07-081

Evolution Analysis of Large-Scale Software Systems Using Design Structure Matrices & Design Rule Theory

Matthew J. LaMantia
Yuanfang Cai
Alan D. MacCormack
John Rusnak

Evolution Analysis of Large-Scale Software Systems Using Design Structure Matrices and Design Rule Theory

Matthew J. LaMantia
VMware, Inc.
lamantia@vmware.com

Yuanfang Cai
Drexel University
yfcai@cs.drexel.edu

Alan D. MacCormack,
John Rusnak
Harvard Business School
amaccormack@hbs.edu
jrusnak@hbs.edu

Abstract

Designers often seek modular architectures to better accommodate expected changes and to enable parallel development. However, we lack a formal theory and model of modularity and software evolution, which can be used for description, prediction, and prescription. According to Baldwin and Clark's [1] theory, modular architectures add value to system designs by creating options to improve the system by substituting or experimenting on individual modules. In this paper, we evaluate their theory by looking at the design evolution of two software product platforms through the modeling lens of design structure matrices (DSMs) and design rule theory. Our analysis shows that DSM models and options theory can explain how real-world modularization activities in one case allowed for different rates of evolution in different software modules and in another case conferred distinct strategic advantages on a firm (by permitting substitution of an at-risk software module without substantial change to the rest of the system). The experiment supports our hypothesis that these formal models and theory can account for important aspects of software design evolution in large-scale systems.

1. Introduction

Microsoft finally released Windows Vista, the new version of its Microsoft's operating system, after more than three years of delay [5], [19]. The trade press has popularly attributed the delays to a deepening complexity disaster resulting from the system's lack of modularity: "With each patch and enhancement, it became harder to strap new features onto the software, since new code could affect everything else in unpredictable ways" [6]. Similarly, Michael Cusumano has called Vista a "60-m-lines-of-code mess of spaghetti" [18].

Designers have long recognized the value of modularity. Constantine's low-coupling, high-cohesion principle has been well known since 1970's [14]. Parnas's famous information hiding criterion [11] similarly has remained influential for decades. Designers are educated to seek modular architectures to better accommodate expected changes and to enable parallel development.

However, because these principles are informal, their successful application depends on the designers' intuition and experience. Intuition and experience, in turn, do not prevent a big company, like Microsoft, from constantly grappling with unanticipated dependencies, modularity decay, and delays in bringing software to market. Thus we are in need of a formal theory and models of modularity and software evolution that can capture the essence of these important but informal design principles and provide the power of description, prediction and prescription.

Baldwin and Clark [1] proposed a theory to explain how modular architectures add value to system designs by creating options to improve the system by substituting or performing experiments on individual modules. Their theory is based on Steward's [15] design structure matrix (DSM) modeling approach (described below).

Baldwin and Clark also proposed that *design rules* can be used to resolve interdependencies and create modular architectures by specifying the interface between modules. Sullivan et al. [16] applied their design rule theory in Parnas's [11] small but canonical Key Word in Context (KWIC) design example. They showed that DSM modeling and the design rule theory can precisely capture Parnas's information hiding criterion. The KWIC experiment provides preliminary evidence of the model and theory's descriptive power.

In this paper, we evaluate the DSM model and design rule theory in large and complex software designs. We examine the design evolution of two software product platforms: (1) Tomcat, an open source web application server from the Apache Software Foundation; and (2) a proprietary application server from a company which remains anonymous. Both systems have been evolving for years. Their

designers have refactored the systems several times and released multiple versions.

We examine the evolution procedure and refactoring activities for these two platforms through the lens of DSM models and design rule theory. We hypothesize that the theory can explain how modularization confers strategic advantage on firms by allowing codebases to evolve in particular ways. The experiment supports our hypothesis: a theory based on DSMs, design rules and options has the power to explain formally phenomena related to the evolution of large-scale software systems.

The designers of the systems we examine made their decisions based on their own visions and prior experiences. They did not use DSM models, design rule theory or options analysis. However, our case studies imply that this set of analytic tools and methods can be used for both prediction and prescription. For example, given two refactoring proposals, designers might use DSMs, design rules and options analysis to quantitatively determine which is better. They can also assess whether the proposals will enable substitution of code that is strategically problematic (because of licensing terms, for example.). Finally, DSM models can be used to see whether *ex post* outcomes match the *ex ante* goals of a refactoring effort.

This paper is organized as follows: Section 2 introduces DSM modeling and Baldwin and Clark's design rule theory. Section 3 presents our evaluation methodology. Section 4 presents the case study of Tomcat. Section 5 presents the case study of the proprietary product platform. Section 6 discusses our results. Section 7 describes related work, and Section 8 concludes.

2. DSM Modeling and Design Rule Theory

This section introduces DSM modeling and explains how design rules decouple otherwise coupled design decisions, create options, and enable independent substitution. In the rest of the paper, we will refer to the formal analysis of design rules and options as "design rule theory."

The design structure matrix (DSM) was initially conceived by Steward [15], and later developed by Eppinger *et al.* [4] as means of modeling interactions between design variables of engineered systems. A DSM is a square matrix, in which each design variable corresponds both to a row and a column of the matrix. A cell is checked if and only if the design decision corresponding to its row depends on the design decision corresponding to the column. Figure 1 shows a simple DSM with three tasks.

	A	B	C
A		X	
B	X		
C	X		

Figure 1: A simple DSM with three design parameters, labeled A, B, and C. In this example, A depends on B. B depends on A. C also depends on A. No modules depend on C.

Building on DSM models, Baldwin and Clark proposed the notion of *design rules* as a means of decoupling otherwise coupled design decisions. Design rules specify the interface between modules, and appear at the left-hand side of the DSM. Figure 2 demonstrates the refactoring of the sample DSM to resolve a cyclical dependency. The design rule (DR) specifies an interface between design decisions A and B, such that, once the DR is introduced, A and B no longer depend on each other. Instead, both depend on DR. In other words, through the agency of design rules, A and B become *independent* modules. Baldwin and Clark define the behavior of introducing design rules that decouple two modules as the *Splitting* operator.

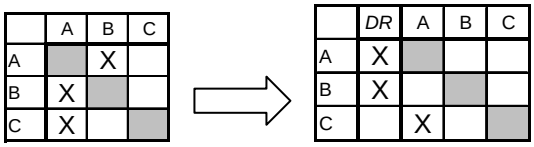

Figure 2: DSM transformation showing addition of a design rule (column "DR") which specifies an interface between A and B, thus resolving their mutual dependency

Given two modules, A and B, resulting from the Splitting operation, experiments on A and B may be performed independently. In other words, module A can be replaced with a better module with advantageous properties, such as higher performance or lower cost, without influencing B. Module B can be similarly substituted with a better version without disturbing A. The ability to select the best candidate for each module increases the value of the entire system. Baldwin and Clark define the behavior of exchanging an existing module for a new module with advantageous properties as the *Substitution* operator. In other words, each module creates an *option* (to substitute), which will only be exercised when substitution is advantageous. Increasing the number of modules increased the number of options, which (under well-defined assumptions) results in higher value for the system as a whole.[1]

[1] Baldwin and Clark proposed a collection of six "modular operators," which together can account for most design structure

The effect of these operations can be captured precisely by DSMs. The introduction of design rules (DR) can be modeled by the left-most columns of the DSM; the effect of splitting is reflected by the absence of dependencies between two modules, and by the fact that the two modules only depend on the design rules. Given two independent modules, substitution becomes possible within each module.

Sullivan *et al.* [16] used DSM models to precisely capture Parnas's [11] information hiding criterion, and used options analysis to show that the information-hiding design of KWIC generates a higher total value of the system. Their experiment provides preliminary evidence of the efficacy of DSM-cum-design-rules analysis based on small but canonical software examples.

We hypothesize that such analysis also has the potential to explain large-scale software evolution phenomena, for example, why some software platforms can survive after many years of evolution, but others can not, and whether a particular modularization effort, such as refactoring, is successful. The next section introduces our evaluation methodology.

3. Methodology

In this section, we introduce our overall evaluation methodology.

3.1 Two Software Systems

We examine two software systems: (1) Tomcat, an open source web application server from the Apache Software Foundation; and (2) a proprietary application server, which has been analyzed with the permission of the company that develops and sells it. We will refer to this company as "Company 2," and to its software system as "Server 2."

We chose the Tomcat server because it is a successful open source software system in which different parts of the system can be expanded or improved independently. Many major software platforms do not have this beneficial property. We hypothesize that DSM modeling and design rule theory can shed light on the properties of this successful, evolvable large-scale software system.

We chose Server 2 because the first author witnessed and participated in a strategic refactoring that addressed a real problem in a commercial software company. We hypothesize that DSM modeling and design rule theory can show formally what the refactoring accomplished and how it benefited the company.

The systems are both web application servers, which implement all or part of the Java 2 Enterprise Edition (J2EE) specification. Server 2 is a much larger software system than Tomcat, by almost an order of magnitude. Tomcat implements only a portion of the J2EE specification, while Server 2 implements the entirety. Server 2 also includes many application "framework" components, which serve as a platform for Company 2's entire product family.

Using methods described below, we analyzed multiple versions of each system to study their evolutionary properties. All versions studied were production releases, and are therefore known to be stably functioning versions of the software. We examined these two software systems using a DSM model based on source code dependencies. Thus our investigation uses static software analysis to extract the dependency relations within software source code.

3.2 Dependency Extraction

Both systems are Java-based projects, and we used an open-source tool, *Dependency Finder*, written by Jean Tessier [17], to extract code dependencies. The basic unit of analysis for our investigation is the Java class. We examined the following kinds of class-to-class dependencies:

- If class *A* is a subclass of *B*, then *A* depends on *B*. The parent class is necessary to compile its children.
- If any portion of class *A* makes explicit reference to *B* as a variable, then it also depends on *B*.
- If a function in class *A* calls or makes reference to a function or data member of class *B*, then *A* depends on *B*.

Java classes are grouped together into "packages." A Java package is a collection of classes which together implement a larger unit of related functionality. The packages are named hierarchically, with each portion of the package name progressively narrowing the scope of the code contained in it. For example, software from the Apache Foundation is contained within other packages starting with "org.apache," and the core functionality of version 3.0 of the Apache Tomcat server is contained in the subpackage "org.apache.tomcat." These class-to-class dependencies are then aggregated hierarchically into package-to-package dependencies.

---

transformations. *Splitting* and *Substitution* are the two most important operators.

3.3 DSM Generation

We now use DSMs to represent the software dependency relationships. Each row and each column of the DSM corresponds to a class, and each dependency is denoted by a mark in the row corresponding to the dependent class and the column corresponding to the depended-upon class. All of the DSMs contained in this paper were rendered using DSAS, the Design Structure Analysis System, created by John Rusnak [12].

The DSMs depicted for the two systems contain several hundred to several thousand classes, and we use black dots to show class dependencies. To make them more comprehensible, classes in the same package are delineated within a square. Packages in the same parent package are surrounded with another square, and so on, to create a hierarchical view. As results are presented, relevant portions of the DSM will be labeled with the subsystems that they represent.

Authors such as Steward [15] and Eppinger *et al.* [4] have observed the importance of sequence in DSM representations of task structure, where marks above the diagonal represent iterative cycles. Similarly, if the elements of a software DSM are ordered so that, wherever possible, a class follows the classes on which it depends, then marks above the diagonal will represent cyclic dependencies.

DSMs in this paper that are labeled as sorted have been reordered in a way that strictly preserves the hierarchical structure of the packages and dependencies, but also minimizes marks above the diagonal. DSMs sorted in this way can help to reveal layered software design structures, as well as cyclical dependencies.

In order to evaluate whether substitution has occurred in a module, the following metric, *architectural change ratio* is used. It is a coarse metric, which is based on the number of classes added or removed from a module between two release versions:

$$changeRatio(version_i \rightarrow version_j) = \frac{(newClassCount_j) + (removedClassCount_j)}{totalClassCount_i}$$

That is to say, the change ratio is simply the sum of the number of new classes added and the number of classes removed, divided by the number of classes in the previous version of the module. This metric captures changes in the class structure, but not code changes within the classes themselves.

The next two sections report our experimental results, showing how DSM modeling and design rule theory can precisely explain software evolution and modularization activities.

4. Tomcat Case Study

We first consider the Apache Tomcat project. Tomcat underwent a change of project structure from commercial to open-source development in 1999. Subsequent to its open-source transition, the Tomcat codebase was partially rewritten, and again "refactored" – redesigned to create a cleaner, more efficient architecture – in its next major release.

We studied five versions of Tomcat: from v3.0, the first open-source version of the server, to v5.0.28. We modeled each version using a DSM, and computed the change ratio from the previous version. Figure 3 shows the DSM generated for Tomcat 3.0.

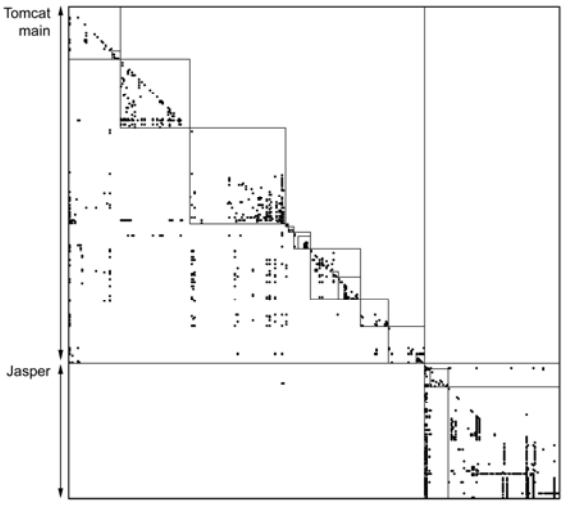

Figure 3: DSM for Tomcat 3.0, showing two independent modules. The DSM is sorted to show module hierarchy

Upon examination of the DSM, it is immediately clear that there are two major and distinct functional modules in the codebase, corresponding to the Tomcat server core ("Tomcat-main"), and a separate module, named Jasper, which processes Java Server Pages.

The fact that there are two distinct functional modules in turn enables a desirable property of the architecture: the two modules can evolve separately. This fact can be shown quantitatively, by observing rates of change of the two modules across successive versions of the code.

Table 1 shows the change ratios of Tomcat-main and Jasper for each version examined relative to the previous version. It shows that from version 3.0 to version 3.3.1, and again from version 3.3.1 to version

4.0, Tomcat-main was almost entirely rewritten or rearchitected (change ratio >= 1.0), while the Jasper module underwent only minor changes (change ratio = 0.2). Across multiple versions of the product, each of the two modules experienced at least one redesign, *but these occurred at different points in time*. Thus the change metrics reveal that there was a different rate of experimentation in the two modules.

version	v3.3.1	v4.0	v4.1.31	v5.0.28
Tomcat-main change ratio	1.0	1.1	0.5	0.4
Jasper change ratio	0.2	0.2	0.5	0.5

Table 1: Architectural change ratios of the Tomcat-main and Jasper module across versions of the software.

The DSM in Figure 3 leaves the impression that Tomcat-main and Jasper are almost totally separated systems because they are only connected at two points. This is not the case, however. The two modules belong to the same system and are not two pieces of unrelated software.

Direct examination of the Tomcat source code shows that the points of connection in the DSM are only calls to a utility function. The real interface between the two chunks of code is defined within the J2EE Servlet API, the specification to which Tomcat conforms. Thus the initial DSM, derived from the source code of Tomcat alone, does not depict the complete relationship between these two modules.

However, we can extend the DSM by adding the J2EE interface, thereby obtaining the larger DSM shown in Figure 4. From this DSM, we observe that the interface between these modules can be considered a "design rule," in the sense proposed by Baldwin and Clark [1]. First, it defines a basic specification, to which both modules must conform: this can be seen from the presence of dependencies in the columns of the J2EE Servlet API and the rows of Tomcat-main and Jasper. As long as both modules conform to this interface, they can interoperate. Second, the interface does not itself depend on either Tomcat-main or Jasper: this is apparent from the *absence* of dependencies in the columns of the two modules and the rows of the Servlet API. Finally, the Tomcat-main and Jasper modules are effectively independent, each depending only on the interface design rule (except for two calls to a utility program).

It is important to note that the existence of these design rules was not immediately observable at the outset. We first observed the highly modularized structure through the DSM shown in Figure 3, and then discovered the key enabler of this desirable characteristic, the J2EE Servlet API as design rules. In many cases, design rules are implicit, such as a data structure agreed among modules (as exemplified by Parnas's KIWC sequential design [11]). Without a formal model like DSM, their existence and their important roles are difficult to recognize.

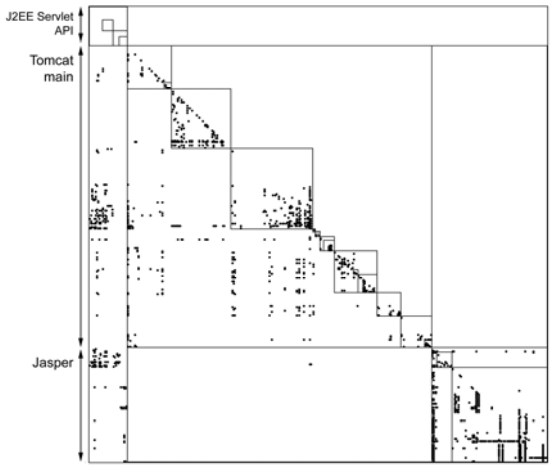

Figure 4 The extended design structure of Tomcat 3.0, showing the Servlet API classes as design rules. The DSM is sorted to show module hierarchy

The two distinctly decoupled modules, as shown in the DSMs, allow for the asynchronous evolution of the two modules. This is an example of the *Substitution* operator proposed by Baldwin and Clark [1] at work in the software domain. Subsequent to Tomcat's donation to the Apache Software Foundation, and its transformation into an open-source project, Apache members redesigned and rewrote the Tomcat-main module. This branch, initially named "Catalina," competed with the older version of Tomcat, and Apache members contributing to the Tomcat project voted to select Catalina as the new primary version of Tomcat (v4.0). In this process, Jasper was only slightly changed.

It is difficult to assess the exact reasons why one version was selected over the other (and, indeed, different members may have chosen the new version for different reasons). However, we can infer from the process itself, and from the result that a new architecture for Tomcat-main was selected, that some advantage was conferred by substituting a new version of the Tomcat-main module (v4.0) for the older version (v3.x). In other words, there was inherent value in the option to substitute at the module level.

Had the two modules, Tomcat-main and Jasper, been tightly coupled by strong code dependencies, changes in one would necessarily have forced changes in the other. Any substitution would then have involved both modules, and been inherently more

difficult (and perhaps more contentious). In this case, the ability conferred by the code architecture to substitute a software module was useful in the context of open-source software development.

The evolution of parts independently of the whole is a critical property of a well-modularized architecture. DSM modeling can reveal whether a complex system is well modularized and explicitly reveal the design rules that enable the separation. Conversely, if the DSM model of a system does not appear to have distinct modules, it may be desirable for architects of the system to further modularize it by identifying additional design rules, and applying the splitting operator.

5. Server 2 Case Study

In the second case study, we examine a closed-source, commercially available product. Company 2's product line is a family of web-based applications – software applications that run on a server and allow user interaction through web pages. Examples of web-based applications include bulletin-board systems that allow users to post and read messages; travel sites that allow users to make and view reservations; commerce sites with "shopping cart" functionality; or any web site that integrates information stored on other systems such as databases. (These examples do not necessarily correspond to Company 2's actual product offerings.)

Company 2's applications are based on its product platform. The platform in turn consists of (1) a J2EE Application Server; (2) an application framework, which implements basic services used by all of its applications; and (3) a business logic engine, which implements more advanced services also used broadly within the product family. Figure 5 shows a module block diagram of this structure. The platform components are shown at the bottom of the diagram, with the applications sitting on top. Arrows indicate dependency relationships: the applications depend on the platform components, and the higher-level platform components depend on the server/framework component at the bottom.

Our analysis focuses on the server/framework platform component, upon which the entire product family depends. This component is a J2EE-compliant application server. If that were its only role, then another, third-party, J2EE-compliant application server could be substituted for this portion of the software. However, the server/framework component also contains framework elements—basic services that the whole product platform and family rely on for functionality. Therefore, a commercially-available third-party application server cannot be substituted wholesale for the server/framework component.

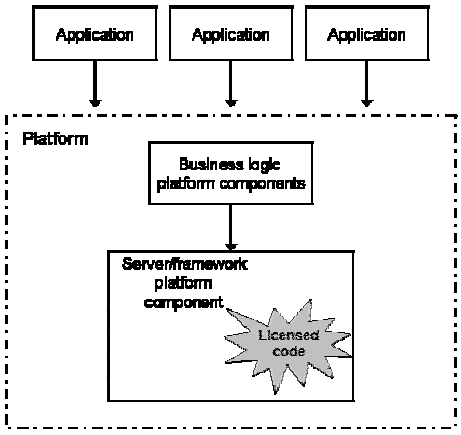

Figure 5: Block architectural diagram of Company 2's product family, including both platform and application components.

We model the original dependency structure of Server 2 at the time of our case study using the DSM shown in **Figure 6**. From the DSM, we observe a large block of highly entangled classes. The highly coupled structure is partially due to the following reason: the server/framework platform component contained code that Company 2 licensed from another vendor, and the licensed code was spread throughout the codebase and could not be readily separated from the rest of the platform.

This situation created distinct strategic risks for Company 2. Upon expiration of the license agreement, the licensor could prohibit Company 2 from releasing new versions of its software containing the licensed code. Or it could raise the price of the license, thereby creating a classic "holdup" scenario. Because the licensed code was intertwined with Company 2's product family, such events could place Company 2's entire product family, revenue, and profitability at risk!

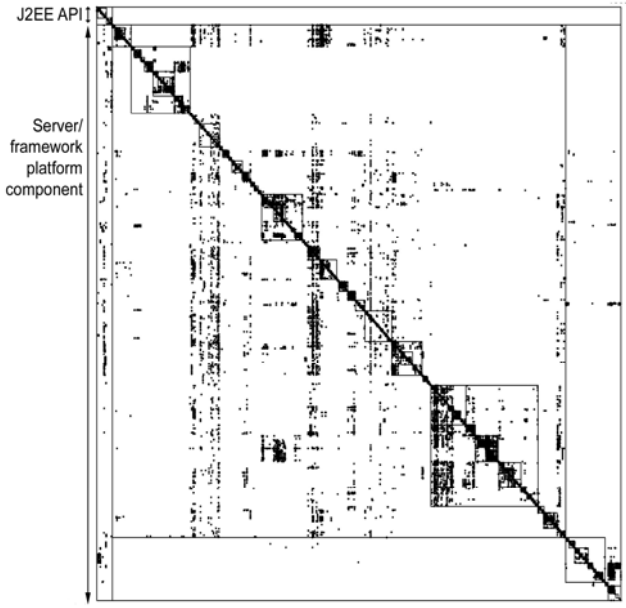

Figure 6: DSM of Server 2, before splitting. Note that the server/framework component is primarily composed of a single, large module. Licensed components are spread throughout this module.

5.1. Splitting and Substitution

To address this problem, the company performed a limited restructuring of the server/framework platform component. The design goal of the restructuring was to isolate the licensed code into a separate module, for which a different third-party software product could be substituted at a later date. As envisioned, this substitution could be performed by Company 2 or by customers in the field. The secondary goals of the restructuring effort were to separate the licensed code with minimal engineering effort, minimal code changes, and minimal technical risk.

In order to achieve these goals, engineers first determined what code was subject to license restrictions. This set of Java classes is denoted by L, the licensed code.

L = {code under license}

The set L had to be separated from the rest of the codebase.

The engineers also identified all Java classes that required the licensed code, a set denoted by R_L.

R_L = {all classes that require some class in L}

However, all of R_L could not simply be split off, because some of it was also required for the rest of the platform and/or applications.

Once R_L had been identified, the classes in R_L were individually examined by a group of engineers, who used their knowledge of the platform and applications to decide what should and should not be excluded from the platform. Any code that was required by other platform and application components could not be excluded from the platform, and thus had to be separated from the licensed code.[2]

In this fashion, a cleaving line was determined that split the server/framework platform component into two separate modules. The code was then rewritten to eliminate dependencies that violated the constraints of separation.

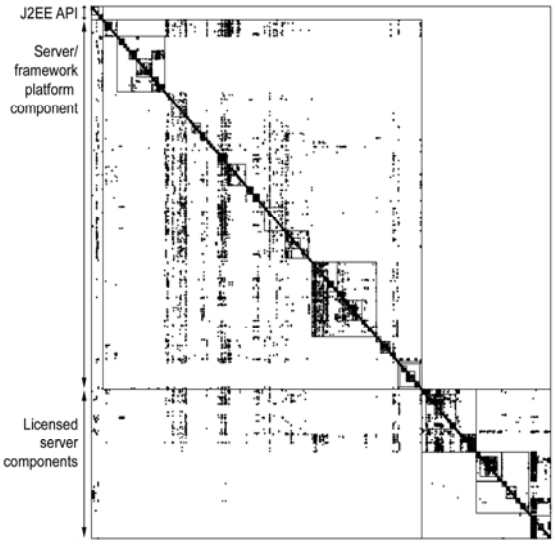

Figure 7: DSM of Server 2, after splitting. Components under third-party license have been separated into a new top-level module. The licensed components depend on company components, but no company component depends on the licensed components.

Figure 7 shows the DSM of the component after splitting. Following the engineering work to resolve the problematic dependencies, the server/framework component was indeed separated into two separate blocks (modules): a new server/framework component and licensed server components. Furthermore, no element in the new server/framework component depended on any part of the licensed code. (This is evident from the absence of dependencies in the upper rectangular quadrant of the DSM.)

In the lower left part of the DSM, we observe that there are significant dependences from the licensed code to the platform components, indicating that the

[2] In this case it was most expedient for engineers to make the determination simply by examining the list. However, the operation could also be performed using formal dependency analysis. Please refer to LaMantia [8] for details.

platform component cannot be substituted independently. These dependencies exist because the licensed server components have evolved interdependently with Company 2's own code, and it was not necessary to resolve these dependencies in order to make the licensed code module substitutable. This outcome was acceptable because only the licensed code module was at risk.

Now that no portion of Company 2's platform or applications depends on the licensed server components module, another implementation can be substituted for it, as long as the substitute module conforms to the underlying design rules which specify the interface between the product family and the licensed code module. In this case, the design rules are defined by the J2EE API module. After the licensed code is replaced by a third-party product, the dependencies in the lower left part of the DSM disappear, and both platform components and the third-party product become independent modules.

Thus a third-party J2EE application server can be used for this purpose, and, in fact, Company 2 supports this configuration today. This substitution scenario is illustrated by the block diagram in **Figure 8**. The company's product platform is unchanged, but the licensed server components are acquired from a third-party J2EE-compliant server:

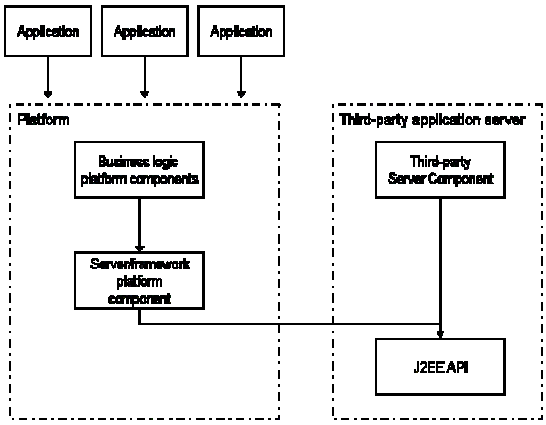

Figure 8: Block diagram of the restructured Company 2 product family, in which a third-party product has been substituted for licensed components previously contained in the platform.

5.2. Strategic value of design structure

Like the Tomcat example, the case of Company 2 illustrates how the splitting of software into modules can facilitate substitution. However, in this case, the split was deliberately engineered to obtain a specific strategic benefit, namely to protect the company's product platform against the loss of licensed components.

There is a difference between the design structure of Tomcat and the modified Company 2 product platform: Whereas the two major modules of Tomcat were effectively independent, relying only on the underlying specification as design rules, Company 2's platform structure is layered. The newly separated "licensed server components" module relies on the platform core. This is acceptable because the component that needs to be substitutable is the dependent module. Nothing else in the product family depends directly upon it. Once a third-party product is plugged in, the dependencies are removed and the two modules become independent.

While this restructuring addressed the problem of having licensed code in the platform, it did not address the vulnerability of the platform to changes in the J2EE specification itself. The entire product family depends, directly or indirectly, on the J2EE specification classes. In this sense, the J2EE specification is a design rule for the entire product family. Company 2 does not control the specification, thus changes in it may expose the product platform to strategic risk.

From the DSM in **Figure 7**, we can still observe a large entangled block of the server/framework platform component, which suggests the opportunity and necessity of further modularization, by splitting and substitution, to mitigate strategic risk.

6. Discussion

In both the Tomcat and Company 2 case studies, the DSM models reveal a key characteristic of modular architectures: the design rules must be explicitly defined so that otherwise dependent modules can be decoupled (splitting) and each independent module can thus be replaced with a better version (substitution) without unwanted perturbations.

For a well modularized system, like Tomcat, the DSM shows the loose coupling characteristic of subsystems by aligning modules as blocks along diagonals, and by the absence of dependences across blocks. By studying the change ratios of multiple versions along its evolutionary path (Table 1), we observed that this architecture has enabled different rates of experimentation in different subsystems, and that the key enabler of this highly flexible architecture is the design rules embodied in the J2EE specification.

The case of Company 2 additionally demonstrates a real-world scenario where the modular partitioning of a software product platform had concrete and quantifiable strategic value. The company significantly

reduced its strategic risk by deliberately splitting elements of a codebase and eliminating problematic dependencies. Comparing the DSMs before and after the refactoring of Server 2, we can see that the platform no longer depends on the licensed code. This means that licensed code can be replaced without jeopardizing the functionality of the company's own code, suggesting that refactoring has succeeded.

The DSM of Server 2 also reveals that rest of the system is still not well-modularized. With a theory of design rules in mind, we can come up with splitting and substitution strategies for further improvement. For example, the new server/framework platform still depends on the J2EE specification. In the future, it might be advantageous to separate all of the application framework and logic that does not require the J2EE specification into its own layered hierarchy of modules.

Although the modularization decisions in these two systems were based on the designer's experiences and intuitions, by using DSMs we were able to capture, both formally and precisely, what the designers did that enabled the substitution of independent modules. Hence the two cases suggest that DSMs and design rule theory can be used as tools for understanding and describing existing software product architectures, and for finding ways to improve them. The patterns observed here in turn have the potential to serve as a model for the deliberate creation of software architectures that enable asynchronous evolution and substitution of modules. Furthermore, they suggest directions to advance our notions of how specific modular decompositions and dependency structures can bring specific strategic advantages to the enterprise.

7. Related Work

Parnas [11] introduced the fundamental concepts which define software modularity. He proposes the principle of "information hiding" as the basis for decomposing software into modules, and defines a module as "a responsibility assignment rather than a subprogram." The essence of information hiding, he said, is to hide the design decisions that are likely to change, and to make the modules communicate only through interfaces.

Sullivan *et al.* [16] applied DSM modeling and design rule theory to Parnas's canonical example, showing that Baldwin and Clark's [1] approach could be used to visualize and formalize Parnas' theory. The dependencies are visualized in a DSM; the interfaces are formalized as design rules; the modules create options; and the risky (volatile) part of the system should be isolated in separate modules to obtain higher option value. Lopes *et al.* [8] later employed similar methods to compare aspect-oriented design vs. object-oriented design.

Sangal *et al.* [13] used a commercial static analysis tool to recover dependency models from source code for the purpose of discovering and communicating software architecture. Rusnak and MacCormack *et al.* [10] used the DSM modeling techniques to compare two complex software systems, the Linux kernel and the Mozilla web browser.

In contrast to prior work, our experiments show how design rules appear as structures in the DSMs of actual codebases. We also characterize the key properties for a system to be adaptive, and explain how splitting and substitution can be enabled by proactively inserting design rules and isolating parts of the system with high risk. By extracting DSMs before and after a modularization activity, we can formally confirm if the activity is successful.

The DSM modeling approach is general enough to model decisions not only in source code, but also in the specification and design stages of codebase development. Cai's [2][3] recent work concentrates on modeling design decisions and dependencies that span the software lifecycle using augmented constraint networks and automatically generating DSMs from logic models. This work shows that the DSM model has the potential to bridge the gap between design and implementation modularity by enabling conformance checking between the two.

8. Conclusion

Important software modularity principles, such as the information hiding criterion, have remained informal. DSM modeling and Baldwin and Clark's design rule theory have the potential to formally account for how design rules create options in the form of independent modules and enable independent substitution.

This paper evaluated the applicability of the model and theory to real-world large-scale software designs by studying the evolution of two complex software platforms through the lens of DSMs and design rule theory. The results showed that (1) DSM models can precisely capture key characteristics of software architecture by revealing independent modules, design rules, and the parts of a system that are not well modularized; (2) design rule theory can formally explain why some software systems are more adaptable, and how a modularization activity, such as refactoring, conveys strategic advantages to a company.

DSM modeling and design rule theory are general enough to model decisions other than those encoded in source code. Having shown the descriptive capability of these techniques, we envision that this approach also has the power of prediction and prescription. For example, designers can use DSM models proactively to design the architecture of a system or to plan a modularization (refactoring) that will increase the system's option value. After the system is built or the refactoring concluded, they can use DSMs extracted from actual source code to check whether the initial architecture or the modularization plan was successful. [7].

9. References

[1] Baldwin, C. Y. and Clark, K. B.. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

[2] Cai, Y. "Modularity in Design: Formal Modeling and Automated Analysis," PhD thesis, University of Virginia, Aug. 2006.

[3] Cai, Y. and Sullivan, K. "Simon: A tool for logical design space modeling and analysis," in *20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, California, USA, Nov 2005.

[4] Eppinger, S. D. "Model-based approaches to managing concurrent engineering," *Journal of Engineering Design*, 2(4):283–290, 1991.

[5] Fried, I. "Vista debut hits a delay." *CNet News.com* (21 March, 2006). 29 April, 2006. <http://news.com.com/2100-1016_3-6052270.html>.

[6] Guth, R. A. "Code Red: Battling Google, Microsoft Changes How It Builds Software." *The Wall Street Journal*, (23 September, 2005): A1. Factiva, MIT Libraries, Cambridge, MA. 25 April, 2006. <http://global.factiva.com>.

[7] Huynh, S. and Cai, Y. "An Evolutionary Approach to Software Modularity Analysis." To appear in the fifth ICSE Workshop on Software Quality (WoSQ 2007), Minneapolis, MN, May 22, 2007.

[8] LaMantia, M. J. "Dependency Models as a Basis for Analyzing Software Product Platform Modularity: A Case Study in Strategic Software Design Rationalization," M.S. thesis, Massachusetts Institute of Technology, 2006.

[9] Lopes, C. V. and Bajracharya, S. K.. "An analysis of modularity in aspect oriented design," in *AOSD '05*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[10] MacCormack, A., Rusnak, J., and Baldwin, C. Y.. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code" *Management Science*, to be published.

[11] Parnas, D. L.. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, 15(12):1053–8, Dec. 1972.

[12] Rusnak, J. The Design Structure Analysis System: A Tool to Analyze Software Architecture, Ph.D. thesis, Harvard University, 2005.

[13] Sangal, N., Jordan, E., Sinha, V., and Jackson, D.. "Using dependency models to manage complex software architecture," in *OOPLSA*, 2005.

[14] Stevens, W. P., Myers, G. J., and Constantine, L. L. "Structured design." *IBM Systems Journal*, 13(2):115–39, 1974.

[15] Steward, D. V. "The Design Structure System: A Method for Managing the Design of Complex Systems." *IEEE Transactions on Engineering Management* EM-28.3: 71-74, August, 1981.

[16] Sullivan, K. Cai, Y., Hallen, B, and Griswold, W. G. "The structure and value of modularity in software design." *SIGSOFT Software Engineering Notes*, 26(5):99–108, Sept. 2001.

[17] Tessier, J. *Dependency Finder*. 4 April, 2006. <http://depfind.sourceforge.net>.

[18] Waters, R.. "Obstacles stand in the way of a cultural shift." *Financial Times* 17 November, 2005: 17. Factiva, MIT Libraries, Cambridge, MA. 25 April, 2006. <http://global.factiva.com>.

[19] "Windows Vista Late – But Why?" *Cnet.com.* (22 March, 2006). 29 April, 2006. <http://crave.cnet.co.uk/desktops/0,39029426,49258743,00.htm>