Analyzing the Evolution of Large-Scale Software Systems using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases

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 have long recognized the value of modularity, but important software modularity principles have remained informal. According to Baldwin and Clark's [1] design rule theory (DRT) , 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 examine the design evolution of two software product platforms through the modeling lens of DRT and design structure matrices (DSMs). We show that DSM models and DRT precisely 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. Our results provide positive evidence that DSM and DRT can inform important aspects of large-scale software structure and evolution, having the potential to guide software architecture design activities.

1. Introduction

Designers have long recognized the value of modularity. Constantine's low-coupling, high-cohesion principle has been well known since the 1970's [17]. Parnas's information hiding criterion [12] 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 intuition and experience. Intuition and experience, in turn, do not prevent a big company from constantly grappling with unanticipated dependencies, modularity decay, and delays in bringing software to market. The significant delay of Windows Vista exemplifies the case: Even a firm with deep expertise in software development, like Microsoft, can still suffer from a complexity disaster resulting from a system's lack of modularity [6].

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.

In recent work, Baldwin and Clark [1] propose a theory to explain how modular architectures add value to system designs by creating options to improve the system by substituting or experimenting on individual modules. Their theory explains the relationship between the modular structure of the IBM 360 computer and its subsequent successful evolution, but it has not been applied to large-scale software systems.

Baldwin and Clark's theory, which is based on Steward's [17] design structure matrix (DSM) modeling approach (described below), argues that *design rules* can be used to resolve interdependencies and create modular architectures by specifying the interface between modules. Sullivan et al. [18] applied this approach to Parnas's [13] small but canonical Key Word in Context (KWIC) design example. They show that DSM models and design rule theory can precisely capture Parnas's information hiding criterion.

To further explore the theory's descriptive power for large and complex software systems, we examine the evolution of two software product platforms through the lens of DSM models and design rule theory: (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.

Our case studies show that a theory based on DSMs, design rules and options precisely explains how modularization conferred strategic advantage on firms by allowing codebases to evolve in particular ways. They provide positive evidence that the model and theory have the power to formally explain phenomena

0-7695-3092-3/08 $25.00 © 2008 IEEE
DOI 10.1109/WICSA.2008.49

related to the evolution of large-scale software systems. The designers of the systems we examine made their decisions based on their visions and prior experiences. We find that design rule theory and DSM models help to formalize their previously informal intuitions and judgments, while also revealing parts of the system that may need to be further modularized.

This paper is organized as follows: Section 2 introduces DSM modeling and Baldwin and Clark's design rule theory. Section 3 presents the methodology of our case study and the research questions we aim to answer. Section 4 presents the case study of Tomcat. Section 5 presents the case study of the proprietary product platform. Section 6 discusses the implications of our results. Section 7 presents our ongoing and future work. Section 8 describes related work, and Section 9 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" (DRT).

The design structure matrix (DSM) was initially conceived by Steward [17], and later developed by Eppinger *et al.* [5] 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. A DSM represents modules as blocks along the diagonal.

The left DSM in Figure 1 shows a simple DSM with three modules. A and B model software design decisions that depend on each other, for example, two procedures that call each other, two classes that refer to each other, or a data structure and algorithm that go hand in hand. The designers usually need to break the cyclical dependency so that A and B can be changed without affecting each other. In contrast, the dependency between C and A models a layered architecture. C refers to A, but A does not refer to C.

Building on DSM models, Baldwin and Clark proposed the notion of *design rules* (DRs) as a means of decoupling otherwise coupled design decisions. Design rules in software are stable design decisions that hide the details of other components. Examples of design rules include: Application Programming Interfaces (APIs) that decouple an application from a library; Open Database Connectivity (ODBC) standards that decouple databases from their clients; a data format agreed among development teams [13]; or even naming conventions [19].

DSM modeling can capture the existence of design rules and their decoupling effects. Figure 1 demonstrates this refactoring activity. The design rule (DR) models 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.

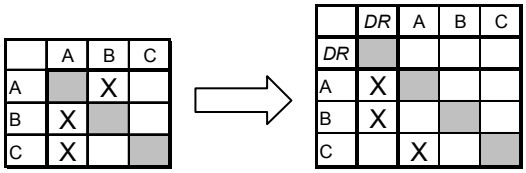

Figure 1: DSM transformation showing addition of a design rule (column "DR") which specifies an interface between A and B, thus resolving their mutual dependency

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 better 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 increases the number of options, which (under well-defined assumptions) results in higher value for the system as a whole.[1]

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. We argue that such

[1] Baldwin and Clark proposed a collection of six "modular operators," which together can account for most design structure transformations. *Splitting* and *Substitution* are the two most important operators.

analysis has the potential to explain large-scale software evolution phenomena, for example, why some software platforms survive after many years of evolution, but others do not; or whether a particular modularization effort, such as refactoring, is successful.

3. Methodology

In this section, we introduce the two systems we use as case studies, the methodology of deriving dependencies and DSM models, and the research questions we aim to answer by these case studies.

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. Because they are associated with our second case study, 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 DR 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 DSM models 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 [20], 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.

3.3 DSM Generation

We 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. The DSMs were rendered using a tool called DSAS, the Design Structure Analysis System, developed in prior work by Rusnak [14].

The DSMs 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.

In order to evaluate whether substitution has occurred in a module, the 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.

In the case studies presented in this paper, we only considered functional/structural coupling caused by syntactic references. There certainly exist other sorts of dependencies, such as information coupling and implicit assumptions. DSM modeling is general enough to express various types of dependencies [18][3]. In this paper, however, since we derive DSMs from source code, only syntactic dependencies are extracted. Note that we equate design decisions with structural elements. That is, we generally view classes, packages, and functions as design decisions, so that we can uniformly represent and analyze dependency structures.

3.4 Research Questions

In the next two sections, we present two case studies to answer the following research questions:

1. Can the DSM models sufficiently reveal the modular structure of these systems?
2. Can the design rules and their decoupling effects be manifested?
3. Can the design rule theory and DSM modeling explain software evolution phenomena, and justify the modularization activities taken by the designers during the evolution process?
4. Can the modeling and theory provide additional insights beyond the designers' intuition?

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 2 shows the DSM generated for Tomcat 3.0. The two major and distinct functional modules correspond to the Tomcat server core ("Tomcat-main") and a separate module, named Jasper, which processes Java Server Pages. The DSM is sorted and reordered in a way that strictly preserves package hierarchy, while minimizing the number of dependencies above the diagonal. DSMs sorted in this way help to reveal layered structures, as well as cyclic dependencies.

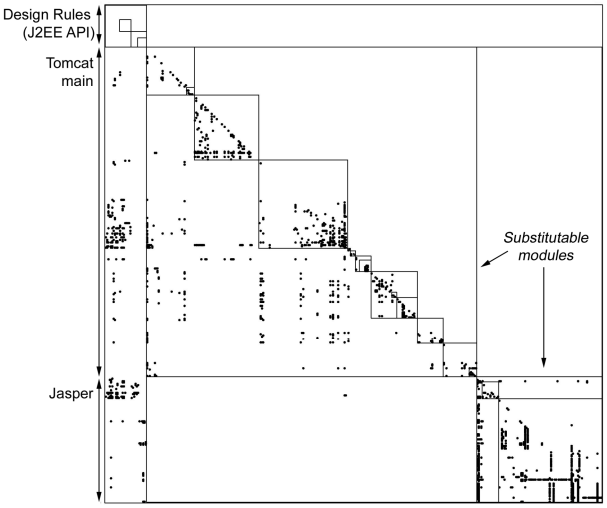

Figure 2: The extended design structure of Tomcat 3.0, showing the Servlet API classes as design rules. The DSM is sorted to reveal module hierarchy.

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 across successive versions of the code.

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 different versions.

Being able to evolve independently is a desirable property. Each module can be substituted with a better version without affecting other modules. The DSM model precisely reveals the key features of the architecture that enable this property: First, the two functional modules, Tomcat main and

Jasper appear to be almost totally separated because they are only connected at two points. The loose coupling between these modules means that changing the classes in one module will have little effects on the other module, enabling their independent evolution. Second, the interface between the two chunks of code, making them an integrated system, is the J2EE Servlet API, the specification to which Tomcat conforms. From this DSM, we observe that the interface between the two 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 *both* 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 function). The DSM model thus precisely reveals the existence of certain key design rules and their important *Splitting* effects.

Having two distinctly decoupled modules allows for the asynchronous evolution of the two modules. And it facilitates the use of the *Substitution* operator proposed by Baldwin and Clark [1]. This dynamic is highlighted by a significant event in the evolution of the design. In particular, 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. Eventually, 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. We have shown that DSM modeling can reveal whether a complex system is well modularized and highlight the design rules that enable the separation. Conversely, if the DSM of a system does not appear to have distinct modules, it may be desirable for its architects 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.)

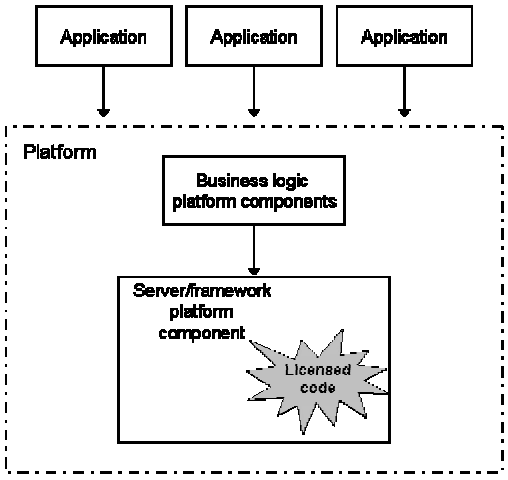

Figure 3: Block architectural diagram of Company 2's product family, including both platform and application components.

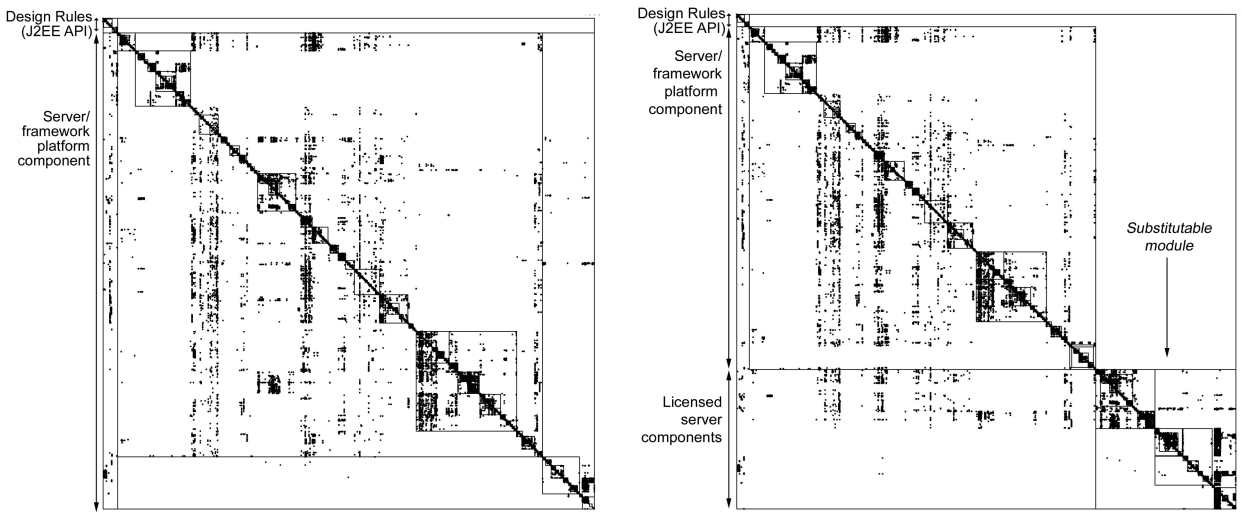

Figure 4: DSMs of Server 2, before and after splitting. Before splitting (left) the server/framework component is primarily composed of a single, large module. Licensed components are spread throughout this module. After splitting (right) the licensed components have been separated into a new top-level module. Some licensed components depend on company components, but no company components depend on licensed components. The module thus may be substituted.

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 3 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 original server/framework component also contained framework elements—basic services upon which the whole product platform and family relied for functionality. Therefore, a commercially-available third-party application server could not be substituted wholesale for the server/framework component. The server/framework platform component also contained code that Company 2 licensed from another vendor.

We modeled the dependency structure of the original server/framework component. The resulting DSM is shown in Figure 4 (left). From the DSM, we observe a large block of highly entangled classes. The DSM also revealed that licensed code was spread throughout the codebase, hence 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!

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 its 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}

The set 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. 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 modified to eliminate dependencies that violated the constraints of separation. The total effort expended, including both analysis and code modification, was approximately two person-months.

Figure 4 (right) shows the DSM after splitting. Following the engineering work to resolve the problematic dependencies, the server/framework component was separated into two separate blocks (modules): a new server/framework component and licensed server components. 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 right 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 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."

After splitting, no portion of Company 2's platform or applications depends on the licensed server components module, so 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 5. The company's product platform is unchanged, but the licensed server components are acquired from a third-party J2EE-compliant server.

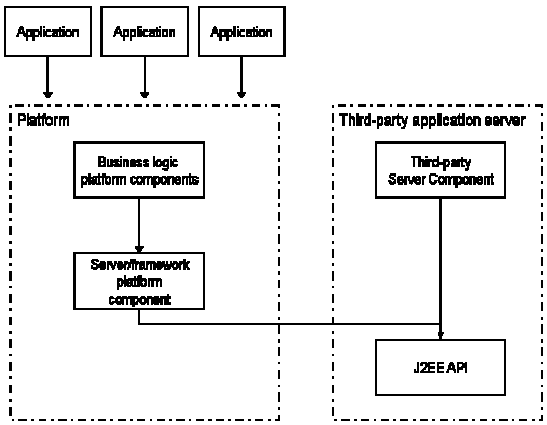

Figure 5: Block diagram of the restructured Company 2 product family, in which a third-party product has been substituted for licensed components previously contained in the platform.

5.2. Strategic value of design structure

The case of Company 2 illustrates how the splitting of software into modules can facilitate substitution. In contrast to Tomcat, however, 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 also 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.

[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.

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.

In summary, the DSM model clearly shows the architecture before and after refactoring, revealing how the value of the system can be improved by independent substitution. The DSM model also reveals other strategic risks of Company 2's platform:

First, 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 a 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.

Second, from the DSM of Server 2 post split in Figure 4, we still observe a large block of code with many interdependencies between elements in the server/framework platform component. This block may be amenable to further re-factoring efforts, in order to produce a clearly layered hierarchy like the one observed in the Tomcat-main module (i.e., a hierarchy with no cyclic dependencies). Such a structure would make the resulting code more adaptable to future changes, by facilitating further splitting or substitution of sub-modules. In sum, the DSM model both helps to explain historical patterns of software evolution, but may also yield insights into potential improvements that are not readily apparent to the designer.

6. Experimental Results

Our results provide positive answers to the research questions proposed in Section 3.

The expressiveness of DSM modeling. In both the Tomcat and Company 2 case studies, the DSM model and DR theory precisely described the modular structure of each version of the system. 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.

The decoupling effects of design rules. In both systems, the DSM models make the existence of key API modules, viewed as design rules, explicit and reveal how the design rules decouple otherwise dependent modules (splitting) allowing each independent module to be replaced with a better version (substitution) without unwanted perturbations.

The explanation of software evolution. The model and the theory clearly explained how the modular structure of software architecture can create strategic value for the system. By studying the change ratios of multiple versions along the evolutionary path for Tomcat (Table 1), we observed that this architecture enabled different rates of experimentation in different subsystems. The key enabler of this highly flexible architecture was identified as the design rules embodied in the J2EE specification. The case of Company 2 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 into separate modules and eliminating problematic dependencies. DSM models and DR theory thus help to formally explain the decisions made by designers based on their informal intuitions and experiences.

The generation of design insight. The DSM of Server 2 post split may be further analyzed to reveal new opportunities for modularization. With a theory of design rules in mind, there may be additional splitting and substitution strategies that can improve the design. 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

In sum, the model and theory suggest new directions to advance our notions of how specific modular decompositions and dependency structures can bring strategic advantages to an enterprise.

7. Ongoing and Future Work

So far we have demonstrated the descriptive power of DSM modeling and DR theory on as-built systems after the dependencies are established, showing that it can uncover the modularity properties of the software and aid in the evolution of software. Our ongoing work suggests the possibility of applying DSM and DTR before coding, and we envision future work of trying it out as an architecture modeling technique integrated with normal software development process.

The patterns observed in these case studies have the potential to serve as a model for the deliberate creation of architectures that enable asynchronous evolution and substitution of modules. For example:

(1) Given two refactoring proposals, designers might use the model and the theory to quantitatively determine which is better.

(2) The designer can use DSM models to reveal the part of the design or source code that needs to be refactored, that is, the module that appears to be a huge block with a lot of cyclical dependencies.

(3) Given a big module that needs to be refactored, the designer can deliberately identify design rules that

are unlikely to change and decouple the tangled system.

(4) By revealing the software architecture before and after refactoring, as in the Company 2 study, the designer can check whether *ex post* outcomes match the *ex ante* goals of a refactoring effort.

Our recent work confirms the predictive potential of DSM models and DRT to aid practitioners. Cai [4] has applied the theory and model to the development process of a real project. In this study, they emphasized the key role of "design rules". For example, design rules were identified early in the development stage after the architecture was designed and modeled using UML sequence diagrams, which enabled the team to discover poorly-modularized design structures and refactor the design prior to coding. DSMs were used to explicitly model and check both the design structure before coding and the source code structure after implementation. Similar analysis was conducted as the system evolved, ensuring a well-modularized implementation conforming to the design.

In ongoing research, two of the authors are working with a large Indian outsourcing firm to understand how DSMs and DR theory can help to improve the conduct of projects. As part of this work, DSM models were recently used to display the output from a project and identify possible opportunities for design refactoring. Source files with problematic dependency structures were identified by their influence on the overall level of coupling in the system. These files, in turn, were independently verified by the project team as being a potential source of design problems. The results have convinced the firm to apply these techniques further.

8. Related Work

Parnas [13] introduced the fundamental concepts which define software modularity. He proposed the principle of "information hiding" as the basis for decomposing software into modules, and defined a module as "a responsibility assignment rather than a subprogram." The essence of information hiding is to hide design decisions that are likely to change, and to make modules communicate through interfaces.

Sullivan *et al.* [18] 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. They also used options analysis to show that the information- hiding design of KWIC generates a higher total value of the system. Lopes *et al.* [11] later employed similar methods to compare aspect-oriented design vs. object-oriented design. Sullivan et al. [19] recently developed a concept of XPI, a special form of design rules that decouple aspect code with the base code based on DSM modeling and DR theory.

Sangal *et al.* [15] used a commercial static analysis tool to recover dependency models from source code for the purpose of discovering and communicating software architecture. Rusnak [13] and MacCormack *et al.* [12] used the DSM modeling techniques to compare two complex software systems, the Linux kernel and the Mozilla web browser.

In contrast to this work and other refactoring techniques, such as Lakos's work [9], 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 inserting design rules and isolating parts of the system with high risk. By extracting DSMs before and after a modularization, we can formally confirm whether 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 development. Cai's [2][3] recent work focuses on modeling design decisions and dependencies that span the software lifecycle using *augmented constraint networks* (ACN) and automatically generating DSMs from logic models. The work shows that DSMs have the potential to bridge the gap between design and implementation modularity, enabling conformance checking between the two.

8. Conclusion

Important software modularity principles, such as the information hiding criterion, have remained informal. DSM models and DR theory have the potential to formally account for how design rules create options in the form of independent modules to enable substitution. This paper represents a first step in this direction.

We present two case studies which apply the model and theory to real-world large-scale software designs, studying the evolution of two complex software systems through the lens of DSMs and DR theory. The results showed that (1) DSM models can precisely capture key characteristics of software architecture by revealing independent modules, design rules, and parts of a system that are not well modularized; (2) DR theory can formally explain why some software systems are more adaptable, and how ex-ante modularization or ex-post refactoring can bring strategic advantages to a company.

DSM models and DR theory are general enough to model decisions other than those encoded in source

code. Having shown the descriptive capability of these techniques, we believe 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) activity 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 [8].

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] Cai, Y. and , Huynh, H., "Logic-Based Software Project Decomposition. " Drexel TR DU-CS-07-02.

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

[6] 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>.

[7] 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>.

[8] 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.

[9] Lakos, J. "Large-Scale C++ Software Design" Addison-Wesley Professional, July, 2006.

[10] 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.

[11] 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.

[12] 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,* July, 2006.

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

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

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

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

[17] Steward, D. V. "The Design Structure System: A Method for Managing the Design of Complex Systems." *IEEE Trans. on Eng. Mgmt* EM-28.3: 71-74, Aug, 1981.

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

[19] Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. "*Information Hiding Interfaces for Aspect-oriented Design.*" *SIGSOFT* ESEC/FSE-13. Page 166-175. September 2005.

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