08-039

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

Alan D. MacCormack
John Rusnak
Carliss Y. Baldwin

Abstract

A variety of academic work asserts that a relationship exists between the structure of a development organization and the architecture of the products that this organization produces. Specifically, products are often said to "mirror" the architectures of the organizations from which they come. Such a link, if confirmed empirically, would be important, given that product architecture has been shown to be an important predictor of, among other things: product performance; product variety; process flexibility; and future industry evolution. We explore this relationship in the software industry by use of a technique called Design Structure Matrices (DSMs), which allows us to visualize the architectures of different software products and to calculate metrics to compare their levels of modularity. We use DSMs to analyze a number of matched-pair products – products that fulfill the same function but that have been developed via contrasting modes of organization; specifically, closed-source (or proprietary) versus open-source (or distributed) development. Our results reveal significant differences in modularity, consistent with a view that distributed teams tend to develop more modular products. We conclude by highlighting some implications of this result and assessing how future work in this field should proceed, based upon these first steps in measuring "design."

1. Introduction

Much recent research points to the critical role of product architecture in the successful development of a firm's new products and services, the competitiveness of its product lines and the successful evolution of its technical 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). Of particular interest to this study, Henderson and Clark (1990) show that incumbent firms often stumble when faced with innovations that are "architectural" in nature. They assert that these dynamics occur because product designs tend to evolve in a way that mirrors the organizations that develop them, a concept sometimes referred to as duality. In essence, the argument is that the space of designs that an organization searches is constrained by the characteristics of the organization itself, in addition to explicit choices made by designers. Unfortunately, the empirical demonstration of such a result remains elusive.

This study aims to provide empirical evidence to address the hypothesized relationship between product and organizational architectures. We do this by applying an analytical technique called design structure matrices (DSMs) to compare a number of products in the software industry. Our analysis takes advantage of the fact that software is an information-based product, meaning that its design comprises a series of instructions (the "source code") that tell a computer what tasks to perform. In this respect, software products can be processed automatically to identify the dependencies that exist between different parts of the design (something that cannot be done with physical products). These dependency relationships can be used to characterize various aspects of product architecture, by both displaying the information visually and calculating metrics that summarize their impact on the system as a whole.

We choose to analyze software products because of a unique opportunity to examine two different organizational modes for development. Specifically, in recent years there has been 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

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

to this product (DiBona et al, 1999). Successful open source software projects tend to be characterized by large, distributed teams of volunteer developers who contribute new features, fix defects in existing code and write documentation for the product (Raymond, 2001; von Hippel and von Krogh, 2003). These developers, which can number in the hundreds, are located around the globe hence may never meet. This approach stands in contrast to the proprietary "closed source" model employed by commercial software firms. In this model, projects tend to be staffed by dedicated teams of individuals who are situated at a single location and have easy access to other team members. Given this proximity, the sharing of information about solutions being adopted in different parts of the design is much easier, and may be encouraged (e.g., if the creation of a dependency between two parts of a design would increase performance). Consequently, the architectures of products developed using a proprietary development model may differ from those of products developed using open source methods: In particular, open source software is likely to be more "modular" than closed source software. Our research explores the magnitude and direction of these differences.

Our paper proceeds as follows. In the next section, we describe the motivation for our research and discuss prior work in the field which pertains to understanding the link between product and organizational architectures. We then describe our research methodology, which involves calculating the level of modularity in a software system by analyzing the dependencies that exist between its component parts. Next, we discuss the how we construct our sample of matched product pairs, each consisting of one open source and one closed source product. Finally, we provide the results of our empirical tests, and highlight the implications of our findings for practitioners and the academy.

2. Research Motivation

The architecture of a product is the scheme by which the functions it performs are allocated to its constituent components (Ulrich, 1995). For any given product, a number of architectures are likely to satisfy its functional requirements. These different architectures may differ along important performance dimensions, such as the quality of the final product, its reliability in operation, its robustness to change and its physical size. They may also imply a differing partitioning of *tasks*, thereby influencing the efficiency

with which development can proceed (Von Hippel, 1990). Understanding the factors that impact how architectures are chosen, how they are developed, how they evolve and how they can be adapted are therefore critical topics for managerial attention.

A variety of work has sought to examine the link between a product's architecture and the characteristics of the organization that develops this product. The roots of this work lie in the field of organization theory, where it has long been recognized that organizations must be designed to reflect the nature of the tasks that they must perform (Lawrence and Lorsch, 1967; Burns and Stalker, 1961). In a similar fashion, transaction cost theory predicts that different organizational forms are required to effectively solve the contracting challenges associated with tasks that present different levels of uncertainty and interdependency to a decision maker (Williamson, 1985; Teece, 1986). To the degree that different product architectures imply a different set of tasks and related challenges, this work suggests that organizations and architectures must also be aligned.

Studies that seek to examine this topic empirically follow one of two approaches. The first explores the need to match patterns of communication within a development project to the interfaces that exist between different parts of a product's design. For example, Sosa et al (2004) examine a large jet engine project, and find a strong tendency for team communications to be aligned with design interfaces. The likelihood of misalignment is shown to be greater across organizational and system boundaries. Cataldo et al (2006) explore the impact of these misalignments in a large software development project. They find that development tasks are completed faster when the patterns of team communication are congruent with the software's dependency structure. Finally, Gokpinar et al (2006) also study the impact of misalignments in an automotive development program. They find that subsystem quality is higher to the degree that team communications are aligned with the patterns of interfaces to other subsystems.

While the prior studies begin with the premise that a development organization must be designed to match the desired structure of a new product, the second stream of work adopts the opposite perspective. It assumes that a development organization's structure is fixed over the medium term, and seeks to understand the impact on new product designs. This view was first articulated by Conway (1968), becoming known as "Conway's Law." It states, "Any organization that designs a system will inevitably produce a design whose

structure is a copy of the organization's communication structure." This dynamic is best illustrated by Henderson and Clark's (1990) study of the photolithography industry, in which market leadership changed hands each time a new generation of equipment was introduced. These observations are traced to the failure of incumbent firms to respond effectively to architectural innovations, which involve changes to the linkages between components. Such innovations challenge firms because they destroy the usefulness of architectural knowledge embedded in their organization structures and information-processing routines, which tend to reflect the existing dominant design (Utterback, 1996). When this design is no longer optimal, organizations find it difficult to adapt.

The contrast between these two perspectives becomes clear when we consider the dynamic that occurs when two different organizations develop the same product. Assuming the functional requirements are identical, one might predict that the product architectures resulting from the two development efforts should be similar. To the degree that architectural choices are driven more by the characteristics of the organizations themselves however, the designs may be quite different. This "mirroring hypothesis" can be tested by comparing the designs of a number of matched-pair products – products that fulfill the same function but that have been developed via contrasting modes of organization. Such a natural experiment exists in software, given two distinctly different development modes are observed: closed source (or proprietary) development versus open source (or distributed) development. To conduct such a test however, we must first establish measures by which to compare different architectures.

2.1 Product Architecture and Modularity

Modularity is a concept that helps us to characterize different product architectures. It refers to the way that a product design is decomposed into different parts or modules. While 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 referred to as "loose-coupling." Modular designs are loosely-coupled in that changes made to one module have little impact on the others. Just as there are degrees of coupling, hence there are degrees of modularity.

The costs and benefits of modularity have been discussed in a stream of research that has sought to examine its impact on a range of activities including the management of complexity (Simon, 1962), product line architecture (Sanderson and Uzumeri, 1995), manufacturing (Ulrich, 1995), process design (MacCormack, 2001) process improvement (Spear and Bowen, 1999) and industry evolution (Baldwin and Clark, 2000). Despite the appeal of this work however, few studies have brought empirical data to bear on the relationship between measures of product modularity, organizational factors assumed to influence this property and outcomes that it might impact (Fleming and Sorenson, 2004). Most studies tend to be conceptually motivated or descriptive in nature, and offer few insights as to how modularity can be measured in a robust and repeatable fashion

Studies which do 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 actual components. Metrics which 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 design task dependencies and project team communications (Sosa et al, 2004). Recent work significantly extends this methodology by showing 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 these methods to compare designs that emerge from different types of organization.

2.2 Software Design Structure

The measurement of modularity has gained significant traction in the field of software, given the information-based nature of the product lends itself to analytical techniques which are not possible with physical products. Critically, software systems are rarely re-built from scratch but instead, use the prior version as a base upon which

new functionality is added. This dynamic increases the importance of understanding techniques by which the resulting complexity can be managed.

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 which sought to measure product complexity for the purposes of predicting development productivity and quality (e.g., McCabe 1976; Halstead, 1976). Whereas measures of software complexity focus on capturing the number of elements in a design, measures of software modularity focus on the number and pattern of dependencies *between* these elements. This implies that a product can be both 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 empirically generally follow one of two approaches. The first focuses on analyzing specific types of dependency between components, for example, the number of non-local branching statements (Banker et al, 1993); the number of global variables (Schach et al, 2002); or the number of function calls (Banker and Slaughter, 2000; Rusovan et al, 2005). The second infers the presence of a dependency between components by assessing whether they tend to be modified at the same time. For example, Eick et al (1999) show that code decays over time as measured by the number of files changed to complete a modification request; and Cataldo et al (2006) show that modifications involving files with higher coupling take longer to complete. While the latter approach avoids the need to specify the type of dependency between components, it requires access to maintenance data that is not always available, nor captured consistently across projects. In multi-project comparisons, a method which extracts dependencies from the source code itself 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 proprietary software (O'Reilly, 1999; Raymond, 2001). Others suggest that modularity is a required property for this method to succeed (Torvalds, as

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

quoted in DiBona, 1999). Empirical work to date yields mixed results on this question. Some studies criticize the number of problematic dependencies between 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 dependencies between their constituent elements (MacCormack et al, 2006). Critically however, none of these studies provides a true apples-to-apples comparison using products that fulfill the same function but that have been developed using different organizational approaches.

In this paper, we explore differences in design structure between software systems of comparable size and function developed using contrasting modes of organization: specifically, closed source (proprietary) versus open source (distributed) software. Our work builds upon recent studies which highlight how DSMs can be used to visualize and measure attributes of software architecture (Sullivan et al, 2001; Lopes and Bajracharya, 2005; MacCormack et al, 2006). We use these methods to examine the hypothesis that the architecture of a product tends to mirror the structure of the organization in which it is developed. In essence, we expect open source products to be more modular than their equivalent closed source counterparts. The use of a matched pair design allows us to control for differences in architecture that are driven by differences in product function.

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

[3] The methods we describe here build on prior work in this field (see MacCormack et al, 2006; 2007).

for version control. And finally, prior work on design uses the source file as the primary level of analysis (e.g., Eick et al, 1999; Rusovan et al, 2005; Cataldo et al, 2006).[4]

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.

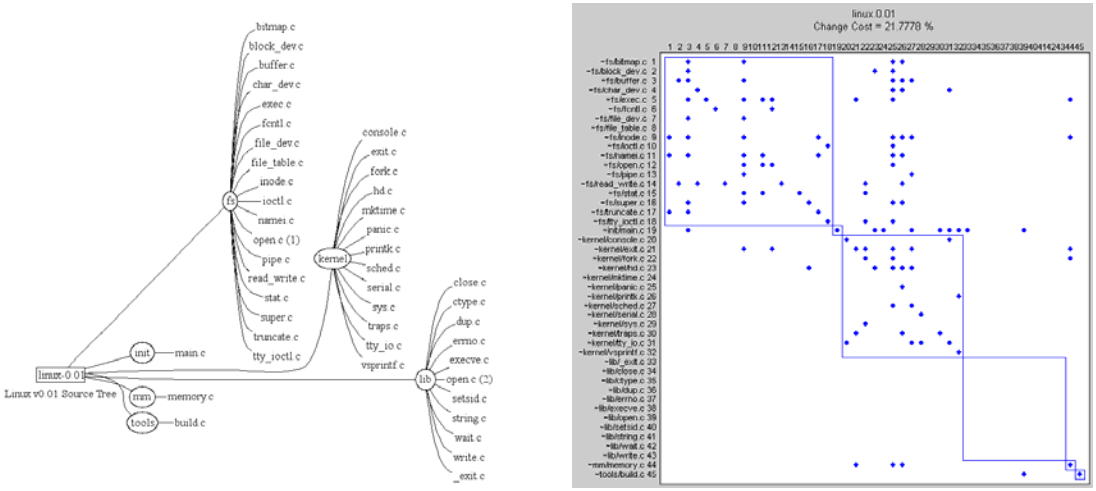
To capture function calls, we input a product's source code into a tool called a "Call Graph Extractor" (Murphy et al, 1998). This tool is used to obtain a better understanding of system structure and interactions between parts of the design.[6] Rather than develop our own extractor, we tested several 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 can be displayed using the *Architectural View*. This groups each source file into a series of nested clusters defined by the directory structure, with boxes drawn around each successive layer in the hierarchy. The result is a map of dependencies, organized by the programmer's perception of the design. To illustrate, the Directory Structure and Architectural View for Linux v0.01 are shown in **Figure 1**. Each "dot" represents a dependency between two particular components (i.e., source files).

---

[4] Source files are akin to physical components; functions are the nuts and bolts within these components.
[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.

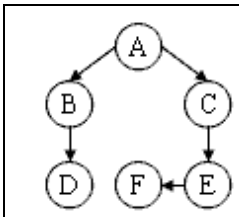


3.1 Measuring System Modularity

The method by which we characterize the structure of a design is by measuring the degree 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 very 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 an 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. In the example below, the system has an overall propagation cost of 42%.

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

Our analysis approach is based upon comparing the architectures of products that perform similar functions developed using different organizational modes. To do this, we construct a sample of matched product pairs, then for each pair, test the hypothesis that the open source product (i.e., the one developed by a larger, more distributed team) is more modular than the closed source product (i.e., the one developed by a smaller, more centralized team). Our matching process takes into account both the function of the software (e.g., a spreadsheet) as well as the relative level of functionality it provides. This is achieved by pairing products of a similar size, thereby controlling for potential differences in modularity related to product scope.

Developing an ideal sample proves difficult for two reasons. First, many open source projects are relatively small efforts involving only a handful of people and a few thousand lines of code (Howison and Crowston, 2004). Yet we need products sufficiently complex for the dynamics we are exploring to produce meaningful differences in architecture. To tackle this problem, we focus only on successful open source efforts resulting in products that are widely used and have a minimum size.[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 release it and cautious about work that seeks to compare it with "free" open source equivalents. Where an ideal match is not available, we therefore adopt two different strategies: First, we try to identify a matched product which was once closed but is now open, and use the first release of the open version as a proxy for the closed source architecture; and second, where information on the nature of the team is available, we try to identify a product developed by a smaller, more centralized team, even if using an open source license.

Table 1 describes the resulting sample of five paired products. Note that there are several well known and successful open source products for which we could not find a suitable match (e.g., the Apache web server project). Note also that we identify two possible closed source matches to the Linux operating system, given that the ideal product pair – Solaris – is significantly larger and more sophisticated than the open

[9] Usage was determined by downloads and other data on the number of user installations. Product size was determined by the number of source files in the product. A minimum threshold of 300 files was used.

source product. While our final sample is admittedly small, it does provide sufficient statistical power for a test of the mirroring hypothesis.[10]

Table 1: The Sample of Matched Pairs

	Open	Closed	Comments
1	Gnucash	MyBooks	Financial management software. MyBooks is a commercial product that has been disguised. It is what we call an "ideal" pair.
2	Abiword	OpenWrite	Word processing software. OpenWrite comes from Star Office, a closed commercial product that was released as OpenOffice in 2000.
3	Gnumeric	OpenCalc	Spreadsheet software. OpenCalc comes from Star Office, a closed commercial product that was released as OpenOffice in 2000.
4	Linux	a) OpenSolaris b) XNU	Operating system software. Solaris is an operating system developed by Sun. Its source code was released in 2004. XNU is the kernel from Apple's Darwin operating system.
5	MySQL	Berkeley DB	Database software. Berkeley DB is developed by a team of less than 10 people.[11] MySQL is developed by a large, distributed team.[12]

5. Empirical Results

Data on the propagation costs for each matched pair is shown in **Table 2**. We find statistically significant differences in propagation cost between all of our matched product pairs. The direction of these differences supports our hypothesis in four out of the five cases. The DSMs for each matched product pair are shown in **Appendix A**. Below, we use these visual comparisons, in conjunction with the data on propagation cost, to discuss the insights revealed by each comparison. Thereafter, we examine the single exception to our hypothesis in further detail.

[10] Assuming the null hypothesis, the chance of finding that the open source product is more modular than the closed source product in each of the five matched pairs is given by $(0.5)^5 = 003125$ ($p<0.05$).
[11] Source: Interview with one of the company founders.
[12] MySQL employs 60 developers in 25 nations; 70% work at home (*The Economist*, Mar 16th 2006).

Table 2: Differences in Propagation Cost for each Product Pair

	Open (Distributed)	Closed (Proprietary)	Test Stat[13]
1: Financial Mgmt	8.8%	**42.5%**	p<0.1%
2: Word Processing	2.8%	**29.7%**	p<0.1%
3: Spreadsheet	**23%**	4.1%	p<0.1%
4a: Operating System	7.2%	**21.8%**	p<0.1%
4b: Operating System	7.4%	**19.1%**	p<0.1%
5: Database	9.4%	**33.6%**	p<0.1%

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 that are called by much of the rest of the system, a structure we call a vertical bus, given it delivers functionality to many other components. By comparison, the closed source product has one very large module, within which there are many dependencies between elements. The system's propagation cost is over 40%, in contrast to the open source product, which is less than 9%.

In pair number two, the visual differences are not as distinctive as the first pair. Each product is divided into many modules of similar size. However, there are a greater number of dependencies between elements in the closed source product, and these dependencies are spread throughout the system, rather than being concentrated within the same module. As a result of this pattern, the propagation cost is almost 30%, in contrast to the open source product, which has a very low figure of only 2.8%.

In pair number three, our hypothesis is not supported. The open source spreadsheet, Gnumeric, has a larger number of dependencies than the closed source product. Many of these dependencies are to a group of files that appear to form one large module, although surprisingly, these files are not isolated within a separate sub-module. The propagation cost of Gnumeric is 23%, the highest of all open source products we examine. By contrast, the closed source product OpenCalc appears to be better structured, possessing a few top-level modules, each of which comprises a number of smaller sub-modules. With relatively few dependencies between elements, its propagation cost is just over 4%. We explore possible explanations for the result of this paired comparison below.

[13] To assess whether the difference between each product pair is statistically significant, we examine the visibility data at the component level and conduct a Mann-Whitney-U test of differences between the two populations (the distribution of visibility is non-normal, hence a simple t-test does not suffice).

In our fourth product category, we consider two potential matched pairs. In the first, which compares Linux with Solaris, our hypothesis is supported. The propagation cost of Solaris is over 20%, 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 2,400 other files. By contrast, the figure for Linux is around 7%. While still large in absolute terms, the difference between these 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 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 propagation cost may therefore be driven, in part, by differences in the functionality they provide.[14] To address this issue, we look at a second matched product – XNU – and compare it to a version of Linux of similar size.[15] The result is remarkably consistent with that of Solaris. The propagation cost of XNU is almost 20%, in comparison to 7.4% for a similarly-sized version of Linux. Of note, the structure of these products looks very similar. Indeed, the density of (direct) dependencies in each system is almost identical. This suggests it is the *pattern* of dependencies in XNU that leads to a higher propagation cost. In essence, this pattern leads to the presence of many more *indirect* linkages between components.

In pair number five, our hypothesis is once again supported. This pair is unusual in that the closed product comprises a large number of very small modules (i.e., it has very flat hierarchy). In a sense, it may therefore appear more modular from an architect's viewpoint. However, the number and pattern of dependencies between source files is such that the product has a very high propagation cost of over 30%. By comparison, the open source product contains an additional layer of hierarchy, with several sub-modules nested within a larger module containing half the system's files. Combined with its lower dependency density, this structure yields a propagation cost of just over 9%.

[14] Note that in every code base we have access to, propagation cost tends to remain broadly constant or decline as the 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 source file pairs, hence the density of the visibility matrix declines with size. This dynamic biases the test *against* our hypothesis when comparing Linux and Solaris.

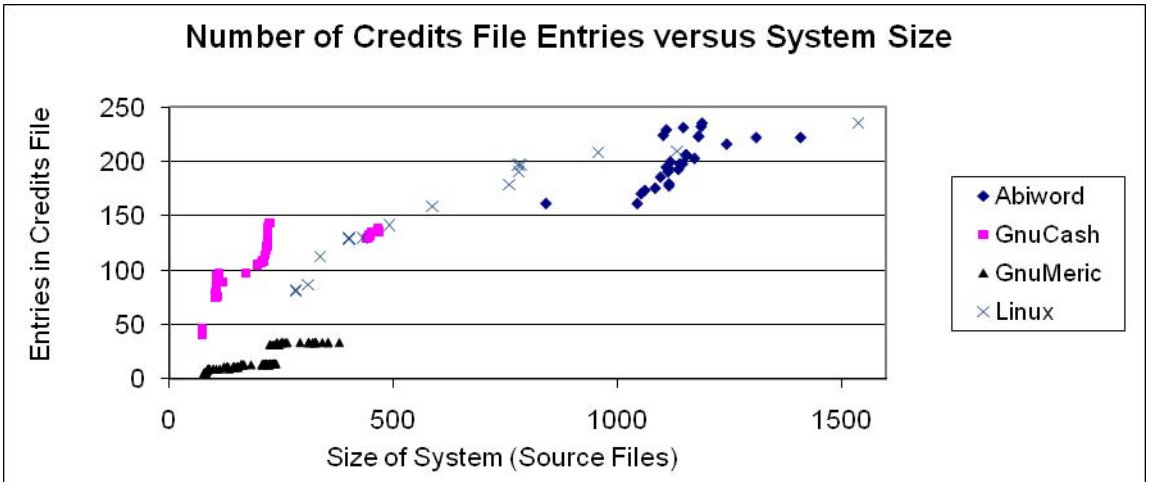
[15] XNU is the kernel of Apple's Darwin operating system. It was developed by a company called NeXT. Its origins lie in a small operating system kernel called Mach, which was developed at Carnegie-Mellon.

5.1 Exploring the Single Negative Result

In an attempt to understand why our hypothesis fails for the spreadsheet applications, we investigate Gnumeric further. In particular, we explore one possible explanation for this result: that this open source project does not involve a highly distributed team. To identify whether this explanation is supported, we examine the number of contributors that worked on GnuMeric in comparison to other open source projects. We gather data from two different sources: the credits file and the change log.[16] The credits file is a list of key individuals who have contributed to a system's development. Each individual's name is listed once, and when added is generally never removed. The change log is a detailed listing of each change made to the product in each new version. Some logs, such as the one used in GnuMeric, list the individuals who contributed each change.

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. The results are shown in **Figure 4**. We see that GnuMeric has a much smaller number of credits file entries than other open source products of similar size. By contrast, Linux, AbiWord and Gnucash all have similar patterns of contributor growth over time, with three to five times as many credits file entries after adjusting for size.

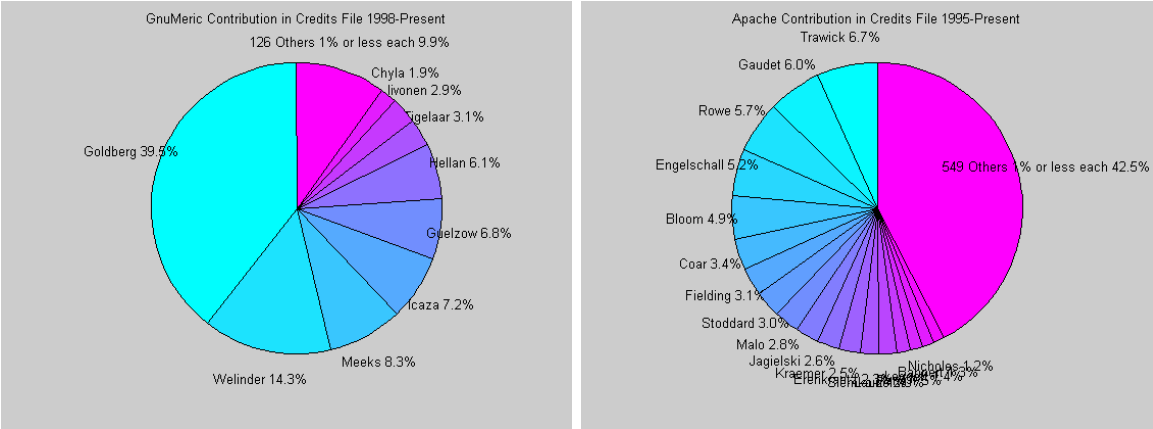
Figure 4: Number of Credits File Entries for Open Source Products



[16] Note that we do not use the Concurrent Versioning System (CVS) system, a tool which is sometimes used to control the submission of new source code to a product, given that in many of these projects, contributions are batched together and submitted by a few individuals who have "source access."

To probe this explanation further we sought to determine the *extent* of each individual's contributions to Gnumeric. To do this, we ran a script to count how many times each unique name appeared in the change log, thus capturing the proportion of submissions attributable to each. For comparison, we conducted the same analysis for an open source product where we could access comparable data, and for which we knew the system had a low propagation cost: the Apache web server.[17] The results are shown in **Figure 5**. The contrast is dramatic. In Gnumeric, one individual accounts for almost 40% of changes, the top four for almost 70% and the top 9 for around 90%. In Apache, the top individual accounts for less than 7% of changes and the top four less than 25%.

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



In **Table 3**, we plot the Gnumeric data by year, highlighting that the pattern of contributions is consistent throughout the project's life. We conclude that Gnumeric's development is far less distributed than that of Apache. Indeed, the data suggests it is more concentrated than a typical closed source or proprietary project. In essence, Gnumeric's high propagation cost provides support for the mirroring hypothesis.[18]

[17] The propagation cost for the Apache web server product closest in size to Gnumeric is less than 1%.
[18] We note that OpenCalc, the closed source product to which we compare Gnumeric, has a low propagation cost compared to other closed source products. Unfortunately, we cannot explain this observation given we have no access to internal data on how the product was developed.

Table 3: Developer Contributions for GnuMeric by Year (1998-2004)

	2004	**2003**	**2002**	**2001**	**2000**	**1999**	**1998**
Goldberg	48.7 %	43.9 %	41.2 %	54.8 %	46.2 %	14.3 %	
Guelzow	9.0 %	21.8 %	17.6 %	3.4 %			
Hellan	5.3 %	5.9 %	13.0 %	4.3 %	8.2 %	0.9 %	
Welinder	25.6 %	22.0 %	18.8 %	10.8 %	9.5 %	11.9 %	0.2 %
Chyla			2.1 %	6.7 %	1.2 %	0.2 %	0.1 %
Tigelaar			0.2 %	11.5 %	4.4 %		
Icaza				0.1 %	5.3 %	20.6 %	12.4 %
Iivonen			0.9 %	0.6 %	3.7 %	10.2 %	
Meeks			0.1 %	0.2 %	10.8 %	30.7 %	1.7 %
Others	11.3 %	6.3 %	6.0 %	7.6 %	10.8 %	11.3 %	85.5 %
Total	*100.0 %*	*100.0 %*	*100.0 %*	*100.0 %*	*100.0 %*	*100.0 %*	*100.0 %*

5. Discussion

Our findings yield insights along several dimensions. First, they reveal substantial differences in relative levels of modularity between software systems of similar size and function. The pairs that we examine vary by a factor of ten in terms of the potential for a design change to propagate to other system components. This result has significant implications for those who must design such systems. In particular, it shows that a product's architecture is not wholly determined by function, but is also influenced by the characteristics of the organization within which development occurs. In essence, the space of possible designs within which solutions are sought is constrained by the nature of the context within which this search occurs. Our work makes such influences visible.

In this respect, our study provides evidence on the question of whether a link exists between a product's architecture and the structure of the organization in which it is developed. In four of the five pairs that we examine, the open source product is more modular than that of a comparable product developed by a smaller, more centralized team. In the one pair in which our hypothesis is not supported, we find that the open source product is *not* in fact developed in a distributed fashion. Rather, the pattern of development is more consistent with that of a small co-located team. Taken in combination, our results provide strong evidence that a product's architecture tends to mirror the structure of the organization within which it is developed.

This finding is consistent with two rival hypotheses. The first is that designs *evolve* to reflect their development environments. In closed source projects, 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 the information and solutions developed in other modules. Even if not an explicit managerial choice, the design naturally becomes more tightly-coupled. By contrast, in open source products, a large and widely distributed team 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 therefore more modular as a consequence.

Alternatively, our observations may be a product of *purposeful choices* made by the original architects. For closed source products, the sole aim is to develop a product that maximizes performance. The benefits of modularity, given the competitive context, may not be seen as significant. By contrast, for open source products, the benefits of modularity are greater. Without it, 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 many other parts of the system. Open source products therefore need to be modular to attract a developer community and facilitate the work of this community. In practice, it is likely that both these causal mechanisms have some validity.

Our results have important implications for development organizations given the recent trend towards "open" approaches to innovation and the increased use of partnering in R&D projects (Chesbrough, 2003; Iansiti and Levian, 2004; MacCormack et al, 2007). In particular, they show that these new organizational arrangements can have a distinct impact on the nature of the resulting design, and hence may affect product performance in unintended ways. In essence, our work highlights that partnering choices, along with the division of tasks that they imply, should not be managed independently of the design process itself (von Hippel, 1990). Decisions taken in one realm will ultimately affect performance in the other, and must be jointly evaluated to achieve effective results.

Several limitations of our study must be considered in assessing the generalizability of results. First, our work is conducted in the software industry, a unique context given that designs exist purely as information. Whether these results will hold for physical products remains uncertain. Second, our sample comprises only five matched pairs, a limitation that stems from the lack of successful open source products of sufficient size,

and the difficulty in obtaining proprietary source code that firms regard as a form of intellectual property. Third, we do not directly test the functional equivalence of the pairs we analyze, instead comparing products of similar size. As a result, some of the differences we observe may be associated with real differences in performance between products. Finally, the pairs that we analyze were not necessarily developed at the same time. Hence our results may partially be explained by learning between the two.[19]

Our work opens up a number of areas for future study. With respect to methods, we show that dependency analysis provides a powerful lens with which to examine product architecture. While we focus on only one type of dependency, our methods can be generalized to others, assuming they can be identified from source code. With respect to studies of modularity, our work provides visibility of a phenomena which was previously hidden, and metrics with which to compare different products. This approach promises to facilitate the study of the performance trade-offs that stem from architectures with different characteristics. There are strong theoretical arguments why such trade-offs exist, yet little empirical evidence to confirm their presence. Does greater modularity require trade-offs with other aspects of product performance? Intriguingly, our own work suggests that many designs are not at the performance frontier where such a trade-off exists, but are below it due to architectural inefficiencies or "slack." If this is true, there may be considerable scope to improve a design along multiple dimensions. Exploring such issues via careful measurement of architecture and product performance will help reveal strategies for moving designs towards the performance frontier. It will also help us to understand the trade-offs involved in moving *along* it.

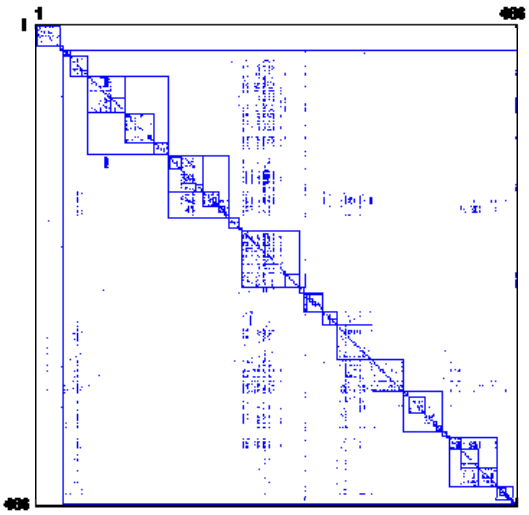
Herbert Simon (1962) was the first to argue for the systematic study of design more than 40 years ago, claiming, '…the proper study of mankind is the science of design.' However, his ambitious vision for the field has proven elusive. The study of design has been constrained by, among other things, limited theory, methods and tools that can deal with the complexity of everyday designs, and more importantly, to make them visible, allowing us to compare their structures. The methods we have developed promise to open up a host of questions that, until now, were beyond our analytical capabilities.

[19] Open source products are often developed only after closed source products have reached some level of maturity. This might allow them to be better designed (i.e., more modular for a given performance level).

APPENDIX A: COMPARISON OF DSMs FOR EACH PRODUCT PAIR

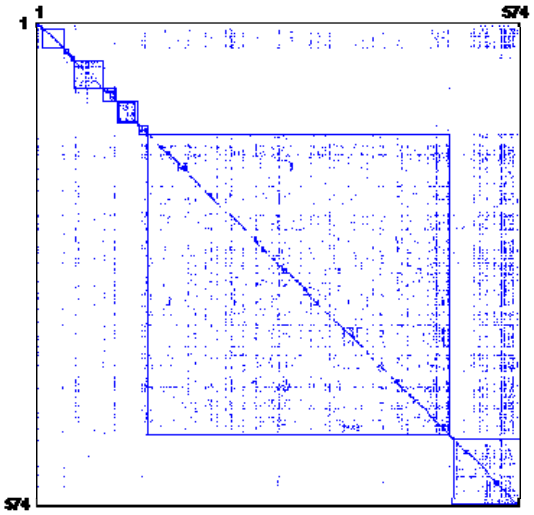
Pair 1: Financial Management Software

Gnucash 1.8.4



Dependency Density = 1.35%

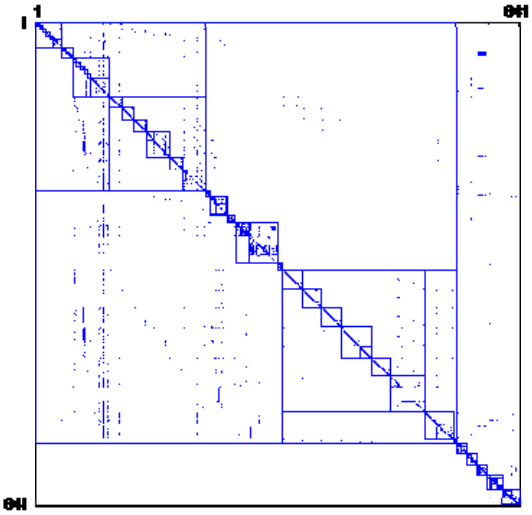
Disguised Product



Dependency Density = 1.68%

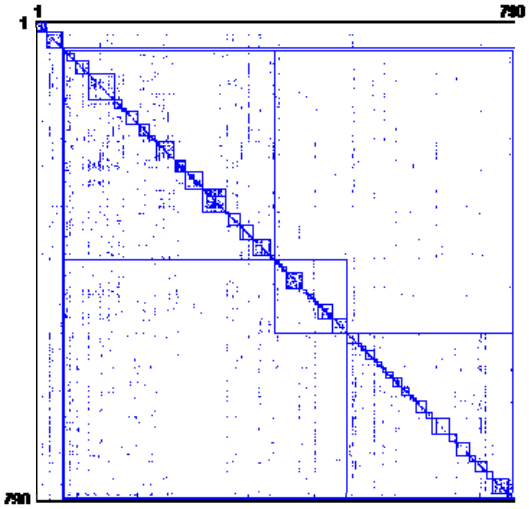
Pair 2: Word Processing Software

Abiword 0.9.1



Dependency Density = 0.33%%

OpenOfficeWrite

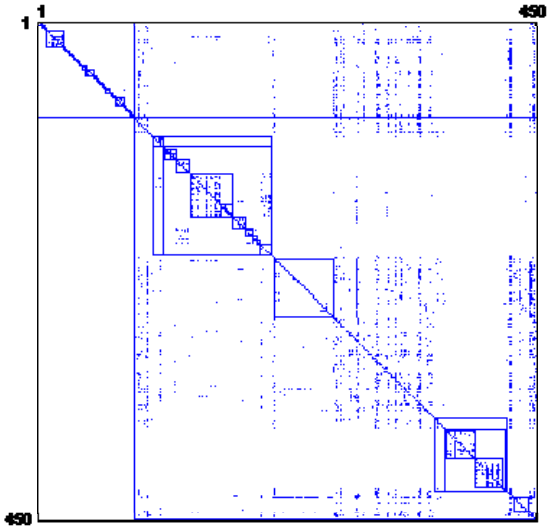


Dependency Density = 0.57%

APPENDIX A: COMPARISONS OF DSMs FOR EACH PRODUCT PAIR

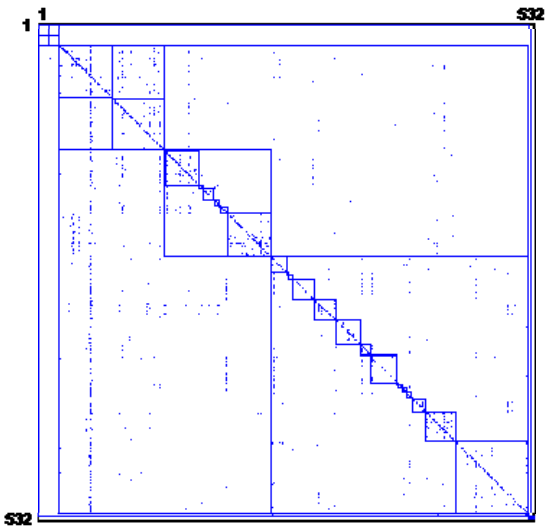
Pair 3: Spreadsheet Software

Gnumeric1.4.2



Dependency Density = 1.58%

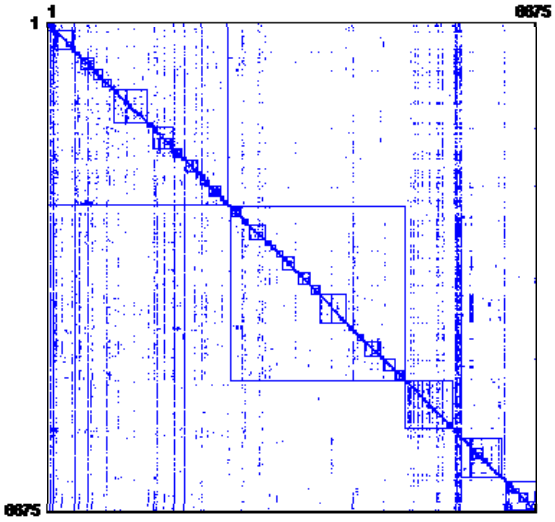
OpenOfficeCalc



Dependency Density = 0.43%

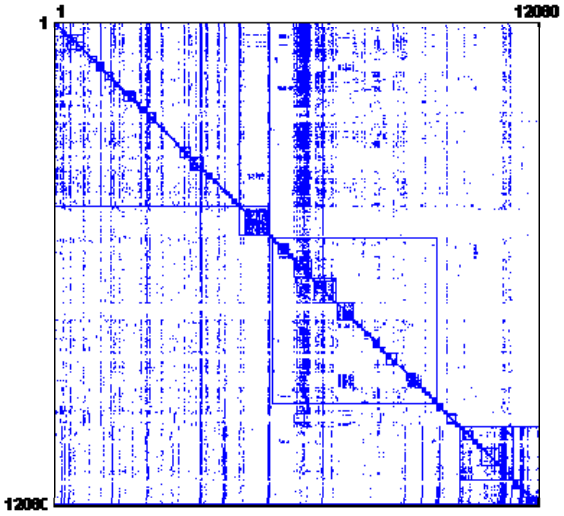
Pair 4a: Operating System Software (Linux versus Solaris)

Linux 2.6.8.1



Dependency Density = 0.11%%

Open Solaris 35

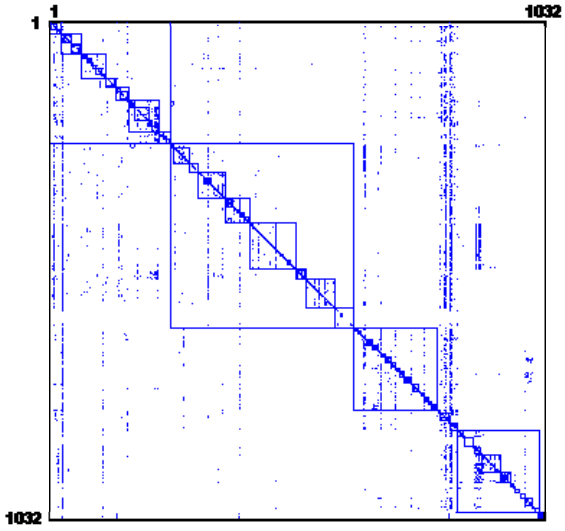


Dependency Density = 0.08%%

APPENDIX A: COMPARISONS OF DSMs FOR EACH PRODUCT PAIR

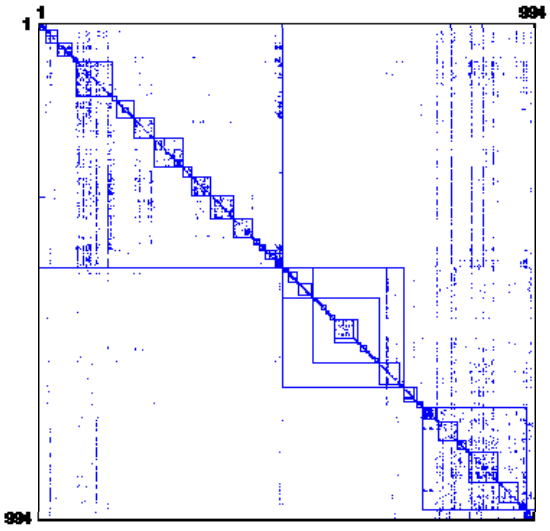
Pair 4b: Operating System Software (Linux versus XNU)

Linux 2.1.32



Dependency Density = 0.56%

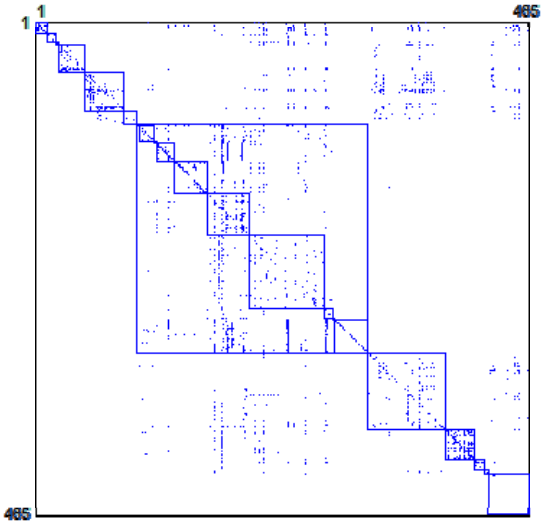
XNU 213.5



Dependency Density = 0.56%

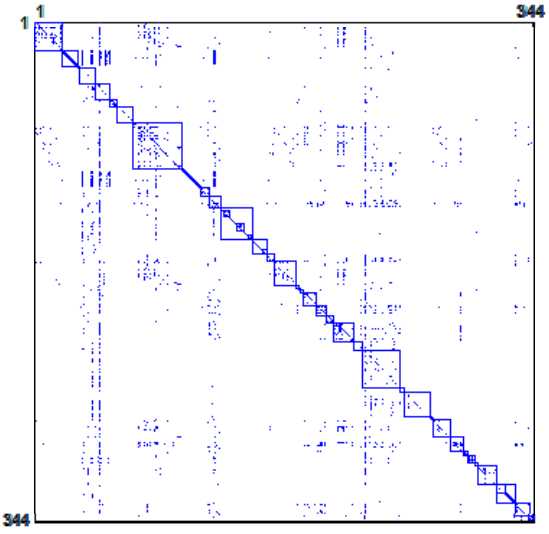
Pair 5: Database Software

MySQL 3.20.32a



Dependency Density = 0.91%

BerkeleyDB 4.3.27



Dependency Density = 1.63%

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.

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

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.

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

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.

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.

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

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

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

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

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

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

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.

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. *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.

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