05-016

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

Alan MacCormack*
John Rusnak**
Carliss Baldwin*

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

* Harvard Business School
** PhD Candidate, Harvard University, Division of Engineering and Applied Sciences.

Copyright © 2004 Alan MacCormack, John Rusnak and Carliss Baldwin.

Harvard Business School Working Paper Number 05-016. Working papers are distributed in draft form for purposes of comment and discussion only. They may not be reproduced without permission of the copyright holder. Copies of working papers are available from the author(s).

Abstract

Much recent research has pointed to the critical role of architecture in the development of a firm's products, services and technical capabilities. A common theme in these studies is the notion that specific characteristics of a product's design – for example, the degree of modularity it exhibits – can have a profound effect on among other things, its performance, the flexibility of the process used to produce it, the value captured by its producer, and the potential for value creation at the industry level. Unfortunately, this stream of work has been limited by the lack of appropriate tools, metrics and terminology for characterizing key attributes of a product's architecture in a robust fashion. As a result, there is little empirical evidence that the constructs emerging in the literature have power in predicting the phenomena with which they are associated.

This paper reports data from a research project which seeks to characterize the differences in design structure between complex software products. In particular, we adopt a technique based upon Design Structure Matrices (DSMs) to map the dependencies between different elements of a design then develop metrics that allow us to compare the structures of these different DSMs. We demonstrate the power of this approach in two ways: First, we compare the design structures of two complex 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 find significant differences in their designs, consistent with an interpretation that Linux possesses a more "modular" architecture. We then track the evolution of Mozilla, paying particular attention to a major "re-design" effort that took place several months after its release as an open source product. We show that this effort resulted in a design structure that was significantly more modular than its predecessor, and indeed, more modular than that of a comparable version of Linux.

Our findings demonstrate that it is possible to characterize the structure of complex product designs and draw meaningful conclusions about the precise ways in which they differ. We provide a description of a set of tools that will facilitate this analysis for software products, which should prove fruitful for researchers and practitioners alike. Empirically, the data we provide, while exploratory, is consistent with a view that different modes of organization may *tend* to produce designs possessing different architectural characteristics. However, we also find that purposeful efforts to re-design a product's architecture can have a significant impact on the structure of a design, at least for products of comparable complexity to the ones we examine here.

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 amounts of value in an industry, given that this strategy creates valuable options for module improvement. However, evidence of how these modularizations occur in practice has not yet been shown. Finally, MacCormack's (2001) work on the management of Internet software projects suggests the importance of a loosely-coupled architecture that "facilitates process flexibility." Without a way to measure the key attributes of a design with empirical data however, this work cannot reach the level of specificity it needs for robust managerial prescriptions to be drawn.

Common to all these research streams (and many 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 area for both researchers and managers to understand. Unfortunately, we lack the appropriate tools, measures and terminology 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 attempts to remedy some of these shortfalls by developing a set of tools and metrics for measuring specific characteristics of a design using a technique called Design Structure Matrices. We use these techniques to compare different product designs from the software industry.

We chose to analyze software product designs for a number of reasons. First of all, the study of software design structure has a long and respected 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 designs 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 have further developed the concept of modularity and hierarchy, proposing a variety of metrics and/or tools to provide indicators of the overall structure of a design (e.g., complexity, coupling and cohesion). Examples include McCabe 1976; Halstead et al, 1976; Parnas at al, 1985; Selby and Basili, 1988; Yourdon, 1993; Dhama, 1995; Wilkie and Kitchenham, 2000.

The second reason we chose to analyze software product designs is that software is an information-based product, meaning that the design comprises a series of instructions (the "source code") that tell a computer what tasks to perform. In this respect, an existing software design can be processed automatically to identify dependencies that exist between different parts of the design (something that cannot easily be done with a physical product). These dependency relationships can be used as inputs to tools which seek to characterize various aspects of design structure, either through displaying the information visually (for example, in a Design Structure Matrix) and/or by calculating various metrics to summarize the patterns of dependencies that exist at the level of the system. Once the appropriate mechanisms have been set up to achieve this, the ability to characterize different design structures is limited only by the computing power available and the access we have to the source code of different designs (not a trivial problem, given many software firms regard source code as a form of proprietary technology).

The third reason we chose to analyze software product designs is because we wanted to take advantage of a unique opportunity to examine two differing organizational modes for developing products that might give rise to systematic differences in design structure (confirming the usefulness of the tools we were developing). Specifically, over recent

years there has been growing interest in the topic of open source (or "free"[1]) software, which is characterized by a) the distribution of a program's source code (programming instructions) along with the binary version of the product[2] and b) a licensing agreement that allows a user to make unlimited copies of, and modifications to, the design (see DiBona et al, 1999). Open source software projects are characterized by highly distributed teams of volunteer developers who contribute new features, fix defects in the existing code and write documentation for the product (von Hippel and von Krogh, 2003). These developers (which can number in the hundreds for some popular projects) are located around the globe, hence most never meet face to face. Among the most famous examples of products developed in this manner are the Linux operating system, the Apache web server, and the Sendmail email routing program.

The development methodology embodied in open source software projects stands in contrast to the "proprietary" development models employed by "for-profit" commercial software firms. These firms tend to staff projects with dedicated teams of individuals, most of whom are located at a single location, and hence have easy access to other members of the development team. Given this proximity, the sharing of information about solutions being adopted in different parts of the design is much easier, and may even be encouraged (for example, if the creation of a dependency between one part of a design and another could lead to a product performance improvement, such as an increase in the speed of operation). Consequently, the design structure that emerges from a proprietary development project is likely to differ from the design structure that emerges from an open source project. Specifically, we assert that the open source design is likely to be "more modular" than the proprietary design, all else being equal.

The final reason that we chose to analyze software product designs relates to the ability to track the evolution of a specific design over time. Most software projects use

---

[1] Free as in speech, not as in beer.

[2] Most commercial software is distributed in binary form (i.e., a series of 1's and 0's) that can be executed directly by a computer. It is extremely difficult to reverse engineer this binary form in order to access the original source code instructions written by the programmer. Hence distributing software in binary form helps keep the methods by which a program performs its tasks proprietary to the author.

sophisticated "version control" systems to track the historical development of a design as it evolves over the life of a project. For a researcher, this represents a unique opportunity to track the "living history" of a design, a technique that is typically not possible in the development of physical products.[3] In this study, such a technique is critically important, given we can observe concentrated efforts to re-design a product, with the aim of making the architecture more (or less) modular. Having access to versions of the design both prior to and after such an effort allows us to say meaningful things about the impact that managerial actions can have on the architecture of an existing product.

In summary, our research agenda can be characterized as follows:

- To develop tools and metrics for characterizing the differences in design structure between complex software products, using source code as input.
- To explore the hypothesis that open source software products are "more modular" than products developed using a proprietary development methodology.
- To identify a concentrated effort to re-design a complex software product, and track the evolution of its design to assess the impact of this effort.

The outline of this paper is as follows. First, we describe our research methodology, which applies the technique of Design Structure Matrices to map the dependencies in software designs, and develops metrics to summarize their structure. We then describe our empirical results, which are based upon a comparison of two software products developed via contrasting modes of organization, and an analysis of the impact of a major re-design effort on the architecture of one of these products. We conclude by discussing the implications of our results for both researchers and practitioners.

---

[3] For example, the first version of the Linux kernel (version 0.01) was posted onto the Internet in 1991. It consists of less than 50 source files. The most recent version of Linux that we have used in our research (version 2.5.75) consists of more than 6,000 source files. We have access to all interim versions that were released between these two points, representing a complete record of the evolution of this design.

2. Research Methodology

We adopt a technique called Design Structure Matrices (DSMs) to analyze and compare the structure of complex software products. A DSM is a tool that highlights the inherent structure of a design by examining the dependencies that exist between its component elements using a symmetric matrix (Steward, 1981). DSMs can be constructed using elements that represent "tasks" to be performed or "parameters" to be defined in a project. In such situations, their main use is to identify a partitioning of design tasks that facilitates the flow of coordinative information in a project. Tasks (or parameters) that have a high level of dependency are grouped into clusters or modules. To aid this process, a range of clustering algorithms have been developed in order to find an optimal allocation of elements to clusters, minimizing the coordination costs involved in developing the product (e.g., see Pimmler and Eppinger, 1994; Idicula, 1995; Gutierrez-Fernandez, 1998; Thebeau, 2001). To illustrate, Exhibit 1 shows a set of dependency relationships between six design elements, a DSM that captures these dependencies in matrix form, and one possible clustering arrangement for these elements.

The use of DSMs has 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 ways that this tool can help to improve the organization of development projects in general (e.g, Eppinger et al, 1994) or describing how DSMs can be used to capture the choices available to an architect in terms of a design's future evolution (e.g., see Baldwin and Clark, 2000). Recently, researchers have taken the first steps in using DSMs to evaluate alternative software design structures. Specifically, Sullivan et al show how this tool can be used to formally model (and value) the concept of information hiding, the principle proposed by Parnas to divide complex designs into a number of smaller modules, each of which can be developed relatively independently (Sullivan et al, 2001; Cai and Sullivan, 2004).

In contrast to studies that use DSMs to understand how the development of a product not yet in existence should proceed, our objective is to use this technique to analyze software designs that already exist. Specifically, we develop a number of metrics based on the analysis of DSMs for specific software products that help to characterize the differences in their design structures. Our aim is to operationalize a concept that has proven somewhat elusive in both the software industry and industries that produce physical products; the degree of "modularity" of a product's architecture.[4] We assert that this concept can only be examined on a comparative basis. That is, we cannot say that a specific design is modular; we can only say that design A is more (or less) modular than design B. Hence our work is based upon the comparisons of different designs, both from a cross-sectional perspective (i.e., product A versus product B) and from a longitudinal perspective (i.e., product A at time T versus product A at time T+n).

Our work is based upon the assumption that the degree of modularity of a software product design can be assessed using metrics developed from a DSM. While we cannot measure modularity directly, we assert that the *manifestation* of different degrees of modularity can be measured in (at least) two different ways. The first is to capture the costs of making a change to the design, in terms of the potential impact this change has on other parts of the design. Specifically, we measure the degree to which a design change to one element propagates through the system, via both direct and indirect dependency relationships with other elements. In designs that are more modular or "loosely-coupled," we assume that this measure, called Change Cost, will be lower, than in a less modular design. The second is to capture the costs of coordination involved in developing the design. Specifically, we measure the degree to which information must be communicated between clusters of related design elements, due to the dependencies that exist between these clusters. In designs that are more modular, we assume that this measure, called Coordination Cost, will be lower, than in a less modular design.

---

[4] We note several authors have attempted categorizations of different types of architectures (e.g., see Ulrich, 1995). However, we are aware of few studies that demonstrate the *measurement* of "the degree to which a given design possesses one type of architecture versus another."

Applying DSMs to Software Designs

The first choice that must be made in applying the use of DSMs to a software design is the level of analysis at which the DSM will be built. The choices available range from high level representations of the major subsystems in a product to the individual component parts that make up these subsystems. The choice of level of analysis should be driven by the research question of interest, but must also be meaningful given the context within which a design is developed. For example, it is likely to be 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). Note that for the team designing a specific subsystem however, the use of a lower level of analysis will likely be more helpful (e.g., brake pads, calipers, discs, etc.).

When considering the design of a software product, there are several potential levels at which a DSM could be built. These are nested in a hierarchical fashion as follows:

- The subsystem level, which corresponds to a group of source code files that relate to (or are perceived to relate to) a specific part of the design.
- The source file level, which corresponds to a collection of programming instructions (source code) that perform a related group of functions.
- The function level, which corresponds to a set of programming instructions (source code) that perform a specific task.

We chose to analyze designs at the source file level for a number of reasons. First of all, source files tend to group functions of a common nature together in a single unit. For example, a programmer may expect to have functions related to the printer in a source file named "printer_driver.c." Source files also typically contain header information and overview comments that apply to all related functions in the group. Secondly, tasks and responsibilities are typically allocated to programmers at the source file level, given this allows them to maintain control over groups of functions that perform related tasks. Hence this is the unit of analysis most relevant to managers and architects of software

projects. Thirdly, source control and configuration management tools typically use the source file as the unit of analysis. With these systems, programmers "check out" source files for editing, add/delete/modify instructions within the file, then "check in" the new version once their work is complete. This is the unit of analysis most tools currently adopt for use in increasing developer productivity.[5]

Capturing Dependencies between Source Files in a Software Design

We capture the dependencies between source files in a design 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 the two source files, in a *specific* direction. For example, if Sourcefile1 calls FunctionA which is located in Sourcefile2, we note that Sourcefile1 depends upon Sourcefile2. This dependency is captured in the DSM in matrix location (1, 2).[6] Note that this dependency does not imply that Sourcefile2 depends upon Sourcefile1. That is, the dependency does not have to be symmetric. This would be true only if Sourcefile2 called a different function, say FunctionB, located in Sourcefile1.

To capture the Function Calls between source files, we input all source files for the product being examined into a tool called a "Call Graph Extractor" (Murphy et al, 1998). This tool is used in software development to allow programmers to obtain a better understanding of code structure and interactions between different parts of the code. The output is typically displayed graphically, with each function being a node, and calls (i.e., dependencies) being shown as lines between these nodes with arrows representing the

---

[5] Further support for this choice comes from the fact that it is not clear that building a DSM at the subsystem level or the function level would be insightful or practical. For example, Linux version 2.5 possesses 10 subsystems, around 4,000 source files and over 60,000 functions. The former level is not granular enough to generate meaningful insights into design structure, whereas the latter is too granular to understand the design – akin to examining the dependencies between nuts, bolts and pieces of metal in an automobile. From a practical standpoint, building a DSM with 60,000 elements and then calculating metrics relating to its structure challenge the computational limits of our current technology.
[6] Note that we use the convention (row_number., column_number.) to describe entries in a DSM.

direction of the dependency. For our purposes however, we merely capture the dependencies output from the extractor in matrix form, representing the existence (or not) of one or more dependencies between each source file in the software (see Exhibit 1).

It should be noted that function calls in software can be extracted either 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 attempts to capture the structure of the design from the programmer's perspective.[7] Rather than develop our own call graph extractor, we tested and reviewed several commercial and academic products that had the ability to process source code written in a variety of languages (e.g., C and C++), captured indirect calls (dependencies that flow through an intermediate source file), could be run in a bulk automated fashion and would output data in a format that could be used as input to a DSM.

The Analysis of Design Structure

Once we have populated a DSM with function call dependencies between different source files, we analyze the structure of the design in three ways: first, by examining the *Architectural View* of the system; second, by determining the impact of a change to each source file, in terms of the percentage of other source files that are potentially affected (summarized by a metric called *Change Cost*); and third, by clustering the DSM into modules containing source files that are highly interdependent and calculating a *Coordination Cost* for the resulting structure. We discuss each of these methods in turn.

---

[7] There are also differences between "Compiler Extractors" and "Design Extractors." The former are used mainly to help compile the software into an executable form. The latter are used to aid programmers in determining source relationships. We therefore use the latter type of extractor.

The Architectural View

In the development of software systems, programmers tend to group source files of a related nature into "directories" or folders. These directories are organized in a nested fashion, representing the programmer's view of the system architecture. The Architectural View of a DSM groups each source file into a series of nested clusters as defined by the directory structure. Boxes are drawn around each successive layer in the hierarchy. The result is a map of source file dependencies, organized by the programmer's perception of the system's design structure. To illustrate, an example of the directory structure of Linux version 0.01 is shown in Exhibit 2. The Architectural View of its DSM is shown in Exhibit 3. This view is useful for visually highlighting differences in design structure between software products of comparable complexity.

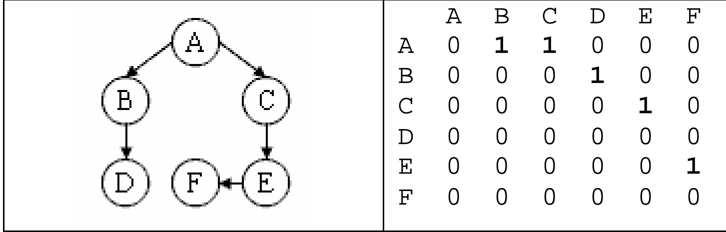
Change Cost

The second method by which we characterize the structure of a design is by measuring the degree of "coupling" it exhibits, as captured by the degree to which a change to any single element causes a (potential) change to other elements in the system, either directly or indirectly (i.e., through a chain of dependencies that exist across elements). This work is very closely related to and builds upon the concept of visibility (Sharmine and Yassine, 2003) which in turn, is based upon the concept of reachability matrices (Warfield, 1973).

To illustrate the concept of visibility, it is useful to consider an example. Consider the element relationships in Figure 1 displayed in both graphical and dependency matrix forms. We can see that element A depends on (or "calls functions within") elements B and C. Hence any change to element B may have a direct impact on 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* may have an indirect impact on element A, with a "path length" of 2. Finally, we also see 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 interdependencies between elements for path lengths of 4 or more.

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

We can 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. Note that we choose to include the matrix for n=0 (i.e., a path length of zero) when calculating the visibility matrix, implying that a change to an element will always affect itself. Figure 2 illustrates the calculation of the visibility matrix for the example above.

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**

From the visibility matrix, we can construct several metrics to give us insight into a system's design structure. The first, called "Fan-Out Visibility," is obtained by summing along the rows of the visibility matrix, and dividing the result 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 the result 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 the example above, element A has a Fan-Out visibility of $6/6^{th}$ (or 100%) meaning that it depends upon all elements in the system. It has a Fan-In visibility of $1/6^{th}$, meaning that it is visible only to itself.

For the purposes of summarizing visibility data at the system level, we compute the average Fan-Out and Fan-In visibility of all elements in the system. Note that due to the symmetric nature of dependency relationships in the system (i.e., for every fan-out, there is a corresponding fan-in) these are identical. We call the resulting metric "Change Cost." Intuitively, this measures the percentage of elements affected, on average, when a change is made to one element in the system. In the example above, using the figures for Fan-out visibility, we can calculate the Change Cost as [6/6+2/6+3/6+1/6+2/6+1/6] divided by 6 elements = 42%. Similarly, when using Fan-In visibility, the system has a change cost of [1/6+2/6+2/6+3/6+3/6+4/6] divided by 6 elements = 42%.

The example above is rather simple in that the design structure does not contain multiple paths between elements and the dependency relationships are purely hierarchical.[8] This will not generally be the case for complex systems, hence complicating the calculation of visibility and change cost. In particular, multiple paths bring the possibility that there is more than one route through which element A depends upon element B (e.g., element A may depend upon elements C and D, both of which depend upon element B). As a result, we limit the 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.

---

[8] A hierarchical system can be identified by the fact that its dependency matrix can be re-arranged such that all non-zero entries are above the diagonal, which is called an "upper-triangular" form.

If a system is hierarchical, the matrix of dependencies will eventually converge to zero. Once the zero matrix is reached, we can sum up the set of visibilities and calculate a change cost. However, when a system is not hierarchical, meaning that it contains one or more cycles, successive powers of the dependency matrix will not converge to zero. The simplest example is a relationship whereby element A depends upon element B which in turn, depends upon element A. To avoid problems like this, in systems that possess cycles of any type, 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 between elements). Given that we also limit entries in the dependency matrix to binary values, the fact that cyclic relationships appear many times ultimately has no impact on the output.

Coordination Cost

Most prior work on the use of DSMs analyzes dependencies between tasks (or design parameters) that have yet to be performed (or defined) rather than the dependencies between elements in an existing design. When used in this manner, the DSM aids managers in partitioning the process of design by re-arranging the sequence and grouping of tasks (or parameters) in a way that minimizes the requirement to communicate information between different parts of the design. While early studies used visual inspection to find better re-arrangements for a DSM, later research developed algorithms to cluster tasks (or parameters) based on the level of interdependency between them.

A range of algorithms can be applied to cluster a DSM, however most operate in a similar fashion, defining a cost function to measure the "goodness" of a proposed solution, and invoking an iterative search technique to find the lowest cost grouping of elements. The cost function operates by allocating a cost to each dependency between elements in the DSM, depending upon whether the elements are in the same cluster or not. Intuitively, this cost can be thought of as the cost of communicating information between the agents (individuals, teams, etc.) responsible for developing each cluster (or element). It is, in essence, a coordination cost. When we apply this technique to a design that already exists, as opposed to a set of tasks or design parameters to be performed or defined, the

result is the *implied* coordination cost of replicating this design, assuming that the design process follows an "ideal" path, as indicated by the lowest-cost grouping of elements in the DSM. Note this is unlikely to reflect the *actual* coordination cost incurred in developing the design, given the iterative nature by which most designs evolve over time.

There is no single best way to cluster a DSM, and no "right" formula for computing what we call coordination cost. The clustering approach we adopt here is therefore based upon prior research (Pimmler and Eppinger, 1994; Idicula, 1995; Gutierrez-Fernandez, 1998; Thebeau, 2001) adapted for the specific context of software development. Specifically, we first identify and account for what we call "Buses" – source files that are called by or call to a large number of other source files (see below). We then adopt an iterative clustering process in which source files chosen at random receive bids to join clusters based upon their level of interdependence, with the winning bid accepted on the condition that it lowers total system cost. We describe this procedure below.

While prior work has identified the role of design elements called "Buses" (e.g., see Ulrich, 1995) these elements have particular importance in software development. In particular, designs often make use of common functions that are called by many other source files in the system – we call these "Vertical Buses," given they are seen in the DSM as an element with many vertical dependencies.[9] Examples include library code, global functions such as "print to screen" or error handling code. The use of such buses avoids duplication of effort, and ensures consistency in the approach to system-wide functions. These files are typically identified early in the design process and have well-defined, stable interfaces. Their use helps minimize coordination costs.

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).

---

[9] Another type of Bus occurs when a specific source file makes many calls to other source files in the system. We call these "Horizontal Buses," given they are seen in the DSM as an element with many horizontal dependencies. They reflect control hubs within the system. As an empirical matter however, most horizontal buses exist at the subsystem level rather than at the system level.

We then examine whether the result is above or below a user-defined parameter called the "Bus Threshold." If it is, we treat it differently in assessing its contribution to coordination cost (described below). Note that there is no "right" answer as to what level of connectivity should constitute the threshold for a Vertical Bus. In our work, we have found it useful to vary this parameter from 10-100% to see what impact this has on results. This approach is appropriate, given our analysis is aimed at comparing different design structures rather than determining the absolute values for cost in a specific design.

Once Vertical Buses have been identified, a clustering algorithm attempts to determine the lowest total coordination cost for the system. It does this by allocating a cost to each dependency in the design, depending upon whether the elements between which the dependency exists are contained within the same cluster or not. Specifically, when considering a dependency between elements i and j, neither of which is a vertical bus, the cost of the dependency takes one of two forms:

$$CoordCost(i \to j \mid \text{in same cluster}) = (i \to j)^{\text{cost_dep}} * cluster_size^{cost_cs}$$

$$CoordCost(i \to j \mid \text{not in same cluster}) = (i \to j)^{cost_dep} * dsm_size^{cost_cs}$$

Where $i \to j$ represents the strength of the dependency (that is, the number of calls between source files) and cost_dep and cost_cs are user-defined parameters that reflect the relative weights given to the strength of dependencies versus the size of the cluster.

If an element has been defined as a Vertical Bus (see above) the cost of a dependency between it and another element takes the form:

$$CoordCost(i \to j \mid \text{where j is a vertical bus}) = (i \to j)^{cost_dep}$$

In essence, we define the coordination cost for a vertical bus to be the sum of dependencies on this element, weighted by the parameter cost_dep. Note that there are no "right" values for cost_dep or cost_cs. In practice, analysis proceeds by setting each

parameter to an initial value, comparing the metrics for the design structures of interest, and testing the sensitivity of results to changes in the parameter values.

The DSM clustering algorithm attempts to determine the optimal allocation of elements to clusters such that the sum of all dependency costs is minimized. The challenge is that for a design of moderate or greater complexity the potential solution space (i.e., the combination of all possible clustering solutions) quickly becomes too large to examine exhaustively.[10] Hence we adopt an iterative search process, whereby elements are clustered via a measure of their affinity for each other, and the evolving results continuously tested to ensure total system costs are decreasing. This is achieved by randomly selecting elements which are the subject of bids from existing clusters (or single elements) to join them. The size of the bid is a function of the dependencies between the selected element (say "i") and the cluster (say "k"). This bid takes the form:

$$Bid(cluster_k, i) = \frac{\left(\sum_{\text{All elements j in } cluster_k} [(i \rightarrow j) + (j \rightarrow i)]\right)^{bid_dep}}{cluster_size(cluster_k)^{bid_cs}}$$

Where bid_dep and bid_cs are parameters defined by the user that reflect the relative weights given to the strength of dependencies versus the size of the cluster.

Intuitively, each cluster bids for new elements based on the weighted average number of dependencies that exist between the new element and existing elements in the cluster. Winning bids are accepted if the proposed new clustering solution would lower the total coordination cost of the system. The bidding process continues in an iterative manner, stopping when the number of iterations with no improvement in total cost exceeds a threshold defined by the user (called "Iteration Stop"). In practice, we have found that setting this threshold equal to the size of the DSM ensures that the algorithm converges to a low cost solution (see Exhibit 4 for an example using a version of Linux). Given the

---

[10] The precise form of the relationship between DSM size and the number of possible clustering solutions is complex. However, calculation of this figure for a 20x20 DSM yields over 10^13 possible combinations. Clearly, for DSMs of the size we examine (1,000+ source files), the number is extremely large.

heuristic nature of the search process, this is not necessarily *the* lowest cost solution, and the outcome will likely differ by a small margin each time the process is repeated.

Note that unlike change cost, coordination cost increases with size, as measured by the number of source files a product contains. Hence when using this metric to compare design structures, we must be careful to ensure that the designs are of similar size.

The Choice of Software Products to Analyze

Our study had two aims. First, to examine whether the tools and metrics we had developed could capture the differences in design structure between two software products that had been developed using different organizational models. And second, to examine whether these tools and metrics could lend insight into the longitudinal development of a design, specifically where a concentrated effort was undertaken to re-design a product's architecture with the goal of making it more (or less) modular.

For the first of our questions, we chose to focus on the comparison of two particular software products, Linux (or more strictly, the Linux Kernel) and Mozilla.[11] Linux is the most widely known open source software product, a UNIX like operating system that has been developed by hundreds of distributed developers since its release onto the Internet in 1991. Linux has a significant market share in the server operating system market, and continues to develop at a rapid pace, at least in terms of the amount of new code and functionality added each year (see MacCormack and Herman, 2000a for a brief history). While Mozilla is also an open source product, its origins lie in a product developed using a proprietary development model. Specifically, in March 1998, Netscape released the code for its Netscape Navigator web browser onto the Internet, calling this product Mozilla. This move was part of a strategy designed to respond to increasing competition in the browser market. The hope was that volunteer developers would contribute to the

[11] Given the difficulty of accessing proprietary source code without being restricted as to the publication of results, we focused our attention on analyzing open source software code that was freely available. Mozilla provided a proxy for a proprietary product design, given its unique history as described above.

design of Mozilla in the same fashion as had happened with Linux. For our purposes, the earliest versions of Mozilla represent a proxy for a design developed using a proprietary methodology (i.e., a product developed by a dedicated team of individuals mostly located in a single location). We note that when released onto the Internet in 1998, Mozilla already comprised over 1500 source files. By contrast, the Linux kernel comprised less than 50 source files when first released onto the Internet in 1991.

For the second of our questions, we take advantage of a "natural experiment" that occurred in the ongoing development of Mozilla during the fall of 1998. At that time, a re-design effort was carried out with the specific aim of making the design more modular. This effort was carried out by a small team of developers, most of whom worked for Netscape (at this time, Netscape sold a commercial browser that used the Mozilla code as its base). Fortunately, because the Mozilla code was open source at this time, we have access to the source code for versions of the product immediately prior to the re-design effort and immediately afterwards. By comparing the design structures of these versions, we can gain insight into the impact that this re-design had on the product's architecture.

3. Empirical Results

We divide our results into two parts. First we consider the comparison of design structures between the earliest complete version of the Mozilla web browser – used as a proxy for a proprietary developed product – and a version of Linux of comparable size (in source files). We then examine the longitudinal evolution of Mozilla's design structure, paying particular attention to a re-design effort that aimed to make explicit changes to the product's architecture to make the design more modular, and thereby facilitate the ease with which contributions could be made by volunteer developers.

A Comparison of the Design Structures of Linux and Mozilla

We first compare the Architectural Views of the DSMs for both Linux and Mozilla (see Exhibit 5). Given the size of these systems, each comprising over 1,500 source files, we

remove the source file names normally printed on the DSM for ease of examination. We identify the source file directory structure several levels deep in the design hierarchy (identified by the nested series of boxes that appear in each DSM). It should be noted that while we are limited by the format of this paper, in practice, designers would be able to print these DSMs at much larger scales to gain insight into a design's structure.

From these plots we can make some overall comments about the structures of these systems. First of all, at the highest level, both designs are architected into a small number of major subsystems, with a few very small subsystems. Noticeably in the case of Linux, the small subsystems appear to contain several vertical buses, meaning that the source files within them are called by many other source files in the system (see annotation A). In Mozilla, while there are a few highly connected source files, these appear to be distributed throughout the system. We should note that empirically, our analysis identifies 14 vertical buses in Linux, using a threshold of 10% connectivity to other source files. In Mozilla, only 2 source files meet this criterion for a vertical bus.

Looking below the highest layer in the directory hierarchy, we note several differences in design structure. First of all, looking at the upper left hand corner of Mozilla's DSM, we note a few large groups of source files that are very tightly connected (see annotation B). By contrast, Linux comprises a greater number of small groups that are less tightly connected (i.e., they have fewer dependencies between them). Second, it is insightful to look at the nature of the dependencies between groups of source files. In Mozilla, we note a group of source files comprising a major subsystem (see annotation C) that has many dependencies above it, indicating that it is tightly connected to other parts of the system. 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.

Another way of characterizing the differences between these designs is to examine the total number of dependencies marked in the DSM. In this respect, the architectural view appears to show that Linux has a greater number of dependencies. A quantitative analysis of the two DSMs confirms this fact. The version of Mozilla we examine has

7880 dependencies. This represents a "density" of 2.8 dependencies per 1000 source file pairs (there are 1684^2 locations in the DSM).[12] By contrast, Linux has a density of just under 4 dependencies per 1000 source file pairs. This result alone however, has little bearing upon the question of whether one design is more modular than the other, as we will see below. It is the specific locations of these dependencies, and the way that they relate to each other that dictates the important architectural characteristics of a design.

We believe that the Architectural View of a DSM gives significant insight into a design's structure, organized in a way that the architect(s) of the system perceives to be most relevant. However, for a quantitative assessment of design structure, we must turn to the metrics developed earlier, and specifically, comparisons of change cost and coordination cost. The metrics for each design are summarized in Table 1 below.

Table 1: Comparisons between comparable versions of Linux and Mozilla

	Linux	**Mozilla**
Number of Source Files (DSM Elements)	1678	1684
Change Cost	5.16%	17.35%
Coordination Cost[13]	20,918,992	30,537,703

When we compare the metrics that summarize design structure at a system level, the results are striking. First we contrast the change cost of these two designs, that is, the percentage of source files that are affected, on average, when a change is made to a source file. For Linux, this figure is 5.16 %. For Mozilla, this figure is 17.35%. This is a significant difference. It implies that the design of Linux is much more loosely-coupled than the design of Mozilla (for the specific versions we examine here). A change to a source file in Mozilla has the potential to impact three times as many source files, on average, as a change to a source file in Linux (of course, the *specific* results obtained

---

[12] These figures suggest that most software DSM's constructed at the source file level are likely to be relatively sparse (i.e., have very few dependencies per possible source-file pair).

[13] Note that this cost is calculated with parameter values as follows: cost_dep and cost_cs = 1; bid_dep and bid_cs = 2. Sensitivity analyses with a number of variations of these basic parameter values shows that the differences in coordination cost between Linux and Mozilla remain.

would depend upon the particular source file chosen for modification). These results are supported by the evidence that comes from clustering each DSM to calculate a coordination cost. The implied coordination cost for Linux is around 2/3rd of that for Mozilla (the absolute values have no literal interpretation).

To conclude, we have compared the design structures of two systems of similar size (in terms of the number of source files) one having been developed using an open source approach to development, the other having been developed using a proprietary approach to development. The architectural view reveals visual differences in design structure consistent with an interpretation that Linux uses more vertical buses in its design and is more modular than Mozilla. The metrics we develop to measure attributes of design structure are consistent with this interpretation. Indeed, we find the extent to which Linux and Mozilla differ is very significant, in terms of these metrics. Specifically, a change to one element in the system is likely to impact three times as many other elements in Mozilla, as compared to Linux. We conclude that the earliest versions of Mozilla were much less modular than comparable versions of Linux.

The question that naturally arises given these results concerns the degree to which the design differences observed are a natural product of the differing organizational modes used in their development, or reflect purposeful design strategies employed by their architects. For Linux, there is evidence that the original author, Linus Torvalds, emphasized modularity as a design criterion early in development (MacCormack and Herman, 2000a). The design was also built on the architectural heritage of both the UNIX and MINIX operating systems, which are regarded as being relatively modular at the system level. By contrast, we know little about the intentions of the original architect(s) of Mozilla (that is, the architect(s) of Netscape's Navigator browser). However, we do know that as an early entrant to a new product category (i.e., the web browser) they are likely to have focused on superior product performance (e.g., speed of operation). That may well have led them to adopt a more tightly-coupled design.

There is however, another dimension of the problem not yet considered that provides an alternative explanation for the results we report. This relates to the inherent design structures that software products that perform different functions may *require*. Put simply, an operating system may require a different design structure to a web browser purely as a result of the different tasks and performance demands it faces. Unless we have a true apples-to-apples comparison, it is therefore difficult to draw any general conclusions. The perfect solution would be to obtain the source code of an operating system developed in a proprietary fashion and compare its design structure to that of Linux (unfortunately, this has not proved possible). 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 the design. Fortunately, such a "natural experiment" did in fact occur during Mozilla's development, as we now discuss.

A Comparison of the Design Structure of Mozilla before and after a "Re-design" Effort

Netscape's Navigator web browser was released onto the Internet in early 1998.[14] At that point, the source code became known as Mozilla, and development began to proceed in the usual fashion associated with other open source software products. Module owners maintained responsibility for reviewing and checking in new code to the master version of the design. Submissions of new functionality, fixes to known defects and reports of new bugs began to come in from around the globe. Within a few months however, the initial enthusiasm for contributing to Mozilla began to decline. The main reason as cited in accounts of Mozilla's history, was the effort involved in understanding and contributing to the code base (e.g., see Cusumano and Yoffie, 1998). In particular, Mozilla's code base was so tightly-coupled that it was hard for would-be contributors to focus on a small part of the design and modify it without having to examine the potential impact on many other parts of the system. Indeed, the results we report above point to

[14] This was not an insignificant task. Third party code that could not be released under an open source license had to be removed. Furthermore, code pertaining to security routines (i.e., encryption) that was not permitted to be exported from the U.S. also had to be removed. The result was a product that did not function particularly well in its initial form.

the nature of this problem – anyone making a change to a Mozilla source file would have to check three times as many possible interactions, on average, as a contributor to Linux.

In the fall of 1998, Mozilla's design team decided to undertake an effort to re-design the codebase, with the objective of making it more modular, and hence more attractive to potential contributors.[15] The re-design effort encompassed two objectives: first, to remove redundant source files and reduce the overall size and complexity of the product; and second, to re-structure the architecture of the product, grouping source files into smaller, related modules, with fewer connections between them. This task was accomplished by a small team of core developers who worked for Netscape (at that time, Netscape sold a commercial browser which used the Mozilla code as its base). The re-design took approximately two months to complete. Prior to the re-design, Mozilla had grown to encompass 2333 source files (version 1998-10-08). After the re-design, the software comprised 1508 source files (version 1998-12-11).

There are two ways to examine the impact of this re-design, given the fact that the number of source files was reduced so dramatically. The first is to compare the design that emerged from the re-design with an earlier Mozilla version of similar size (around 1500 source files). The second is to compare the design immediately *prior* to the re-design with a later Mozilla version of similar size (around 2300 source files).

Exhibit 6 shows the Architectural Views for 2 versions of Mozilla that are of a similar size but are separated by the re-design effort: version 1998-04-08 and version 1998-12-11.[16] The results are striking – indeed, they are more significant than the differences between the original Mozilla release and Linux. The re-designed version of Mozilla comprises much smaller clusters of source files. Furthermore, there are very few dependencies *between* these clusters, or indeed between any of the individual source files. Much of the DSM is "white space." Illustrating this point, the density of the DSM (i.e.,

---

[15] Source: Interviews conducted with Mozilla's core team members, during the spring of 1999.

[16] 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 coordination cost, though not necessarily a higher change cost.

number of non-zero entries) for the re-designed product is only 1.8 dependencies per thousand source file pairs, as compared to 2.8 for the earlier version. We show in table 2 the calculations of change cost and coordination cost for this paired comparison.

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

	Mozilla (before)	**Mozilla (after)**
Number of Source Files (DSM Elements)	1684	1508
Change Cost	17.35%	2.78%
Coordination Cost[17]	30,537,703	10,234,903

These metrics confirm the dramatic impact of this re-design effort on the design structure of the product. Most significantly, the change cost of the design has dropped by a factor of five. That is, changes to a source file impact far fewer other source files, on average, after the re-design effort. This dramatic reduction is also mirrored by the reduction in coordination cost, which has fallen to a third of its previous level. We therefore conclude that the efforts made by the Mozilla team to make the design more modular were extremely successful. These results are confirmed by the data in table 3, where we look at a different matched pair, consisting of the Mozilla version immediately prior to the re-design, and a later design of comparable complexity.

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

	Mozilla (before)	**Mozilla (after)**
Number of Source Files (DSM Elements)	2333	2325
Change Cost	18.00%	3.80%
Coordination Cost[18]	55,395,336	28,581,517

---

[17] Note that this cost is calculated with parameter values as follows: cost_dep and cost_cs = 1; bid_dep and bid_cs = 2. Sensitivity analyses with a number of variations of these basic parameter values show that the differences in coordination cost between these versions of Mozilla remain.

[18] Note that this cost is calculated with parameter values as follows: cost_dep and cost_cs = 1; bid_dep and bid_cs = 2. Sensitivity analyses with a number of variations of these basic parameter values show that the differences in coordination cost between these versions of Mozilla remain.

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 change cost and coordination cost for 24 months in Exhibits 7 and 8 respectively. Also included on these plots are variations in the size of the design over time (in terms of the number of source files). The results are striking. Considering first the change cost of the design, we see the impact of the re-design effort in late 1998, which reduced change cost from a level varying between 15-18% to a level varying between 2-6%. Similarly, coordination cost drops from over 50,000,000 to around 10,000,000 (although much of this fall comes from the reduction in the number of source files in the product – hence the need for the paired comparisons described earlier). Note that these plots provide useful information to those responsible for managing the evolution of a product's architecture. Specifically, problematic trends can be identified quickly and corrective action taken as a result. Indeed, this appears to have happened in late 1999, where we see an upward trend in change cost in August, but note that the figure is much lower in the subsequent release.

The final piece of analysis we do is to compare the re-designed version of Mozilla with a version of Linux of similar size (version 2.1.88). The results are shown in table 4.

Table 4: Comparisons of Mozilla *immediately* after the re-design and Linux

	Mozilla	**Linux**
Number of Source Files (DSM Elements)	1508	1538
Change Cost	2.78%	3.72%
Coordination Cost[19]	10,234,903	15,814,993

We see that the re-design has succeeded in making Mozilla at least as modular as Linux. Indeed, it appears to have made the design more modular than the open source product. Assuming that the re-design did not lower the performance of the product on some dimension (an assumption supported by the competitive nature of the browser market at

[19] Note that this cost is calculated with parameter values as follows: cost_dep and cost_cs = 1; bid_dep and bid_cs = 2. Sensitivity analyses with a number of variations of these basic parameter values shows that the differences in coordination cost between Linux and Mozilla remain.

this point in time) we reach a compelling conclusion: There was no *requirement* for Mozilla to possess as tightly-coupled a design structure as it did prior to the re-design effort. That is, it is unlikely that our earlier results are explained by inherent differences in design structure dictated 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 the specific choices that were made by Mozilla's original architects.

4. Discussion

The primary contribution of this paper has been to develop a conceptual framework and a set of tools and metrics for analyzing the structure of complex software designs. Using the technique of Design Structure Matrices, we have shown that it is possible to characterize differences in structure between designs of comparable size, using data on the *actual* dependencies that exist between elements in a design. The techniques we develop make use of both the visual representation of design structure, as well as system-level metrics which characterize the degree of modularity in its architecture. Our metrics make use of the assumption that modularity in a design is manifested in (at least) two ways; first, in the degree to which changes to one element in a design propagate to other elements in the design (called Change Cost); and second, in the degree to which information must be communicated between agents responsible for developing clusters of related design elements in the process of development (called Coordination Cost).

At a detailed level, we have demonstrated the power of the tools and metrics we have developed by comparing the design structures of two software products of moderate to high complexity. We do this at the source file level, the level most relevant to the software architect or programmer wishing to understand a design, and to the manager wishing to prescribe actions that can influence the evolution of the design in a specific direction. Our results show that these tools and metrics help to capture both the differences in structure that have evolved over time in an evolutionary manner, as well as purposeful efforts to change the nature of a design in a single concentrated effort.

Our specific results are based upon the examination of only two software products and therefore must be regarded as highly exploratory. Nevertheless, they generate some useful empirical evidence that sheds light on the question of whether there exists a link between the architecture of a product and the characteristics of the organization that produced it. We compare the design of the Linux operating system, an open source software product developed by a distributed team of "volunteer" developers, to the design of a web browser called Mozilla that was originally developed in a proprietary fashion (i.e., by a focused team of engineers who worked for a single firm and who were mostly located at a single location). Our results are consistent with an interpretation that Linux was more modular than Mozilla. However, we also show that a single concentrated effort to re-design Mozilla had a significant impact on its design. Indeed, the architecture of Mozilla, post re-design, was more modular than that of Linux.

Our results are noteworthy for two reasons. First of all, they suggest that purposeful efforts to re-design a product's architecture can have a significant effect. The results are surprising, given that much work suggests decisions made at the highest "architectural" layers of a design are difficult to change without having to make major changes to other portions of the design as a consequence (Marples, 1961; Henderson and Clark, 1992). In this respect, we offer three possible explanations: First, this may be a unique feature of software products (i.e., the ease with which their architectures can be changed) as compared to physical products. While there is likely some truth to this notion, we note 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 that time-span (e.g., see MacCormack and Herman, 2000b). Second, it could be that the complexity of Mozilla was not sufficient for major architectural changes to represent a problem. We note however, that the product contained well over a million lines of code, which would take a team of 100 developers around 2 years to complete using standard

productivity metrics.[20] Finally, there may be something specific about a web browser's design (or the category of products to which it belongs) that makes architectural change easier. Yet we can find no evidence to support such an assertion. We are therefore left to speculate that with a sufficiently talented and motivated team, it is possible to achieve rather substantial changes in the design of a product's architecture.

The second aspect of our results that is of interest relates to the differences between the design of Linux and the design of Mozilla. We show that the original design of Mozilla (a proxy for proprietary developed code) is significantly less modular than a comparable version of Linux. Assuming that there are benefits to modularity, for example, in terms of product reliability and the ease with which parts of the design can be changed over time, one might ask why this was the case? One explanation is that we do not have a true apples-to-apples comparison, so we are merely capturing the natural differences that emerge given the differing performance requirements of these two product categories. However, we have also shown that a purposeful effort to re-design Mozilla resulted in an architecture that was *more* modular than a comparable version of Linux. So there is no reason to expect, ex-ante, that Mozilla *should* be less modular than Linux.

We speculate that there are two intertwined reasons for our observations. The first is that the design structure that evolved in each case was a direct reflection of its development environment. In Mozilla's case, a focused team located mostly at a single site, all of whom worked for the same firm (Netscape) was involved in developing the design. Design problems could be solved by face-to-face interactions, and performance could be "tweaked" by taking advantage of the ease of access that one team of module developers could have to the information and design solutions being developed in another. The design therefore became tightly-coupled. In Linux' case however, hundreds of distributed developers situated around the world contributed to its design in a loosely-coupled fashion. Face-to-face communication was almost non-existent, as most

---

[20] Studies of developer productivity tend to quote standards of 10-20 lines of code per person-day, depending upon programming language, programmer experience etc. Of course, using more advanced metrics, such as function points, is a far more reliable way of predicting lead times for any specific project.

developers never met. Hence the design structure naturally evolved to reflect this mode of organization. Indeed, this mode of organization was only possible given that the design structure, and specifically, the partitioning of design tasks, was loosely-coupled.[21]

The second reason for observing the more modular nature of Linux reflects the particular choices that were made by the original *architects* of each system in response to the contextual challenges each faced. 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 highly significant. By contrast, for Linus Torvalds, the benefits of modularity were substantial. Without such an architecture, there was little hope that other contributors could a) understand enough of the design to contribute in a meaningful way, and b) develop new features or fix existing defects without affecting many other parts of the design. Linux needed to be modular to attract and facilitate a developer community.

This work has many potential implications for practitioners. First of all, we introduce a new tool (DSMs) and a set of metrics based upon this tool that have the power to characterize aspects of design structure which have often proved elusive. While we acknowledge that the absolute value of metrics such as change cost and coordination cost have no real meaning considered in isolation, their use in comparing different designs, from both a cross-sectional and a longitudinal perspective, are a useful addition to a developers toolkit. Indeed, developers and maintainers of software products may be able to use such tools to benchmark design efforts against those in similar product categories, as well as those of their own earlier versions. These tools may also be useful as a way of illustrating the impact of (or setting performance objectives for) major re-design efforts.

For the academy, our work contributes to the current efforts of scholars to apply the concept of Design Structure Matrices to the analysis of software design structures. Our

---

[21] Linus Torvalds, the original architect of Linux, makes this point (see MacCormack and Herman, 2000a).

specific contribution has been in operationalizing the concepts of DSM analysis in an empirical setting, and measuring aspects of design structure for two products of moderate to high complexity. We acknowledge that these are early results, and the designs we analyze are not an "ideal" matched pair. Hence much work remains to be done. Specifically, in the domain of tools and metrics, we continue to search for clustering algorithms which more accurately reflect the coordinative costs involved in large scale software development efforts. It is likely that such algorithms will differ from those found to be optimal in non-software settings, given the inherent differences in constraints (e.g., the increased desirability of buses in software systems). In the domain of empirical application, we are assembling a database of historical versions of many different types of software products, both open source and proprietary. This will allow us to examine questions of whether software design structures differ according to category (e.g., infrastructure versus application) and test hypotheses about the effect of different organizational modes for development. Finally, we have begun to assess whether the aspects of design structure that we measure can predict product performance. For example, ongoing work uses our cost metrics assembled at the cluster and source file level to predict the *future* occurrence of defects ("bugs") in that part of the design.

With regard to the generalization of this work to non-software settings, several comments are relevant. First of all, there is a trend towards an increased software content in many physical products, hence the direct application of the techniques we describe are likely to prove increasingly valuable in other industries over time. Secondly, much of the design work involved in physical products now involves extensive computer aided design (CAD) techniques which capture design information in digital form – that is, as a set of values for a range of design parameters. To the extent that we can identify the means by which such design parameters interact (i.e., the analogy to software "function calls") we may be able to process CAD information in a manner similar to the way we handle software dependencies. Building the capability to perform such analyses will be costly, involving theoretical research and empirical studies such as this one, to explore the value and relevance of the outputs that these tools generate. However, we believe our early results demonstrate the significant potential that exists for this field of study.

References

Austin, S. A., A. N. Baldwin, 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, Clark, Design Rules: The Power of Modularity, MIT Press, 2000.

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, WP 3208, 1990.

Cai, Y. and K.J. Sullivan, Software Design Space Modelling and Analysis, University of Virginia Working Paper, 2004.

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

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

Dibona, C., S. Ockamn and M. Stone, "Open Sources: Voices from the Open Source revolution," O'Reilly and Associates, Sebastopol, CA, 1999.

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.

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.

Gutierrez, F, Integration Analysis of Product Architecture to Support Effective Team Co-location, M.I.T. SM Thesis, 1998.

Halstead, Gordon, Elshoff, On Software Physics and GM's PL.I Programs. GM Research Publication GMR-2175, General Motors Research Laboratories, Warren, MI, 1976.

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

Idicula, Planning for Concurrent Engineering, Nanyang Technological University SM Thesis, 1995.

Kusiak, Andrew , Larson, Nick and Wang, Juite, "Reengineering of Design and Manufacturing Processes," Computers and Industrial Engineering, Vol. 26, No. 3, 1994, pp. 521-536.

MacCormack, Alan, and Kerry Herman. "Red Hat and the Linux Revolution." Harvard Business School Case 600-009, 2000a.

MacCormack, Alan D., and Kerry Herman. "Microsoft Office 2000: Multimedia." Harvard Business School Multimedia Case 600-023, 2000b.

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

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

McCabe, T., A Complexity Measure. IEEE Transactions of SoftwareEngineering, Volume SE-2, No. 4, pp 308-320, Dec 1976.

Murphy, Notkin, Griswold, Lan, An Empirical Study of Static Call Graph Extractors, ACM Transactions on Software Engineering and Methodology (TOSEM), v7, n2, p158-191, 1998.

Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053-8, December 1972.

Parnas, Clements, and Weiss. The modular structure of complex systems. IEEE Transactions on Software Engineering, SE-11(3):259-66, March 1985.

Pimmler, Eppinger, Integration Analysis of Product Decompositions, Proceedings of the ASME Sixth International Conference on Design Theory and Methodology, Minneapolis, MN, Sept., 1994. Also, M.I.T. Traditional School of Management, Cambridge, MA, Working Paper no. 3690-94-MS, May 1994.

Pinkett, Randy, Product Development Process Modeling And Analysis Digital Wireless Telephones, S.M. Thesis (EE/CS), Massachusetts Institute of Technology, Cambridge, MA, 1998.

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

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

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

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

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

Steward, Donald 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.

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, Knowledge Management of System Interfaces and Interactions for Product Development Processes, M.I.T. SM Thesis, 2001.

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

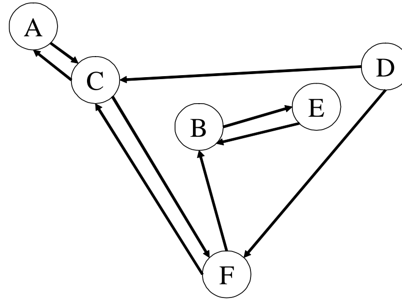
Von Hippel, E., and G. von Krogh, "Open Source Software and the Private-Collective Innovation Model: Issues for Organizational Science," Organization Science, Vol 14, No.2 Mar-Apr, 2003.

Warfield, Binary Matricies in System Modeling, IEEE Transactions on Systems, Management, and Cybernetics, v3, 1973.

Wilkie, Kitchenham, Coupling measures and change ripples in C++ application software, Journal of Systems and Software, v52, n2-3, 2000.

Yourdon, "Software Metrics: The Next Productivity Frontier", Computerworld Australia 1993.

Exhibit 1: Design Dependencies, a Design Structure Matrix and a Clustered DSM



	A	B	C	D	E	F
A	.		X			
B		.			X	
C	X		.			X
D			X	.		X
E		X			.	
F		X	X			.

	B	E	A	C	D	F
B	.	X				
E	X	.				
A			.	X		
C			X	.		X
D				X	.	X
F	X			X		.

Exhibit 2: The Directory Structure of Linux version 0.01

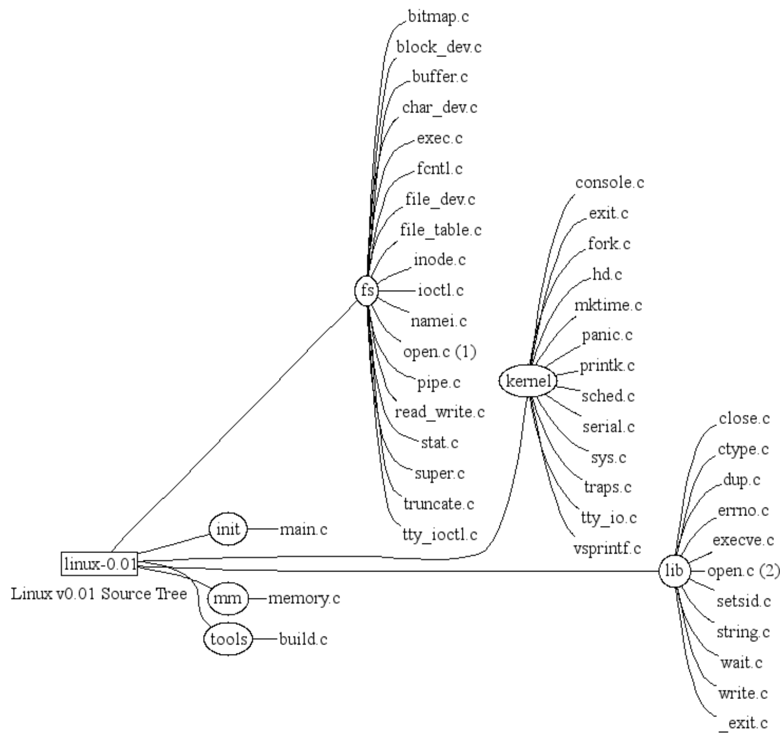


Exhibit 3: The Architectural View of Linux version 0.01

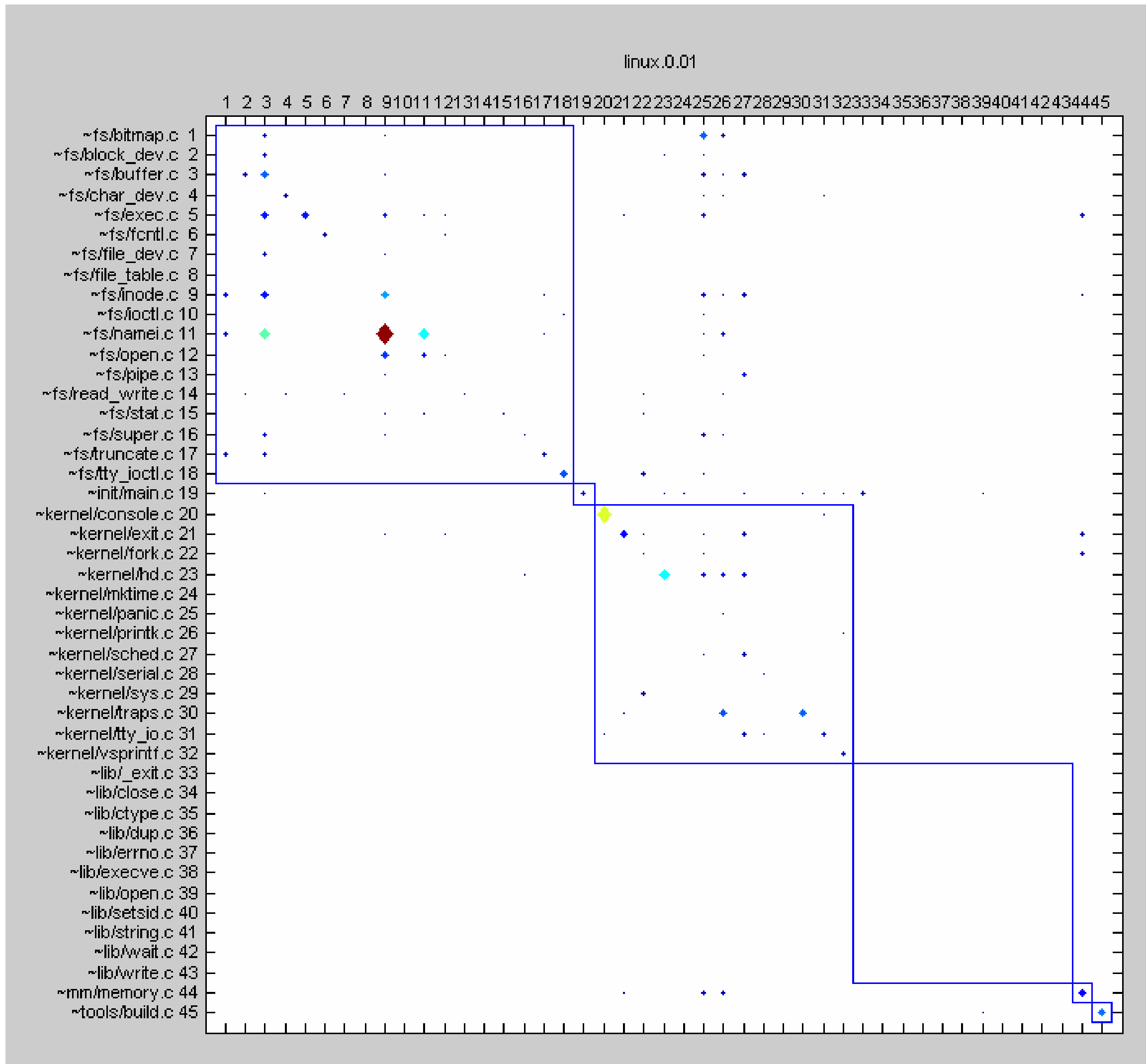
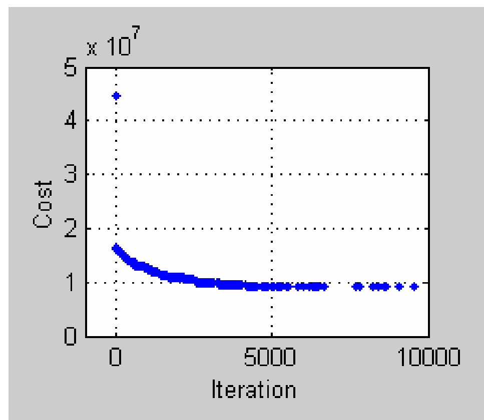
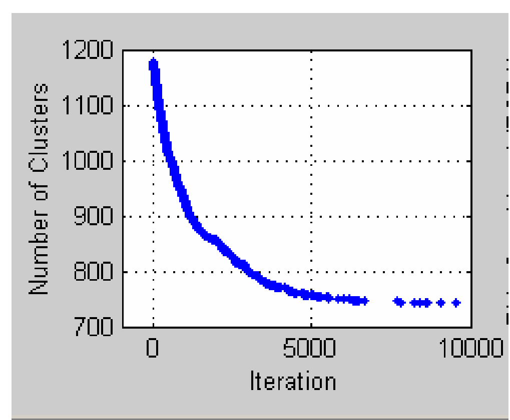


Exhibit 4: Illustration of Clustering Convergence for Linux version 2.1.51

4A: Total Coordination Cost per Iteration



4B: Number of Clusters per Iteration



4C: Number of Bids Considered per Successful Iteration

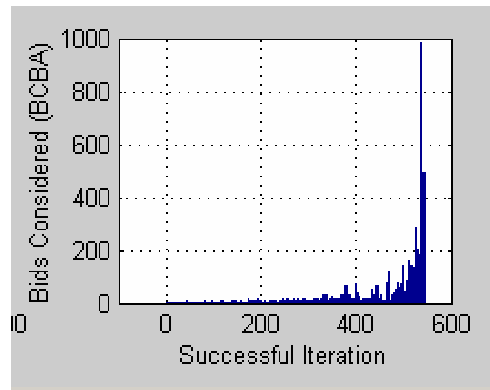


Exhibit 5: A Comparison of the earliest complete version of Mozilla and a comparable version of Linux.

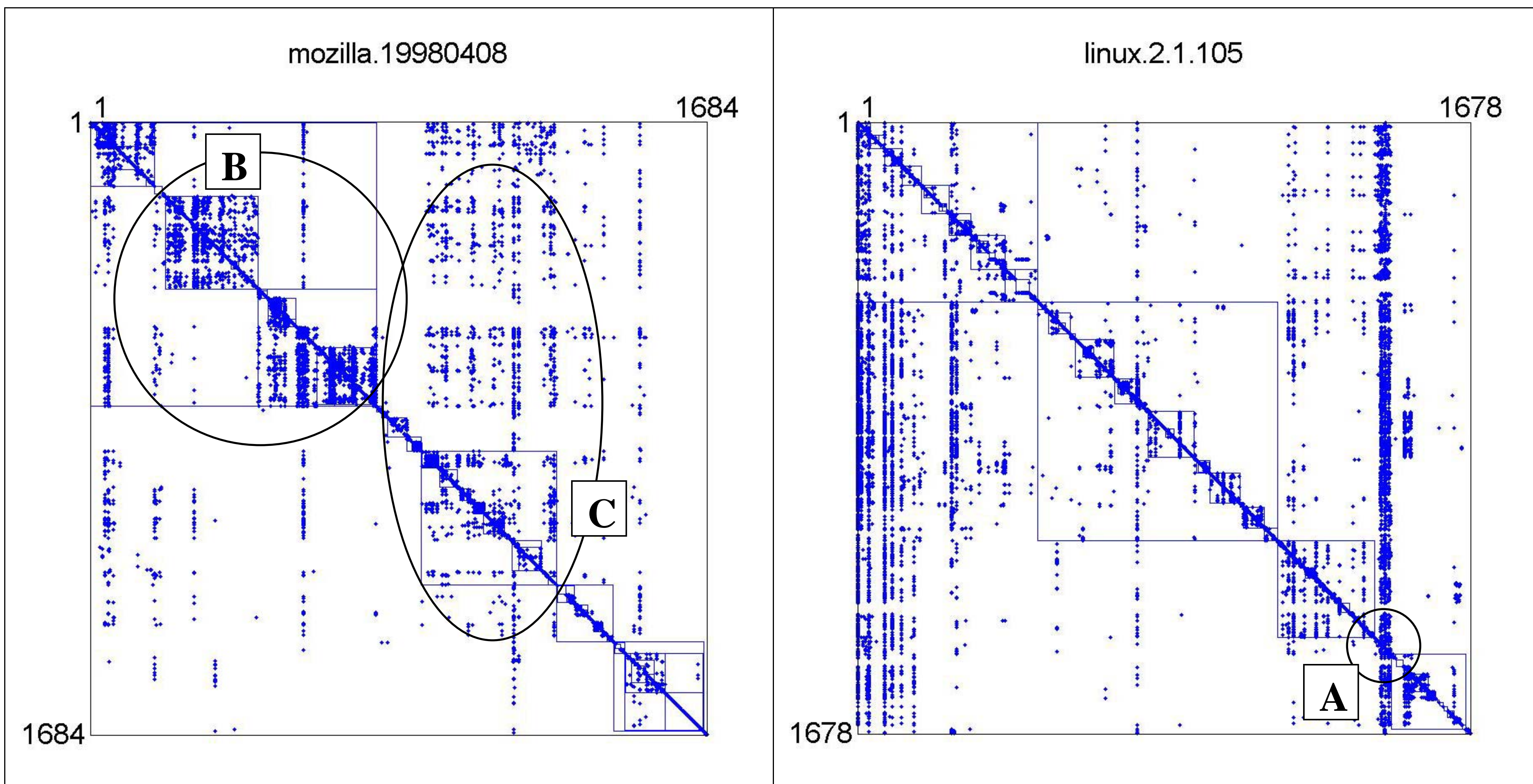


Exhibit 6: A Comparison of Mozilla before and after a single concentrated re-design effort.

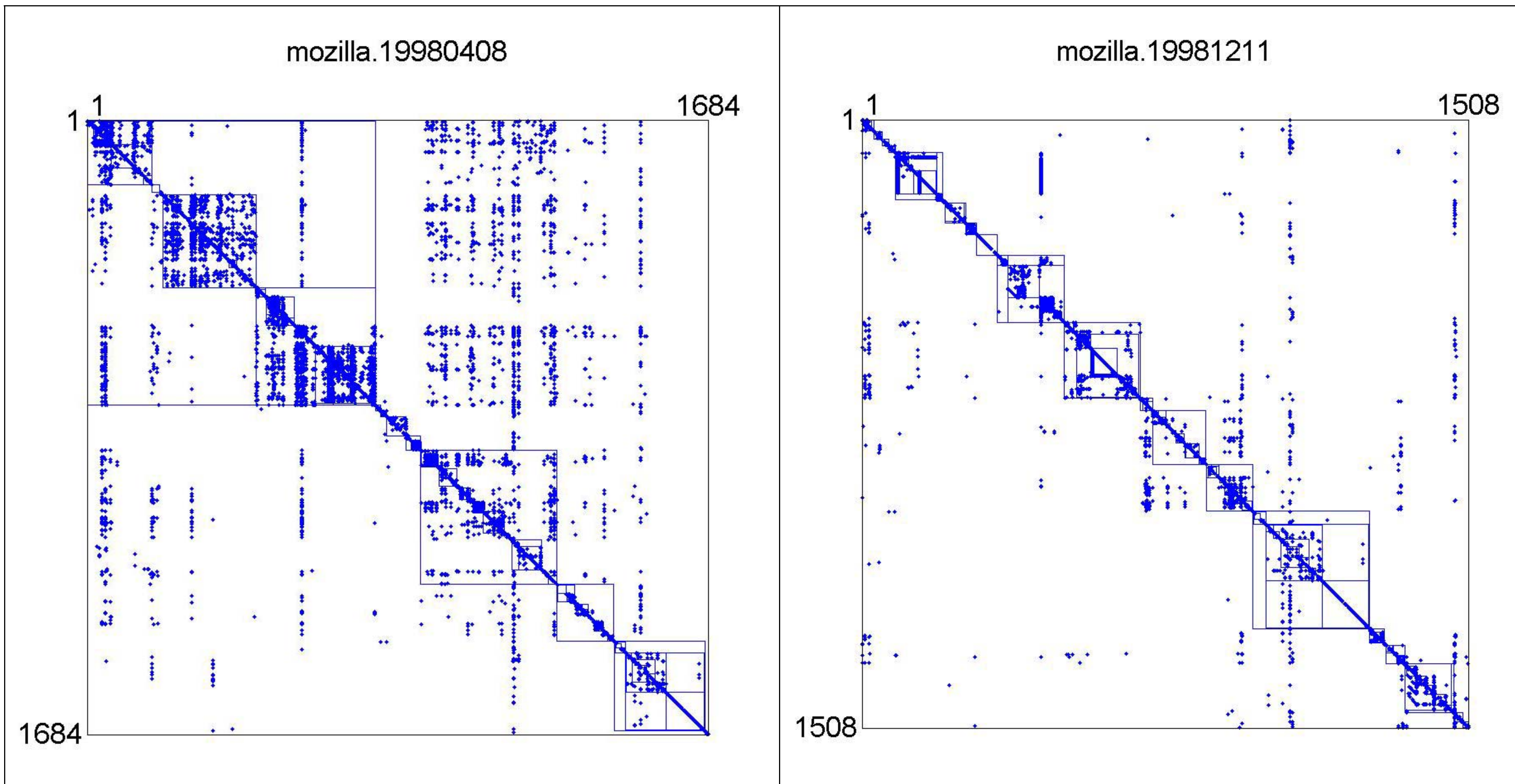


Exhibit 7: The Impact of a Re-design Effort on Mozilla's Change Cost

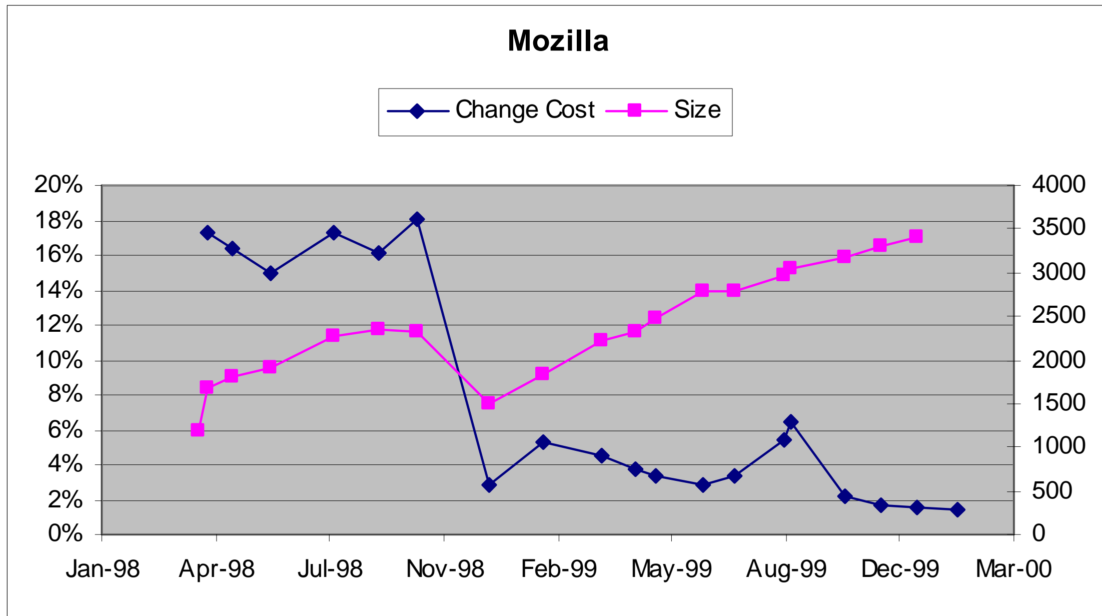


Exhibit 8: The Impact of a Re-design Effort on Mozilla's Coordination Cost