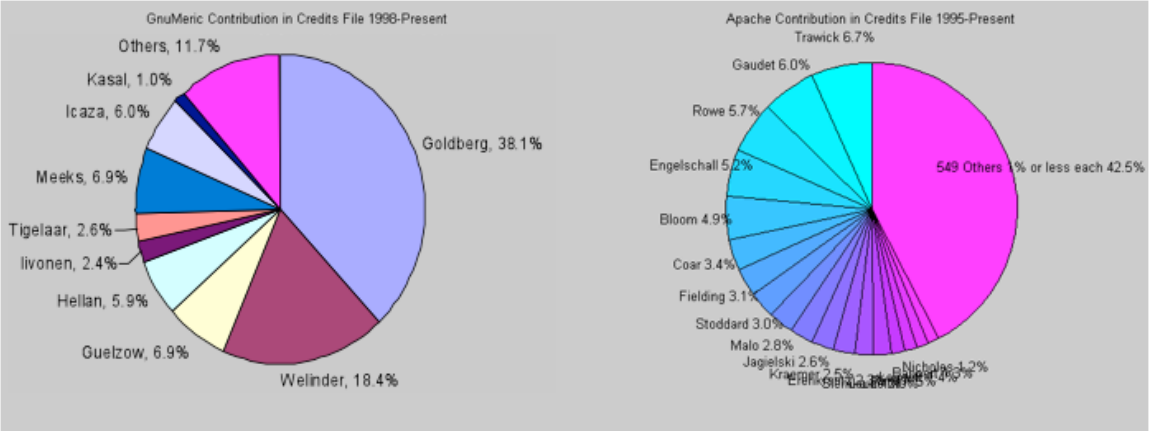
Figure 4: Number of Credits File Entries for Open Source Products



To capture the *extent* of each individual's contributions we developed a script to count how many times each unique name appeared in Gnumeric's change log, providing a proxy for the number of submissions attributable to each. For comparison, we conducted the same analysis for an open source project with similar data, and for which the system was known to have a low propagation cost: the Apache web server.[20] Results are shown in **Figure 5**. The contrast is clear. In Gnumeric, one individual accounts for almost 40% of changes, the top four for ~70% and the top 9 for ~90%. In Apache, the top individual accounts for less than 7% of changes and the top four less than 25%.

[19] Note that MySQL dos not have a credits file of equivalent structure to the other open source products.
[20] The propagation cost for the version of Apache web server closest in size to Gnumeric is less than 1%.

Figure 5: Developer Contributions for GnuMeric (left) and Apache (right)



In **Table 5**, we plot the GnuMeric contributor data by year. This data highlights that the pattern of contributions has been consistent throughout the project's life. In any given year, development activity is concentrated within a few key individuals.

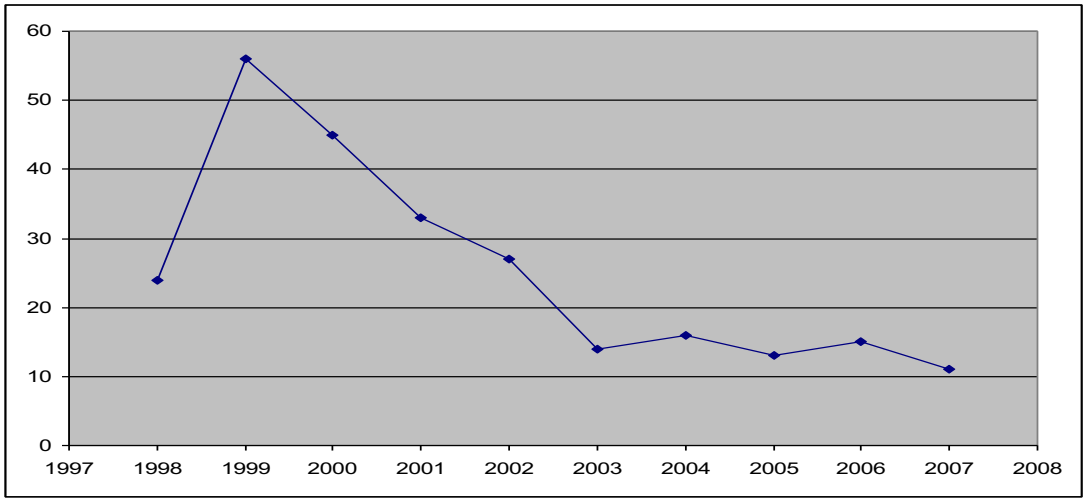
Table 5: Developer Contributions for GnuMeric by Year (1998-2004)

	2007	2006	2005	2004	2003	2002	2001	2000	1999	1998	Total
Goldberg	29.2%	32.2%	35.4%	48.7%	43.9%	41.2%	55.0%	46.2%	14.2%		38.1%
Welinder	37.9%	46.4%	32.8%	25.6%	22.0%	18.8%	10.8%	9.5%	11.9%	1.0%	18.4%
Guelzow	18.5%	3.8%	0.3%	9.0%	21.8%	17.9%	3.4%				6.9%
Hellan	2.0%	5.0%	6.8%	5.3%	5.9%	13.0%	4.4%	8.2%	0.9%		5.9%
Iivonen						0.9%	0.6%	3.7%	10.2%		2.4%
Tigelaar						0.2%	11.5%	4.4%			2.6%
Meeks						0.1%	0.2%	10.8%	30.6%	7.7%	6.9%
Icaza							0.1%	5.3%	20.7%	58.8%	6.0%
Kasal		0.3%	11.9%	2.5%							1.0%
Others	12.4%	12.3%	12.9%	8.8%	6.3%	7.9%	14.1%	12.0%	11.5%	32.5%	11.7%
Total	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

In **Figure 6**, we plot the number of unique contributors in the change log by year. This yields a fascinating insight. While the number of contributors increased in the first year of the project, the trend thereafter was one of decline. From a peak of over 50 people, the number falls to between 10-15.[21] In combination, these data suggest that Gnumeric is not developed by a loosely-coupled organization. Rather, development is concentrated among a few developers. Hence the higher propagation cost we observe relative to other open source projects is consistent with the mirroring hypothesis.

[21] The tightly-integrated nature of Gnumeric's design may a major factor in explaining the declining trend in participation. Developers contributing to Gnumeric need to understand far more of the system than in other open source projects, in order to ensure their changes do not affect other parts.

Figure 6: The Number of Unique Contributors to GnuMeric over time



6. Discussion

This study makes an important contribution to the academy and practicing managers. We find strong evidence to support the hypothesis that a product's architecture tends to mirror the structure of the organization in which it is developed. In all the pairs we examine, the loosely-coupled organization develops a product with a more modular design than that of the tightly-coupled organization. Furthermore, the open source software product with the highest propagation cost comes from an organization that is more tightly-coupled than the typical open source project. The results have important implications, in that we show a product's architecture is not wholly determined by function, but is influenced by contextual factors. The search for a new design is constrained by the nature of the organization within which this search occurs.

The differences in levels of modularity within each product pair are surprisingly large, especially given each matches products of similar size and function. We find products vary by a factor of eight, in terms of the potential for a design change in one component to propagate to other system components. Critically, these differences are not driven by differences in the number of *direct* dependencies between components – in only three of the pairs does the tightly-coupled organization produce a design with significantly higher density (see **Table 3**). Rather, each direct dependency gives rise to many more *indirect* dependencies in products developed by tightly-coupled organizations, as compared to those developed by loosely-coupled organizations.

The mirroring phenomenon is consistent with two rival causal mechanisms. First, designs may *evolve* to reflect their development environments. In tightly-coupled organizations, dedicated teams employed by a single firm and located at a single site develop the design. Problems are solved by face-to-face interaction, and performance "tweaked" by taking advantage of the access that module developers have to information and solutions developed in other modules. Even if not an explicit managerial choice, the design naturally becomes more tightly-coupled. By contrast, in loosely-coupled organizations, a large, distributed team of volunteers develops the design. Face-to-face communications are rare given most developers never meet. Hence fewer connections between modules are established. The architecture that evolves is more modular as a result of the limitations on communication between developers.

Alternatively, our observations may stem from *purposeful choices* made by designers. For commercial firms, the main aim is to develop a product that maximizes performance (e.g., speed, functionality, etc.). The benefits of modularity, given the competitive context, may not be viewed as significant. In open source projects however, the benefits of modularity are greater. Without modularity, there is little hope that contributors can understand enough of a design to contribute to it, or develop new features and fix defects without affecting other parts of the system. Open source products need to be modular to attract a developer community and to facilitate the work of this community.

While our data can be explained by either of these mechanisms, in practice, both likely work in parallel. This is particularly true in this industry, given software products rarely die. Instead, each version becomes a platform upon which subsequent versions are built. Early design choices can have a profound impact on a system, however, as the system grows, the organizational form through which development occurs likely plays an increasingly prominent role. For example, when Linus Torvalds released the first version of Linux in 1991, his architectural design choices were critical to its early success. Twenty years later however, with 95% of the code having been developed by others, the nature of the design likely owes much more to its organizational heritage.

Building on this argument, we believe that early design choices play a critical role in explaining the evolution of successful open source software projects. In particular, assuming the availability of a large pool of "seed" designs with a variety of different

architectures, potential contributors operate as a "selection environment," choosing only the most modular of them to work on, thereby minimizing the costs in making their contributions. Such a mechanism explains why large, successful open source projects are associated with more modular products, even though the set of initial designs they emerge from may be just as heterogeneous as those developed by commercial firms. Conversely, inside a firm, there are rarely alternative designs from which to select – resources are usually too scarce. Furthermore, the selection environment is such that modularity may not be the most critical criterion for filtering. Instead, measures associated with product performance (e.g., speed or memory usage) are likely to be the priority. Such a process might even lead firms to select *against* more modular designs.

We note that the pairs we examine were not developed contemporaneously; open source projects are often started after a product category has reached maturity. Hence our results could be explained, in part, by learning that occurs between the dates of release. While commercial firms restrict access to source code, preventing the copying of code, general knowledge about a product category might still benefit later projects. In this respect, it is useful to consider the comparison of Linux and Solaris, operating systems developed at similar times, each of which drew upon the heritage of UNIX (a system developed at AT&T Bell Labs in the 1960's). Despite the conceptual foundations shared by these projects, they were organized very differently. Solaris was developed inside a commercial firm – Sun Microsystems – headquartered in Silicon Valley. By contrast, the first version of Linux, comprising only 45 components, was posted on the Internet in 1991, and thereafter gained a huge contributor-base worldwide. Despite the common ancestry of these systems, and the objectives they shared, these organizations developed in very different ways. Ultimately, they produced very different designs with respect to the levels of modularity observed.

Our work suggests that managers of the innovation process must strive to understand the influences on their design choices that stem from the way they are organized. These influences are seldom explicit, but are a result of the interplay between a firm's problem solving and information processing routines, and the space of designs that must be searched to locate a solution. While a firm can look backwards, and see what kinds of designs it has developed in the past, it is harder to look forward, and imagine what

designs might be possible. Managers also face a cognitive problem, in that the architecture of a system depends critically on the *indirect* dependencies in a design, which are difficult to observe using existing tools and techniques. Indeed, the commercial managers we work with almost always think their designs are highly modular. Unfortunately, the pristine black boxes they draw on their whiteboards rarely reflect the *actual* file-to-file interactions embedded in the source code.

More broadly, our findings have important implications for R&D organizations given the recent trend towards "open" innovation and the use of partners in R&D projects (Brusoni and Prencipe, 2001; Chesbrough, 2003; Iansiti and Levian, 2004; MacCormack et al, 2007). They imply that these new organizational arrangements will have a distinct impact on the designs they produce, and hence may affect product performance in unintended ways. Our results suggest that R&D partnering choices, and the division of tasks that these choices imply, cannot be managed independently of the design process itself (von Hippel, 1990; Brusoni et al, 2001). Decisions taken in one realm will ultimately affect performance in the other, suggesting the need for a more integrated approach to project, product and procurement management.

Our study has several limitations that must be considered in assessing the generalizability of our results. First, our work is conducted in the software industry, a unique context in which designs exist purely as information, without physical constraints. Whether the results hold for physical products requires confirmation. Second, our sample comprises only five matched pairs, a limitation that stems from the dearth of successful open source products of sufficient size and complexity, and the difficulty accessing commercial source code that firms regard as intellectual property. Further large-scale testing would help discern the conditions under which mirroring holds from those where this link is weaker. Finally, we do not directly test the functional equivalence of the pairs we analyze, comparing products only of a similar size. While there are no consistent differences in the *number* of functions across pairs (see **Table 3**) some of the results we observe may still be associated with differences in the functionality of sample products.

Our work opens up a number of areas for future study. Most importantly, our work highlights differences in product design that emerge from two distinct organizational forms that differ along many dimensions – including the degree to which participants

share common goals, work at the same location, and possess formal authority over others to govern development. Yet our research design does not disentangle which of these dimensions are most critical in driving the phenomenon. Further work could usefully adopt a much more fine-grained definition of organizational design parameters, in order to determine those with the most effect on product design decisions.

Our results also provoke questions as to whether there are performance implications from the design differences that we observe. For example, one reason that organizations might rationally choose different designs relates to the trade-offs that exist between architectures with different characteristics. Unfortunately, there is little empirical evidence to confirm the existence or magnitude of these trade-offs. Does achieving greater modularity require accepting lower performance along other important dimensions? Our ongoing work suggests that, in practice, many designs may not be at the performance "frontier" where such a trade-off exists, but sit below it due to architectural inefficiencies or "slack" (MacCormack et al, 2006). If this is true, there may be scope to improve a design along multiple dimensions without any penalty. Exploring these issues will help reveal managerial strategies for moving designs *towards* the frontier. And they will help to understand the trade-offs involved in moving *along* it.

Appendix A: Details for Sample Organizations

Loosely-Coupled Organizations

GnuCash[22]

GnuCash is an open source financial management application. The GnuCash project was started by developer Rob Clark in 1997, with the goal of providing "an easy to use, yet powerful utility to help keep your finances in order." In 1998, developer Linas Vepstas joined him, and the 1.0 version of Gnucash was released under an open license. As of version 1.8.4, GnuCash had 138 unique contributors in the credits file.

Abiword[23]

Abiword is an open source word processing application. It was originally part of an effort to develop an open source productivity suite, called AbiSuite. The project was started in 1998 by a firm called AbiSource, which ceased operations one year later. Thereafter, a team of volunteers took over development. One team member stated in 2008, "We have no policies whatsoever. Everyone is free to work on whatever he or she loves." As of version 0.9.1, Abiword had 160 unique contributors in the credits file.

GnuMeric[24]

Gnumeric is an open source spreadsheet application. It was originally developed by developer Miguel de Icaza, who released the 1.0 version in December 2001. Note that Gnumeric's change log does not contain developer emails, hence we use the "feature log" for quantitative data reported in the text. As of version 1.4.2, GnuMeric had 19 unique contributors listed in the credits file.

Linux Kernel[25]

Linux is an open source operating system. A typical release consists of several hundred software packages, at the center of which is the "kernel." The first version of the Linux kernel was developed by developer Linus Torvalds in 1991, as part of an effort to develop a "UNIX-like" operating system that ran on Intel-based computers. As of version 2.1.32, Linux had 228 unique contributors in the credits file. As of version 2.6.8, Linux had 515 unique contributors in the credits file.

MySQL[26]

---

[22] Source: Clark, R. X-accountant, www.cs.hmc.edu/~rclark/xacc/ accessed September 30th, 2010; Gnucash developers & contributors, www.gnucash.org/old-website/developers.php3, accessed November 18th, 2010.
[23] Source: Sundaram, R. Abiword team interview, *Red Hat Magazine*, May 8th, 2008.
[24] Source: Wikipedia entry, "Gnumeric," accessed November 18th, 2010.
[25] Source: MacCormack, A. and Herman. Red Hat and the Linux Revolution, *Harvard Business School Case*, No. 600-009, 1999.
[26] Source: Wikipedia entry, "MySQL," accessed September 30th, 2010.

MySQL is an open source relational database. The development of MYSQL was begun by developers Michael Widenius and David Axmark in 1994, with the first release coming in 1995. The source code is released under an open source license. A Swedish firm called MySQL AB sponsors the project, selling software and services to enterprise customers. MySQL has no credits file in the source code (a web-based database names all major developers as of the present day). As of 2006, published reports said that MySQL involved 60 developers from 25 nations, 70% of whom work from home.[27] MySQL AB was bought by Sun Microsystems in 2008, which in turn was bought by Oracle in 2009.

Tightly-Coupled Organizations

MyBooks

MyBooks is a commercial application for managing a consumer's personal finances. The software was introduced in the mid 1980's. In the 1990s, it was ported to other operating system platforms, with new versions released every 2-4 years. Early versions of the software, including the one we analyze from the early 1990s, were developed by a "handful" of programmers working at a single location.[28]

StarWriter[29]

StarWriter is the word processing application from an office productivity suite called StarOffice. StarOffice was developed by StarDivision, a firm founded in Hamburg, Germany in 1984. Originally developed for the Microsoft DOS platform, StarOffice was ported to Windows between 1995-1996. StarDivision was purchased by Sun Microsystems in 1999 for $73.5 million. At the time, StarDivision had between 150-200 employees. The StarOffice suite was renamed OpenOffice and released under an open source license. Sun continued to fund development of applications in the suite, using 50 former StarDivision employees, located in Germany.

StarCalc[30]

StarCalc is the spreadsheet application from StarOffice (see above).

Solaris[31]

Solaris is a UNIX-based operating system introduced by Sun Microsystems in 1992. Solaris superseded SunOS, Sun's earlier operating system, which was developed with

[27] Source: *The Economist*, March 16th, 2006.
[28] Source: Personal communication with company developer, 2007.
[29] Sources: CNET news, August 20th, 1999; CNET news, August 30th, 1999WindowsITPro, August 31st, 1999; ComputerWorld, 19th April 2005. Wikipedia entry, "Oracle Open Office," accessed September 29th, 2010.
[30] Source: Ditto
[31] Sources: Wikipedia entry, "Solaris (operating system)," accessed November 18th, 2010; Kranz, T. The History of Solaris, content4reprint.com/computers/ operating_systems/ accessed September 30th, 2010.

AT&T. New versions of Solaris were typically released annually, until Solaris 10, the last version as of November 2010. In 2005, Sun released much of the Solaris source code under an open source license, and founded the OpenSolaris project to facilitate external contributions to the product. When Oracle bought Sun in 2009 however, it terminated the OpenSolaris distribution and project.

XNU[32]

XNU is an operating system kernel from Apple's Mac OSX operating system. XNU was originally developed by a computer company called NeXT in the late 1980s, as part of the NeXTSTEP operating system. NeXT was founded in 1985, and introduced its first computer workstation in 1989. XNU combined the "Mach" microkernel, developed at Carnegie-Mellon University, with software components developed by NeXT. NeXT withdrew from the hardware business in 1993, shedding 300 of its 540 staff. NeXT was acquired by Apple in 1996. Apple subsequently released the source code for many components of its operating system under the name Darwin. XNU is part of Darwin.

Berkeley DB

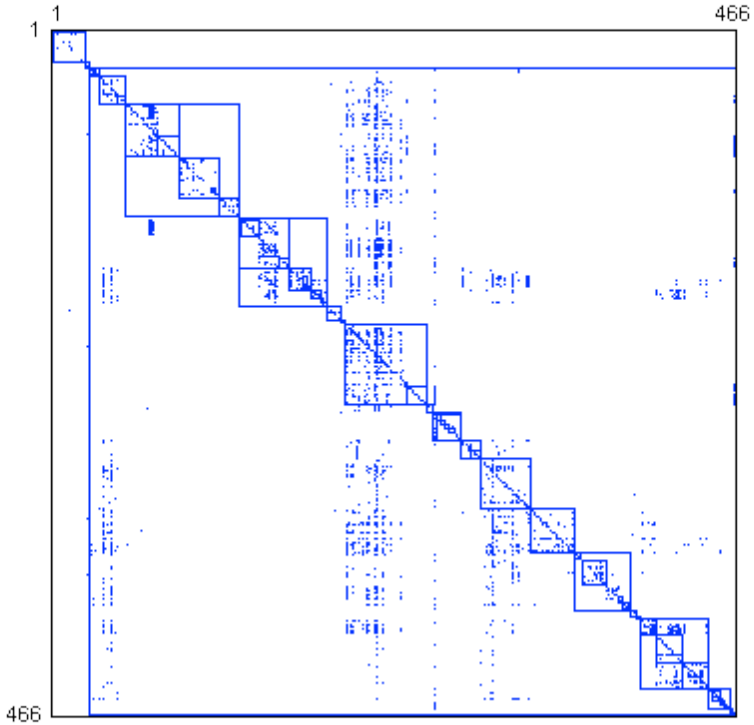
Berkeley DB is an embedded database. It originated at the University of California, Berkeley, from code developed by developers Margo Seltzer and Ozan Yigit in 1991.[33] In 1996, Netscape requested that the authors improve and extend the software hence Seltzer and Keith Bostic created a commercial firm called Sleepycat Software, to develop and commercialize the product.[34] While the product was distributed using an open source license, ongoing development was limited to a handful of developers who worked closely together.[35] Given this method of development, there is no credits file for Berkeley DB.

[32] Sources: Wikipedia entries, "XNU" and "NeXT," accessed November 18th, 2010.
[33] Source: Seltzer and Yigit, A New Hashing Package for UNIX, *Proceedings of the 1991 Winter Usenix*.
[34] Source: Wikipedia entry, "Berkeley DB," accessed September 29th, 2010.
[35] Source: Personal communication with company founder, 2007.

APPENDIX B: COMPARISON OF DSMs FOR EACH PRODUCT PAIR

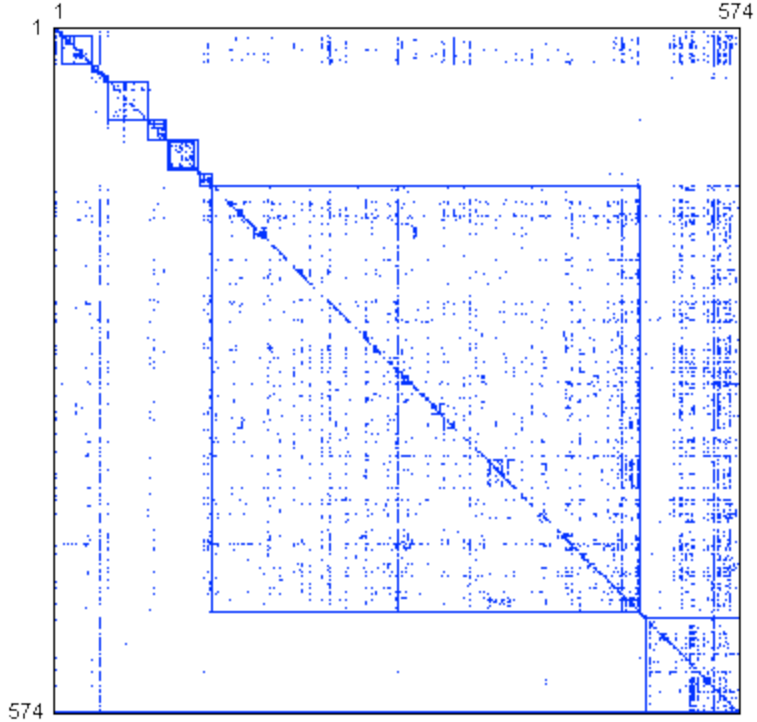
Matched Pair 1: Financial Management Software

Gnucash 1.8.4



Size = 466
Dependency Density = 1.3672%
Propagation Cost = 7.7428%

MyBooks

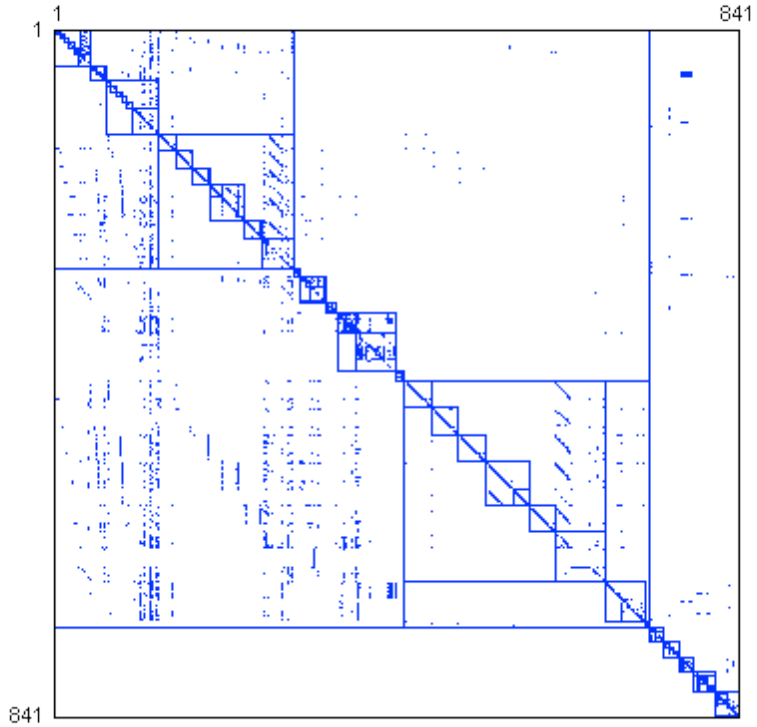


Size = 574
Dependency Density = 1.8903%
Propagation Cost = 47.1394%

APPENDIX B: COMPARISON OF DSMs FOR EACH PRODUCT PAIR

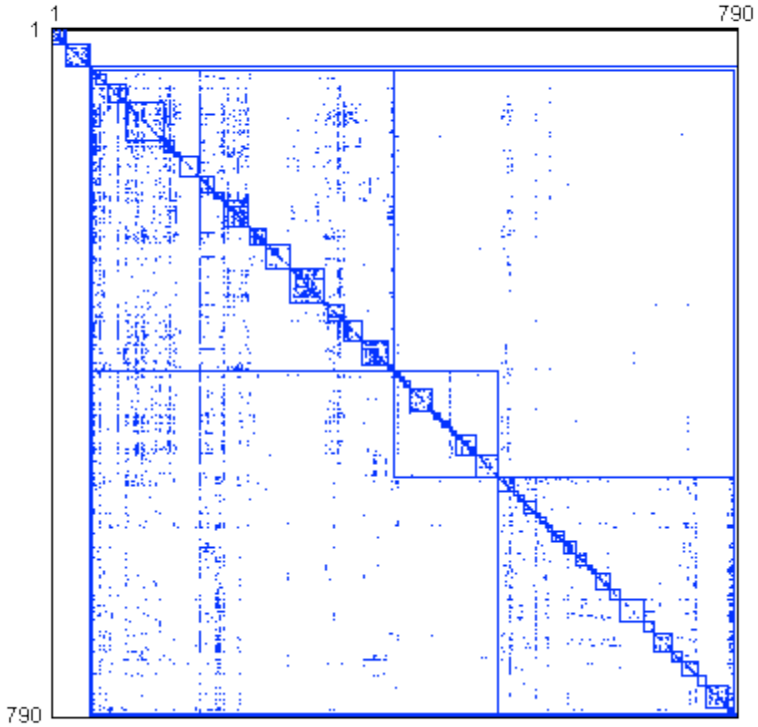
Matched Pair 2: Word Processing Software

Abiword 0.9.1



Size = 841
Dependency Density = 0.51832%
Propagation Cost = 8.2524%

OpenOfficeWrite 1.0

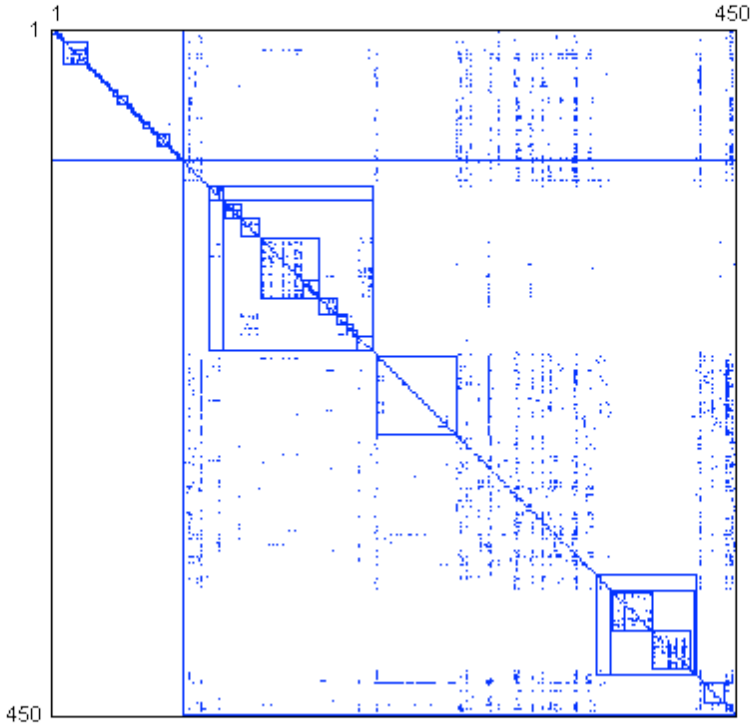


Size = 790
Dependency Density = 1.0276%
Propagation Cost = 41.7699%

APPENDIX B: COMPARISON OF DSMs FOR EACH PRODUCT PAIR

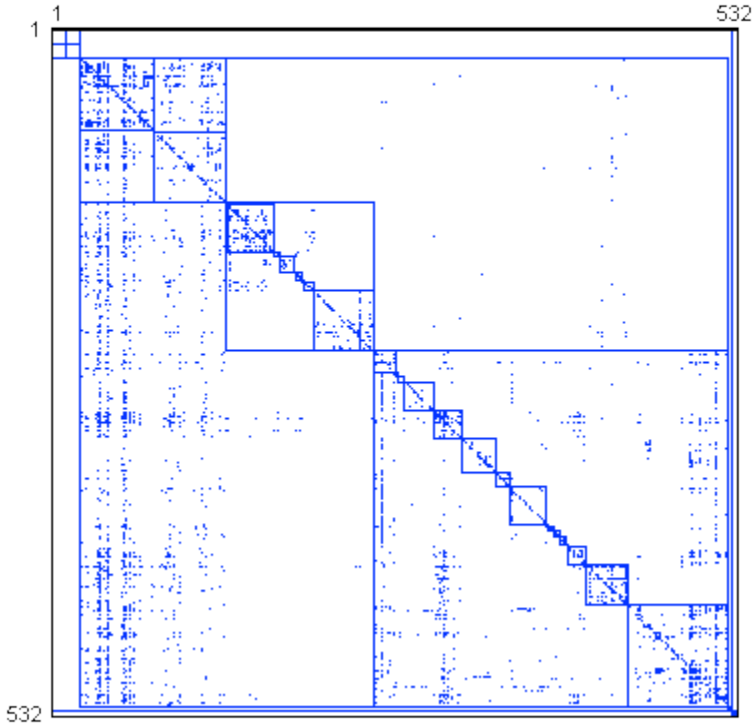
Matched Pair 3: Spreadsheet Software

GnuMeric 1.4.2



Size = 450
Dependency Density = 1.6119%
Propagation Cost = 23.6222%

OpenOfficeCalc 1.0

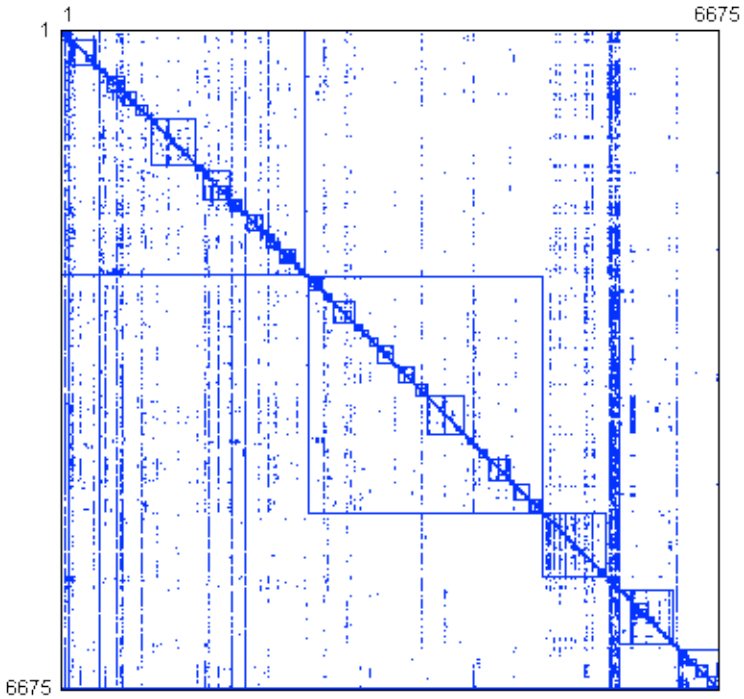


Size = 532
Dependency Density = 1.3773%
Propagation Cost = 54.3071%

APPENDIX B: COMPARISON OF DSMs FOR EACH PRODUCT PAIR

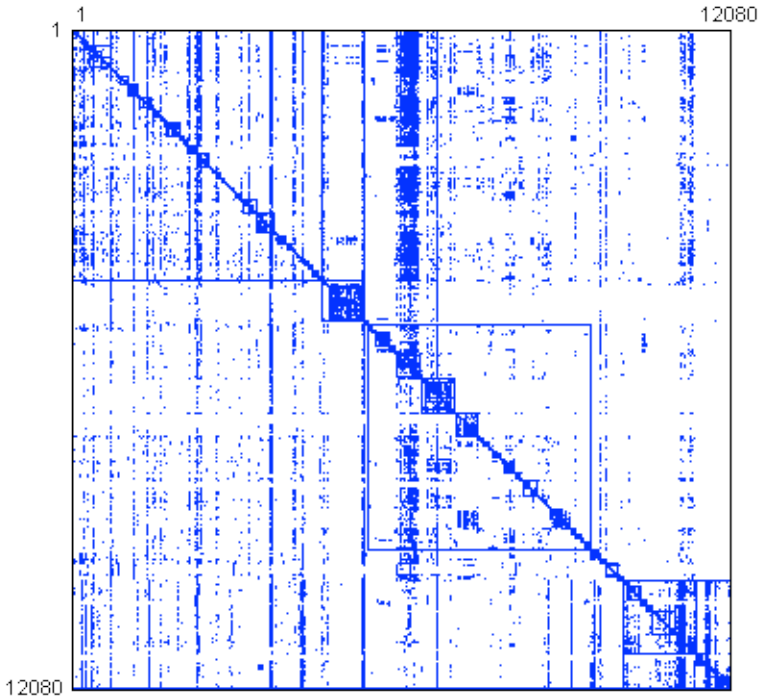
Matched Pair 4a: Operating System (Linux versus Solaris)

Linux 2.6.8.1



Size = 6675
Dependency Density = 0.11118%
Propagation Cost = 7.1827%

OpenSolaris 35

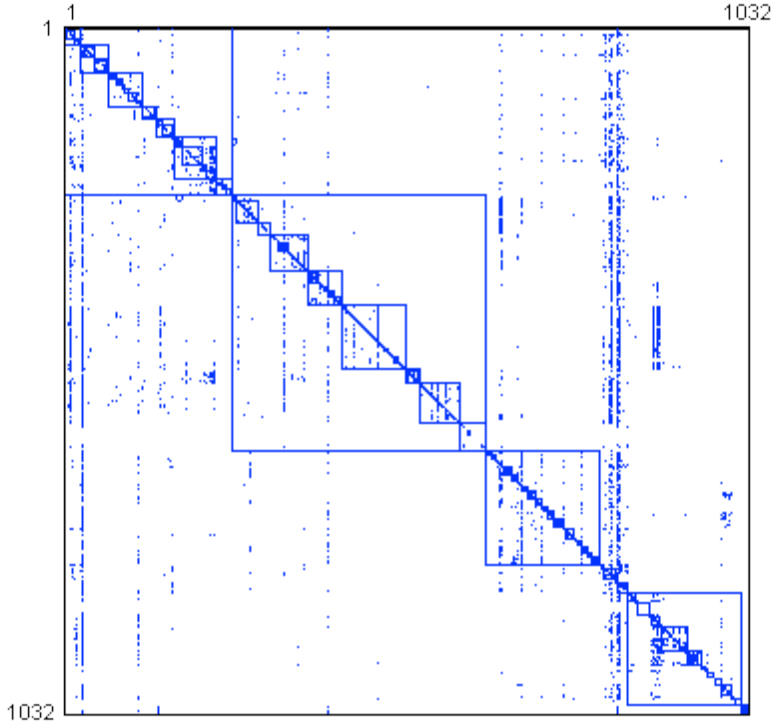


Size = 12080
Dependency Density = 0.07714%
Propagation Cost = 22.5903%

APPENDIX B: COMPARISON OF DSMs FOR EACH PRODUCT PAIR

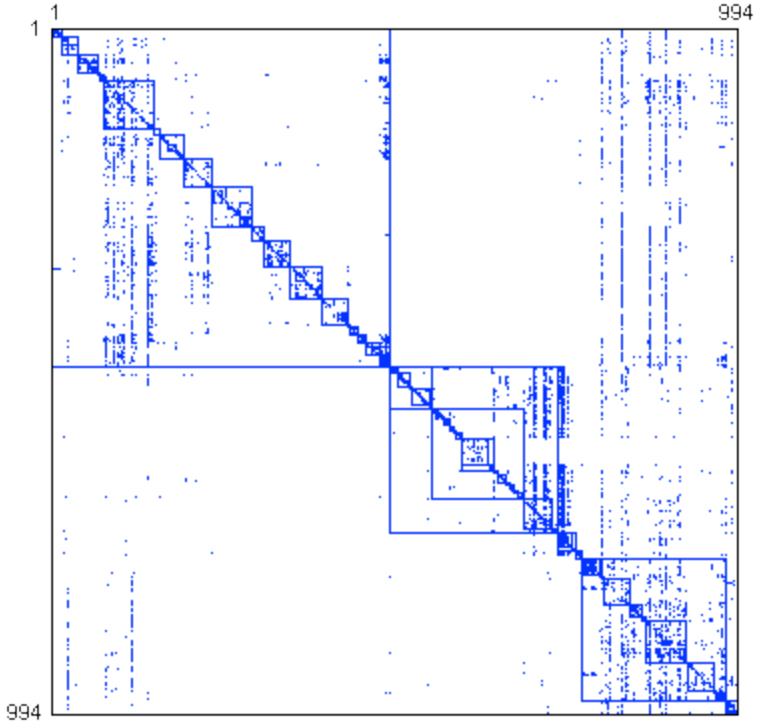
Matched Pair 4b: Operating System (Linux versus XNU)

Linux 2.1.32



Size = 1032
Dependency Density = 0.56402%
Propagation Cost = 7.2139%

XNU 123.5

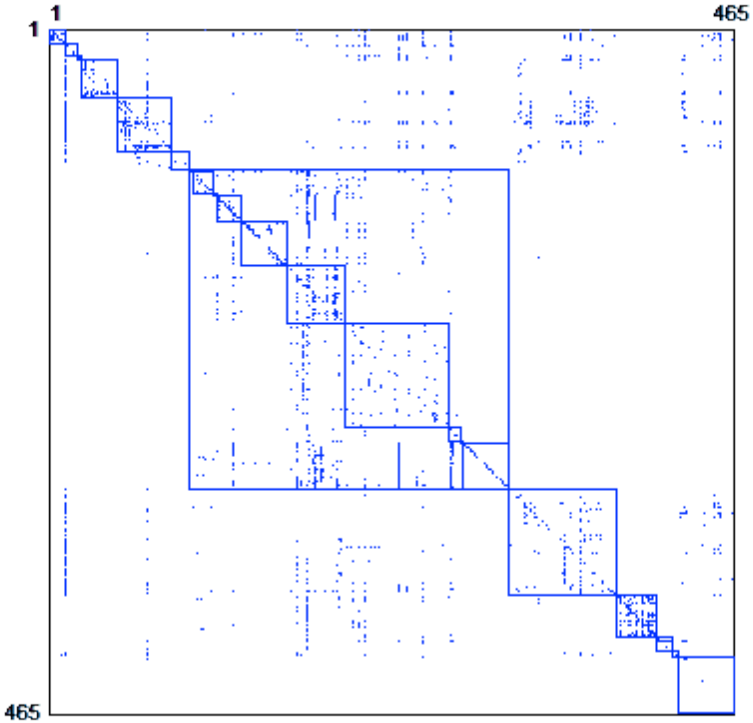


Size = 994
Dependency Density = 0.69836%
Propagation Cost = 24.8286%

APPENDIX B: COMPARISON OF DSMs FOR EACH PRODUCT PAIR

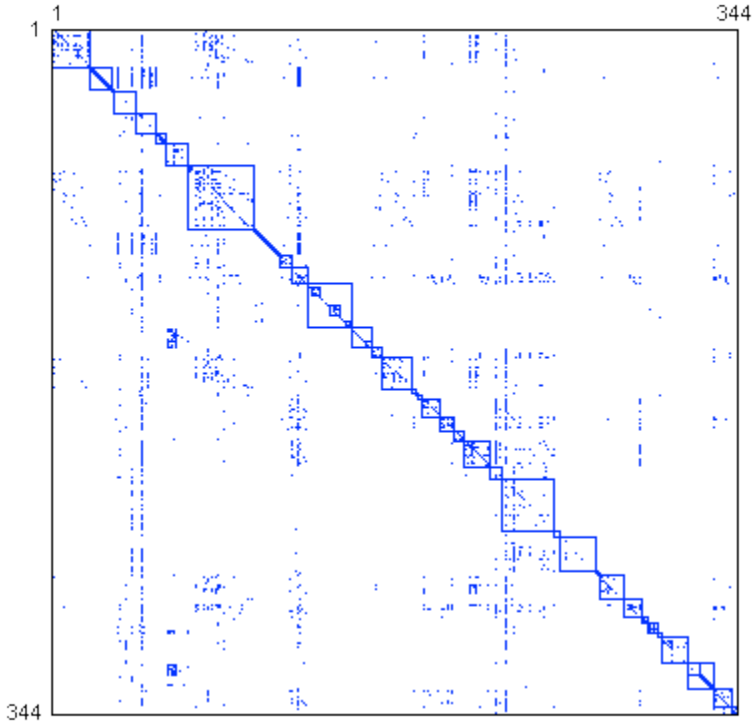
Pair 5: Database Software

MySQL 3.20.32a



Size = 465
Dependency Density = 0.94485%
Propagation Cost = 11.3049%

BerkeleyDB 4.3.27



Size = 344
Dependency Density = 1.8794%
Propagation Cost = 43.2311%

References

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

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

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

Brusoni, Stefano and Andrea Prencipe (2001) "Unpacking the Black Box of Modularity: Technologies, Products and Organizations," *Industrial and Corporate Change,* 10(1):179-205.

Brusoni, Stefano, Andrea Prencipe and Keith Pavitt (2001) "Knowledge Specialization, Organizational Coupling and the Boundaries of the Firm: Why Do Firms Know More Than They Make?" *Administrative Science Quarterly*, 46(4):597-621.

Burns, T., and G.M. Stalker (1961) *The Management of Innovation,* Tavistock Publications, London, England.

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

Chesbrough, Henry. (2003) *Open Innovation,* Harvard Business School Press, Boston MA.

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

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

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

Dibona, C., S. Ockman and M. Stone (1999) *Open Sources: Voices from the Open Source Revolution*, Sebastopol, CA: O'Reilly and Associates.

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

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

Fayol, H. (1949) *General and Industrial Management,* London: Pitman (first published in 1919).

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.

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

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

Howison, J. & Crowston K (2004) "The perils and pitfalls of mining SourceForge," *Proceedings of Mining Software Repositories Workshop, International Conference on Software Engineering*, May 2004.

Iansiti, M. and R. Levian (2004) *The Keystone Advantage,* Harvard Business School Press, Boston, MA.

Lawrence, Paul R. and Jay W. Lorsch (1967) *Organization and Environment,* Harvard Business School Press, Boston, MA.

Lopes, Cristina V. and Sushil K. Bajracharya (2005) "An Analysis of Modularity in Aspect-oriented Design," in *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development*, ACM

Press, pp. 15-26.

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

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

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

MacCormack, Alan, Theodore Forbath, Patrick Kalaher and Peter Brooks (2007) "Innovation through Collaboration: A New Source of Competitive Advantage," Harvard Business School Working Paper 07-079.

March, J.G. and H.A. (1958) Simon, *Organizations,* New York: John Wiley

Mayo, E. (1945) *The Social Problems of an Industrial Civilization,* Boston, MA: HBS Press

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

McGregor, D. (1960) *The Human Side of the Enterprise,* New York: Macmillan

Mockus, Audris, Roy T. Fielding and James D. Herbsleb (2002) "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, 11(3):309-346.

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

O'Reilly, T. (1999) Lessons from Open Source Software Development, Comms. ACM **42**(4) 33-37.

Orton, J.D. and K.E. Weick, Loosely-Coupled Systems: A Reconceptualization, *Academy of Management Review,* 15:203-23

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

Paulson, James W., Giancarlo Succi and Armin Eberlein (2003) "An Empirical Study of Open-Source and Closed-Source Software Products," *IEEE Transactions on Software Engineering*, 30(4):246-256.

Raymond, Eric S. (2001) *The Cathedral and the Bazaar* O'Reilly & Associates, Inc., Sebastopol, CA.

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

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

Sanchez, Ronald A. and Joseph T. Mahoney (1996). "Modularity, flexibility and knowledge management in product and organizational design". *Strategic Management Journal*, 17: 63-76, reprinted in *Managing in the Modular Age: Architectures, Networks, and Organizations* (G. Raghu, A. Kumaraswamy, and R.N. Langlois, eds.), Blackwell, Oxford/Malden, MA.

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

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

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.

Scott, W.R. (1981) *Organizations: Rational, Natural and Open Systems,* New Jersey: Prentice-Hall, Inc.

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

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

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

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

Simon H.A. (1976) *Administrative Behavior* (3rd Edition), New Yrok: Macmillan (first published in 1945)

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

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

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

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

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

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

Taylor, F.W. (1911) *The Principles of Scientific Management,* New York: Harper.

Teece, David J. (1986) "Profiting from Technological Innovation: Implications for Integration, Collaboration, Licensing and Public Policy," *Research Policy*, 15: 285-305.

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 (1994). *Mastering the Dynamics of Innovation,* Harvard Business School Press, Boston, MA.

von Hippel, Eric (1990) "Task Partitioning: An Innovation Process Variable," *Research Policy* 19: 407-18.

von Hippel, Eric and Georg von Krogh (2003) "Open Source Software and the 'Private Collective' Innovation Model: Issues for Organization Science," *Organization Science*, 14(2):209-223.

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

Weber, M. (1947) *The Theory of Social and Economic Organization,* ed. A.H. Henderson and T. Parsons, Illinois: Free Press (first published in 1924)

Weick, K..E. Educational Organizations as Loosely-Coupled Systems, *Administrative Science Quarterly,* 21: 1-19

Williamson, Oliver E. (1985). *The Economic Institutions of Capitalism*, New York, NY: Free Press.