MANAGEMENT SCIENCE
Vol. 52, No. 7, July 2006, pp. 1015–1030
ISSN 0025-1909 | EISSN 1526-5501 | 06 | 5207 | 1015

informs

DOI 10.1287/mnsc.1060.0552

Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code

Alan MacCormack, John Rusnak, Carliss Y. Baldwin
Harvard Business School, Soldiers Field, Boston, Massachusetts 02163
{amaccormack@hbs.edu, jrusnak@hbs.edu, cbaldwin@hbs.edu}

This paper reports data from a study that seeks to characterize the differences in design structure between complex software products. We use design structure matrices (DSMs) to map dependencies between the elements of a design and define metrics that allow us to compare the structures of different designs. We use these metrics to compare the architectures of two software products—the Linux operating system and the Mozilla Web browser—that were developed via contrasting modes of organization: specifically, open source versus proprietary development. We then track the evolution of Mozilla, paying attention to a purposeful "redesign" effort undertaken with the intention of making the product more "modular." We find significant differences in structure between Linux and the first version of Mozilla, suggesting that Linux had a more modular architecture. Yet we also find that the redesign of Mozilla resulted in an architecture that was significantly more modular than that of its predecessor and, indeed, than that of Linux. Our results, while exploratory, are consistent with a view that different modes of organization are associated with designs that possess different structures. However, they also suggest that purposeful managerial actions can have a significant impact in adapting a design's structure. This latter result is important given recent moves to release proprietary software into the public domain. These moves are likely to fail unless the product possesses an "architecture for participation."

Key words: architecture; software; design; product development; modularity; open source
History: Accepted by Eric von Hippel and Georg von Krogh, guest editors; received August 23, 2004. This paper was with the authors 4 months for 2 revisions.

1. Introduction

Much recent research points to the critical role of design structure 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). For example, Henderson and Clark (1990) show that incumbent firms often stumble when faced with innovations that are "architectural" in nature. They argue that these dynamics occur because product designs tend to mirror the organizations that develop them. However, the empirical demonstration of such a result remains elusive. Similarly, Baldwin and Clark (2000) argue that the modularization of a system can generate tremendous value in an industry, given that this strategy creates valuable options for module improvement. However, evidence of how and why such modularizations occur in practice has not yet been shown. Finally, MacCormack's (2001) work on the management of software projects suggests the importance of a modular architecture that "facilitates process flexibility." Without a way to measure the structural attributes of a design in a robust fashion, however, this work cannot reach the level of specificity it needs for managerial prescriptions to be drawn.

Common to all these research streams (and others not mentioned above) is a growing body of evidence that a product's architecture, or more broadly, the specific decisions made with regard to the partitioning of design tasks and the resulting design structure, is a critical topic for both researchers and managers to understand. Unfortunately, we lack metrics with which to pursue research on this topic, hence we have little *empirical* evidence that these constructs have power in predicting the phenomena they are associated with. This paper addresses the shortfall by defining a number of metrics that measure the degree of modularity of a design, based on an analytical technique called the design structure matrix (DSM). We use these metrics to compare the structures of different product designs from the software industry.

We chose to analyze software products because of a unique opportunity to examine two differing organizational modes for development. Specifically, over

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 (programming instructions) 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 highly 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 and hence may never meet face to face. Among the most popular examples of products developed in this manner are the Linux operating system and the Apache Web server.

The development methodology embodied in open source software projects stands in contrast to the "proprietary" development model employed by commercial software firms. In this model, projects tend to be staffed by dedicated teams of individuals who are located 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 (for example, if the creation of a dependency between two parts of a design could increase product performance). Consequently, the architectures of products developed using a proprietary development model are likely to differ from those of products developed using open source methods. Specifically, open source software is often claimed to be more "modular" than proprietary software (O'Reilly 1999, Raymond 2001). Our research seeks to explore the magnitude and direction of these presumed differences in design structure.

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 design structure, through displaying the information visually and calculating metrics to summarize the pattern of dependencies at the system level. Once mechanisms have been set up to do this, the ability to characterize different designs is limited only by the computing power available and the access we have to the source code of these designs (which is not a trivial problem, given firms regard source code as a form of proprietary technology).

The information-based nature of software products brings another benefit in that we can track the evolution of a design. This is possible because software developers use sophisticated version-control systems to track design changes. For a researcher, this presents an opportunity to follow the living history of a design, a technique that is typically not possible for physical products. In this study, this technique is valuable, given that we identify purposeful efforts to redesign a product. Having access to versions both before and after a redesign allows us to determine the impact of such managerial actions.

In the next section, we describe the motivation for our research and discuss prior work in this field. We then describe our research methodology, which uses DSMs to map the dependencies in software designs and defines metrics to summarize their structures. Next, we discuss our results, which are based on (a) a comparison of two products developed via contrasting modes of organization, and (b) an analysis of the impact of a major redesign effort for one of these products. We conclude by highlighting the implications of our work for both researchers and practitioners.

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 however, a number of architectures may satisfy its functional requirements. These different architectures are likely to 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 (Ulrich 1995). They may also imply a differing partitioning of design *tasks*, thereby influencing the efficiency with which development can proceed (von Hippel 1990). Understanding how architectures are chosen, how they evolve and how they are adapted are therefore critical topics for managerial attention. For this to occur however, we must better understand how to measure and characterize differences between them.

Modularity refers to the manner in which a design is decomposed into different "modules." Although authors vary in their definitions of modularity, they tend to agree on the concepts that lie at its heart: the notion of interdependence within modules and independence between modules (Baldwin and Clark 2000). This latter concept, in turn, encompasses two

[1] Most commercial software is distributed in a binary form (i.e., a series of 1s and 0s) that is executed directly by a computer. It is difficult to reverse-engineer this binary form to derive the original source code. Hence distributing software in binary form keeps the methods by which a program performs its tasks proprietary to the author.

related ideas: The need to allow work on a given module to be carried out without affecting other modules in the design, a concept known as "loose-coupling," and the need for well-designed interfaces between these modules. The *degree* of modularity is an important property of a product's architecture that is often used to characterize how designs differ.

The costs and benefits of modularity have been discussed in a stream of academic 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), and process improvement (Spear and Bowen 1999). Despite the appeal of this work however, few studies show correlation between measures of modularity, organizational factors assumed to influence this property, and outcomes it might impact (e.g., see Fleming and Sorenson 2004).

The most promising technique for understanding and measuring modularity has come from the field of engineering, in the form of the DSM. A DSM highlights the inherent structure of a design by examining the dependencies that exist between its component elements in a square matrix (Steward 1981, Eppinger et al. 1994). DSMs can be constructed using elements that represent "tasks" to be performed or "parameters" to be defined. In such situations, their use is to identify a partitioning of tasks (or parameters) that facilitates the flow of information in a project. Tasks (or parameters) that have a high level of dependency are grouped into clusters or modules. DSMs have been studied in a wide variety of industries including aerospace (Grose 1994), automotive (Black et al. 1990), building design (Austin et al. 1994), manufacturing (Kusiak et al. 1994), and telecommunications (Pinkett 1998). Much of the work has been of a theoretical nature, exploring the ways that DSMs can help to better organize projects (e.g., Eppinger et al. 1994) or value the options available to a designer (e.g., Baldwin and Clark 2000). A key contribution of the DSM literature has been to highlight that the degree of modularity of a design depends not only on the *number* of dependencies between elements, but also on their *pattern* of distribution. Several studies therefore explore different types of architectures and their properties (Sosa et al. 2003, Sharman and Yassine 2004). Recent work uses DSMs to provide insights into the relationship between product architecture and organizational structure. For example, Sosa et al. (2004) find a strong tendency for design interactions and team interactions to be aligned and show instances of misalignment are more likely to occur across organizational and system boundaries.

Studies of Software Design Structure

In the field of software, the study of design structure has a long tradition. Parnas (1972) first proposed the concept of *information hiding* as a mechanism for dividing code into modular units. This concept allowed designers to separate the details of internal module design from external module interfaces, reducing the coordination costs required to develop complex software systems while increasing the ability to make changes to the design without affecting other parts of a system. Subsequent authors developed these concepts, proposing a variety of metrics, such as coupling and cohesion, to provide indicators of design structure (e.g., Selby and Basili 1988, Offut et al. 1993, Dhama 1995). However, many of these studies were theoretical in nature, using stylized examples rather than actual code, and provided little data to suggest the metrics were predictive or prescriptively useful. Recent empirical work has made progress on this front. For example, Eick et al. (2001) find code tends to decay over time, measured by the number of files that must be changed to fulfill a modification request. Banker and Slaughter (2000) show greater "structure" can mediate the effect of volatility and complexity on software enhancement costs. This study accounts for the *number* of design dependencies but not their *pattern* of distribution and, hence, does not fully capture the notion of modularity we describe.

With the recent rise in popularity of open source software, interest in the topic of architecture and modularity has received further stimulus. Prior work suggests a duality exists between a product's design and the organization that developed it (Henderson and Clark 1990) hence this new mode of organization may be expected to give rise to software that exhibits different architectural properties than that of proprietary products. Some authors suggest 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 of development to succeed (Torvalds, as quoted in DiBona et al. 1999). Such issues are important, given that theory predicts different types of architecture may incur trade-offs in terms of performance (Ulrich 1995, von Hippel 1990). Furthermore, attempts to move software from one domain to the other may be destined to fail unless we understand whether a product's architecture must first be changed, and the impact that such changes can have on a system of reasonable complexity.

Studies seeking insights into these issues yield few general conclusions, in part due to the differing approaches they take. Paulson et al. (2004) find a correlation between functions added and functions modified in open source projects but not in

proprietary projects.[2] They measure the impact of modifications rather than inherent system structure so cannot draw broader conclusions about the nature of open source architectures. Schach et al. (2002) and Yu et al. (2005) compare Linux with other open source products, showing that its design has a larger number of "unsafe" dependencies between kernel source files and nonkernel source files through the use of global variables (variables defined and used in many parts of the design). Yet we do not know how proprietary products would compare in such an analysis. Rusovan et al. (2005) also examine Linux, looking at one important design element (the Address Resolution Protocol (ARP) source file). They find it has many dependencies on other elements through direct and indirect "function calls" (requests for specific tasks to be performed). Yet this analysis may not be representative of the system as a whole or differ from proprietary systems. Finally, Mockus et al. (2002, p. 343) report in-depth case data from the Apache and Mozilla projects. They note Mozilla modules "are not as independent from one another" relative to Apache and speculate that this may be due to Mozilla's commercial legacy, but do not provide metrics to support these qualitative observations.

In sum, despite a long tradition of studying modularity dating back to Parnas's (1972) paper, studies of software design structure often exhibit common pitfalls: They examine only a small part of a design, hence cannot make claims about the system as a whole; they rely on qualitative or manual analyses, which are subjective and may not scale to larger systems; they focus on different types of dependencies, limiting the cumulative impact of the work; and they measure the "effects" of system structure (e.g., the number of file changes per modification) rather than the "causes" of these effects. As Shaw and Garlan (1996) state in their seminal work on the topic, software architecture remains "an emerging discipline."

This study contributes to the literature on modularity in general, and the study of software design structure in particular. Specifically, we use DSMs to analyze different software designs. We argue that this technique provides a rigorous and valuable way to characterize software architecture, addressing many of the pitfalls noted in the literature. Indeed, recent work has explored the use of DSMs to evaluate software design choices (Sullivan et al. 2001, Lopes and Bajracharya 2005). For example, Sullivan et al. (2001) use DSMs to formally model (and value) the concept of information hiding, the principle proposed by Parnas (1972) to divide designs into modules. Furthermore, Gomes and Joglekar (2004) use metrics derived from a DSM to predict the likelihood of outsourcing software tasks.

3. Research Methodology

In contrast to studies that use DSMs to understand how a set of future tasks or parameters should be organized, we use this technique to analyze *existing* designs. We calculate metrics from a DSM to characterize product structure. Our objective is to measure the degree of modularity of a design. We argue that this concept must be measured comparatively. That is, the absolute level of modularity for a product has no meaning; we can only determine that product A is more (or less) modular than product B. Our work is based on comparing different designs, using both cross-sectional (i.e., product A versus product B) and longitudinal (i.e., product A at time T versus product A at time $T+N$) comparisons.

Although we cannot measure modularity directly, we argue that the manifestation of different degrees of modularity can be measured from a DSM by examining the costs of dependencies between its elements. These costs depend on both the number and pattern of distribution of dependencies. We define two metrics to capture these costs, and each makes different assumptions about their nature. The first metric, *propagation cost*, assumes that all dependencies between elements, both direct and indirect, incur the same cost, regardless of where the elements are located or how long the path length is between them. The second metric, *clustered cost*, assumes that the cost of dependencies between elements will differ depending on whether elements are in the same "cluster" (or module) or are in different clusters. We define each metric below, after discussing the unit of analysis and the type of dependency we focus on.

The Unit of Analysis: The Source File

The first decision that must be made in applying the use of DSMs to a software design is the level of analysis at which the DSM is built. Choices range from high-level representations of major subsystems to the individual components that make up these subsystems. The level of analysis should be driven by the research question but must also be meaningful given the context.[3] When considering the design of a software product, there are three levels at which a DSM could be built, nested in a hierarchical fashion as follows: (a) the subsystem level, which corresponds to a group of source files that all relate to a specific

[2] This correlation does not necessarily imply open source projects modify more files per function added than proprietary projects. It only implies that there is a relationship between these two variables.

[3] For example, it is more insightful to examine the DSM for an automobile at the subsystem level (e.g., brakes, suspension, and powertrain) than at the raw component level (e.g., nuts and bolts).

part of the design; (b) the source-file level, which corresponds to a collection of programming instructions (source code) that performs a related group of functions; and (c) the function level, which corresponds to a set of programming instructions (source code) that performs a highly specific task.

We choose to analyze designs at the source-file level for several reasons. First, source files tend to group functions of a common nature into a single place and, hence, are a logical unit of analysis. For example, a programmer may expect to have functions related to the printer in a source file named "printer_driver.c." Source files also contain header information and overview comments that apply to all functions in the file. Second, tasks and responsibilities are typically allocated to programmers at the source-file level. Hence, this unit of analysis reflects how managers of software projects exert influence on product structure. Third, source control and configuration management tools use the source file as their unit of analysis. Programmers "check out" source files for editing, modify the code, then "check in" the new version once work is complete. Finally, other empirical studies that focus on software structure typically use the source file as the unit of analysis (e.g., Schach et al. 2002, Rusovan et al. 2005).[4]

The Type of Dependency: A "Function Call"

We capture dependencies between source files by examining the "function calls" that each source file makes. A function call is an instruction that requests a specific task to be executed by the program. The function called may or may not be located within the source file that originated 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, which is located in Sourcefile2, then Sourcefile1 depends on (or "needs to know" about) Sourcefile2. This dependency is marked in the DSM in location $(1, 2)$.[5] Note that this dependency does not imply that Sourcefile2 depends on Sourcefile1; that is, the dependency is not symmetric.[6] This would be true only if a function in Sourcefile2 called a function in Sourcefile1.

We note that function calls are only one important type of dependency that exists in a software design. Several authors have developed comprehensive categorizations of dependency types (e.g., Garlan and Shaw 1996, Dellarocas 1996). Others focus on a single type, such as global variables, to provide insight into system structure (Schach et al. 2002). We argue that DSMs provide a general technique by which different notions of dependency can be analyzed, each providing a different "lens" through which to view the system (Sangal et al. 2005). We choose to focus on function calls given this type of dependency is at the heart of many analysis tools (e.g., call graphs, see Murphy et al. 1998) and is used in prior work that examines system structure (Banker and Slaughter 2000, Rusovan et al. 2005).

To capture function calls between source files, we input the source code for a product into a tool called a "call graph extractor" (Murphy et al. 1998). This tool is used to obtain a better understanding of code structure and interactions between different parts of the code. We note that function calls can be extracted statically or dynamically. Static calls are extracted from code that is not in an execution state and use source code for input. Dynamic calls are extracted from code in an execution state and use executable code and the program state as input. 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 structure of the design from the programmer's perspective.[7] Rather than develop our own call extractor, we tested several products that had the ability to process source code written in both procedural and object-oriented languages (e.g., C and C++), captured indirect calls (dependencies that flow through intermediate files), could be run in an automated fashion, and output data in a format that could be input to a DSM (i.e., in matrix form). A product called Understand C++[8] was selected given it best met all these criteria.

Analyzing Design Structure Using a DSM

Once a DSM is populated with function call dependencies, we can examine it visually using what we call the *architectural view* of the system. In software designs, programmers tend to group source files of a related nature into "directories" that are organized in a nested fashion. The architectural view of a DSM groups each source file into a series of clusters as

[4] The source-file level is granular enough to be meaningful without being overwhelming. For example, Linux version 2.5 has 10 subsystems, over 4,000 source files, and over 60,000 functions.

[5] We use the convention (row number, column number) to describe entries in a DSM.

[6] We assert that the designer of a calling function needs to "know about" the function being called. Although the reverse may be true (a function may need to know about the functions that call it) such reverse dependence would limit the usefulness of the function; it could be used only for purposes that were known at the time of its definition.

[7] "Compiler extractors" are used mainly to help compile the software into an executable form. "Design extractors" are used to aid programmers in determining source relationships. We use the latter type of extractor.

[8] Understand C++ is distributed by Scientific Toolworks, Inc.; see http://www.scitools.com for details.

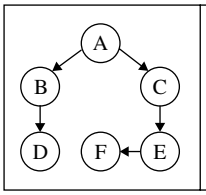
defined by the directory structure, with boxes drawn around each layer in the hierarchy. The result is a map of source-file dependencies, organized by the programmer's view of the system.[9] As we shall see later, these maps are a powerful aid to assessing design structure. But to compare different designs we must define metrics that characterize these maps. Below, we define two metrics to achieve this objective.

Propagation Cost. The first method by which we characterize the structure of a design is by measuring the degree of "coupling" it exhibits, 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 builds on the concept of visibility (Sharman and Yassine 2004), which in turn is based on the concept of reachability matrices (Warfield 1973). To clarify how we measure this concept, we use a simple example. Consider the element relationships shown in Figure 1. We note that element A depends on elements B and C. So any change to element B has the potential to impact element A. Similarly, elements B and C depend on elements D and E, respectively, so any change to element D may have a direct impact on element B and an indirect impact on element A with a "path length" of 2. Finally, we note that any change to element F may have a direct impact on element E, an indirect impact on element C with a path length of 2, and an indirect impact on element A with a path length of 3. In this system, there are no indirect dependencies for path lengths of 4 or more.

We use the technique of matrix multiplication to identify the "visibility" of any given element for any given path length. 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. By summing these matrices together we derive the visibility matrix V, showing the dependencies that exist for all possible path lengths up to n. We choose to include the matrix for $n = 0$ (i.e., a path length of 0) when calculating the visibility matrix, implying that a change to an element will always affect itself. Figure 2 illustrates the derivation of the visibility matrix for the example we describe above.

From the visibility matrix, we can construct several metrics to give insight into a system's structure. The first, called "fan-out visibility," is obtained by summing along the rows of the visibility matrix, and dividing by the total number of elements. An element with high fan-out visibility depends on (or calls functions within) many other elements. The second, called "fan-in visibility," is obtained by summing down the columns of the visibility matrix, and dividing by the total number of elements. An element with high fan-in visibility has many other elements that depend on it (or call functions within it). In our example, element A has a fan-out visibility of 6/6th (or 100%), meaning that it depends on all elements in the system, and has a fan-in visibility of 1/6th, meaning that it is visible only to itself.

Figure 1 Example System in Graphical and Dependency Matrix Form



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

To summarize visibility at the system level, we compute the average fan-out and fan-in visibility of all elements in the system. Due to the symmetric nature of dependency relationships (i.e., for every fan-out, there is a corresponding fan-in) these are identical. We call the resulting metric "propagation cost." Intuitively, this measures the proportion of elements that could be affected, on average, when a change is made to one element in the system. In the example above, we can calculate propagation cost from fan-out visibility ($[6+2+3+1+2+1]/6*6 = 42\%$) or fan-in visibility ($[1+2+2+3+3+4]/6*6 = 42\%$).

We note that in the example we consider, the design does not contain multiple paths between elements and the dependency relationships are purely hierarchical.[10] This will not generally be the case for complex systems. In particular, multiple paths bring the possibility that there is more than one route through which element A depends on element B. As a result, we limit values in the visibility matrix to be binary, capturing only the fact that there exists a dependency, and not the number of possible paths that this dependency can take. Similarly, a nonhierarchical system will contain one or more "cycles," meaning successive powers of the dependency matrix will not converge to zero.[11] To avoid this problem, we raise the dependency matrix to a maximum power dictated by the number of elements in the system (i.e., the longest possible path length). Given that we limit entries in

[9] The directory structure and architectural view of Linux version 0.01 are displayed in Appendix A; the appendices are available on the *Management Science* website at http://mansci.pubs.informs.org/ecompanion.html.

[10] A hierarchical system can be identified by the fact that its dependency matrix can be rearranged such that all nonzero entries are below the diagonal, which is called a "lower-triangular" form.

[11] For example, a relationship whereby element A depends on element B which depends on element A.

Figure 2 Successive Powers of the Dependency 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**

the visibility matrix to binary values, the fact that cyclic relationships appear multiple times ultimately has no impact on the output.

Clustered Cost. The assumption behind the measure of propagation cost is that each dependency between elements incurs the *same* cost, wherever these elements are located in the design and however long the path length between them. By contrast, our second measure attributes a *different* cost to each dependency according to where the elements are located. In essence, we posit that there is an "idealized modular form" in which dependencies are grouped into clusters of tightly connected elements (or "modules") with no dependencies between these clusters. Dependencies between elements in the same cluster are assumed to incur a low cost; those between elements in different clusters are assumed to incur a high cost. This type of structure is related to the notion of decomposability (Simon 1962) and is a characteristic of modular systems (Baldwin and Clark 2000). Figure 3 shows a DSM with dependencies grouped in this way.

To assess the degree to which a design approximates this idealized modular form, we cluster the DSM to determine the most appropriate grouping of elements. Although a range of algorithms can be used for clustering, most define a cost function to measure the "goodness" of a solution and invoke a search technique to find the lowest cost grouping of elements. The approach we adopt is based on prior DSM research (Pimmler and Eppinger 1994, Idicula 1995, Fernandez 1998, Thebeau 2001) adapted for the context of software development.[12] In particular, we first identify and account for what we call "buses"—source files that are called by a large number of other source files. We then allocate a cost to each dependency, depending on whether the dependency is between elements that are in the same cluster. Finally, we adopt an iterative clustering process in which source files chosen at random receive bids to join clusters based on their level of interdependence. We describe this process below.

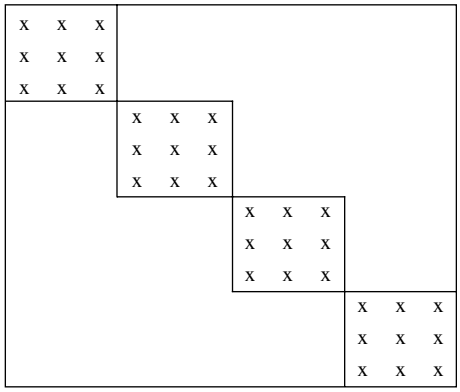
Prior work has highlighted the role of design elements called buses (e.g., Ulrich 1995). These elements have particular importance in software development in that designs often make use of common functions that are called by many other elements.[13] We call these "vertical buses," given they appear in a DSM as an element with many vertical dependencies. The use of vertical buses avoids duplication of effort and ensures a consistent approach to systemwide functions. Buses are typically identified early in the design process and have well-defined, stable interfaces (Baldwin and Clark 2000). To identify vertical buses, we assess the degree to which a particular source file is called by other elements in the system (in terms of the percentage of source files that call it). We then examine whether the result is above or below a parameter called the "bus threshold." If higher, we treat dependencies on this source file differently in assessing their cost (described below). Note that there is no correct answer as to the connectivity that should constitute the threshold for a bus. In our work, we initially set this parameter to 10%, then vary it from 10%–100% to evaluate the potential impact this has on results.

Once vertical buses have been identified, a cost is allocated to each dependency depending on the location of the elements between which the dependency exists. Specifically, when considering a dependency

[12] Prior work clusters DSMs to find a grouping of tasks involved in developing a *new* design. We use clustering to identify an appropriate grouping of related elements in an *existing* design.

[13] Examples include library code, global functions such as "print to screen," or error-handling code.

Figure 3 A DSM with Dependencies in an "Idealized Modular Form"



between elements i and j, the cost of the dependency takes one of the following three forms:

$$DependencyCost(i \rightarrow j \mid j \text{ is a vertical bus}) = d_{ij}$$

$$DependencyCost(i \rightarrow j \mid \text{in same cluster}) = d_{ij} * n^{\lambda}$$

$$DependencyCost(i \rightarrow j \mid \text{not in same cluster}) = d_{ij} * N^{\lambda}.$$

Where d_{ij} is a binary variable[14] indicating the presence of a dependency between i and j; n is the size of the cluster that i and j are located within; N is the DSM size; and λ is a user-defined parameter.[15] Note that prior to clustering, all elements are assumed to be in a "singleton" cluster consisting of only itself.

The clustering algorithm attempts to determine the optimal allocation of nonbus elements to clusters such that the sum of dependency costs is minimized. The challenge is that for a design of moderate or greater complexity the solution space becomes too large to search exhaustively. Hence an iterative search process is adopted, whereby randomly selected elements are the subject of "bids" from existing clusters that have a dependency on this element. Each bidding cluster bids an amount equal to the marginal reduction in cost that would result from the element joining it.[16] The element is awarded to the cluster with the highest bid. If there are no positive bids, the element remains in its current location.[17] The bidding process continues in an iterative manner, stopping when the number of iterations with no improvement in cost exceeds a threshold defined by the user. In practice, we find that setting this threshold equal to the size of the DSM ensures the algorithm converges to a low cost solution.[18] We should note that although the clustering process will tend to group elements with high levels of interdependency together, there are limits to the size of clusters. In particular, adding an element increases the cost of other dependencies in the cluster, through its impact on cluster size (i.e., an increase in the parameter n). Hence, the algorithm will allocate an element to a cluster only when the reduced costs of dependencies with the element *exceed* the added costs borne by other dependencies.

[14] d_{ij} can also be set to the *strength* of dependency based on the number of calls between elements.

[15] There is no "correct" value for λ. Analysis proceeds by setting this parameter to some value, comparing the metrics for the design structures of interest, and testing the sensitivity of results to parameter changes. We nominally set $\lambda = 2$, given the number of potential interactions increases by a power-law with the number of elements in a cluster.

[16] The method by which bidding occurs is described in Appendix B.

[17] Our clustering algorithm is hierarchical in that it does not allow elements to be in more than one cluster.

4. Empirical Setting

Our study had two aims. First, to explore the differences in design structure between two software products that had been developed using different organizational modes, namely open source and proprietary development. And, second, to examine a concentrated effort to redesign a product's architecture with the goal of making it more modular. These objectives led directly to the choice of the two software products that we examine: Linux (or more strictly, the Linux Kernel) and Mozilla.

Linux is a UNIX-like operating system that is the best known of all open source software products. Since its release onto the Internet in 1991, hundreds of developers have contributed to its development (DiBona et al. 1999). Linux has gained significant market share in the server operating system market and continues to develop at a rapid pace in terms of the amount of new code and functionality added each year (MacCormack and Herman 2000a). It represents one of the most successful examples of open source software and, hence, is well represented in studies that seek to evaluate the performance and nature of open source development (e.g., Godfrey and Tu 2000, Schach et al. 2002, Rusovan et al. 2005). For our purposes, Linux represents a "prototypical" example of the open source development model.

Mozilla is also an open source product, however its origins lie in a product developed using a proprietary development model. Specifically, in March 1998, Netscape released the code for its Navigator Web browser onto the Internet, renaming the software Mozilla. The release of the source code was part of a strategy to respond to increasing competition in the browser market. Netscape hoped that volunteer developers would contribute to the design in the same fashion as was happening with other open source products. We use the first release of Mozilla as a proxy

[18] Given the nature of the search process, the outcome will differ by a small margin each time it is repeated.

for a design developed using a proprietary methodology (i.e., a dedicated team of individuals located in a single location).[19] This product comprised more than 1,500 source files. By contrast, the Linux kernel comprised fewer than 50 source files when first released in 1991. Comparing the first release of Mozilla with a similarly sized version of Linux allows us to compare a product developed with a proprietary model to one in which 95% of the design (in terms of source files) has evolved through the use of an open source development model.

We chose to use Mozilla as a proxy for a proprietary product to take advantage of a "natural experiment" that occurred during the fall of 1998. Specifically, at this time, a redesign effort was carried out with the aim of making Mozilla "more modular."[20] The reason for this redesign, as indicated in accounts of Mozilla's history, was the effort involved in understanding and contributing to the code base (Cusumano and Yoffie 1998). The code was thought too tightly coupled for would-be contributors to modify without having to examine the impact on many other parts of the system. The redesign aimed to make the design more modular and, hence, more attractive to potential contributors. The effort was carried out over several months by a small team of developers, most of who worked for Netscape (the firm sold a commercial browser that used Mozilla as its base). Given Mozilla is in the public domain, we can examine versions of the product both before and after the redesign effort to evaluate its impact.

5. Empirical Results

We divide our results into two parts. First, we compare the first version of Mozilla—a proxy for a proprietary developed product—with a version of Linux of comparable size (in source files). We then examine the longitudinal evolution of Mozilla, paying particular attention to the redesign effort that aimed to make the product more modular. For each design, we report data on the number of source files, the number of dependencies, the density of the DSM (i.e., the number of dependencies per source-file pair), the propagation cost, and the clustered cost. We also provide data on the average complexity of source files, in terms of the number of functions and lines of code.

A Comparison of the Design Structures of Linux and Mozilla

We first compare the architectural views of Linux and Mozilla (see Figure 4). From these plots we can make some overall comments about their structure. At the highest level, both are architected into a small number of major subsystems, within which are smaller groups of source files. Linux however, possesses more vertical buses—source files that are called by many other source files in the system (see annotation A). Indeed, our analysis identifies 14 vertical buses in Linux versus two in Mozilla. In the upper left-hand corner of Mozilla's DSM we see a few large groups of source files that are very tightly connected (see annotation B). By contrast, Linux comprises a larger number of small groups that are less tightly connected (i.e., fewer dependencies between them). Finally, in Mozilla, we note one group of source files composing a major subsystem that has many dependencies above it, indicating it is tightly connected to other parts of the system (see annotation C). By contrast, apart from the vertical buses discussed above, the subsystems in Linux appear to be less tightly connected to the rest of the system.

Table 1 shows quantitative data comparing Mozilla and Linux. We first note that Mozilla has fewer dependencies between source files than Linux. Mozilla has a density of 2.4 dependencies per 1,000 source-file pairs.[21] By contrast, Linux has a density of 3.4 dependencies per 1,000 source-file pairs, over 40% greater than that of Mozilla. At first sight, one might imagine that this would imply Mozilla is more modular than Linux. But these data only give insight into the *number* of dependencies and not the *pattern* of distribution of these dependencies. Our measures of propagation cost and clustered cost, by contrast, take both these aspects of design structure into account. The propagation cost for Mozilla is 17.35% versus 5.16% for Linux, a striking difference. This implies that the design of Linux is much more loosely coupled than the first version of Mozilla. A change to a source file in Mozilla has the potential to impact three times as many source files, on average, as a similar change in Linux (of course, the specific results would depend on the particular source file being modified). This finding is supported by the evidence that comes from clustering each DSM. The clustered cost for Linux is around 70% of that for Mozilla, despite having a larger number of dependencies. Finally, we also note differences in the average complexity of source files. Mozilla contains 50% more functions per file, although the difference in terms of lines of code is not as significant.

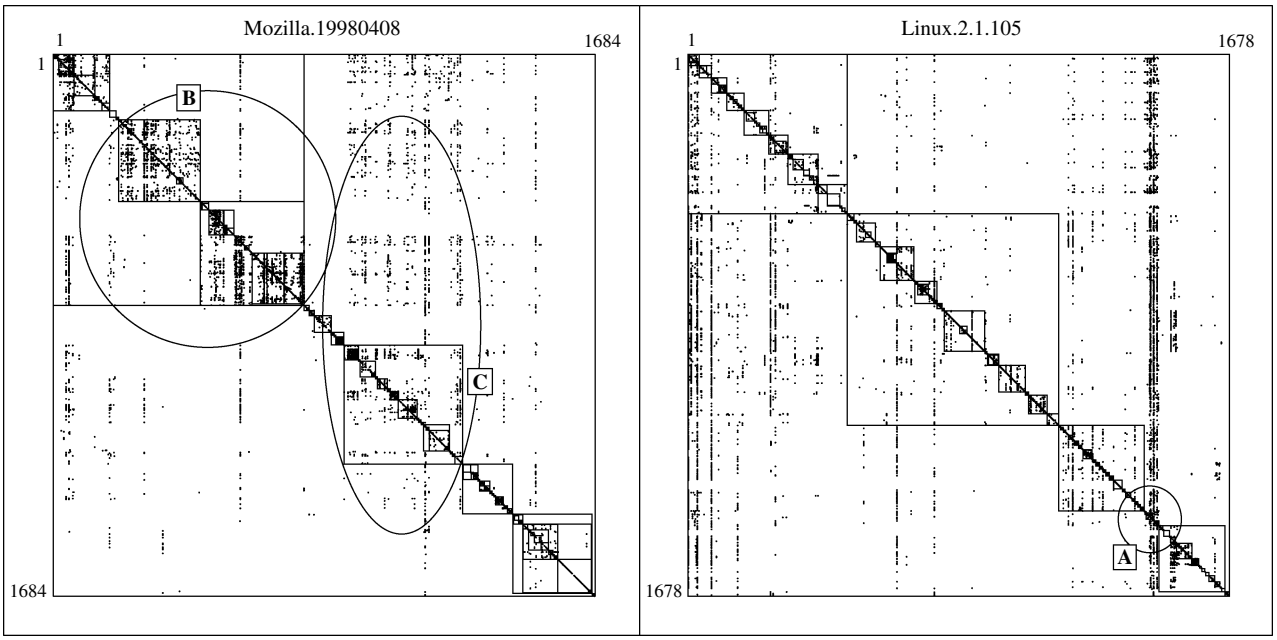
To summarize, we have compared the design structures of two systems of similar size (in source files), one having been developed with an open source

[19] The idea for Netscape's browser came from a product called Mosaic, developed at the University of Illinois. Netscape's code, however, was developed independently, was far more complex, and targeted commercial users.

[20] *Source.* Interviews with Mozilla team members during spring 1999.

[21] Our figures illustrate that most software DSMs constructed at the source-file level are relatively sparse (i.e., they have very few dependencies per possible source-file pair).

Figure 4 **A Comparison of the First Version of Mozilla and a Comparable Version of Linux**



development model, the other having been developed using a proprietary model. The architectural view reveals visual differences in design structure consistent with an interpretation that Linux uses more vertical buses and is more loosely coupled than Mozilla. The metrics that measure design structure are consistent with this view. We conclude that the first version of Mozilla was significantly less modular than a comparable version of Linux.

The question that arises given these results concerns the degree to which the differences observed are a product of the organizational modes used in development, or reflects purposeful strategies employed by their architects. For Linux, there is evidence that the original author, Linus Torvalds, emphasized modularity as a design criterion (DiBona et al. 1999). The design was also built on the heritage of the UNIX and MINIX operating systems, which are relatively modular at the system level. By contrast, we know little about the intentions of the original architect(s) of Mozilla. We can speculate that as an early entrant to a new product category (the Web browser) there was likely a greater focus on superior product performance (e.g., speed of operation) than on achieving high levels of product modularity. Given the potential trade-off that exists between product performance and modularity (Ulrich 1995) this may have led Mozilla's original architect(s) to adopt a more tightly coupled or "integral" design.

There is, however, another explanation for our results, relating to the different functions these products perform. Put simply, an operating system may require a different design structure to a Web browser as a result of the different tasks and performance demands it faces. Unless we have a true apples-to-apples comparison, it is difficult to draw general conclusions. One solution would be to obtain the source code of an operating system developed in a proprietary fashion and compare its design to that of Linux. Another option however, is to identify a purposeful effort to change the architecture of Mozilla to a more modular form, and gauge the impact of this effort on its design. Fortunately, such a "natural experiment" did in fact occur during Mozilla's ongoing development. We now examine its impact.

Table 1 **Comparisons Between Comparable Versions of Linux and Mozilla***

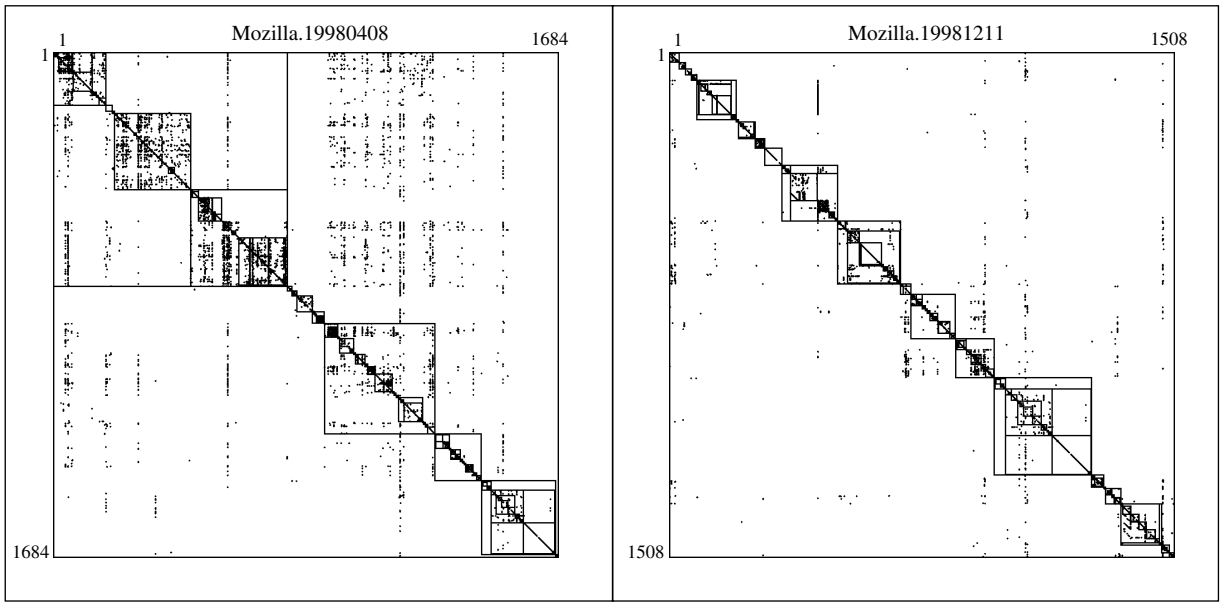
	Mozilla (98-04-08)	Linux 2.1.105
Source files (DSM elements)	1,684	1,678
Dependencies	6,717	9,110
Density (per 1,000 source-file pairs)	2.4	3.4
Propagation cost	17.35%	5.82%
Clustered cost	5,323,915,033	3,697,742,107
Functions per source file	17.7	12.8
Lines of code per source file	733	670

*The clustered cost results we report use binary dependencies for d (rather than strengths) and use $\lambda = 2$. Sensitivity analyses using a range of other assumptions did not change the comparative results.

A Comparison of the Design Structures of Mozilla Before and After a Redesign

Netscape's Navigator browser was released under an open source license in March 1998. At that point, the

Figure 5 A Comparison of Mozilla Before and After a Purposeful Redesign Effort



source code became known as Mozilla, and developers from around the world began to contribute to the project. In the fall of 1998, a group of Mozilla's core developers, most of whom worked for Netscape, redesigned the code base to make it more modular and, hence, more attractive to contributors. The results above point to the nature of the problem Mozilla faced; anyone making a change to a source file would have to check three times as many possible interactions, on average, as a contributor to Linux. The redesign effort that followed encompassed several objectives: to remove redundant or obsolete source files; to rewrite major parts of the code to run more efficiently; and to rearchitect the product, grouping source files into smaller modules with fewer connections between them.

Prior to the redesign, Mozilla comprised 2,333 source files. Afterwards, the software comprised only 1,508 source files. There are two ways to examine the impact of this redesign, given the number of source files was reduced so dramatically. The first is to compare the design that emerged with an earlier Mozilla version of similar size. The second is to compare the design immediately *prior* to the redesign with a later Mozilla version of similar size. Figure 5 shows the comparison of the architectural views for the former. The results are striking. The redesigned version of Mozilla comprises much smaller clusters of source files. There are very few dependencies *between* these clusters, or indeed between any of the individual source files. Much of the DSM is "white space."

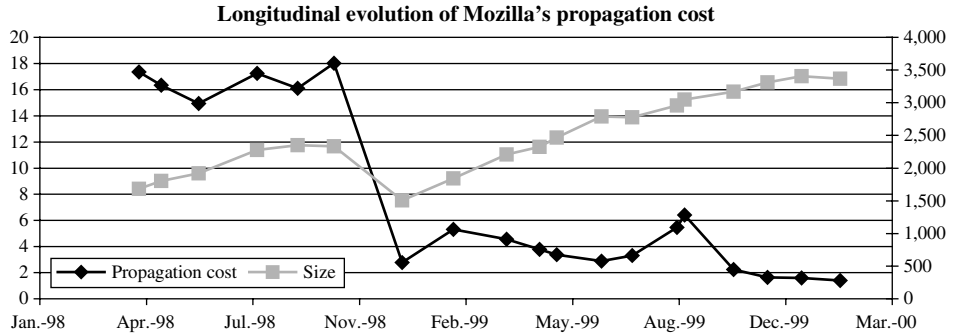
Table 2 shows quantitative data comparing Mozilla before and after the redesign. We first note that the redesigned product has only 1.3 dependencies per thousand source-file pairs, compared to 2.4 in the earlier version. Hence the redesign significantly reduced the *number* of dependencies between different source files in the design. Furthermore, there are significant changes when we look at the metrics that account for both the number and *pattern* of dependencies. Specifically, the propagation cost of the design has dropped from 17.35% to 2.78%. That is, changes to a source file have the potential to impact 80% fewer source files, on average, after the redesign. This reduction is mirrored by the reduction in clustered cost, which drops to around 30% of its previous level. These results are obtained with a small reduction in the complexity of source files, as measured by the number of functions and lines of code. We conclude that the efforts

Table 2 Comparisons of Mozilla Prior To and *Immediately* After the Redesign*

	Mozilla (98-04-08)	Mozilla (98-12-11)
Source files (DSM elements)	1,684	1,508
Dependencies	6,717	3,037
Density (per 1,000 source-file pairs)	2.4	1.3
Propagation cost	17.35%	2.78%
Clustered cost	5,323,915,033	1,636,799,495
Functions per source file	17.7	16.8
Lines of code per source file	733	530

*Note that there is no public version of Mozilla prior to the redesign with around 1,500 source files, hence our comparison design—version 1998-04-08—has a greater number (1,684 source files) and would therefore be expected to have a higher clustered cost, although not necessarily a higher propagation cost.

Figure 6 The Impact of a Redesign Effort on Mozilla's Propagation Cost



to make the design more modular were extremely successful.[22]

It is insightful to look at the impact of this redesign effort in the context of the evolution of Mozilla's design structure over time. To this effect, we plot the evolution of Mozilla's propagation cost and clustered cost in Figures 6 and 7, respectively. The results are striking. The redesign effort reduced propagation cost from a level varying between 15%–18% to a level varying between 2%–6%. Similarly, the redesign effort reduced clustered cost by an order of magnitude (although much of this fall comes from the reduction in the number of source files, and hence the need for the paired comparisons we conduct above). To our knowledge, these plots represent the first quantitative view of the evolution of a product's architecture over time. They show that the impact of the Mozilla redesign effort was unambiguous and significant in magnitude. Specifically, the architecture of the product was much more modular after the redesign effort. Furthermore, we have found no public data that suggests the redesigned product was substantially inferior in terms of functionality or performance. Indeed, discussions with Mozilla developers suggest the redesign *improved* performance on some dimensions.[23] We are left with an intriguing conclusion. There was no requirement for Mozilla to possess as tightly coupled a design structure as it did prior to the redesign effort. Hence our previous results are not explained by the differing functions that a browser and an operating system must perform. The evidence therefore suggests that the design structure that evolved was due to either the mode of organization involved in its development, or to specific choices made by the team's architects (discussed below), or both.[24]

In Table 3, we compare the metrics for the redesigned version of Mozilla with a comparable version of Linux. We find that the redesign succeeded in making Mozilla more modular than Linux. Specifically, a change to a source file in the redesigned Mozilla, on average, has the potential to impact fewer than half the source files than a change in Linux. Similarly, the clustered cost of the design is now only around 50% of that of Linux. Interestingly, however, we note that the density of dependencies in Mozilla has dropped to around 35% of that of Linux. Hence, on a per-dependency basis, Mozilla does not fare quite as well as perhaps it should. Such a comparison, while exploratory, implies that much of Mozilla's post-redesign advantage in modularity relative to Linux comes from the reduction in the *number* of dependencies as opposed to a superior configuration in terms of their *pattern* of distribution.[25]

6. Discussion

The distinctive contribution of our work is in applying the technique of DSMs to the analysis of software architecture. In particular, we process source code for existing software designs, extract and map dependencies between their component elements, and calculate metrics that give us insight into system structure. We use these metrics both to compare designs developed via different modes of organization—open source versus proprietary development—and to track the evolution of one design over time, giving insight into the impact of a redesign effort. Our study examined only two products, and hence must be regarded as exploratory. However, the novel research design we employ, in using both cross-sectional and longitudinal

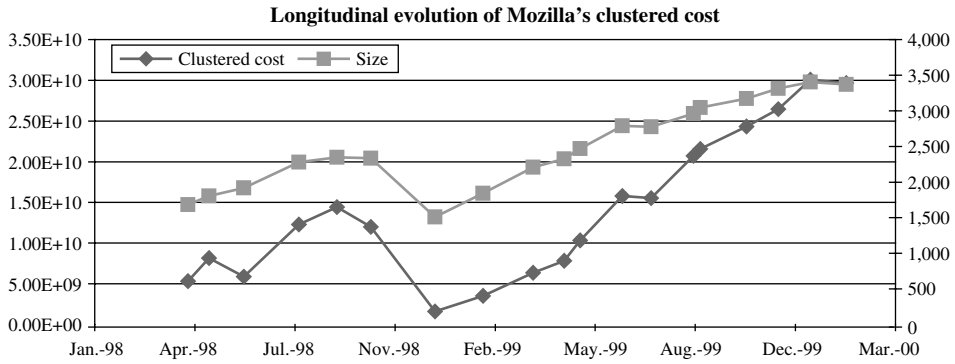
[22] We see similar results when we compare versions of Mozilla *immediately* before and after the redesign.

[23] *Source.* Interviews with Mozilla team members during spring 1999.

[24] We should note that plots such as these have a very practical use, by making visible the evolution of a product's architecture. Specifically, problematic trends can be identified quickly and corrective action planned with defined objectives for the impact on a system's architecture. While experienced architects routinely perform this role using an intuition of the structure of a code base, we previously lacked measures to explain their intuitions.

[25] Mozilla's redesign clearly improved the pattern of distribution of dependencies *relative to the prior version of the design*, given its propagation cost declined so significantly (see Table 2).

Figure 7 The Impact of a Redesign Effort on Mozilla's Clustered Cost



comparisons, yields insight on several important dimensions.

First, our results demonstrate that there are substantial differences in modularity between different software systems of comparable size and function. Although this may not be surprising, the fact that we can *measure* the size of such differences is a critical advance in our understanding of architecture. We find that the systems we examine vary by a factor of five in terms of the potential for a design change to propagate to other source files in a system, a measure we refer to as propagation cost. Such a result has significant implications for those responsible for software system development and maintenance. It suggests that systems are likely to vary dramatically in terms of their robustness to change and the costs and efficiency of future enhancements. Indeed, given we examine two versions of a product that perform the same function but with very different architectures, our results highlight the value of design as a *managerial choice*. That is, architecture is not wholly determined by function, but results from purposeful choices by designers reacting to the incentives and structures that surround them.

In this respect, our study generates useful data on the question of whether a link exists between a product's architecture and the structure of the organization from which it comes. We show that the architecture of a product developed by a highly distributed team of developers (Linux) was more modular than another product of similar size developed by a colocated team of developers (Mozilla). Critically, however, we find that a purposeful effort to redesign Mozilla resulted in an architecture with greater modularity. Hence, the initial differences between Linux and Mozilla were not driven by the different functional requirements of these products. These results are consistent with the idea that a product's design mirrors the organization that develops it. We must ask, however, how such a dynamic occurs?

Table 3 Comparisons of Mozilla *Immediately* After the Redesign and Linux

	Mozilla (98-12-11)	Linux (2.1.88)
Source files (DSM elements)	1,508	1,538
Dependencies	3,037	8,519
Density	1.3	3.6
Propagation cost	2.78%	5.65%
Clustered cost	1,636,799,495	3,175,246,929
Functions per source file	16.8	12.6
Lines of code per source file	530	670

Our observations are consistent with two rival hypotheses. The first is that each design *evolved* to reflect its development environment. In Mozilla's case, a focused team employed by one firm and located at a single site developed the design. Problems were solved by face-to-face interaction, and performance "tweaked" by taking advantage of the access that module developers had to the information and solutions developed in other modules. Even if not an explicit managerial choice, the design naturally became more tightly coupled. In Linux's case, however, hundreds of distributed developers situated around the world developed the design in a loosely coupled fashion. Face-to-face communication was almost nonexistent as most developers never met. The design structure that evolved therefore relied on fewer connections between modules and, hence, was more modular. Alternatively, our observations may reflect *purposeful choices* made by the original designers in response to specific contextual challenges. For Netscape, the aim was to develop a product that maximized product performance, given a dedicated team of developers and a competitive environment that demanded a product be shipped as quickly as possible. The benefits of modularity, given the context at the time, would not necessarily have been seen as significant. By contrast, for Linus Torvalds, the benefits of modularity were substantial. Without modularity, there was little hope that contributors could understand enough of the design to contribute in a meaningful way, or develop new features and fix existing defects without affecting many other parts of

the design. Linux therefore *needed* to be modular to attract and facilitate a developer community.

We note that not all open source projects are characterized by broad participation by many developers or highly distributed team structures. Indeed, open source projects are often small efforts that involve only a handful of developers (Healy and Schussman 2003). In addition, proprietary projects are not always colocated, but may involve developers from multiple sites and organizations. In this respect, we caution that our observations here are related to specific organizational characteristics of the projects we examine, and not the mere fact that the software is open source or proprietary. This is an important distinction, given it points to the type of work needed to further explore the rival hypotheses we describe. Specifically, questions on the relationship between organization and architecture can be answered *only* by capturing measures of both factors. Indeed, our ongoing work supports such a view, given that we have found open source products with tightly coupled architectures in which contributions are concentrated among a handful of developers (Rusnak 2005). Our study therefore opens the door to a deeper understanding of the complex relationship between a product's architecture and the community that grows around it.

Our results show that purposeful efforts to redesign a product's architecture can have a significant impact. This is surprising, given much work suggests architectural change is difficult (Marples 1961, Henderson and Clark 1990). In this respect, we consider three possible explanations for our findings: First, this may be a unique feature of software products (i.e., the ease of architectural change) as compared to physical products. We note, however, that several popular software products sold today contain code that is many years old despite major efforts to update their designs (and one assumes their architectures) over time (e.g., MacCormack and Herman 2000b). Second, Mozilla may not be sufficiently complex to make architectural change a challenge. We note, however, that at the time of the redesign, the product contained over one million lines of code, which would take a team of 100 developers more than two years to complete using standard productivity metrics. Finally, there may be something specific about a Web browser's design that makes architectural change easier to achieve. Still, we can find no evidence to support this view. We are left to speculate that with a sufficiently talented and motivated team, it is possible to achieve substantial changes in a product's architecture.

This latter result is important in the context of open source software. Recent years have seen an increase in the number of proprietary products released under an open source license, with the hope that a developer community will converge around them. Examples include Mozilla, Open Office (a personal productivity suite), and Eclipse (a suite of developer tools). The Mozilla experience, however, suggests that proprietary products may not be well suited to distributed development if they have tightly coupled architectures. There is a need to create an "architecture for participation" that promotes ease of understanding by limiting module size and ease of contribution by minimizing the propagation of design changes. In Mozilla's case, the redesign to a more modular form was followed by an increase in the number of contributors[26] (Mockus et al. 2002). Subsequent improvements led to a product called Firefox, which gained both critical acclaim and market share.[27] We speculate that without the changes made in 1998, such developments would have been unlikely, and the Mozilla experiment may well have failed.

So how is such structural change achieved? Our results illustrate that increased modularity can be realized by reducing the number of design dependencies or by rearranging their pattern of distribution. With regard to the first strategy, our work suggests that substantial latitude exists to reduce dependencies without a major impact on performance (the redesign of Mozilla reduced dependencies by around 50%). This implies the need for a deeper understanding of how and why dependencies exist and the mechanisms through which they can be managed. With regard to the second strategy, our work shows that designs with greater numbers of dependencies are not necessarily less modular than those with fewer. Poorly placed dependencies, especially those that link otherwise independent modules, may result in a cascade of unwanted and hard-to-detect *indirect* interactions. Our results suggest that purposeful actions to reduce such "rogue" dependencies can be effective (the redesign of Mozilla reduced propagation cost by over 80%).

Our work opens up a number of areas for future study. With respect to methods, we believe DSMs provide a powerful lens with which to examine issues of dependency in software. Although we focus on one type of dependency, our method can be generalized to others, assuming that they can be identified from source code. With respect to studies of architecture, our work provides visibility of a phenomena that was previously hidden and metrics with which to conduct further studies. In particular, there is value in asking whether different types of product (e.g., databases versus operating systems) require different types of architecture, and if so, identifying their characteristics.

[26] *Source.* Interviews with Mozilla team members during spring 1999.

[27] See http://www.mozilla.org/products/firefox/ for details.

Our ongoing work reveals large variations in the modularity of different systems (Rusnak 2005), but we lack sufficient data to determine causes. Building a database of different system types will help answer this question. Finally, our work facilitates the study of a topic of critical importance that has proven elusive to quantify: the potential performance trade-offs from architectures with different characteristics. There are strong theoretical arguments why such trade-offs exist, yet little empirical evidence to confirm their presence. The Mozilla redesign is a case in point. How was it possible to increase modularity without degrading other aspects of performance? We speculate that many designs are not at the performance "frontier" where a trade-off exists but are positioned below it due to architectural inefficiencies or "slack." If this is true, there may be considerable scope to improve a design along one or more dimensions without incurring a performance penalty. Exploring such issues through careful measurement of architectural characteristics and product performance should help reveal strategies for moving designs *toward* the performance frontier. It will also help us understand the trade-offs that are ultimately involved in moving a design *along* it.

An online supplement to this paper is available on the *Management Science* website (http://mansci.pubs.informs.org/ecompanion.html).

Acknowledgments

The authors are grateful for the comments of Steve Eppinger, Jim Herbsleb, Josh Lerner, Stephen Schach, Sandra Slaughter, Sidney Winter, and participants from The Wharton School Technology miniconference; the Information Systems seminar at the Tepper School of Business, Carnegie Mellon University, and the Technology and Operations Management seminar at Harvard Business School. All errors remain the authors' own.

References

Austin, S. A., A. N. Baldwin, A. Newton. 1994. Manipulating the flow of design information to improve the programming of building design. *Construction Management Econom.* **12**(5) 445–455.

Baldwin, C. Y., K. B. Clark. 2000. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, MA.

Banker, R. D., S. A. Slaughter. 2000. The moderating effects of structure on volatility and complexity in software enhancement. *Inform. Systems Res.* **11**(3) 219–240.

Black, T. A., C. H. Fine, E. M. Sachs. 1990. A method for systems design using precedence relationships: An application to automotive brake systems. Working Paper 3208, Sloan School of Management, MIT, Cambridge, MA.

Cusumano, M., D. Yoffie. 1998. *Competing on Internet Time*. Free Press, New York.

Dellarocas, C. D. 1996. A coordination perspective on software architecture: Towards a design handbook for integrating software components. Ph.D. thesis, MIT, Cambridge, MA.

Dhama, H. 1995. Quantitative models of cohesion and coupling in software. *J. Systems Software* **29** 65–74.

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

Eick, S., T. L. Graves, A. F. Karr, J. S. Marron, A. Mockus. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Software Engrg.* **27**(1) 1–12.

Eppinger, S. D., D. E. Whitney, R. P. Smith, D. Gebala. 1994. A model based method for organizing tasks in product development. *Res. Engrg. Design* **6**(1) 1–13.

Fernandez, C. I. G. 1998. Integration analysis of product architecture to support effective team co-location. Masters thesis, Sloan School of Management, MIT, Cambridge, MA.

Fleming, L., O. Sorenson. 2004. Science as a map in technological search. *Strategic Management J.* **25** 909–928.

Garlan, M., D. Shaw. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, New York.

Godfrey, M., Q. Tu. 2000. Evolution in open source software: A case study. *Proc. 16th IEEE Internat. Conf. Software Maintenance*, IEEE, Los Alamitos, CA.

Gomes, P. J., N. R. Joglekar. 2004. The costs of coordinating distributed software development tasks. Working paper, School of Management, Boston University, Boston, MA.

Grose, D. L. 1994. Reengineering the aircraft design process. *Proc. 5th AIAA/USAF/NASA/ISSMO Sympos. Multidisciplinary Anal. Optim., Panama City Beach, FL.*

Healy, K., A. Schussman. 2003. The ecology of open-source software development. Working paper, University of Arizona, Tucson, AZ.

Henderson, R., K. B. Clark. 1990. Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms. *Admin. Sci. Quart.* **35**(1) 9–30.

Idicula, J. 1995. Planning for concurrent engineering. Research report, Gintic Institute, Singapore.

Kusiak, A., N. Larson, J. Wang. 1994. Reengineering of design and manufacturing processes. *Comput. Indust. Engrg.* **26**(3) 521–536.

Lopes, C. V., S. K. Bajracharya. 2005. An analysis of modularity in aspect oriented design. *Proc. 4th Internat. Conf. Aspect-Oriented Software Development*, Chicago, IL, 15–26.

MacCormack, A. D. 2001. Product-development practices that work: How Internet companies build software. *Sloan Management Rev.* **42**(2) 75–84.

MacCormack, A., K. Herman. 2000a. Red Hat and the Linux revolution. HBS Case 600-009, Harvard Business School, Boston, MA.

MacCormack, A., K. Herman. 2000b. Microsoft Office 2000. HBS Case 600-023, Harvard Business School, Boston, MA.

Marples, D. L. 1961. The decisions of engineering design. *IEEE Trans. Engrg. Management* **8**(2) 55–71.

Mockus, A., R. T. Fielding, J. D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Software Engrg. Methodology* **11**(3) 309–346.

Murphy, G. C., D. Notkin, W. G. Griswold, E. S. Lan. 1998. An empirical study of static call graph extractors. *ACM Trans. Software Engrg. Methodology* **7**(2) 158–191.

Offutt, A. J., M. J. Harrold, P. A. Koltee. 1993. A software metric sytem for module coupling. *J. Systems Software* **20** 295–308.

O'Reilly, T. 1999. Lessons from open source software development. *Comm. ACM* **42**(4) 33–37.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Comm. ACM* **15**(12) 1053–1058.

Paulson, J. W., G. Succi, A. Eberlein. 2004. An empirical study of open-source and closed-source software products. *IEEE Trans. Engrg.* **30**(4) 246–256.

Pimmler, T. U., S. D. Eppinger. 1994. Integration analysis of product decompositions. *Proc. ASME Sixth Internat. Conf. Design Theory Methodology*, ASME, New York.

Pinkett, R. 1998. Product development process modeling and analysis digital wireless telephones. Masters thesis. MIT, Cambridge, MA.

Raymond, E. S. 2001. *The Cathedral and the Bazaar*. O'Reilly and Associates, Sebastopol, CA.

Rivkin, J., N. Siggelkow. 2005. Patterned interactions in complex systems: Implications for exploration. *Management Sci.* Forthcoming.

Rusnak, J. 2005. The design structure analysis system. Doctoral dissertation, Harvard University, Boston, MA.

Rusovan, S., M. Lawford, D. Parnas. 2005. Open source software development: Future or fad? J. Feller, B. Fitzgerald, S. A. Hissam, K. R. Lakhani, eds. *Perspectives on Free and Open Source Software*. MIT Press, Cambridge, MA.

Sanchez, R., J. T. Mahoney. 1996. Modularity, flexibility, and knowledge management in product and organization design. *Strategic Management J.* **17** 63–76.

Sanderson, S., M. Uzumeri. 1995. Managing product families: The case of the Sony Walkman. *Res. Policy* **24**(5) 761–782.

Sangal, N., E. Jordan, V. Sinha, D. Jackson. 2005. Using dependency models to manage complex software architecture. *Proc. 20th Annual ACM Conf. Object-Oriented Programming, Systems, Languages, Appl.* ACM, New York.

Schach, S. R., B. Jin, D. R. Wright, G. Z. Heller, A. J. Offutt. 2002. Maintainability of the Linux Kernel. *IEE Proc. Software*, Vol. 149. IEE, Washington, D.C., 18–23.

Schilling, M. A. 2000. Toward a general modular systems theory and its application to interfirm product modularity. *Acad. Management Rev.* **25**(2) 312–334.

Selby, R., V. Basili. 1988. Analyzing error-prone system coupling and cohesion. Technical Report UMIACS-TR-88-46, CS-TR-2052, University of Maryland Computer Science, College Park, MD.

Sharman, D., A. Yassine. 2004. Characterizing complex product architectures. *Systems Engrg. J.* **7**(1) 35–60.

Shaw, M., D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, New York.

Simon, H. A. 1962. The architecture of complexity. *Proc. Amer. Philos. Assoc.* **106** 467–482.

Sosa, M. E., S. D. Eppinger, C. M. Rowles. 2003. Identifying modular and integrative systems and their impact on design team interactions. *ASME J. Mech. Design* **125** 240–252.

Sosa, M. E., S. D. Eppinger, C. M. Rowles. 2004. The misalignment of product architecture and organizational structure in complex product development. *Management Sci.* **50**(12) 1674–1689.

Spear, S., K. H. Bowen. 1999. Decoding the DNA of the Toyota production system. *Harvard Bus. Rev.* (September–October) 96–106.

Steward, D. V. 1981. The design structure system: A method for managing the design of complex systems. *IEEE Trans. Engrg. Management* **28** 71–74.

Sullivan, K. J., W. G. Griswold, B. J. Cai, B. Hallen. 2001. The structure and value of modularity in software design. *Proc. Joint Eur. Software Engrg. Conf./Foundations Software Engrg.*, ACM, New York.

Thebeau, R. E. 2001. Knowledge management of system interfaces and interactions for product development processes. Masters thesis, MIT, Cambridge, MA.

Ulrich, K. T. 1995. The role of product architecture in the manufacturing firm. *Res. Policy* **24** 419–440.

von Hippel, E. 1990. Task partitioning: An innovation process variable. *Res. Policy* **19** 407–418.

von Hippel, E., G. von Krogh. 2003. Open source software and the "private-collective" innovation model: Issues for organizational science. *Organ. Sci.* **14**(2) 209–223.

Warfield, J. N. 1973. Binary matricies in system modeling. *IEEE Trans. Systems, Management, Cybernetics* **3** 441–449.

Yu, L., S. R. Schach, K. Chen, G. Z. Heller, J. Offutt. 2006. Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD. *J. Systems Software*. Forthcoming.

Figure 1: The Directory Structure of Linux version 0.01

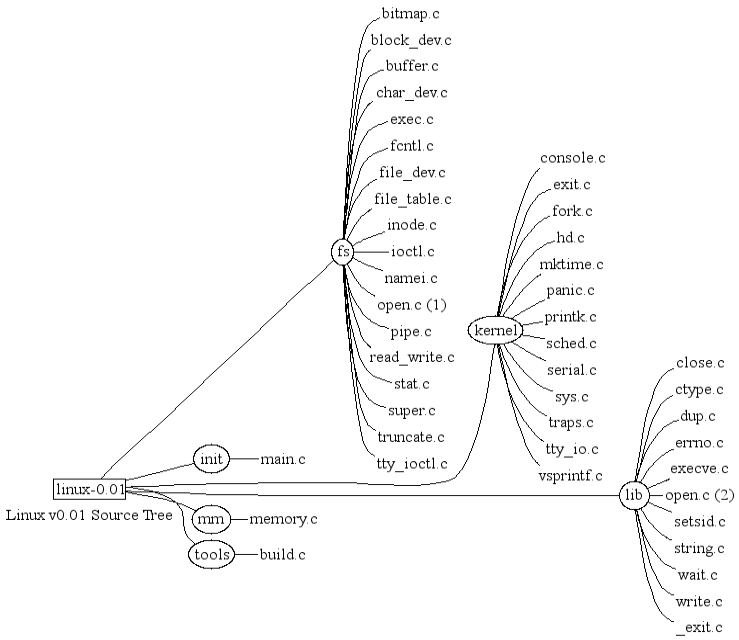
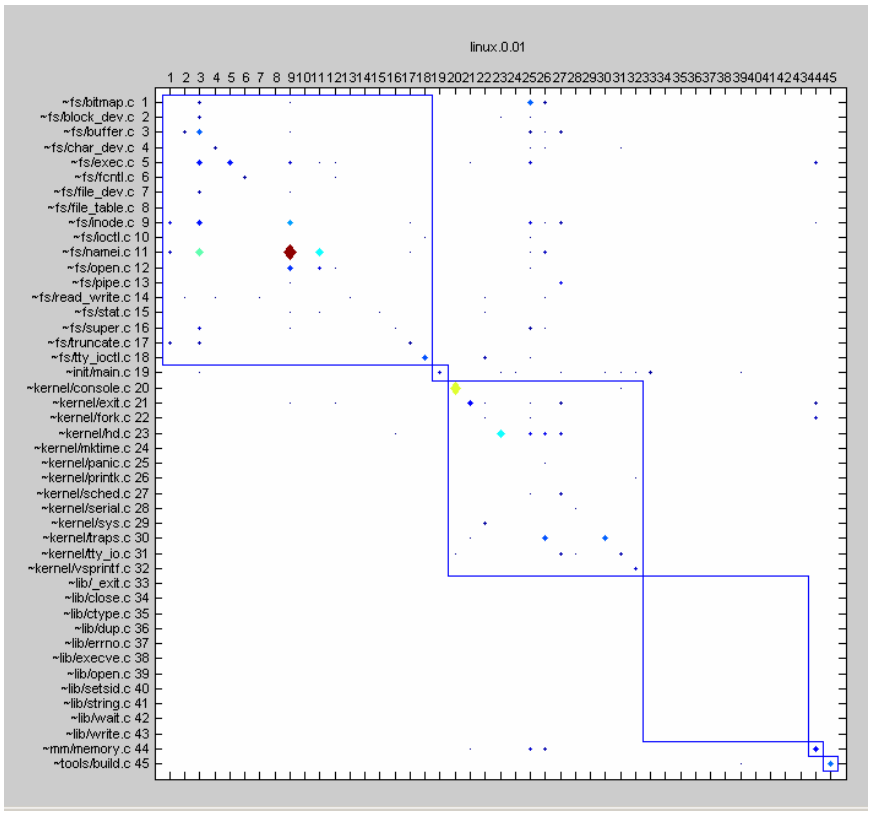


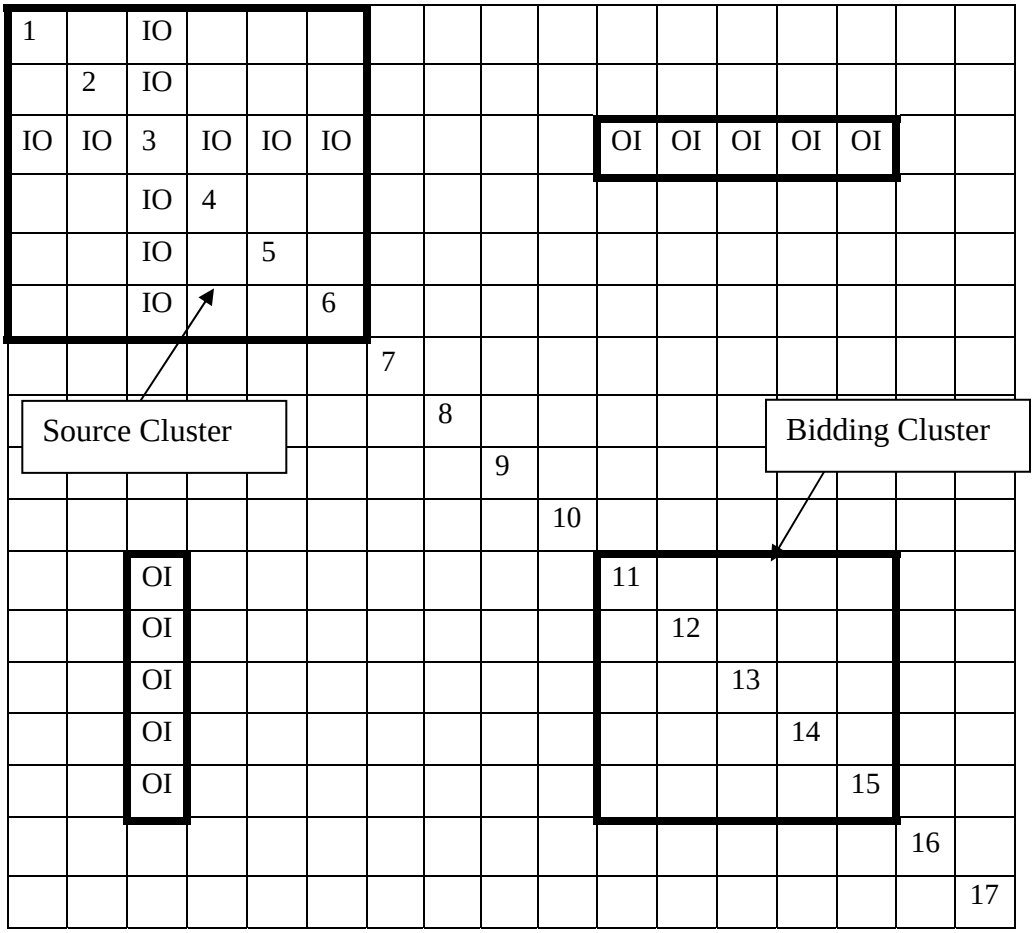
Figure 2: The Architectural View of Linux version 0.01



Online Appendix B: The Method of Marginal Cost Calculation for Clustering

During the bidding process, when an element ”proposes” to leave one cluster and join another, only a few entries in the clustered cost sum will change. Because the number of iterations and bids within iterations is high, it is computationally efficient to compute the marginal cost of a change, rather than the total cost. Exhibit 1 illustrates how this is done. The figure shows two existing clusters (“source” and “bidding”). If element 3 in the source cluster were to move to the bidding cluster, only the highlighted regions would have different dependency costs. The costs of dependencies in non-highlighted regions (should any exist) would not need to be recomputed.

Figure 1: An Example of the Costs that Change when Bidding for an Element



To compute the change in cost in the highlighted regions we first consider the elements that move in and out of a cluster. These fall into two categories: IO dependencies are "in to out" dependencies that move from being "in" the source cluster to "out" of any cluster. OI dependencies are "out to in" dependencies that move from being "out" of any cluster to "in" the bidding cluster. The change in cost for each type of dependency is as follows:

Change in cost for "IO" dependencies $\equiv d_{IO}(-m^{\lambda} + N^{\lambda})$

Change in cost for "OI" dependencies $\equiv d_{OI}(-N^{\lambda} + n^{\lambda})$

Where: N ≡ the size of the DSM

m ≡ the size of the source cluster (before the move)

n ≡ the size of the bidding cluster (after the move)

$\lambda \equiv$ The cluster size weight (set by the user)

Now, because the size of a cluster changes if an element leaves or joins, dependency cost changes for source cluster and bidding cluster must also be recomputed as follows:

Change in cost for a dependency, d_s, in source cluster = $d_s(-m^{\lambda} + (m-1)^{\lambda})$

Change in cost for a dependency, d_b, in bidding cluster = $d_b(-(n-1)^{\lambda} + n^{\lambda})$

The total change in the cost formula equals the sum of all these changes:

$$\text{Total change in cost} = \sum_{d_{IO}} d_{IO}(-m^{\lambda} + N^{\lambda}) + \sum_{d_{OI}} d_{OI}(-N^{\lambda} + n^{\lambda}) + \sum_{d_s} d_s(-m^{\lambda} + (m-1)^{\lambda}) + \sum_{d_b} d_b(-(n-1)^{\lambda} + n^{\lambda})$$

During each iteration, every cluster calculates a bid using this formula. The cluster with the largest negative bid (i.e., biggest cost reduction) "wins" the element. The total cost of the system is decreased by the change in cost of the winning cluster. If no bids lower total cost, the element stays in the same cluster and total cost remains the same.

05-016

Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code

Alan MacCormack
John Rusnak
Carliss Baldwin

Forthcoming: *Management Science* 2006

Corresponding Author:
Alan MacCormack
Morgan Hall T39
Harvard Business School
Soldiers Field
Boston, MA 02163
amaccormack@hbs.edu

We are grateful for the comments of Steve Eppinger, Jim Herbsleb, Josh Lerner, Stephen Schach, Sandra Slaughter, Sidney Winter and participants from the Wharton School Technology mini-conference; the Information Systems seminar at the Tepper School of Business, Carnegie Mellon University and the Technology and Operations Management seminar at Harvard Business School. All errors remain our own.

Abstract

This paper reports data from a study that seeks to characterize the differences in design structure between complex software products. We use Design Structure Matrices (DSMs) to map dependencies between the elements of a design and define metrics that allow us to compare the structures of different designs. We use these metrics to compare the architectures of two software products – the Linux operating system and the Mozilla web browser – that were developed via contrasting modes of organization: specifically, open source versus proprietary development. We then track the evolution of Mozilla, paying attention to a purposeful "re-design" effort undertaken with the intention of making the product more "modular." We find significant differences in structure between Linux and the first version of Mozilla, suggesting that Linux had a more modular architecture. Yet we also find that the re-design of Mozilla resulted in an architecture that was significantly more modular than that of its predecessor, and indeed, than that of Linux. Our results, while exploratory, are consistent with a view that different modes of organization are associated with designs that possess different structures. However, they also suggest that purposeful managerial actions can have a significant impact in adapting a design's structure. This latter result is important given recent moves to release proprietary software into the public domain. These moves are likely to fail unless the product possesses an "architecture for participation."

I. Introduction

Much recent research points to the critical role of design structure 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). For example, Henderson and Clark (1992) show that incumbent firms often stumble when faced with innovations that are "architectural" in nature. They argue that these dynamics occur because product designs tend to mirror the organizations that develop them. However, the empirical demonstration of such a result remains elusive. Similarly, Baldwin and Clark (2000) argue that the modularization of a system can generate tremendous value in an industry, given that this strategy creates valuable options for module improvement. However, evidence of how and why such modularizations occur in practice has not yet been shown. Finally, MacCormack's (2001) work on the management of software projects suggests the importance of a modular architecture that "facilitates process flexibility." Without a way to measure the structural attributes of a design in a robust fashion however, this work cannot reach the level of specificity it needs for managerial prescriptions to be drawn.

Common to all these research streams (and others not mentioned above) is a growing body of evidence that a product's architecture, or more broadly, the specific decisions made with regard to the partitioning of design tasks and the resulting design structure is a critical topic for both researchers and managers to understand. Unfortunately, we lack metrics with which to pursue research on this topic, hence have little *empirical* evidence that these constructs have power in predicting the phenomena they are associated with. This paper addresses this shortfall, by defining a number of metrics that measure the degree of modularity of a design, based upon an analytical technique called Design Structure Matrices. We use these metrics to compare the structures of different product designs from the software industry.

We chose to analyze software products because of a unique opportunity to examine two differing organizational modes for development. Specifically, over 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

(programming instructions) 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 highly 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 face to face. Among the most popular examples of products developed in this manner are the Linux operating system and the Apache web server.

The development methodology embodied in open source software projects stands in contrast to the "proprietary" development model employed by commercial software firms. In this model, projects tend to be staffed by dedicated teams of individuals who are located 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 (for example, if the creation of a dependency between two parts of a design could increase product performance). Consequently, the architectures of products developed using a proprietary development model are likely to differ from those of products developed using open source methods. Specifically, open source software is often claimed to be more "modular" than proprietary software (O'Reilly, 1999; Raymond, 2001). Our research seeks to explore the magnitude and direction of these presumed differences in design structure.

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 design structure, through displaying the information visually and calculating metrics to summarize the pattern of dependencies at

[1] Most commercial software is distributed in a binary form (i.e., a series of 1's and 0's) that is executed directly by a computer. It is difficult to reverse engineer this binary form to derive the original source code. Hence distributing software in binary form keeps the methods by which a program performs its tasks proprietary to the author.

the system level. Once mechanisms have been set up to do this, the ability to characterize different designs is limited only by the computing power available and the access we have to the source code of these designs (not a trivial problem, given firms regard source code as a form of proprietary technology).

The information-based nature of software products brings another benefit in that we can track the evolution of a design over time. This is possible because software developers use sophisticated "version control" systems to track design changes. For a researcher, this presents an opportunity to follow the "living history" of a design, a technique that is typically not possible for physical products. In this study, this technique is valuable, given we identify purposeful efforts to re-design a product. Having access to versions both prior to and after a re-design allows us to determine the impact of such managerial actions.

In the next section, we describe the motivation for our research and discuss prior work in this field. We then describe our research methodology, which uses Design Structure Matrices to map the dependencies in software designs and defines metrics to summarize their structures. Next, we discuss our results, which are based upon a) a comparison of two products developed via contrasting modes of organization, and b) an analysis of the impact of a major re-design effort for one of these products. We conclude by highlighting the implications of our work for both researchers and practitioners.

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 however, a number of architectures may satisfy its functional requirements. These different architectures are likely to 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 (Ulrich, 1995). They may also imply a differing partitioning of design *tasks*, thereby influencing the efficiency with which development can proceed (Von Hippel, 1990). Understanding how architectures are chosen, how they evolve and how they are adapted are therefore critical topics for managerial attention. For this to occur however, we must better understand how to measure and characterize differences between them.

Modularity refers to the manner in which a design is decomposed into different "modules." While authors vary in their definitions of modularity, they tend to agree on the concepts that lie at its heart: the notion of interdependence within modules and independence between modules (Baldwin and Clark, 2000). This latter concept, in turn, encompasses two related ideas: The need to allow work on a given module to be carried out without affecting other modules in the design, a concept known as "loose-coupling," and the need for well-designed "interfaces" between these modules. The *degree* of modularity is an important property of a product's architecture that is often used to characterize how designs differ.

The costs and benefits of modularity have been discussed in a stream of academic research that has sought to examine its impact on a range of activities including the management of complexity (Simon, 1969), product line architecture (Sanderson and Uzumeri, 1995), manufacturing (Ulrich, 1995), process design (MacCormack, 2001) and process improvement (Spear and Bowen, 1999). Despite the appeal of this work however, few studies show correlation between measures of modularity, organizational factors assumed to influence this property, and outcomes it might impact (e.g., see Fleming and Sorenson, 2004).

The most promising technique for understanding and measuring modularity has come 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 component elements in a symmetric matrix (Steward, 1981; Eppinger et al, 1994). DSMs can be constructed using elements that represent "tasks" to be performed or "parameters" to be defined. In such situations, their use is to identify a partitioning of tasks (or parameters) that facilitates the flow of information in a project. Tasks (or parameters) that have a high level of dependency are grouped into clusters or modules. DSMs have been studied in a wide variety of industries including aerospace (Grose, 1994), automotive (Black et al, 1990), building design (Austin et al, 1994) manufacturing (Kusiak, 1994) and telecommunications (Pinkett, 1998). Much of the work has been of a theoretical nature, exploring the ways that DSMs can help to better organize projects (e.g, Eppinger et al, 1994) or value the options available to a designer (e.g., Baldwin and Clark, 2000). A key contribution of the DSM literature has been in highlighting that the degree of modularity of a design depends not only on the *number* of dependencies between elements, but

also their *pattern* of distribution. Several studies therefore explore different types of architectures and their properties (Sosa et al, 2003; Sharman and Yasine, 2004). Recent work uses DSMs to provide insights into the relationship between product architecture and organizational structure. For example, Sosa et al (2004) find a "strong tendency for design interactions and team interactions to be aligned," and show instances of misalignment are more likely to occur across organizational and system boundaries.

Studies of Software Design Structure

In the field of software, the study of design structure has a long tradition. Parnas (1972) first proposed the concept of *information hiding* as a mechanism for dividing code into modular units. This concept allowed designers to separate the details of internal module design from external module interfaces, reducing the coordination costs required to develop complex software systems, while increasing the ability to make changes to the design without affecting other parts of a system. Subsequent authors developed these concepts, proposing a variety of metrics, such as coupling and cohesion, to provide indicators of design structure (e.g., Selby and Basili, 1988; Offut et al, 1993; Dhama, 1995). However, many of these studies were theoretical in nature, using stylized examples rather than actual code, and providing little data to suggest the metrics were predictive or prescriptively useful. Recent empirical work has made progress on this front. For example, Eick et al (1999) find code tends to "decay" over time, as measured by the number of files that must be changed to fulfill a modification request. Banker and Slaughter (2000) show greater "structure" can mediate the effect of volatility and complexity on software enhancement costs. This study accounts for the *number* of design dependencies but not their *pattern* of distribution, hence does not fully capture the notion of modularity we describe.

With the recent rise in popularity of open source software, interest in the topic of architecture and modularity has received further stimulus. Prior work suggests a duality exists between a product's design and the organization that developed it (Henderson and Clark, 1992) hence this new mode of organization might be expected to give rise to software that exhibits different architectural properties than that of proprietary products. Some authors suggest 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 of development to succeed (Torvalds, as quoted in DiBona et al, 1999). Such issues are important, given theory predicts different types of architecture may incur trade-offs in terms of performance (Ulrich, 1995; Von Hippel, 1990). Furthermore, attempts to move software from one domain to the other may be destined to fail unless we understand whether a product's architecture must first be changed, and the impact that such changes can have on a system of reasonable complexity.

Studies seeking insights into these issues yield few general conclusions, in part due to the differing approaches they take. Paulson et al (2004) find a correlation between functions added and functions modified in open source projects, but not in proprietary projects.[2] They measure the impact of modifications rather than inherent system structure so cannot draw broader conclusions about the nature of open source architectures. Schach et al (2003) and Yu et al (2005) compare Linux with other open source products, showing that its design has a larger number of "unsafe" dependencies between kernel source files and non-kernel source files through the use of global variables (variables defined and used in many parts of the design). Yet we do not know how proprietary products would compare in such an analysis. Rusovan et al (2005) also examine Linux, looking at one important design element (the Address Resolution Protocol [ARP] source file). They find it has many dependencies on other elements through direct and indirect "function calls" (requests for specific tasks to be performed). Yet this analysis may not be representative of the system as a whole or differ from proprietary systems. Finally, Mockus et al (2002) report in-depth case data from the Apache and Mozilla projects. They note Mozilla modules "are not as independent from one another..." relative to Apache and speculate that this may be due to Mozilla's commercial legacy, but do not provide metrics to support these qualitative observations.

In sum, despite a long tradition of studying modularity dating back to Parnas' (1972) paper, studies of software design structure often exhibit common pitfalls: They examine only a small part of a design hence cannot make claims about the system as a whole; they rely on qualitative or manual analyses which are

[2] This correlation does not necessarily imply open source projects modify more files per function added than proprietary projects. It only implies that there is a relationship between these two variables.

subjective and may not scale to larger systems; they focus on different types of dependencies, limiting the cumulative impact of the work; and they measure the "effects" of system structure (e.g., the number of file changes per modification) rather than the "causes" of these effects. As Shaw and Garlan (1996) state in their seminal work on the topic, software architecture remains "…an emerging discipline."

This study contributes to the literature on modularity in general, and the study of software design structure in particular. Specifically, we use Design Structure Matrices to analyze different software designs. We argue that this technique provides a rigorous and valuable way to characterize software architecture, addressing many of the pitfalls noted in the literature. Indeed, recent work has explored the use of DSMs to evaluate software design choices (Sullivan et al, 2001; Lopes and Bajracharya, 2005). For example, Sullivan et al (2001) use DSMs to formally model (and value) the concept of information hiding, the principle proposed by Parnas to divide designs into modules. Furthermore, Gomes and Joglekar (2004) use metrics derived from a DSM to predict the likelihood of outsourcing software tasks. The domain to which we apply this technique is the analysis of open source software. Specifically, we first explore the differences in design structure between an open source product and one developed via a proprietary development model. Prior work suggests such different organizational models are likely to yield different architectures (Henderson and Clark, 1990) but provides little empirical evidence to support this idea. We then evaluate the impact of a purposeful effort to re-design a product's architecture. Previous studies highlight the difficulty of identifying the *need* for architectural change (Henderson and Clark, 1990) but provide little empirical evidence on organizations' abilities to *effect* such change. Given recent interest in moving proprietary software into the public domain, this is a critical question.

3. Research Methodology

In contrast to studies that use DSMs to understand how a set of future tasks or parameters should be organized, we use this technique to analyze *existing* designs. We calculate metrics from a DSM to characterize product structure. Our objective is to measure the degree of modularity of a design. We argue that this concept must be measured comparatively. That is, the absolute level of modularity for a

product has no meaning; we can only determine that product A is more (or less) modular than product B. Our work is based upon comparing different designs, using both cross-sectional (i.e., product A versus product B) and longitudinal (i.e., product A at time T versus product A at time T+N) comparisons.

While we cannot measure modularity directly, we argue that the manifestation of different degrees of modularity can be measured from a DSM by examining the costs of dependencies between its elements. These costs depend upon both the number and pattern of distribution of dependencies. We define two metrics to capture these costs, each of which makes different assumptions about their nature. The first metric, *propagation cost*, assumes that all dependencies between elements, both direct and indirect, incur the same cost, regardless of where the elements are located or how long the path length is between them. The second metric, *clustered cost*, assumes that the cost of dependencies between elements will differ depending upon whether elements are in the same "cluster" (or module) or are in different clusters. We define each metric below, after discussing the unit of analysis and the type of dependency we focus on.

The Unit of Analysis: The Source File

The first decision that must be made in applying the use of DSMs to a software design is the level of analysis at which the DSM is built. Choices range from high level representations of major subsystems to the individual components that make up these subsystems. The level of analysis should be driven by the research question, but must also be meaningful given the context[3]. When considering the design of a software product, there are three levels at which a DSM could be built, nested in a hierarchical fashion as follows: a) the subsystem level, which corresponds to a group of source files that all relate to a specific part of the design; b) the source file level, which corresponds to a collection of programming instructions (source code) that perform a related group of functions; and c) the function level, which corresponds to a set of programming instructions (source code) that perform a highly specific task.

[3] For example, it is more insightful to examine the DSM for an automobile at the subsystem level (e.g., brakes, suspension and powertrain) than at the raw component level (e.g., nuts and bolts).

We choose to analyze designs at the source file level for several reasons. First, source files tend to group functions of a common nature into a single place hence are a logical unit of analysis. For example, a programmer may expect to have functions related to the printer in a source file named "printer_driver.c." Source files also contain header information and overview comments that apply to all functions in the file. Second, tasks and responsibilities are typically allocated to programmers at the source file level. Hence this unit of analysis reflects how managers of software projects exert influence on product structure. Third, source control and configuration management tools use the source file as their unit of analysis. Programmers "check out" source files for editing, modify the code, then "check in" the new version once work is complete. Finally, other empirical studies that focus on software structure typically use the source file as the unit of analysis (e.g., Schach et al, 2003; Rusovan et al, 2005).[4]

The Type of Dependency: A "Function Call"

We capture dependencies between source files by examining the "Function Calls" that each source file makes. A Function Call is an instruction that requests a specific task to be executed by the program. The function called may or may not be located within the source file that originated 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 which is located in Sourcefile2, then Sourcefile1 depends upon (or "needs to know" about) Sourcefile2. This dependency is marked in the DSM in location (1, 2).[5] Note that this dependency does not imply that Sourcefile2 depends upon Sourcefile1; that is, the dependency is not symmetric.[6] This would be true only if a function in Sourcefile2 called a function in Sourcefile1.

We note that function calls are only one important type of dependency that exists in a software design. Several authors have developed comprehensive categorizations of dependency types (e.g., Garlan

---

[4] The source file level is granular enough to be meaningful without being overwhelming. For example, Linux version 2.5 has 10 subsystems, over 4,000 source files and over 60,000 functions.
[5] We use the convention (row number, column number) to describe entries in a DSM.
[6] We assert that the designer of a calling function needs to "know about" the function being called. While the reverse may be true (a function may need to know about the functions that call it) such reverse dependence would limit the usefulness of the function; it could be used only for purposes that were known at the time of its definition.

and Shaw, 1996; Dellarocas, 1996). Others focus on a single type, such as global variables, to provide insight into system structure (Schach et al, 2001). We argue that DSMs provide a general technique by which different notions of dependency can be analyzed, each of which provides a different "lens" through which to view the system (Sangal et al, 2005). We choose to focus on function calls given this type of dependency is at the heart of many analysis tools (e.g., call graphs, see Murphy et al, 1998) and is used in prior work that examines system structure (Banker and Slaughter, 2000; Rusovan et al, 2005).

To capture Function Calls between source files, we input the source code for a product into a tool called a "Call Graph Extractor" (Murphy et al, 1998). This tool is used to obtain a better understanding of code structure and interactions between different parts of the code. We note that function calls can be extracted statically or dynamically. Static calls are extracted from code not in an execution state and use source code for input. Dynamic calls are extracted from code in an execution state and use executable code and the program state as input. 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 structure of the design from the programmer's perspective.[7] Rather than develop our own call extractor, we tested several products that had the ability to process source code written in both procedural and object oriented languages (e.g., C and C++), captured indirect calls (dependencies that flow through intermediate files), could be run in an automated fashion and output data in a format that could be input to a DSM (i.e., in matrix form). A product called Understand C++[8] was selected given it best met all these criteria.

Analyzing Design Structure using a DSM

Once a DSM is populated with function call dependencies, we can examine it visually using what we call the *Architectural View* of the system. In software designs, programmers tend to group source files of a related nature into "directories" that are organized in a nested fashion. The Architectural View of a

[7] "Compiler Extractors" are used mainly to help compile the software into an executable form. "Design Extractors" are used to aid programmers in determining source relationships. We use the latter type of extractor.
[8] Understand C++ is distributed by Scientific Toolworks, Inc. see <www.scitools.com> for details.

DSM groups each source file into a series of clusters as defined by the directory structure, with boxes drawn around each layer in the hierarchy. The result is a map of source file dependencies, organized by the programmer's view of the system.[9] As we shall see later, these maps are a powerful aid to assessing design structure. But to compare different designs we must define metrics that characterize these maps. Below, we define two metrics to achieve this objective.

Propagation Cost

The first method by which we characterize the structure of a design is by measuring the degree of "coupling" it exhibits, 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 builds upon the concept of visibility (Sharmine and Yassine, 2003) which in turn, is based upon the concept of reachability matrices (Warfield, 1973). To clarify how we measure this concept, we use a simple example. Consider the element relationships shown in Figure 1. We note that element A depends on elements B and C. So any change to element B has the potential to impact element A. Similarly, elements B and C depend on elements D and E respectively. So any change to element D may have a direct impact on element B and an indirect impact on element A with a "path length" of 2. Finally, we note that any change to element F may have a direct impact on element E, an indirect impact on element C with a path length of 2, and an indirect impact on element A with a path length of 3. In this system, there are no indirect dependencies for path lengths of 4 or more.

We use the technique of matrix multiplication to identify the "visibility" of any given element for any given path length. 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. By summing these matrices together we derive the visibility matrix V, showing the dependencies that exist for all possible path lengths up to n. We choose to include the matrix for n=0 (i.e., a path length of zero) when

[9] The directory structure and Architectural View of Linux version 0.01 are displayed in Online Appendix A.

calculating the visibility matrix, implying that a change to an element will always affect itself. Figure 2 illustrates the derivation of the visibility matrix for the example we describe above.

From the visibility matrix, we can construct several metrics to give insight into a system's structure. The first, called "Fan-Out Visibility," is obtained by summing along the rows of the visibility matrix, and dividing by the total number of elements. An element with high Fan-Out visibility depends upon (or calls functions within) many other elements. The second, called "Fan-In Visibility," is obtained by summing down the columns of the visibility matrix, and dividing by the total number of elements. An element with high Fan-In visibility has many other elements that depend upon it (or call functions within it). In our example, element A has a Fan-Out visibility of $6/6^{th}$ (or 100%) meaning that it depends upon all elements in the system, and has a Fan-In visibility of $1/6^{th}$ meaning that it is visible only to itself.

To summarize visibility at the system level, we compute the average Fan-Out and Fan-In visibility of all elements in the system. Due to the symmetric nature of dependency relationships (i.e., for every fan-out, there is a corresponding fan-in) these are identical. We call the resulting metric "Propagation Cost." Intuitively, this measures the proportion of elements that could be affected, on average, when a change is made to one element in the system. In the example above, we can calculate propagation cost from Fan-Out Visibility ([6+2+3+1+2+1]/6*6 = 42%) or Fan-In visibility ([1+2+2+3+3+4]/6*6 = 42%).

We note that in the example we consider, the design does not contain multiple paths between elements and the dependency relationships are purely hierarchical.[10] This will not generally be the case for complex systems. In particular, multiple paths bring the possibility that there is more than one route through which element A depends upon element B. As a result, we limit values in the visibility matrix to be binary, capturing only the fact there exists a dependency, and not the number of possible paths that this dependency can take. Similarly, a non-hierarchical system will contain one or more "cycles" meaning successive powers of the dependency matrix will not converge to zero.[11] To avoid this problem, we raise

[10] A hierarchical system can be identified by the fact that its dependency matrix can be re-arranged such that all non-zero entries are below the diagonal, which is called a "lower-triangular" form.
[11] For example, a relationship whereby element A depends upon element B which depends upon element A.

the dependency matrix to a maximum power dictated by the number of elements in the system (i.e., the longest possible path length). Given that we limit entries in the visibility matrix to binary values, the fact that cyclic relationships appear multiple times ultimately has no impact on the output.

Clustered Cost

The assumption behind the measure of propagation cost is that each dependency between elements incurs the *same* cost, wherever these elements are located in the design and however long the path length between them. By contrast, our second measure attributes a *different* cost to each dependency according to where the elements are located. In essence, we posit that there is an "idealized modular form" in which dependencies are grouped into clusters of tightly-connected elements (or "modules") with no dependencies between these clusters. Dependencies between elements in the same cluster are assumed to incur a low cost; those between elements in different clusters are assumed to incur a high cost. This type of structure is related to the notion of decomposability (Simon, 1962) and is a characteristic of modular systems (Baldin and Clark, 2000). Figure 3 shows a DSM with dependencies grouped in this way.

To assess the degree to which a design approximates this idealized modular form, we cluster the DSM to determine the most appropriate grouping of elements. While a range of algorithms can be used for clustering, most define a cost function to measure the "goodness" of a solution, and invoke a search technique to find the lowest cost grouping of elements. The approach we adopt is based upon prior DSM research (Pimmler and Eppinger, 1994; Idicula, 1995; Fernandez, 1998; Thebeau, 2001) adapted for the context of software development.[12] In particular, we first identify and account for what we call "Buses" – source files that are called by a large number of other source files. We then allocate a cost to each dependency, depending upon whether the dependency is between elements that are in the same cluster or not. Finally, we adopt an iterative clustering process in which source files chosen at random receive bids to join clusters based upon their level of interdependence. We describe this process below.

[12] Prior work clusters DSMs to find a grouping of tasks involved in developing a *new* design. We use clustering however, to identify an appropriate grouping of related elements in an *existing* design.

Prior work has highlighted the role of design elements called "Buses" (e.g., Ulrich, 1995). These elements have particular importance in software development, in that designs often make use of common functions that are called by many other elements[13]. We call these "Vertical Buses," given they appear in a DSM as an element with many vertical dependencies. The use of vertical buses avoids duplication of effort and ensures a consistent approach to system-wide functions. Buses are typically identified early in the design process and have well-defined, stable interfaces (Baldwin and Clark, 2000). To identify vertical buses, we assess the degree to which a particular source file is called by other elements in the system (in terms of the percentage of source files that call it). We then examine whether the result is above or below a parameter called the "Bus Threshold." If it is, we treat dependencies on this source file differently in assessing their cost (described below). Note that there is no "right" answer as to the connectivity that should constitute the threshold for a Bus. In our work, we initially set this parameter to 10%, then vary it from 10-100% to evaluate the potential impact this has on results.

Once vertical buses have been identified, a cost is allocated to each dependency depending upon the location of the elements between which the dependency exists. Specifically, when considering a dependency between elements i and j, the cost of the dependency takes one of the following three forms:

$$DependencyCost(i \rightarrow j \mid \text{j is a vertical bus}) = d_{ij}$$
$$DependencyCost(i \rightarrow j \mid \text{in same cluster}) = d_{ij} * n^{\lambda}$$
$$DependencyCost(i \rightarrow j \mid \text{not in same cluster}) = d_{ij} * N^{\lambda}$$

Where d_{ij} is a binary variable indicating the presence of a dependency between i and j[14]; n is the size of the cluster that i and j are located within; N is the DSM size; and λ is a user-defined parameter.[15] Note that prior to clustering, all elements are assumed to be in a "singleton" cluster consisting of only itself.

---

[13] Examples include library code, global functions such as "print to screen" or error handling code.
[14] This parameter can also be set to the *strength* of dependency based on the number of calls between elements.
[15] There is no "right" value for λ. Analysis proceeds by setting this parameter to some value, comparing the metrics for the design structures of interest, and testing the sensitivity of results to parameter changes. We nominally set λ=2, given the number of potential interactions increases by a power-law with the number of elements in a cluster.

The clustering algorithm attempts to determine the optimal allocation of non-bus elements to clusters such that the sum of dependency costs is minimized. The challenge is that for a design of moderate or greater complexity the solution space becomes too large to search exhaustively. Hence an iterative search process is adopted, whereby randomly selected elements are the subject of "bids" from existing clusters that have a dependency on this element. Each bidding cluster bids an amount equal to the marginal reduction in cost that would result from the element joining it.[16] The element is "awarded" to the cluster with the highest bid. If there are no positive bids, the element remains in its current location.[17] The bidding process continues in an iterative manner, stopping when the number of iterations with no improvement in cost exceeds a threshold defined by the user. In practice, we find that setting this threshold equal to the size of the DSM ensures the algorithm converges to a low cost solution.[18] We should note that while the clustering process will tend to group elements with high levels of interdependency together, there are limits to the size of clusters. In particular, adding an element increases the cost of other dependencies in the cluster, through its impact on cluster size (i.e., an increase in the parameter n). Hence the algorithm will allocate an element to a cluster only when the reduced costs of dependencies with the element *exceed* the added costs borne by other dependencies.

4. Empirical Setting

Our study had two aims. First, to explore the differences in design structure between two software products that had been developed using different organizational modes, namely open source and proprietary development. And second, to examine a concentrated effort to re-design a product's architecture with the goal of making it more modular. These objectives led directly to the choice of the two software products that we examine: Linux (or more strictly, the Linux Kernel) and Mozilla.

---

[16] The method by which bidding occurs is described in Online Appendix B.
[17] Our clustering algorithm is hierarchical in that it does not allow elements to be in more than one cluster.
[18] Given the nature of the search process, the outcome will differ by a small margin each time it is repeated.

Linux is a UNIX-like operating system that is the best known of all open source software products. Since its release onto the Internet in 1991, hundreds of developers have contributed to its development (DiBona et al, 1999). Linux has gained significant market share in the server operating system market, and continues to develop at a rapid pace, in terms of the amount of new code and functionality added each year (MacCormack and Herman, 2000a). It represents one of the most successful examples of open source software, hence is well represented in studies that seek to evaluate the performance and nature of open source development (e.g., Godfrey and Tu, 2001; Scach et al, 2001; Rusovan et al 2005). For our purposes, Linux represents a "prototypical" example of the open source development model.

Mozilla is also an open source product, however its origins lie in a product developed using a proprietary development model. Specifically, in March 1998, Netscape released the code for its Navigator web browser onto the Internet, renaming the software Mozilla. The release of the source code was part of a strategy to respond to increasing competition in the browser market. Netscape hoped that volunteer developers would contribute to the design in the same fashion as was happening with other open source products. We use the first release of Mozilla as a proxy for a design developed using a proprietary methodology (i.e., a dedicated team of individuals located in a single location).[19] This product comprised over 1500 source files. By contrast, the Linux kernel comprised less than 50 source files when first released in 1991. Comparing the first release of Mozilla with a similarly sized version of Linux allows us to compare a product developed with a proprietary model to one in which 95% of the design (in terms of source files) has evolved through the use of an open source development model.

We chose to use Mozilla as a proxy for a proprietary product to take advantage of a "natural experiment" that occurred during the fall of 1998. Specifically, at this time, a re-design effort was carried out with the aim of making Mozilla "more modular."[20] The reason for this re-design, as indicated in accounts of Mozilla's history, was the effort involved in understanding and contributing to the code base

[19] The idea for Netscape's browser came from a product called Mosaic, developed at the University of Illinois. Netscape's code, however, was developed independently, was far more complex, and targeted commercial users.
[20] Source: Interviews with Mozilla team members during the spring of 1999.

(Cusumano and Yoffie, 1998). The code was thought too tightly-coupled for would-be contributors to modify without having to examine the impact on many other parts of the system. The re-design aimed to make the design more modular, and hence more attractive to potential contributors. The effort was carried out over several months by a small team of developers, most of who worked for Netscape (the firm sold a commercial browser that used Mozilla as its base). Given Mozilla is in the public domain, we can examine versions of the product both prior to and after the re-design effort to evaluate its impact.

5. Empirical Results

We divide our results into two parts. First, we compare the first version of Mozilla – a proxy for a proprietary developed product – with a version of Linux of comparable size (in source files). We then examine the longitudinal evolution of Mozilla, paying particular attention to the re-design effort that aimed to make the product more modular. For each design, we report data on the number of source files, the number of dependencies, the density of the DSM (i.e., the number of dependencies per source file pair) the propagation cost and the clustered cost. We also provide data on the average complexity of source files, in terms of the number of functions and lines of code.

A Comparison of the Design Structures of Linux and Mozilla

We first compare the Architectural Views of Linux and Mozilla (see Figure 4). From these plots we can make some overall comments about their structure. At the highest level, both are architected into a small number of major subsystems, within which are smaller groups of source files. Linux however, possesses more vertical buses – source files that are called by many other source files in the system (see annotation A). Indeed, our analysis identifies 14 vertical buses in Linux, versus two in Mozilla. In the upper left hand corner of Mozilla's DSM we see a few large groups of source files that are very tightly connected (see annotation B). By contrast, Linux comprises a larger number of small groups that are less tightly connected (i.e., fewer dependencies between them). Finally, in Mozilla, we note one group of source files comprising a major subsystem that has many dependencies above it, indicating it is tightly

connected to other parts of the system (see annotation C). By contrast, apart from the vertical buses discussed above, the subsystems in Linux appear to be less tightly connected to the rest of the system.

Table 1 shows quantitative data comparing Mozilla and Linux. We first note that Mozilla has fewer dependencies between source files than Linux. Mozilla has a density of 2.4 dependencies per 1000 source file pairs.[21] By contrast, Linux has a density of 3.4 dependencies per 1000 source file pairs, over 40% greater than that of Mozilla. At first sight, one might imagine that this would imply Mozilla is more modular than Linux. But these data only give insight into the *number* of dependencies, and not the *pattern* of distribution of these dependencies. Our measures of propagation cost and clustered cost, by contrast, take both these aspects of design structure into account. The propagation cost for Mozilla is 17.35% versus 5.16% for Linux, a striking difference. This implies that the design of Linux is much more loosely-coupled than the first version of Mozilla. A change to a source file in Mozilla has the potential to impact three times as many source files, on average, as a similar change in Linux (of course, the specific results would depend upon the particular source file being modified). This finding is supported by the evidence that comes from clustering each DSM. The clustered cost for Linux is around 70% of that for Mozilla, despite having a larger number of dependencies. Finally, we also note differences in the average complexity of source files. Mozilla contains 50% more functions per file, although the difference in terms of lines of code does not appear to be as significant.

To summarize, we have compared the design structures of two systems of similar size (in source files), one having been developed with an open source development model, the other having been developed using a proprietary model. The architectural view reveals visual differences in design structure consistent with an interpretation that Linux uses more vertical buses and is more loosely-coupled than Mozilla. The metrics that measure design structure are consistent with this view. We conclude that the first version of Mozilla was significantly less modular than a comparable version of Linux.

---

[21] Our figures illustrate that most software DSM's constructed at the source file level are relatively sparse (i.e., they have very few dependencies per possible source-file pair).

The question that arises given these results concerns the degree to which the differences observed are a product of the organizational modes used in development, or reflects purposeful strategies employed by their architects. For Linux, there is evidence that the original author, Linus Torvalds, emphasized modularity as a design criterion (DiBona et al, 1999). The design was also built on the heritage of the UNIX and MINIX operating systems, which are relatively modular at the system level. By contrast, we know little about the intentions of the original architect(s) of Mozilla. We can speculate however, that as an early entrant to a new product category (the web browser) there was likely a greater focus on superior product performance (e.g., speed of operation) than on achieving high levels of product modularity. Given the potential trade-off that exists between product performance and modularity (Ulrich, 1995) this may have led Mozilla's original architect(s) to adopt a more tightly-coupled or "integral" design.

There is however, another explanation for our results, relating to the different functions these products perform. Put simply, an operating system may require a different design structure to a web browser as a result of the different tasks and performance demands it faces. Unless we have a true apples-to-apples comparison, it is difficult to draw general conclusions. One solution would be to obtain the source code of an operating system developed in a proprietary fashion and compare its design to that of Linux. Another option however, is to identify a purposeful effort to change the architecture of Mozilla to a more modular form, and gauge the impact of this effort on its design. Fortunately, such a "natural experiment" did in fact occur during Mozilla's ongoing development. We now examine its impact.

A Comparison of the Design Structures of Mozilla before and after a Re-design

Netscape's Navigator browser was released under an open source license in March 1998. At that point, the source code became known as Mozilla, and developers from around the world began to contribute to the project. In the fall of 1998, a group of Mozilla's core developers, most of whom worked for Netscape, re-designed the code base to make it more modular, and hence more attractive to contributors. The results above point to the nature of the problem Mozilla faced; anyone making a change to a source file would have to check three times as many possible interactions, on average, as a

contributor to Linux. The re-design effort that followed encompassed several objectives: to remove redundant or obsolete source files; to re-write major parts of the code to run more efficiently; and to re-architect the product, grouping source files into smaller modules with fewer connections between them.

Prior to the re-design, Mozilla comprised 2333 source files. Afterwards, the software comprised only 1508 source files. There are two ways to examine the impact of this re-design, given the number of source files was reduced so dramatically. The first is to compare the design that emerged with an earlier Mozilla version of similar size. The second is to compare the design immediately *prior* to the re-design with a later Mozilla version of similar size. Figure 5 shows the comparison of the Architectural Views for the former. The results are striking. The re-designed version of Mozilla comprises much smaller clusters of source files. There are very few dependencies *between* these clusters, or indeed between any of the individual source files. Much of the DSM is "white space."

Table 2 shows quantitative data comparing Mozilla before and after the re-design. We first note that the re-designed product has only 1.3 dependencies per thousand source file pairs, compared to 2.4 in the earlier version. Hence the re-design significantly reduced the *number* of dependencies between different source files in the design. Furthermore, there are significant changes when we look at the metrics that account for both the number and *pattern* of dependencies. Specifically, the propagation cost of the design has dropped from 17.35% to 2.78%. That is, changes to a source file have the potential to impact 80% fewer source files, on average, after the re-design. This reduction is mirrored by the reduction in clustered cost, which drops to around 30% of its previous level. These results are obtained with a small reduction in the complexity of source files, as measured by the number of functions and lines of code. We conclude that the efforts to make the design more modular were extremely successful.[22]

It is insightful to look at the impact of this re-design effort in the context of the evolution of Mozilla's design structure over time. To this effect, we plot the evolution of Mozilla's propagation cost and clustered cost in Figures 6 and 7 respectively. The results are striking. The re-design effort reduced

[22] We see similar results when we compare versions of Mozilla *immediately* prior to and after the re-design.

propagation cost from a level varying between 15-18% to a level varying between 2-6%. Similarly, the re-design effort reduced clustered cost by an order of magnitude (although much of this fall comes from the reduction in the number of source files, hence the need for the paired comparisons we conduct above). To our knowledge, these plots represent the first quantitative view of the evolution of a product's architecture over time. They show that the impact of the Mozilla re-design effort was unambiguous and significant in magnitude. Specifically, the architecture of the product was much more modular after the re-design effort. Furthermore, we have found no public data that suggests the re-designed product was substantially inferior in terms of functionality or performance. Indeed, discussions with Mozilla developers suggest the re-design *improved* performance on some dimensions.[23] We are left with an intriguing conclusion. There was no requirement for Mozilla to possess as tightly-coupled a design structure as it did prior to the re-design effort. Hence our previous results are not explained by the differing functions that a browser and an operating system must perform. The evidence therefore suggests that the design structure that evolved was due to either the mode of organization involved in its development, and/or to specific choices made by the team's architects (discussed below).[24]

In Table 3, we compare the metrics for the re-designed version of Mozilla with a comparable version of Linux. We find that the re-design succeeded in making Mozilla more modular than Linux. Specifically, a change to a source file in the re-designed Mozilla, on average, has the potential to impact fewer than half the source files that a change in Linux. Similarly, the clustered cost of the design is now only around 50% of that of Linux. Interestingly however, we note that the density of dependencies in Mozilla has dropped to around 35% of that of Linux. Hence on a per dependency basis, Mozilla does not fair quite as well as perhaps it should. Such a comparison, while exploratory, implies that much of

[23] Source: Interviews with Mozilla team members during the spring of 1999.

[24] We should note that plots such as these have a very practical use, by making visible the evolution of a product's architecture. Specifically, problematic trends can be identified quickly and corrective action planned with defined objectives for the impact on a system's architecture. While experienced architects routinely perform this role using an intuition of the structure of a code base, we previously lacked measures to explain their intuitions.

Mozilla's post re-design advantage in modularity relative to Linux comes from the reduction in the *number* of dependencies as opposed to a superior configuration in terms of their *pattern* of distribution.[25]

6. Discussion

The distinctive contribution of our work is in applying the technique of Design Structure Matrices to the analysis of software architecture. In particular, we process source code for existing software designs, extract and map dependencies between their component elements, and calculate metrics that give us insight into system structure. We use these metrics both to compare designs developed via different modes of organization – open source versus proprietary development – and to track the evolution of one design over time, giving insight into the impact of a re-design effort. Our study examined only two products, hence must be regarded as exploratory. However the novel research design we employ, in using both cross-sectional and longitudinal comparisons, yields insight on several important dimensions.

First, our results demonstrate that there are substantial differences in modularity between different software systems of comparable size and function. While this may not be surprising, the fact that we can *measure* the size of such differences is a critical advance in our understanding of architecture. We find that the systems we examine vary by a factor of five in terms of the potential for a design change to propagate to other source files in a system, a measure we refer to as propagation cost. Such a result has significant implications for those responsible for software system development and maintenance. It suggests that systems are likely to vary dramatically in terms of their robustness to change, and the costs and efficiency of future enhancements. Indeed, given we examine two versions of a product that perform the same function yet with very different architectures, our results highlight the value of design as a *managerial choice*. That is, architecture is not wholly determined by function, but results from purposeful choices by designers reacting to the incentives and structures that surround them.

---

[25] Mozilla's re-design clearly improved the pattern of distribution of dependencies *relative to the prior version of the design*, given its propagation cost declined so significantly (see Table 2).

In this respect, our study generates useful data on the question of whether a link exists between a product's architecture and the structure of the organization from which it comes. We show that the architecture of a product developed by a highly distributed team of developers (Linux) was more modular than another product of similar size developed by a co-located team of developers (Mozilla). Critically however, we find that a purposeful effort to re-design Mozilla resulted in an architecture with greater modularity. Hence the initial differences between Linux and Mozilla were not driven by the different functional requirements of these products. These results are consistent with the idea that a product's design mirrors the organization that develops it. We must ask however, how such a dynamic occurs?

Our observations are consistent with two rival hypotheses. The first is that each design *evolved* to reflect its development environment. In Mozilla's case, a focused team employed by one firm and located at a single site developed the design. Problems were solved by face-to-face interaction, and performance "tweaked" by taking advantage of the access that module developers had to the information and solutions developed in other modules. Even if not an explicit managerial choice, the design naturally became more tightly-coupled. In Linux' case however, hundreds of distributed developers situated around the world developed its design in a loosely-coupled fashion. Face-to-face communication was almost non-existent as most developers never met. The design structure that evolved therefore relied on fewer connections between modules, and hence was more modular. Alternatively, our observations may reflect *purposeful choices* made by the original designers in response to specific contextual challenges. For Netscape, the aim was to develop a product that maximized product performance, given a dedicated team of developers and a competitive environment that demanded a product be shipped as quickly as possible. The benefits of modularity, given the context at the time, would not necessarily have been seen as significant. By contrast, for Linus Torvalds, the benefits of modularity were substantial. Without modularity, there was little hope that contributors could understand enough of the design to contribute in a meaningful way, or develop new features and fix existing defects without affecting many other parts of the design. Linux therefore *needed* to be modular to attract and facilitate a developer community.

We note that not all open source projects are characterized by broad participation by many developers or highly distributed team structures. Indeed, open source projects are often small efforts that involve only a handful of developers (Healy and Schussman, 2003). In addition, proprietary projects are not always co-located, but may involve developers from multiple sites and organizations. In this respect, we caution that our observations here are related to specific organizational characteristics of the projects we examine, and not the mere fact that the software is open source or proprietary. This is an important distinction, given it points to the type of work needed to further explore the rival hypotheses we describe. Specifically, questions on the relationship between organization and architecture can be answered *only* by capturing measures of both factors. Indeed, our ongoing work supports such a view, given we have found open source products with tightly-coupled architectures in which contributions are concentrated among a handful of developers (Rusnak, 2005). Our study therefore opens the door to a deeper understanding of the complex relationship between a product's architecture and the community that grows to surround it.

Our results show that purposeful efforts to re-design a product's architecture can have a significant impact. This is surprising, given much work suggests architectural change is difficult (Marples, 1961; Henderson and Clark, 1992). In this respect, we consider three possible explanations for our findings: First, this may be a unique feature of software products (i.e., the ease of architectural change) as compared to physical products. We note however, that several popular software products sold today contain code that is many years old despite major efforts to update their designs (and one assumes their architectures) over time (e.g., MacCormack and Herman, 2000b). Second, Mozilla may not be sufficiently complex to make architectural change a challenge. We note however, that at the time of the re-design, the product contained over one million lines of code, which would take a team of 100 developers over two years to complete using standard productivity metrics. Finally, there may be something specific about a web browser's design that makes architectural change easier to achieve. Yet we can find no evidence to support this view. We are left to speculate that with a sufficiently talented and motivated team, it is possible to achieve substantial changes in a product's architecture.

This latter result is important in the context of open source software. Recent years have seen an increase in the number of proprietary products released under an open source license, with the hope that a developer community will converge around them. Examples include Mozilla, Open Office (a personal productivity suite) and Eclipse (a suite of developer tools). The Mozilla experience however, suggests that proprietary products may not be well-suited to distributed development if they have tightly-coupled architectures. There is a need to create an "architecture for participation" that promotes ease of understanding by limiting module size and ease of contribution by minimizing the propagation of design changes. In Mozilla's case, the re-design to a more modular form was followed by an increase in the number of contributors[26] (Mockus et al, 2002). Subsequent improvements led to a product called Firefox which gained both critical acclaim and market share.[27] We speculate that without the changes made in 1998, such developments would have been unlikely, and the Mozilla experiment may well have failed.

So how is such structural change achieved? Our results illustrate that increased modularity can be realized by reducing the number of design dependencies or by re-arranging their pattern of distribution. With regard to the first strategy, our work suggests substantial latitude exists to reduce dependencies without a major impact on performance (the re-design of Mozilla reduced dependencies by around 50%). This implies the need for a deeper understanding of how and why dependencies exist and the mechanisms through which they can be managed. With regard to the second strategy, our work shows that designs with greater numbers of dependencies are not necessarily less modular than those with fewer. Poorly placed dependencies, especially those that link otherwise independent modules, may result in a cascade of unwanted and hard-to-detect *indirect* interactions. Our results suggest purposeful actions to reduce such "rogue" dependencies can be effective (the re-design of Mozilla reduced propagation cost by over 80%).

Our work opens up a number of areas for future study. With respect to methods, we believe DSMs provide a powerful lens with which to examine issues of dependency in software. While we focus on one type of dependency, our method can be generalized to others, assuming that they can be identified from

---

[26] Source: Interviews with Mozilla team members during the spring of 1999.
[27] See <http://www.mozilla.org/products/firefox/> for details.

source code. With respect to studies of architecture, our work provides visibility of a phenomena which was previously hidden, and metrics with which to conduct further studies. In particular, there is value in asking whether different types of product (e.g., databases versus operating systems) require different types of architecture, and if so, identifying their characteristics. Our ongoing work reveals large variations in the modularity of different systems (Rusnak, 2005) but we lack sufficient data to determine causes. Building a database of different system types will help answer this question. Finally, our work facilitates the study of a topic of critical importance that has proven elusive to quantify: the potential performance trade-offs from architectures with different characteristics. There are strong theoretical arguments why such trade-offs exist, yet little empirical evidence to confirm their presence. The Mozilla re-design is a case in point. How was it possible to increase modularity without degrading other aspects of performance? We speculate that many designs are not at the performance "frontier" where a trade-off exists but are positioned below it due to architectural inefficiencies or "slack." If this is true, there may be considerable scope to improve a design along one or more dimensions without incurring a performance penalty. Exploring such issues through careful measurement of architectural characteristics and product performance should help reveal strategies for moving designs *towards* the performance frontier. It will also help us understand the trade-offs that are ultimately involved in moving a design *along* it.

REFERENCES

Austin, S. A., A. N. Baldwin and A. Newton. Manipulating the flow of design information to improve the programming of building design. *Construction Management & Economics*. 12(5): 445-455. Sep 1994.

Baldwin, C. and Clark, K. B. *Design Rules: The Power of Modularity*, MIT Press, Cambridge, MA, 2000.

Banker, R.D. and S.A. Slaughter. The Moderating Effect of Structure on Volatility and Complexity in Software Enhancement, *Information Systems Research*, Vol. 11, No. 3, September 2003.

Black, T. A., C. H. Fine, and E. M. Sachs. A Method for Systems Design Using Precedence Relationships: An Application to Automotive Brake Systems. M.I.T. Sloan School of Management, WPaper 3208, 1990.

Cusumano, M. and D. Yoffie. *Competing on Internet Time*. Free Press, New York., 1998.

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

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

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

Eick, S., T.L. Graves, A.F. Karr, J.S. Marron and A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data, IEEE Transactions on Software Engineering, Vol. 27, No. 1, Jan 2001.

Eppinger, S.D., D.E. Whitney, R.P. Smith and D. Gebala, A Model Based Method for Organizing Tasks in Product Development. *Research in Engineering Design* 6(1): 1-13. !994.

Fernandez, C. I. G. Integration Analysis of Product Architecture to Support Effective Team Co-location. *Unpublished Masters Thesis*, MIT Sloan School of Management, 1998.

Fleming, L. and O. Sorenson. Science as a Map in Technological Search, *Strategic Management Journal*, 25: 909-928, 2004.

Godfrey, M., and Q. Tu. Evolution in Open Source Software: A Case Study, *16th IEEE International Conference on Software Maintenance*, 2000.

Gomes, P. J., and N. R Joglekar. The Costs of Coordinating Distributed Software Development Tasks, *Boston University School of Management WPaper*, October, 2004.

Grose, D.L., Reengineering the Aircraft Design Process, *Proceedings of the Fifth AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Panama City Beach, FL, Sept. 7-9, 1994.

Healy, K. and A. Schussman, The Ecology of Open-Source Software Development, *University of Arizona WPaper*, 2003.

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

Idicula, J. Planning for Concurrent Engineering, *Gintic Institute Research Report*, Singapore, 1995.

Kusiak, A., N Larson, and J Wang,, Reengineering of Design and Manufacturing Processes, *Computers and Industrial Engineering*, Vol. 26, No. 3, 1994, pp. 521-536.

Lopes, C. V. and S. K Bajracharya. An Analysis of Modularity in Aspect Oriented Design, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, March 2005.

MacCormack, A, and K. Herman. Red Hat and the Linux Revolution. *Harvard Business School Case Study* 600-009, 2000 (a).

MacCormack, A., and K. Herman. Microsoft Office 2000, *Harvard Business School Multimedia Case Study*, 600-023, 2000 (b).

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

Marples, D.L., The Decisions of Engineering Design, *IEEE Transactions on Engineering Management*, 8(2):55-71, 1961.

Mockus, A. R.T. Fielding and J.D. Herbsleb, Two Case Studies of Open Source Software Development: Apache and Mozilla, *ACM Proceedings on Software Engineering and Methodology*, Vol. 11, No. 3 July 2002.

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

O'Reilly, T. Lessons from Open Source Software Development, *Communications of the ACM*, Vol. 42, No. 4, 1999.

Parnas, D.L. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053-8, December 1972.

Paulson, J.W., G. Succi and A Eberlein, An Empirical Study of Open-Source and Closed-Source Software Products, *IEEE Transactioons on Engineering*, Vol. 30, No. 4, April 2004.

Pimmler, T. U. and S.D. Eppinger. Integration Analysis of Product Decompositions, *Proceedings of the ASME Sixth International Conference on Design Theory and Methodology*, Minneapolis, MN, Sept., 1994.

Pinkett, R. Product Development Process Modeling And Analysis Digital Wireless Telephones, *Unpublished Masters Thesis*, M.I.T, 1998.

Raymond, E. S. *The Cathedral & the Bazaar*, O'Reilly and Associates, Sebastopol, CA, 2001.

Rivkin, J. and N. Siggelkow, Patterned Interactions in Complex Systems: Implications for Exploration, *Harvard Business School WPaper* #05-044, 2005.

Rusnak, J. The design Structure Analysis System, *Unpublished Doctoral Dissertation*, Harvard University, June 2005.

Rusovan S., M. Lawford and D. Parnas. Open Source Software Development: Future or Fad? Forthcoming in *Perspectives on Free and Open Source Software*, Feller J. et al., eds., MIT Press, Cambridge, MA, 2005.

Sanchez, R., J.T. Mahoney. Modularity, Flexibility, and Knowledge Management in Product and Organization Design. Strategic Management Journal 17: 63-76, 1996.

Sanderson, S. and M. Uzumeri. Managing Product Families: The Case of the Sony Walkman. Research Policy, Vol 24, No. 5, 761-782, 1995.

Schach, S.R., B. Jin, D.R. Wright, G.Z. Heller and A.J. Offutt, Maintainability of the Linux Kernel, *IEE Proceedings—Software*, vol. 149, pp 18-23, 2003.

Schilling, M.A. Toward a General Modular Systems Theory and its Application to Interfirm Product Modularity. *Academy of Management Review*. Vol.25 No.2 312-334, 2000.

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

Sangal, N., E. Jordan, V. Sinha and D. Jackson, Using Dependency Models to Manage Complex Software Architecture, forthcoming in *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, 2005.

Sharman, D. and A. Yassine, Characterizing Complex Product Architectures, *Systems Engineering Journal*, Vol. 7, No. 1, 2004.

Shaw, M. and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, New York, 1996.

Simon, H. A. The Architecture of Complexity, *Proceedings of American Philosophical Association*, 1962.

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

Steward, D. V. The Design Structure System: A Method for Managing the Design of Complex Systems, *IEEE Transactions on Engineering Management*, Vol. 28, pp. 71-74, 1981.

Sosa, M. E., S. D., Eppinger and C.M. Rowles. Identifying Modular and Integrative Systems and their Impact on Design Team Interactions, *ASME Journal of Mechanical Design*, Vol. 125, pp. 240-252, 2003.

Sosa, M. E., S. D., Eppinger and C.M. Rowles. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development, *Management Science*, vol. 50, No. 12, 2004.

Sullivan, K.J., B.J. Cai, B.Hallen and W.G. Griswold, "The Structure and Value of Modularity in Software Design", *Proceedings of the Joint International Conference on Software Engineering and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Sep 2001.

Thebeau, R.E. Knowledge Management of System Interfaces and Interactions for Product Development Processes, *Unpublished Masters Thesis*, M.I.T, 2001.

Ulrich, K. T. The role of Product Architecture in the Manufacturing Firm. *Research Policy* 24: 419-440, 1995.

Von Hippel, E. Task Partitioning: An Innovation Process Variable," *Research Policy* 19, 407-418, 1990.

Von Hippel, E., and G. von Krogh, Open Source Software and the Private-Collective Innovation Model: Issues for Organizational Science, *Organization Science*, Vol 14, No.2 Mar-Apr, 2003.

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

Yu, L., S. R. Schach, K. Chen, G. Z. Heller and J. Offutt, Maintainability of the Kernels of Open-Source Operating Systems: A Comparison of Linux with FreeBSD, NetBSD, and OpenBSD, *Journal of Systems and Software*, forthcoming, 2005.

Table 1: Comparisons between comparable versions of Linux and Mozilla[28]

	Mozilla (98-04-08)	**Linux 2.1.105**
Source Files (DSM Elements)	1684	1678
Dependencies	6717	9110
Density (per 1000 source-file pairs)	2.4	3.4
Propagation Cost	17.35%	5.82%
Clustered Cost	5,323,915,033	3,697,742,107
Functions per Source File	12.8	17.7
Lines of Code per Source File	670	733

Table 2: Comparisons of Mozilla prior to and *immediately* after the re-design[29]

	Mozilla (98-04-08)	**Mozilla (98-12-11)**
Source Files (DSM Elements)	1684	1508
Dependencies	6717	3037
Density (per 1000 source-file pairs)	2.4	1.3
Propagation Cost	17.35%	2.78%
Clustered Cost	5,323,915,033	1,636,799,495
Functions per Source File	17.7	16.8
Lines of Code per Source File	733	530

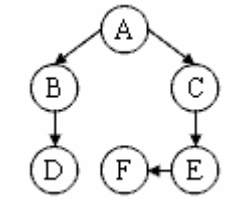
Table 3: Comparisons of Mozilla *immediately* after the re-design and Linux

	Mozilla (98-12-11)	**Linux (2.1.88)**
Source Files (DSM Elements)	1508	1538
Dependencies	3037	8519
Density	1.3	3.6
Propagation Cost	2.78%	5.65%
Clustered Cost	1,636,799,495	3,175,246,929
Functions per Source File	16.8	12.6
Lines of Code per Source File	530	670

[28] The clustered cost results we report use binary dependencies for d (rather than strengths) and use $\lambda = 2$. Sensitivity analyses using a range of other assumptions did not change the comparative results.

[29] Note that there is no public version of Mozilla prior to the re-design with around 1500 source files, hence our comparison design – version 1998-04-08 – has a greater number (1684 source files) and would therefore be expected to have a higher clustered cost, though not necessarily a higher propagation cost.

Figure 1: Example System in Graphical and Dependency Matrix 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

Figure 2: Successive Powers of the Dependency 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**

Figure 3: A DSM with dependencies in an "Idealized Modular Form"

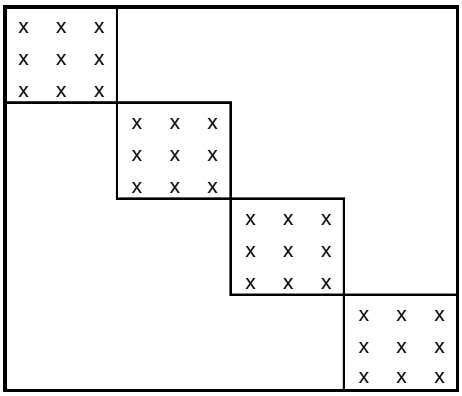


Exhibit 4: A Comparison of the first version of Mozilla and a comparable version of Linux.

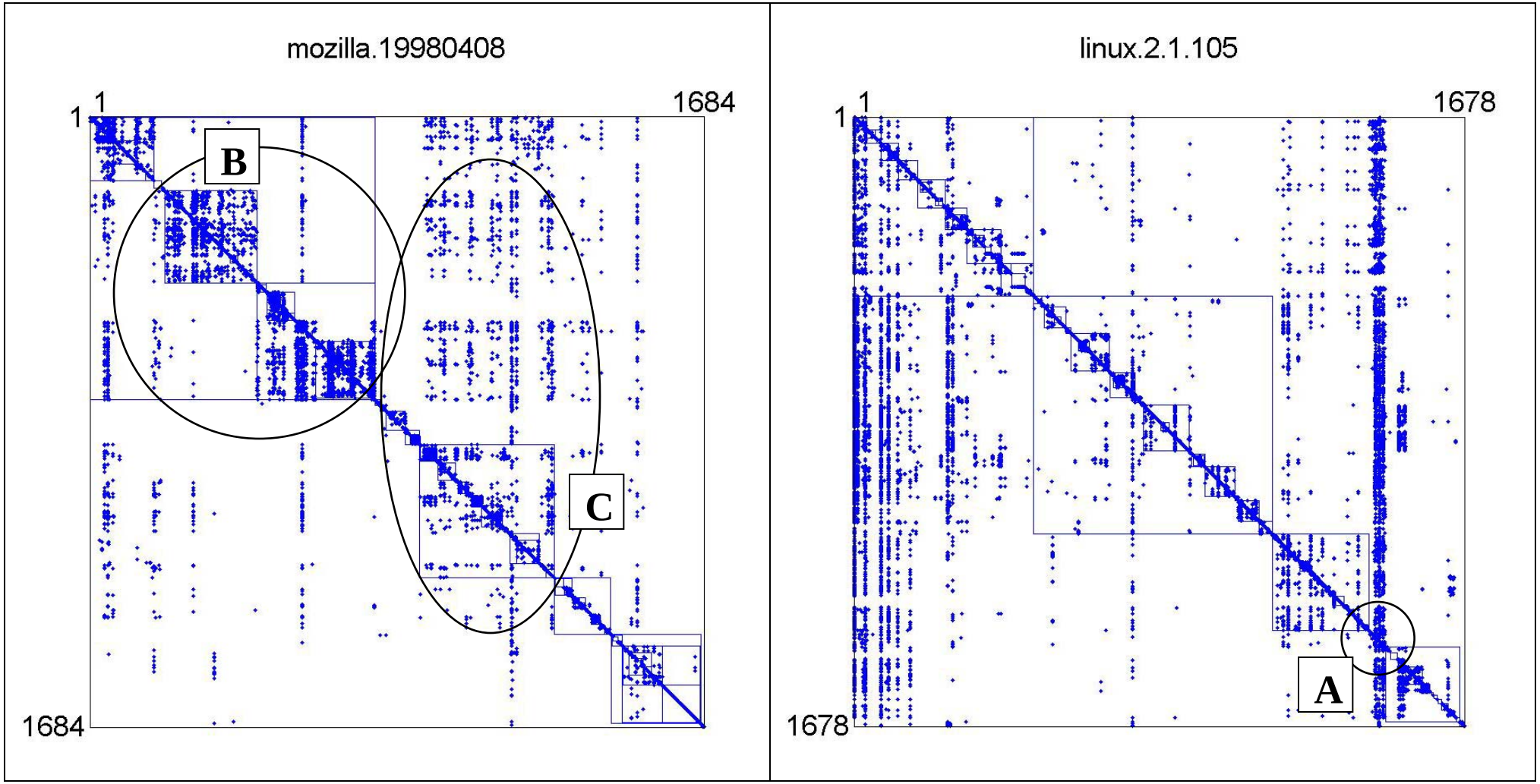


Exhibit 5: A Comparison of Mozilla before and after a purposeful re-design effort.

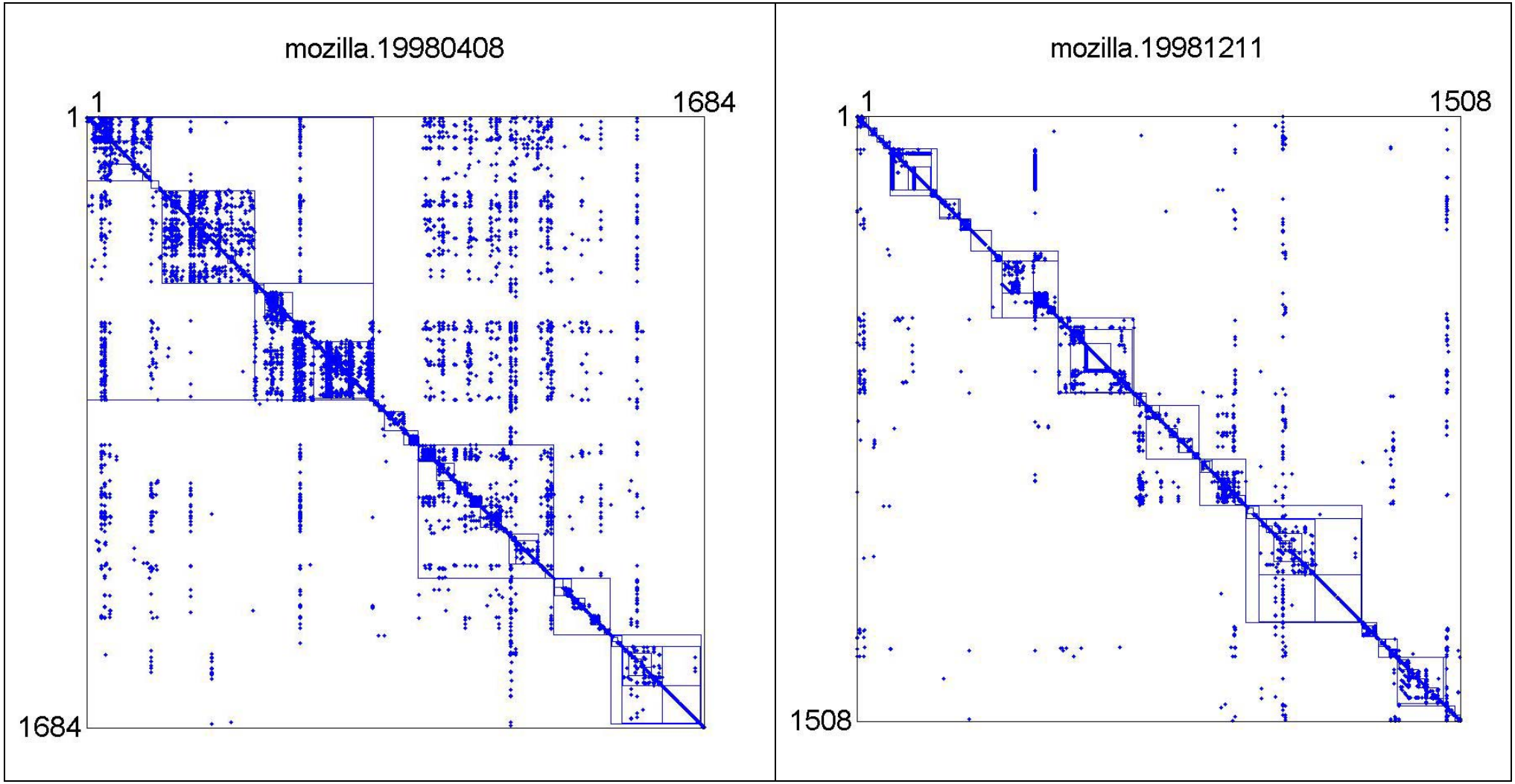


Exhibit 6: The Impact of a Re-design Effort on Mozilla's Propagation Cost

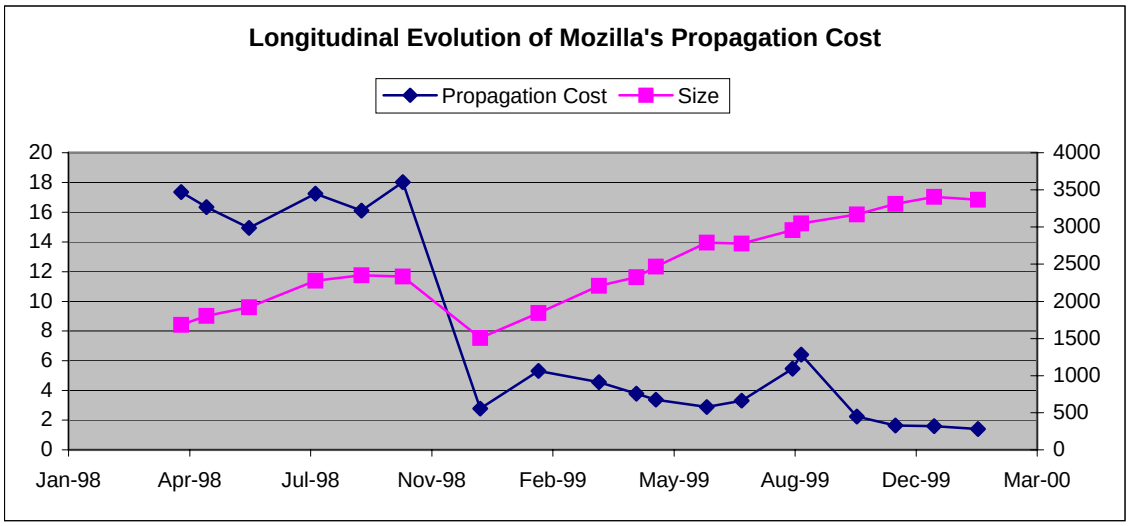


Exhibit 7: The Impact of a Re-design Effort on Mozilla's Clustered Cost