Dependency Models as a Basis for Analyzing Software Product Platform Modularity: A Case Study in Strategic Software Design Rationalization

by
Matthew J. LaMantia

Submitted to the System Design and Management Program
In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 2006

Signature of Author

Matthew J. LaMantia
System Design and Management Program
June 2006

Certified By

Professor James M. Utterback
Thesis Supervisor
David J. McGrath jr. Professor of Management and Innovation and Professor of Engineering Systems

Certified By

Patrick Hale
Director
System Design and Management Program

Abstract

It is broadly accepted among software managers and architects that maintaining the integrity of software designs is important for the long-term health and viability of software product platforms. The use of modular, information-hiding architectures is considered beneficial in this regard, because they enable the addition of new features and the correction of software defects without widespread changes (Parnas, 1972; Parnas, 1978). Moreover, modular architectures in general add value to system designs by creating options to improve the system by substituting or performing experiments on individual modules (Baldwin and Clark, 2000). Recent research has sought to clarify and to define formally these notions of modularity, and their value in the software domain, though the use of software dependency models and software design structure matrices. These models provide a graphic representation of the relationships between the building blocks of large software systems, which can be used to aid visual understanding of their structure, and which also form the basis for quantitative metrics for evaluating the degree of modularity of a software system (MacCormack *et al.*, 2005; Sangal *et al.*, 2005; Sullivan *et al.*, 2006). The goal of this thesis is to contribute to the development and validation of formal models of modularity and value, by examining the design evolution of two similar software systems through the lens of software-dependency-based design structure matrices. We find that the design structure matrix model demonstrates a form of information-hiding modularity that allows different rates of experiment in different software modules, and that also allows substitution of a software module without substantial change to the rest of the software system. Moreover, the cases demonstrate that such a substitution scenario can confer distinct strategic advantages to a firm. Based on these results, we suggest that software managers and architects explore modular architectures that localize areas of risk – technical and otherwise – in software modules with well-defined interfaces.

Acknowledgements

I am indebted to many members of the academic community, without whose help it would have been impossible for me to complete – or even to begin – this thesis.

I wish to thank Professor Carliss Baldwin, Dr. John Rusnak, and Professor Alan MacCormack of the Harvard Business School, whose paper on the design structure of the Mozilla web browser initially sparked my interest in this topic. They have been extremely generous in sharing their time, insights, resources, and especially Dr. Rusnak's software tools, which made this analysis possible.

At the Massachusetts Institute of Technology, Professor James Utterback, who advised this thesis, provided invaluable counsel on how to approach the whole endeavor, and directed me to the people who could help me begin. Those people included Dr. Karim Lakhani, Stefano Mazzocchi, and Professor Marc Meyer of the Northeastern College of Business Administration.

I am grateful to my fellow graduate students who helped, inspired, and occasionally prodded me to complete this thesis, especially David Schiller and Erica Fuchs.

Throughout my time at MIT, my classmates and the administration of the System Design and Management program have been extremely supportive. Thanks to Pat Hale, Helen Trimble, Ted Hoppe, and William Foley.

My former colleagues, Charles Morehead and Adam Belmont, provided invaluable assistance in obtaining data necessary for the execution of this analysis.

Finally, I wish to thank my family and friends for their patience and support: Charles, Ann, and Elise LaMantia; Steven Allen Hoey; Marilyn Fontenrose; Daniel Lavoie; and Patrick Lamerson.

Table of Contents

Table of Figures

Chapter 1: Introduction

Longhorn is late.

At the time of this writing, Windows Vista – the next version of its Microsoft's operating system that was originally code-named Longhorn – is scheduled to be released in early 2007, more than three years after its initial delivery date in the fourth quarter of 2003 (Fried, 2006; Cnet.com, 2006).

The trade press has popularly attributed these 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" (Guth, 2005). Similarly, Michael Cusumano has called Vista a "60-m-lines-of-code mess of spaghetti" (quoted by Waters, 2005).

Microsoft Windows is not a single product. It is available in configurations for home use, professional use, media applications, and servers. It is a product family, and at its core is a software product platform. Windows happens to be an operating system, but a software product platform is not necessarily an operating system. Rather, it is the software component, or set of components, that forms the basis for the architecture and functionality of a software product family (Meyer and Selinger, 1998).

The story of the Windows delays highlights the ways in which a lack of platform modularity can have grievous consequences for product families. Conversely, authors such as Meyer and Webb (2005) have documented how a modular architecture can

facilitate platform renewal, and the continued health of software product platforms and the product families that rely upon them.

Software modularity

Parnas (1972) introduced the fundamental concepts which define software modularity. He proposes the principle of "information hiding" as the basis for decomposing software into modules. He notes that modular decomposition is a design decision, which must be completed before work can begin on actual implementation. The modular design defines an implicit division of labor (or at least a divisibility of labor): "a 'module' is considered to be a responsibility assignment rather than a subprogram."

Modalities of decomposition should be judged by their ability to accrue the benefits of modular design, which include allowing separate people or groups to work on components, thus minimizing development time; increasing flexibility by allowing a single module to be modified without worrying about the whole program; and making programs easier to design and understand, because a each module may be examined and understood apart from the others.

In addition, Parnas (1978) argues that the structural design of software determines its ability to be changed. He introduces the notion of program hierarchy, which establishes a partial ordering among modules according to a 'uses' relationship. In this paradigm, the process of changing software by expanding or contracting it can be viewed as creating additional families of software modules that are supersets or subsets of the initial family, or as substituting some modules in the family. The critical role of the quality of the

initial decomposition in making this possible highlights the need to anticipate potential changes during the planning process.

Software Design Structure Matrices

Although Parnas' notions of modularity are broadly accepted as principles of good software architecture, they have largely been used as informal guidelines which guide software architects' intuitions. The application of Design Structure Matrices to software is an effort to create a formal model as a basis for analyzing software structure, and for quantifying the value created by these structures.

The Design Structure Matrix (DSM) was initially conceived by Steward (1981) as means of understanding interactions between design parameters of engineered systems. Eppinger *et al.* (1994) extended this model to capture more deeply the correspondence between design structure and task structure in product development.

A design parameter, in this context, is essentially a subproblem or task of a design, which must be solved in the final detailed design. A DSM is a square matrix, in which each design parameter corresponds both to a row and a column of the matrix. Squares in the matrix have a binary value, either checked or empty. A square is checked if and only if the task corresponding to its row depends on the task 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.

Baldwin and Clark (2000) integrate DSMs, Parnas' notion of information-hiding modularity, and the notion of real options from finance theory, to create a general framework for value in modular product design. They propose that modular architectures create value by creating options that allow the designer to choose, at a later date, to change or improve submodules without redesigning the whole system.

Baldwin and Clark also propose the notion of *design rules* as a means of resolving interdependencies to create modular architectures. 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.

	A	B	C
A		X	
B	X		
C	X		

	DR	A	B	C
A	X			
B	X			
C		X		

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.

They propose a collection of six "modular operators," which together form a complete set of design structure transformations. Two of these will be of particular import to this thesis: *Splitting* is the act of separating one module into two. *Substitution* is the act of exchanging an existing module for a new module with advantageous properties, such as higher performance or lower cost.

Based on this foundation, several authors have recently applied the concept of design structures to software. Sullivan *et al.* (2001) apply DSMs to the same classic software design problem that was originally used by Parnas to illustrate the concept of information-hiding modularity, demonstrating how this framework can both formalize and visualize Parnas' concept in the software domain. They also offer an extension to the DSM model, the *environment and design structure matrix* (EDSM), in which external entities on which the design parameters depend, such as the computer operating system or the nature of user interaction, are explicitly added at the left-hand side of the matrix.

Sullivan *et al.* further demonstrate that the modular decomposition creates option value. They simulate the results of experiments which refine the value of individual modules within two architectures which address the same software problem. The architecture based on the principle of information-hiding modularity shows a greater increase in overall technical performance, and thus system value. This demonstration is thus a preliminary validation of Baldwin and Clark's real options model in the software domain. However, the authors note that the problem is a self-contained example, "divorced from real markets."

Sangal *et al.* (2005) illustrate the use of software DSMs as a means of discovering and communicating software architecture. They propose that software dependencies may be used to manage the architectural integrity of software in ongoing development, though the definition and enforcement of rules constraining dependency relationships of software modules. They further illustrate this approach through the use of a commercial software DSM tool to analyze two medium-sized open source projects.

Rusnak (2005) and MacCormack *et al.* (2005) have studied the relationship between software architecture and team structure. MacCormack *et al.* apply the DSM model as a means of comparing two complex software systems, the Linux kernel and the Mozilla web browser. They define several metrics based on this model for comparing the degree of modularity of design structures. This thesis follows closely upon their methods, which will be discussed in detail in later chapters.

Research goals

The research goal of this thesis is to contribute to the development and validation of formal models of modularity and value, by examining two case studies of real-world software systems through the lens of software-dependency-based Design Structure Matrices. Does the model speak to the value-creating processes in the evolution of these software systems? What extensions or refinements to the model are suggested by the cases? Finally, what insights does the model offer for software managers and architects?

This thesis begins, in Chapter Two, with a detailed description of the model and associated methods. Chapters Three and Four present the two studies, which apply the model to two similar software systems, one open source and one proprietary. Chapter

Five presents a discussion of the results of the case studies, the model itself, and their implications for software managers and architects.

Two appendices are attached. The first presents some computational and algorithmic details of the model and its implementation, and the second presents the source code developed in support of this investigation.

Chapter 2: Methodology

Two distinct server implementations

This thesis will examine two software systems, both written in the Java programming language. The first of these systems is Tomcat, an open source web application server from the Apache Software Foundation. The second of these systems is a proprietary application server, which has been analyzed for this thesis with the permission of the company that develops and sells it. In the interest of protecting the company's strategic interests, it will remain anonymous. We will refer to this company as "Company 2," and to its software system as "Server 2."

In the interest of full disclosure, it must be stated that the author of this thesis was previously employed by Company 2 and has performed firsthand some of the restructuring of Server 2 that is described in this thesis. The system was chosen as a case for examination because it was known to illustrate an example of the use of dependency information to address a strategic problem in a commercial software company.

The Tomcat server was chosen because it is an open-source software system that, like Server 2, functions as a web application server. Because it is open source, it is available for analysis without any special access or permission. In addition, MacCormack *et al.* (2005) have conjectured that software projects with highly distributed collaboration may require modular architectures.

The J2EE Standard

The systems examined in this chapter are web application servers, which implement all or part of the Java 2 Enterprise Edition (J2EE) specification. J2EE is a standard suite of technologies for creating dynamic web-based applications. It is an open standard, to the extent that it is defined though the Java Community Process (JCP), in which groups of industry experts discuss and define the specification standards. However, the intellectual property associated with the specification, including the application programming interfaces and the J2EE brand, is owned by Sun, and licensed to the groups which implement it in their software products (Byous, 2003).

The J2EE suite includes technologies for creating large-scale software applications, using Java technology. It includes technologies for presenting data and user interfaces within a web browser, and for many services which support such applications. In order for an application server product to receive the J2EE brand, the product must fully pass the J2EE Compatibility Test Suite, consisting of several thousand separate test scenarios, which verify that the software system complies with the behavior described in the specification (Sun Microsystems, 2003).

The J2EE standard is defined both by textual specifications and by the J2EE Application Programming Interface classes (J2EE API). The integration of the J2EE API as a software component in compliant servers facilitates the interoperability of compliant servers.

There are several distinct implementations of the J2EE specification in existence: At the time of this writing, the Sun website lists 23 for J2EE version 1.3, and 12 for J2EE

version 1.4 (Sun Microsystems, 2006a; Sun Microsystems, 2006b). Some of these projects are open-source software.

Because the systems have at least some nominally identical behavior, they provide a way to compare design structures while minimizing the effect of problem domain on the design. However, this congruency is highly imperfect. While J2EE servers all implement the same basic functionality, they also offer additional functionality, not required or described by the J2EE specification, to add value and differentiate themselves from competing products. Such functionality might include administrative and management services; security mechanisms; support for running multiple servers in tandem on separate machines in a "server farm;" and functionality which acts as a framework or product platform for vertical applications built on the server platform.

It must also be stated that Server 2 is a much larger software system than Tomcat, by almost an order of magnitude. Whereas Tomcat only implements a portion of the J2EE specification, Server 2 implements the entire specification. Server 2 also includes many application "framework" components, which serve as a platform for Company 2's entire product family.

Versions of the software examined

All of the software versions examined in this investigation were production releases of the software, and are therefore known to be stably functioning versions of the software.

Source code dependencies

This thesis will examine these two software systems using a DSM model based on their source code dependencies. Before describing this model in detail, it is important define the dependency relationship, and how the data is obtained.

The basic notion of software dependency was defined by Parnas (1978) as a "uses" relationship. A software module *A* or entity uses, or depends on, *B* just in case *A* "requires the presence of a correct version of" *B*.

In practice, there are two kinds of software dependencies: Static, or "compile-time," dependencies and dynamic, or "run-time," dependencies.

Static dependencies capture the notion that one software module is necessary in order to compile another. In other words, *A* depends on *B* just in case the source code of *A* makes an explicit reference to *B*. In such a case, in order to compile *A* to an executable form, it is necessary to have access to *B*.

Run-time dependencies, in contrast, are based on actual calling patterns of the software during operation, and vary according its use and deployment. They cannot, in general, be inferred from any *a priori* examination of the software code. For this reason, this investigation focuses on analyzing static software dependencies.

Dependency extraction

Tools used

For the Java-based software projects examined in this thesis, an open-source tool, *Dependency Finder*[1], written by Jean Tessier, was used to extract code dependencies. In Java, it is possible to extract dependencies from the executable, packaged product, so it is not necessary to have access to source code in order to get the necessary data. This was an important factor in gaining access to data about the proprietary server product examined in this investigation.

Unit of analysis

In Java, all software functions and code are encapsulated within classes. A class is a basic unit of code which corresponds to an entity or unit of software functionality within the software system.

For this reason, the basic unit of analysis for this investigation is the Java class. This is roughly, but not exactly, equivalent to the use of the source code file as the basic unit of analysis by MacCormack *et al.*[2] Typically, one Java class is defined in each source code file. However, it is possible, on occasion, to define an "inner" class, which appears inside another Java class. In such cases, two or more Java classes may be defined in the same source file.

---

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

[2] Unlike Java, in C++, not all functions are encapsulated within classes; it is possible to organize functions simply by containing them in files. For this reason, the author proposes that classes and files are appropriate and correspondent units of analysis for Java and C++, respectively.

For the purpose of this investigation, all of the dependency data available from the Java class files were used. In Java, there are several kinds of class-to-class compile-time 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*.

We assign a strength to the dependency from *A* to *B*, equal to the number of unique references, like those described above, from *A* to *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. This information is useful for implementing algorithms to simplify and summarize software DSMs. Details about how this was performed are deferred to Appendix A.

The Software DSM model

From class dependencies to software DSM

The software DSM is a matrix representation of the software dependency relationships. Each row and each column of the matrix corresponds to a class, and each dependency of nonzero strength is denoted by a mark in the row corresponding to the dependent class and the column corresponding to the depended-upon class:

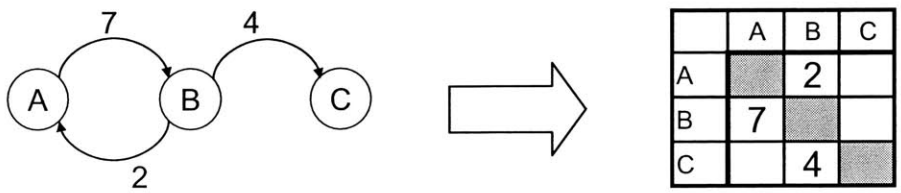

Figure 3: Example transformation from class dependencies to software DSM.

Most of the DSMs contained in the chapters of this thesis were rendered using DSAS, the Design Structure Analysis System. This software was created by Rusnak (2005).

Hierarchical DSMs

The software DSMs depicted in this thesis are far more complex than the example above, containing as many as 12,000 classes. For this reason, they have black dots instead of numbers 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.

Ordering in software DSMs

Authors such as Steward (1981) and Eppinger *et al.* (1994) have observed the importance of sequence in DSM representations of task structure, in which 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 thesis that are labeled as sorted have been reordered in a manner that strictly preserves the hierarchical structure of the packages and dependencies, but also minimizes marks above the diagonal. DSMs that have been sorted in this way can help to reveal layered software design structures, as well as cyclical dependencies which represent interdependent modules. The details of the algorithm are deferred to Appendix A.

An example of this reordering is shown in Figure 4.

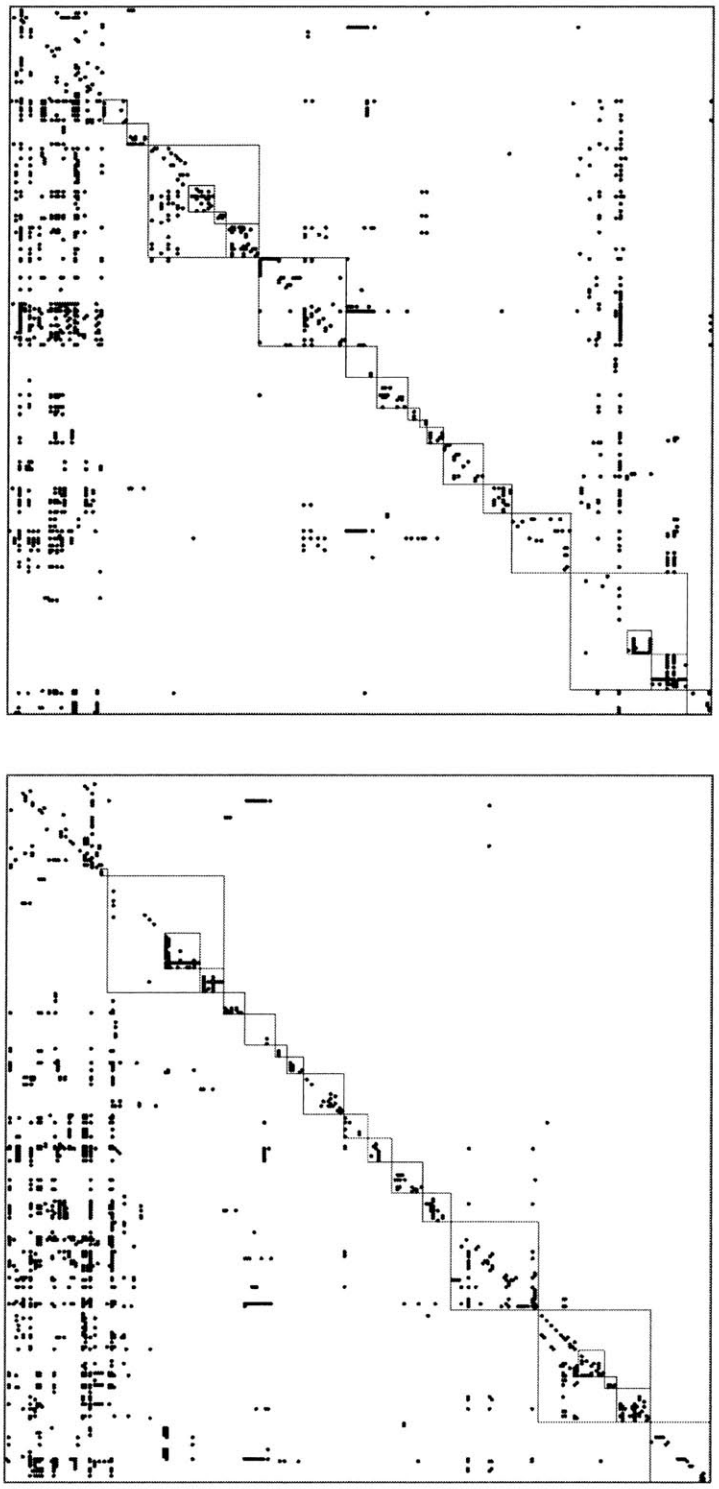

Figure 4: A software DSM, before (top) and after (bottom) sorting of modules. The dependency structure is the same, but in the lower DSM, marks above the diagonal denote cyclic dependencies.

Modularity Metrics

MacCormack *et al.* (2005) propose a collection of metrics for evaluating the degree of modularity of a software design. These methods were repeated for the two cases examined in this thesis, using software provided by author John Rusnak. Two novel metrics were computed using software developed for this thesis.

Dependency density

The first modularity metric proposed by MacCormack *et al.* (2005) is *dependency density*, which is simply the number of possible dependencies that are actually present in the code base. Conceptually, it is the proportion of cells of the DSM that are filled in with a dependency.

Propagation cost

The second metric proposed by MacCormack *et al.* (2005) is *propagation cost*. The intuitive basis for this metric is the idea that, if a class or source file is changed, it might necessitate changes in any of the files which depend on it, and, in turn, in any of the files that depend on those files, and so on. In mathematical terms, a change in any file or class has the potential to affect all of the files in the transitive closure if its "dependents" graph.

The propagation cost of a change in a single file is thus the size of the transitive closure of its dependent graph (as a number of files), divided by the total size of the codebase (again, as a number of files). For a discussion of methods for computing propagation cost, see Appendix A.

Clustered cost

The final metric proposed by MacCormack *et al.*, is *clustered cost.* It reflects the notion that in a modular design, components within the same parent module may be tightly coupled, but components in different parent modules should be less connected. The clustered cost metric models this notion by assigning different costs to dependencies within the module (or cluster) than to dependencies that cross a cluster boundary.

In order to assign cost in this way, it is necessary first to assign the individual units (files) to clusters – in other words, to create the modular groupings as a basis for assigning cost. In the methodology of MacCormack *et al.*, these clusters are based on an optimization algorithm for finding a clustering of units which minimizes this cost metric.

Packaged cost

As mentioned above, Java classes are grouped into packages, and this structure represents an inherent clustering. The *packaged cost* metric is similar to clustered cost, but uses the "natural" or "architectural" clustering available in the package structure of the codebase.

The exact formula for the metric is based on the clustered cost formula proposed by MacCormack *et al.*, but modified slightly. A "cost" is assigned to each class-to-class dependency. The cost is based only on the existence of any dependency, regardless of the strength of the dependency. However, the cost varies according to whether the two classes are in the same package, or in different packages. The following formula is closely adapted from MacCormack *et al.* (2005) and the textual description is a direct quotation:

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

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

"where d_{ij} is a binary variable indicating the presence of a dependency between i and j; n is the size of the cluster [in this case, the package]; N is the DSM size, and λ is a user-defined parameter."

The two differences between the package-clustered cost metric as used in this investigation and the minimized used by MacCormack *et al.* are: Firstly, whereas the clustered cost is an optimized least-cost clustering, the packaged cost uses the clustering as given by the package structure in the source code. Secondly, MacCormack *et al.* make special allowance for "vertical busses" in the DSM structure – that is to say, if the number of modules that depend on a given module exceed a certain threshold, such as 10% of all modules, that class is considered a "bus" class, and dependencies upon it are assigned a lower cost. The packaged cost metric makes no such allowance.

Packaging efficiency (Packaged / Clustered ratio)

The final metric is the ratio of the packaged cost to the minimized clustered cost. This metric, *packaging efficiency*, may be viewed as an evaluation of whether the classes are arranged within the package structure in a manner that maximizes the independence of the individual packages.

Change metrics

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 the module, during the period 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 number of new classes added and the number of classes removed, divided by the number of classes in the previous version of the module.

Toolkit development

In support of this thesis, Java code was developed to analyze and manipulate dependency graph information. This software translates between the formats provided and required by the other tools used in this thesis. It also implements the hierarchical dependency model and associated algorithms described in Appendix A. The source code itself is presented in Appendix B.

Chapter 3: Tomcat case study: Asynchronous Design Evolution

Background

The first J2EE web application server that we will consider is the Apache Tomcat project. The Tomcat project is an interesting case to study for at least two reasons.

Firstly, Tomcat underwent a change of project structure from commercial to open-source development. Tomcat was initially designed as the reference implementation of the Servlet specification, one of the original J2EE technologies. As such, it was provided by Sun to J2EE technology licensees as a sample of a system which correctly implements the specification. In this initial arrangement, the project was developed commercially by a team at Sun Microsystems, and the source was available to licensees. Later, in 1999, the Tomcat source code and intellectual property were donated to the Apache Foundation, and became Apache Tomcat. The initial open-source release was version 3.0 (Apache Software Foundation, 2006).

Secondly, 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. These changes to the codebase offer an opportunity to correlate changes in design structure and metrics with architectural intent.

Design structure of the Tomcat server

Two distinct functional modules with limited interface

The following is the DSM for Tomcat 3.0, which is the first open-source version of the server:

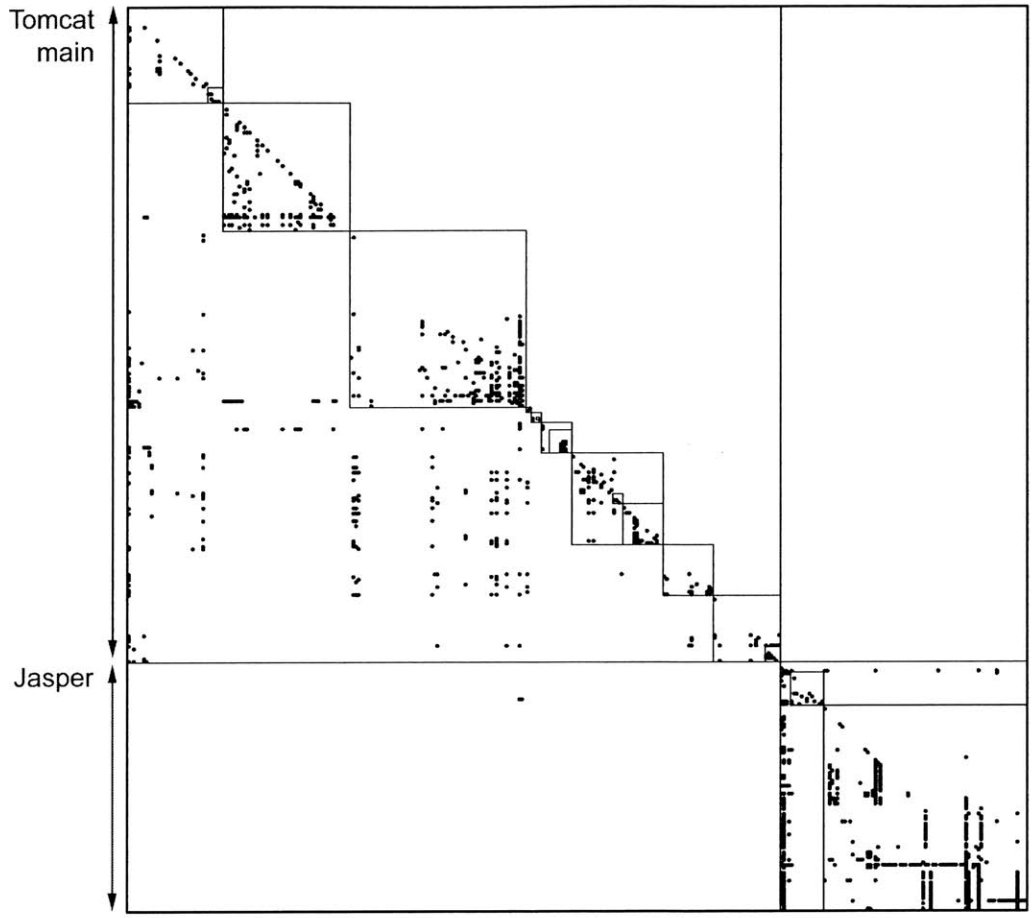

Figure 5: DSM for Tomcat 3.0, showing two independent modules. The DSM is sorted to show module hierarchy.

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

The initial DSM reveals that the Jasper depends on the Tomcat-main module. In addition, the DSM shows graphically that the interface between the two modules is very limited – In fact, it is only two points.

This relationship could be distilled to the following architectural block diagram, which can be inferred from the DSM. However, the block diagram does not convey as much information about the limited scope of the interface between the modules, or about their relative sizes or complexities. The block diagram is shown in Figure 6:

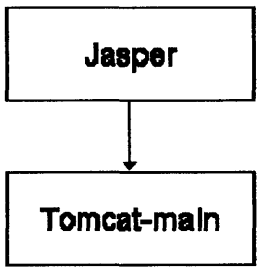

Figure 6: Block diagram of the high-level modular structure of Tomcat 3.0.

This initial DSM and the resulting architectural block diagram do not represent the complete relationship between these two modules.. Upon examination of the Tomcat source code, it becomes apparent that the points of interface shown in the above DSM are only calls to a utility function; the real interface is defined within the J2EE Servlet API, the specification to which Tomcat conforms. This is illustrated in the extended DSM in Figure 7.

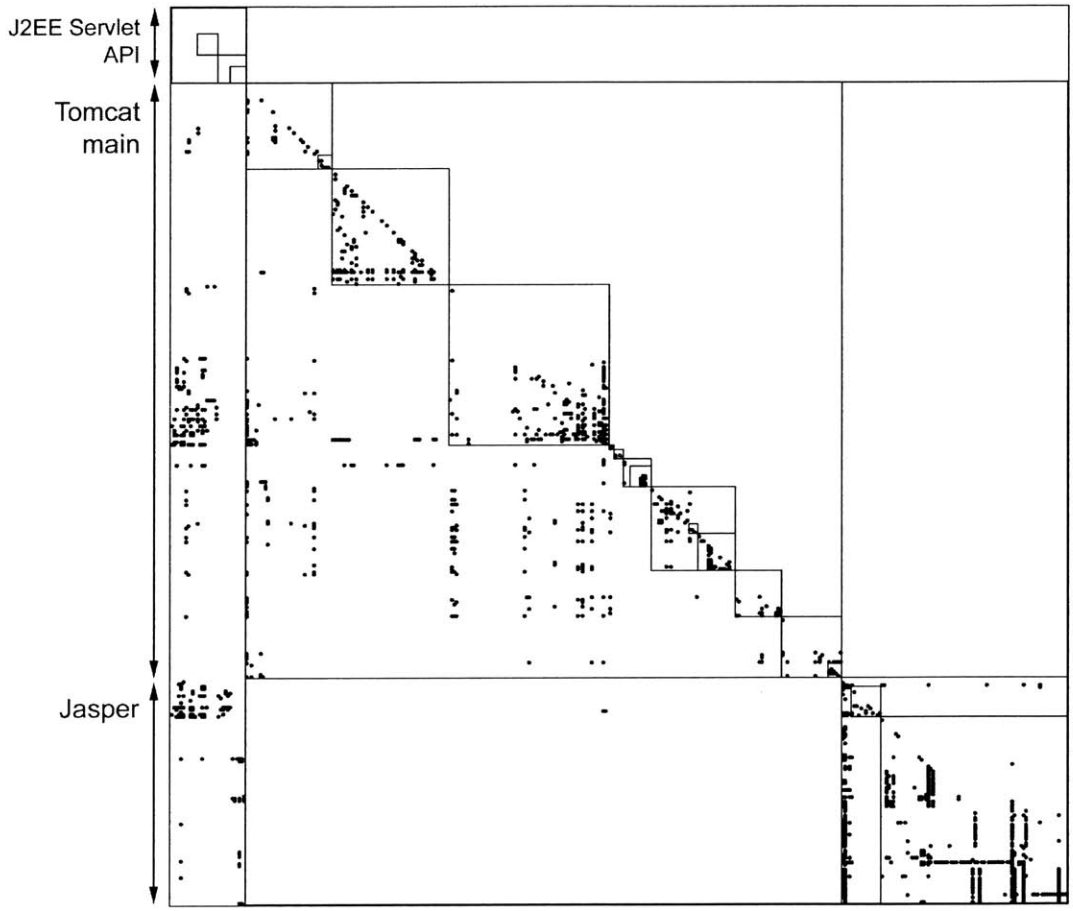

Figure 7: The extended design structure of Tomcat 3.0, showing the Servlet API classes as design rules. The DSM is sorted to show module hierarchy.

In this sense, the interface between these modules can be considered a "design rule," in the sense proposed by Baldwin and Clark (2000), because it defines a basic specification, which is independent of either module, and to which both modules must conform. As long as both modules conform to this interface, they may interoperate.

Based on this refinement, we can see that the Tomcat-main and Jasper modules are effectively independent, each depending only on the interface design rule. This relationship between is shown in the modified block diagram in Figure 8:

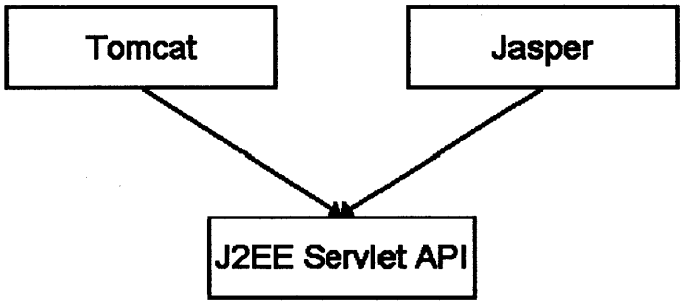

Figure 8: The revised block structure of Tomcat 3.0.

It is important to note that this final reordering of the DSM, and the discovery of the role of the interface as a design rule, was not performed automatically by the tools at hand. However, the DSM aided this discovery by graphically illustrating the nature of the relationship between the modules. Based on this insight, it was possible to examine the source code and refine the relationship.

Asynchronous evolution of platform modules

This decoupling of the two modules has enabled them to evolve separately. Across multiple versions of the product, each of the two modules undergoes at least one redesign, but these occur at different points in time.

This fact can be shown quantitatively, by observing the degree of change of the different modules across subsequent versions of the code. Table 1 shows the change ratio for each of the versions examined, from the previous version. In particular, even as the Tomcat-main code was entirely replaced or restructured entering versions 3.3.1 and 4.0, the Jasper code was only slightly modified.

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.

Modularity Metrics

Unit of evolution: The Tomcat server core vs. the complete project

Because the Tomcat-main and Jasper modules evolved separately, it makes sense to evaluate the modularity metrics for both the whole server project and for just the Tomcat-main module by itself. The results are presented below.

Results

Table 2 shows the results of computing these metrics for several successive versions of the entire Tomcat codebase, including Jasper:

	v3.0	v3.3.1	v4.0	v4.1.31	v5.0.28
number of classes	353	385	510	644	667
number of dependencies	1110	1774	2086	2829	3046
dependency density (per 1000 source-file pairs)	8.9	12.0	8.0	6.8	6.9
propagation cost	9.7%	11.1%	9.5%	7.5%	13.5%
packageCost (package-clustered cost)	49,278,865	146,593,900	300,666,439	692,142,963	665,515,426
finalCost (minimized clustered cost)	28,746,765	64,969,484	106,753,648	234,065,273	204,083,241
package/final cost ratio	1.71	2.26	2.82	2.96	3.26

Table 2 Results of modularity metrics for the full Tomcat server, across multiple versions.

Table 3 shows the results of computing these metrics for several successive versions of the main modules of Tomcat, excluding Jasper:

	v3.0	v3.3.1	v4.0	v4.1.31	v5.0.28
number of classes	256	261	296	527	463
number of dependencies	655	1143	1296	2215	1908
dependency density (per 1000 pairs)	10.0	16.8	14.8	8.0	8.9
propagation cost	10.5%	12.4%	14.8%	8.6%	18.4%
packageCost (package-clustered cost)	16,810,352	52,410,152	78,413,781	422,668,174	267,120,315
finalCost (minimized clustered cost)	11,140,993	20,588,043	29,282,159	182,986,033	100,249,761
package/final cost ratio	1.51	2.55	2.68	2.31	2.66

Table 3: Results of modularity metrics for the main module of the Tomcat server, across multiple versions

Discussion

From these results we can draw several important conclusions, both about the Tomcat project and about the model itself.

Asynchronous evolution and experimentation in a modular architecture

The DSM analysis reveals a design structure – two distinctly decoupled modules – that allowed the asynchronous evolution of these two modules. From version 3.0 to version 3.3.1, and again from version 3.3.1 to version 4.0, the main module of Tomcat was almost entirely rewritten or rearchitected, while the Jasper module was mostly unchanged. Moreover, the change metrics reveal that there was a different rate of experimentation in the two modules.

This is an example of the substitution operator proposed by Baldwin and Clark (2000) 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 one of the two branches to be adopted as the new primary version of Tomcat (Mazzocchi, 2006). Although 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) we can infer from the process itself, and from the result that a new architecture was selected, that some advantage was conferred by substituting a new version.

Of course, this example does not prove the converse – that without the design structure described above, the substitution could not have occurred in this manner. However, the formal properties of software dependencies suggest that this is the case: Had the two modules been tightly coupled by strong code dependencies, changes in one would

necessarily have forced changes in its dependent modules. What this example does show is that, at least in this case, the theoretical ability to substitute a software module conferred by the structure is actually useful in this open-source software development environment.

Reflections on the modularity metrics

The metrics data show that, even where the architectural intent of a set of changes is to refactor, simplify, and make the codebase more modular, code sprawl – the increase in the overall size of the project – is a major risk.

The evolution from version 4.0, in which the Tomcat-main module was completely replaced, to version 4.1.31, which was the "refactoring" of this rewritten module, to 5.0.38 is interesting, because the metrics give conflicting results.

During this timeframe, the overall size of the system increased. From version 4.0 to version 4.1.31, the propagation cost decreased considerably – from 9.5% to 7.5% – which suggests successful refactoring, but the clustered cost more than doubled. This may be a result of the increase in number of modules, but it may also suggest a degradation of the quality of the package structure. Indeed, an examination of the list of packages in the DSM for this version reveals the re-introduction of several packages which were eliminated in the version 4.0 rewrite. For this reason, we can conjecture from the data that, while version 4.1.31 was is considered a "production-quality" release, and is functionally stable, it was in a state of architectural flux at this point. The low propagation cost may actually reflect the fragmentation of the version's base classes. That is to say, the reintroduced packages may represent redundant functionality in the

basic utility classes which are widely used throughout the system; because different portions of the system use different pieces of redundant functionality, the propagation cost of changes in this area is lowered.

In contrast, in the transition from version 4.1.31 to version 5.0.28, the propagation cost increases very significantly, from 7.5% to 13.5%. However, the clustered cost actually decreases, which is very significant, given that the number of modules actually increased.

Examination of the list of packages in the DSM suggests that the increase in propagation cost resulted from the rationalization of the product's package structure and the removal of the redundant functionality – several packages that were eliminated in version 4.0, then reintroduced in version 4.1.31, were again eliminated. However, this rationalization, and elimination of potentially redundant functionality increases the propagation cost, because more of the system depends on the same base classes.

The data suggest that the metrics are sensitive to the number of modules (source files or classes) under consideration. For this reason, special care must be taken when using these metrics to track the evolution of a project across time, because the number of modules in a software project often changes. Increases and decreases in the values of these metrics are neither inherently good nor inherently bad. Rather, the metrics may be used to discover and highlight the nature of the system's design evolution.

Chapter 4: Company 2 case study: Splitting to enable substitution

Background

The product platform, and its conundrum

Company 2's product line is a family of web-based applications – software applications which run on a server, and which allow user interaction through web pages. Examples of such web 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 consists of a J2EE Application Server; an application framework, which implements basic services used by all of its applications; and a business logic engine, which implements more advanced services also used broadly within the product family. Figure 9 shows a module block diagram of this structure.

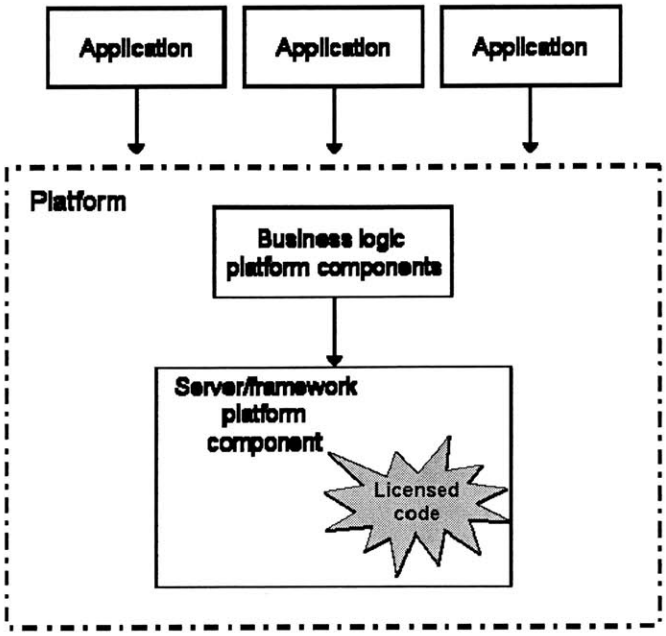

Figure 9: Block architectural diagram of Company 2's product family, including both platform and application components.

The platform components are shown at the bottom of the diagram, with the applications sitting on top. Arrows indicate the dependency relationships of the modules – the applications depend on the platform components, and the higher-level platform components depend on the framework/server component at the bottom.

This chapter will focus on the server/framework application component, upon which the entire product family depends. It will examine the partial redesign of the platform structure to increase modularity and create a new option for addressing a strategic vulnerability.

The server/framework component is a J2EE-compliant application server. That is to say, it is an application server which conforms to the design rules of the J2EE Specification.

If that were its only role, then another, third-party, J2EE-compliant could be substituted for this portion of its functionality.

However, this component is not only a J2EE Server; it also contains framework components, basic services on which the whole product platform and family rely. Therefore, a commercially-available third-party application cannot be substituted wholesale for this major platform component.

Moreover, this component contains licensed code – some parts of it were derived from code which Company 2 licensed from another vendor – and this code could not be readily separated from the rest of the platform.

This situation created distinct strategic risks for Company 2: The expiration of this license agreement could prohibit Company 2 from releasing new versions of its software containing the licensed code. Because this licensed code was intertwined with Company 2's product family, such an event could place Company 2's entire product family at risk!

Application of the dependency structure model to this example

Use of dependency-based analysis to find a cleaving line

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. This substitution could be performed by Company 2, licensing the software component and integrating it directly into the platform, or by customers, integrating the products in the field. The secondary goals

were to perform this separation 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.

$$L = \{\text{code under license}\}$$

However, if this code is to be separated, all the code which requires it must also be separated. This set is denoted by R_L, the classes which require licensed code.

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

However, all of R_L cannot simply be split off and substituted, because some of it is also required for the rest of the platform and application. At this stage, the set R_L of classes was 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 which was required by other platform and application components cannot be excluded from the platform.[3]

---

[3] 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: To perform this operation formally, one could use the following steps.

First let A be the set of all classes in the product family:

$$A = \{\text{all classes in the product family}\}$$

The define A' as all of the classes except those which require the licensed code:

$$A' = A - R_L$$

Let $R_{A'}$ be the classes which are required by some class in A':

Once the appropriate cleaving line was determined to split the server/framework component into two separate modules, the code dependencies that violated the constraints imposed by the separation were systematically eliminated.

$R_{A'}$ = {all classes that are required by some class in A'}

Then the set of classes P which must be preserved in the platform, and therefore must be extricated from dependence upon the licensed code L is simply the intersection of the two requirements sets:

$$P = R_L \cap R_{A'}$$

In order to do so, any dependencies which go from the preserved code to the licensed code must be broken. The set D_{elim} is the set of dependencies which must be eliminated:

D_{elim} = {all dependencies from class i to class j such that $i \in P$ and $j \in R_L - P$}

Figure 10 illustrates the design structure of Server-2, prior to splitting.

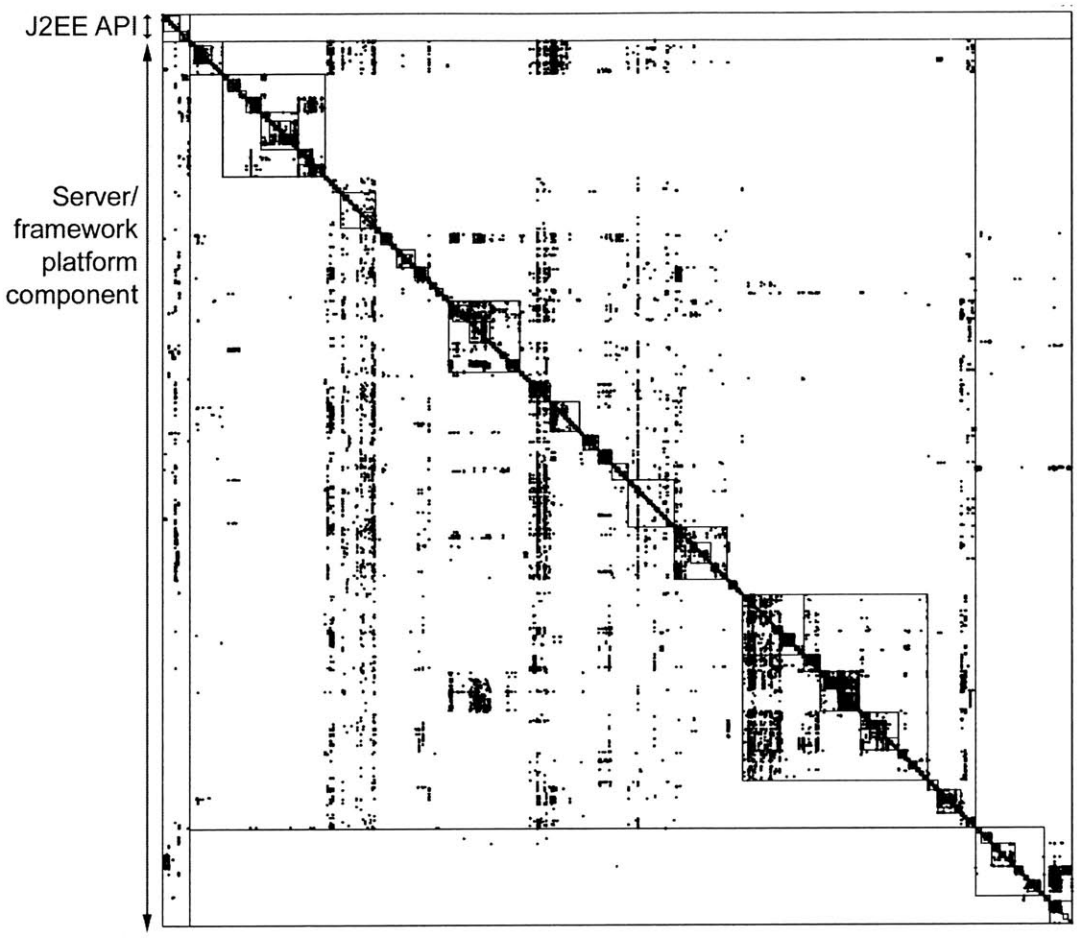

Figure 10: DSM of Server 2, before splitting. Note that the server/framework component is composed of a single, large module.

In contrast, following engineering work to resolve the deleterious dependencies, the DSM shows clearly that the main server/framework component no longer relies on the licensed components. This is shown in Figure 11.

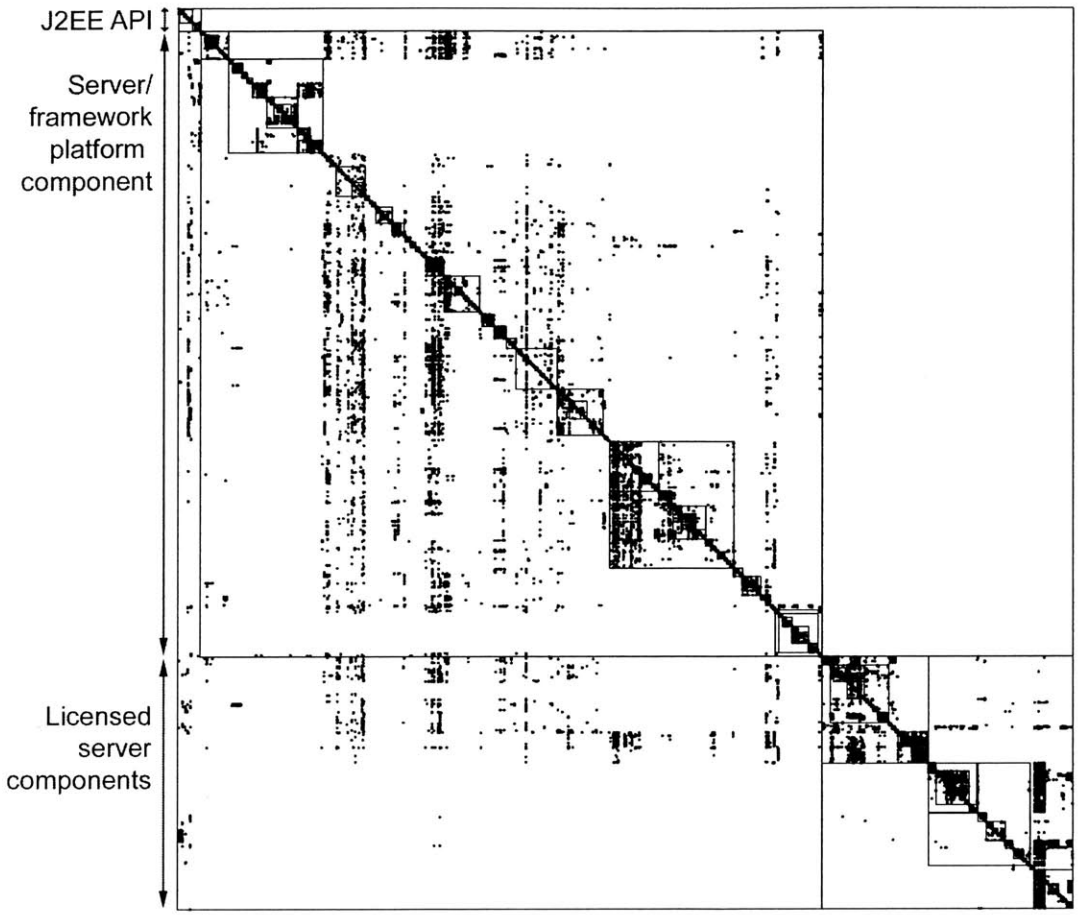

Figure 11: DSM of Server 2, after splitting. Components under third-party license have been separated into a new top-level module.

This new structure is illustrated by the block diagram in Figure 12:

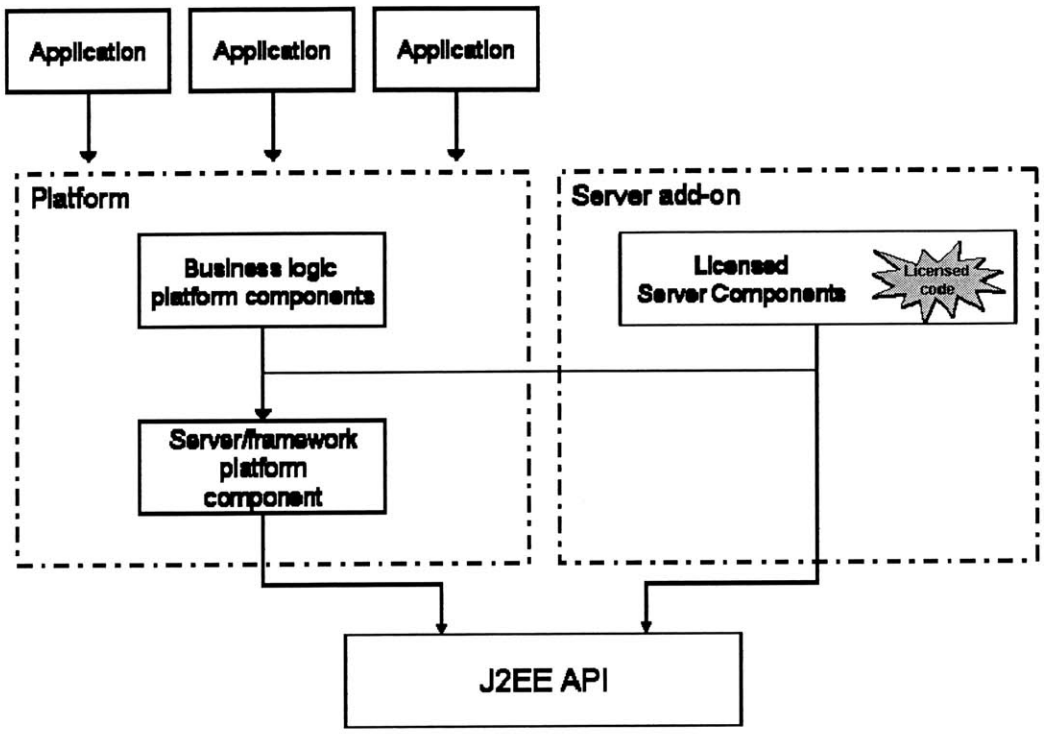

Figure 12: Block diagram of the restructured Company 2 product family, in which licensed components are separated from the main product platform.

Because no portion of the Company 2 platform or applications depends on the licensed server components module, another implementation may be substituted for it, as long as the substitute module conforms to the underlying design rules which are the interface between the product family and the licensed code module. Because, in this case, the design rules are defined by the J2EE API module, a third-party J2EE application server can be used for this purpose. In fact, Company 2 supports this configuration.

This substitution scenario is illustrated by the following block diagram (Figure 13). The company's product platform is unchanged, but the licensed server components are acquired from a third-party J2EE-compliant server:

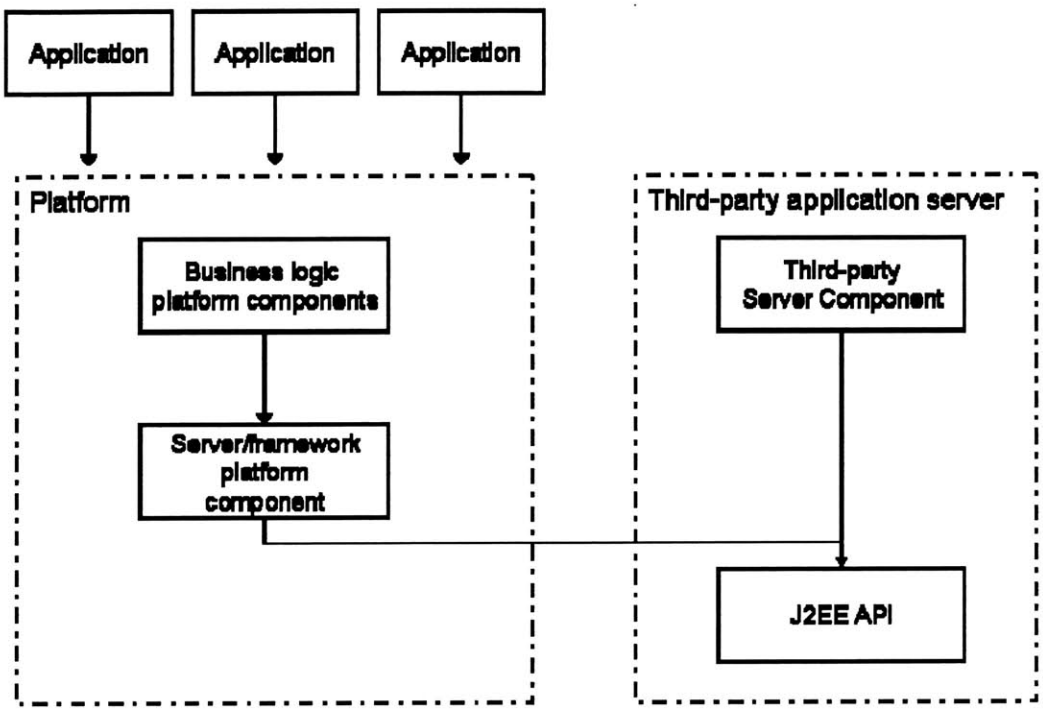

Figure 13: Block diagram of the restructured Company 2 product family, in which a third-party product has been substituted for licensed components previously contained in the platform.

The following DSM (Figure 14) shows the structure of the Company 2 platform with the licensed code eliminated, and running on a third-party application server.

	A	B	C	D	E	F	G
A J2EE API							
B Server/framework component	X						
C Business logic component		X					
D Third-party Server components	X	X					
E Application 1	X	X	X				
F Application 2	X	X	X				
G Application 3	X	X					

Figure 14: DSM showing the design structure of the Company 2 product family operating with a third-party J2EE server.

Modularity metrics

Table 4 shows the metrics results for the Company 2. Due to the large size of this software system, the minimized clustered cost metric was not computed.

	Before split	After Split
number of classes	10603	12750
number of dependencies	54285	65114
dependency density (per 1000 pairs)	0.5	0.4
propagation cost	18.7%	10.3%
packageCost (package-clustered cost)	3,490,093,345,221	6,139,113,728,992

Table 4: Modularity metrics for the Company 2 product platform, before and after splitting.

The most notable change in these metrics is a significant decrease in propagation cost, from 18.7% to 10.3%. It makes sense that the partitioning of the design structure in this way would decrease the propagation cost, because the main "server/framework"

component is insulated from all changes in the "server add-on" component. However, it must also be noted that some of this decrease may be attributable to the addition of a significant number of new classes. If these classes are not highly connected to the existing structure, such an addition could also decrease propagation cost.

Discussion

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. Unlike the Tomcat example, this split was deliberately engineered to accrue a specific strategic benefit, namely to protect the Company's product platform against the loss of licensed components.

There is an important difference between the design structure of Tomcat and that of 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, the Company 2 platform structure is layered; the newly separated "licensed server components" platform module relies on the platform core. However, 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.

Remaining strategic vulnerabilities

While this restructuring addressed the problem of having licensed code in the platform, it did not address the vulnerability of this platform to changes in the J2EE specification itself. As illustrated in Figure 15, the block diagram of the platform architecture, the entire product family depends, directly or indirectly, on these specification classes. In

this sense, the J2EE specification is a design rule for the entire product family, and for its interface with third-party software. However, Company 2 does not control this specification. For this reason, changes in this environment still expose the product platform to obsolescence.

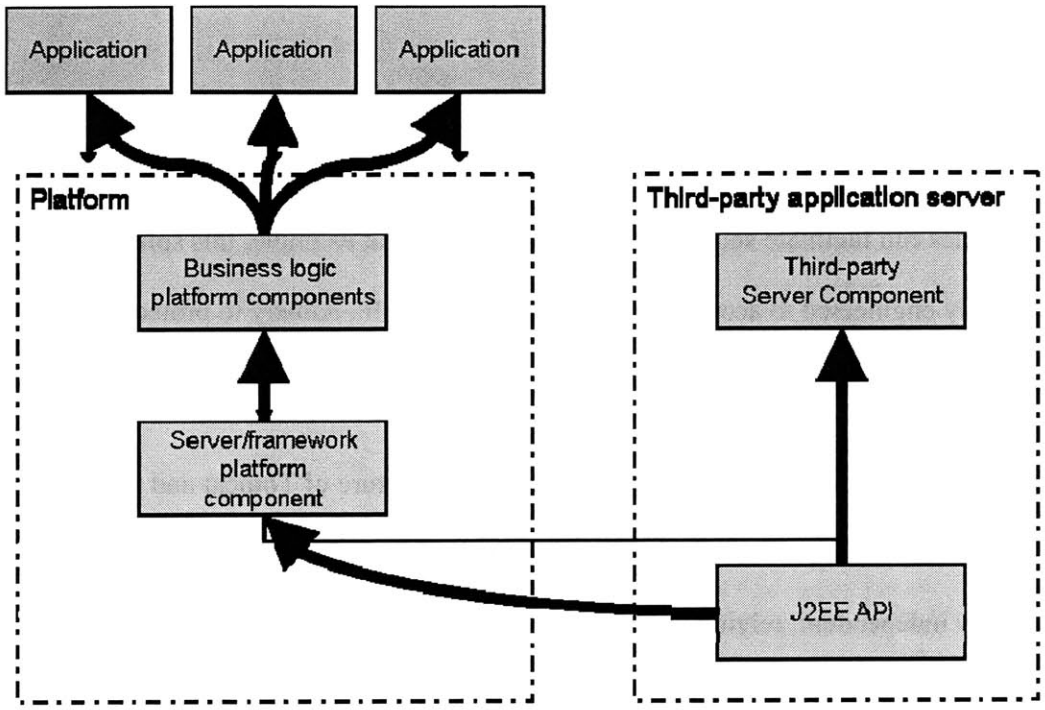

Figure 15: Block diagram showing propagation of changes in the J2EE API specification through the Company 2 product family. Dark arrows indicate the route of propagation. Modules that are potentially affected are colored gray – which is all of them!

Figure 16 is a block diagram that illustrates a potential architectural strategy to mitigate this exposure. In particular, all of the application framework and logic that does not require the J2EE specification is separated into its own layered hierarchy of modules. A separate layered hierarchy of modules integrates this application framework and logic

with the functionality for generating web pages and user-interfaces provided by the J2EE specification.

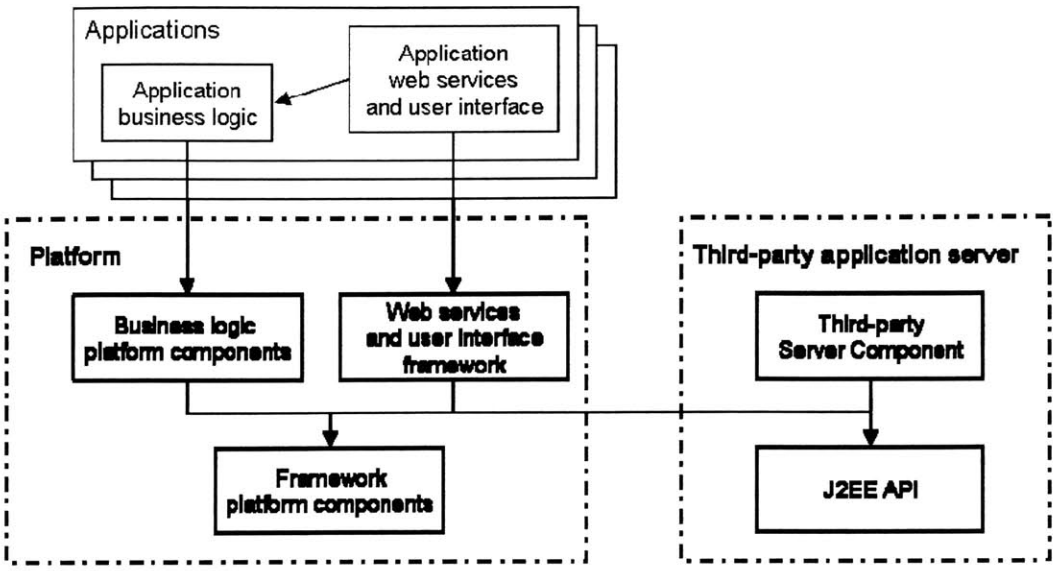

Figure 16: Block diagram of an alternate structuring of the Company 2 product family, designed to contain the affects of changes in the J2EE API specification.

As demonstrated in Figure 17, this alternative architecture limits the exposure of the product family to a class of changes in the technical environment.

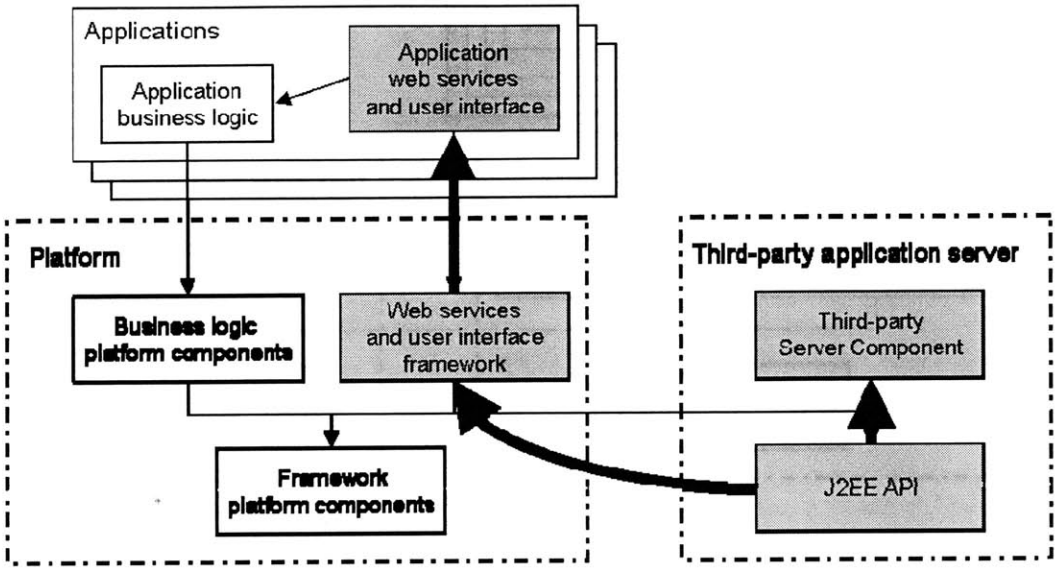

Figure 17: Block diagram showing the affect of the alternate structuring of the Company 2 product family, limiting propagation of changes in the J2EE API specification. Dark arrows indicate the route of propagation. Modules that are potentially affected are colored gray.

Chapter 5: Discussion and conclusions

An architecture that enables experimentation and substitution

The results for both the Tomcat and the Company 2 product architectures demonstrate an example of a design structure that enables different rates of experimentation in different subsystems of the software platform. The case of Company 2 additionally demonstrates a real-world scenario in which a firm has accrued a distinct and quantifiable advantage from deliberately modifying its architecture to fit this pattern.

It is, however, important to note that these results do not show that the design structure shared by Tomcat and the refactored Server 2 platform is necessary in order to accrue the benefits. It is possible that other architectures exist which would also allow experimentation and substitution of modules. However, the evidence of asynchronous evolution of the decoupled modules does suggest that this architecture is sufficient to allow such experimentation.

These results also do not demonstrate that the design evolution of software naturally gravitates towards modular architectures like those depicted in these cases. Indeed, there is evidence that, without special effort, the modularity of software tends to decay over time. In a longitudinal study of a large software system, Eick *et al.* (2001) found that the span of individual changes (the collection of files or modules changed to make a single modification) increased significantly. Additionally, whereas initially such change spans tended to form distinct clusters of associated files, in late stages of system evolution no such clustering was apparent.

Moreover, the modular design structure that emerged in these two cases may have been facilitated by the J2EE specification, which provided a set of ready-made design rules. The J2EE specification is the result of the Java Community Process, a broad-based iterative specification effort (Byous, 2003). The facts that these specifications were fixed before implementation, that they formally defined interfaces within the programming environment, and that conformance to them was a firm requirement of every implementation, together essentially forced the promotion of J2EE API to the status of design rules.

Nonetheless, the cases examined in this thesis do suggest that dependency models can be used as a tool for understanding and describing existing software product architectures, and for finding ways to improve them. Even if the design evolution documented in these cases is the result of a fortuitous and atypical alignment of factors influencing the software architectures, the patterns observed here can still 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.

Strategic value of modularity in software design

Validation of the option value of substitution in a commercial software environment

The case of Company 2 represents an example where the modular partitioning of a software product platform has concrete and quantifiable strategic value. Because of the license restrictions associated with the code embedded in the product platform, the

viability of the entire product family hung in the balance. In this case, the benefit vastly outweighed the engineering cost of increasing the modularity of the platform.

The software architecture was partitioned, and modifications were made to assure that the dependencies on the licensed code module were completely removed, but the software was not, on the whole, rewritten to accomplish this. In this case, a single cleavage of the software, into two primary modules, conferred tremendous value. The value was conferred because the module which was made substitutable had a high strategic risk associated with it.

Baldwin and Clark (2000, 2002) demonstrate that the process of splitting a single-module system into several independent modules, resulting in the ability to experiment and substitute at the module level, will always yield a positive change in the overall value of the system, subject to the assumption that the values of the resulting modules are a normally distributed random variable.

The Company 2 case strengthens the case for attributing positive option value to modular structures, by exemplifying a situation where the module relationships are highly asymmetric, and the value scenarios are discontinuous, but there is still positive and quantifiable option value in the modular decomposition. In fact, it is precisely because of the asymmetry of the modules and the discontinuous variance in the value of the licensed code module that the incremental value of the modular splitting was so high. A visual inspection of the block diagram in Figure 12 reveals the asymmetry in this modular decomposition: The "Server Add-on" module containing the licensed code is dependent on the platform core component, and not vice versa. Moreover, whereas the value of the

platform core module is relatively stable, the value of the licensed module is highly volatile. Depending on the license status of this module, it might have the incremental positive value of completing the functionality of the server platform, or it might have a value that is so negative that it cancels the value of the entire product family by rendering it unsellable.

Banker and Slaughter (2000) argue that modular structuring of software code is most useful as a means of mitigating the negative effects of volatility (frequent changes) and complexity (difficulty of comprehension of the problem domain). They find, both theoretically and empirically, that when code is frequently modified, programmers become familiar with its structure. Programmers are able to limit their scope of search during maintenance and enhancement activity, thus reducing costs. Similarly, when a problem domain is highly complex, software structuring aids problem decomposition and comprehension. Conversely, when volatility and complexity are low, the benefits accrued may not repay the investment in software structuring, and too much may even impair comprehensibility.

While the example of Company 2 certainly does not contradict these notions, it does suggest that there may be a more general principle at play: Rather than confining the analysis to complexity and frequency of changes, this example suggests that the key criterion is volatility in the financial sense of the word, encompassing any form of value risk. In this paradigm, high frequency of changes entails risk because frequent modification increases the risk of introducing functional defects. Similarly, complexity entails risk because code that is difficult to comprehend is more likely to contain flaws.

In the case of Company 2, the risk associated with the code was legal in nature, and the source of risk was entirely external to the software system and the problem domain. Nonetheless, all three sources of risk have the same potential negative outcome: Partial or total loss of value of the module.

Moreover, the dependency model suggests that it is not necessary to apply notions of risk monolithically to an entire codebase. Rather, risk may be attributed to specific entities or modules within the software system. Because the value of each module is contingent on that of the modules it requires, risk propagates through the dependency graph. That is to say, if a module is at risk – and thus has a wide variance in its estimated value – then the modules that depend upon it are similarly affected.

Recommendations

Isolate areas of risk in substitutable modules

Based on this model, we would make the following recommendations to software managers and architects: Firstly, managers and architects should identify areas of risk – technical and otherwise – that could impair the value of their platforms or products. Secondly, managers and architects should select modular decompositions that isolate these risks in substitutable modules wherever possible. In particular, dependence upon an at-risk module should be avoided if possible. Architects may wish to create new design rules, in order to define interfaces which eliminate undesirable dependencies. Such a practice requires extremely careful consideration when defining such interfaces, because if they are later discovered to be incorrect, revising them can require widespread changes (Baldwin and Clark, 2000).

Potential sources of uncertainty include:

- Intra-firm technical risks associated with a module, such as difficulty of implementation;
- Inter-firm technical risks, such as design rules controlled by competing firms, or standards which may change;
- Legal factors, such as intellectual property restrictions; and
- Market factors, such as uncertainty about market opportunities or user requirements.

Software engineering teams should adopt practices to maintain the integrity of modular structures during ongoing development and evolution of software platforms. This requires, firstly, communicating the design intent to software developers. Design structure matrices and block architectural diagrams may be useful for this purpose. Secondly, measures should be taken to maintain the integrity of the modular architecture during software evolution. Because static dependencies are computable properties of software code, it is possible to formalize and verify allowable dependencies automatically, every time software is built. Sangal *et al.* (2005) have suggested such a constraint-based approach to maintaining dependency rules.

Risks of pursuing strategic modularity

This process and its potential benefits, however, do not come without their own risks. These risks include increased costs; the creation of suboptimal solutions; problems created by prematurely fixing a modular decomposition; hampering the creativity of researchers and engineers; policy resistance from researchers and engineers; and increased software defects due to erroneous software reuse.

Cost of modularity

Numerous authors have noted that modularity is not free; there is expense associated with front-end design to create design rules and achieve modular decomposition (Banker and Slaughter, 2000; Baldwin and Clark, 2000). Banker *et al.* (1987) observed that some kinds of software structuring techniques had negative short-term effects on programmer productivity. This result, however, is not surprising; the authors observe that their window of measurement was very short, and a planning cost associated with these methods is expected.

Software mangers should weigh the expense associated with modular design and implementation against the anticipated benefits. It is clear that some level of modular decomposition carries significant net value. And it is also clear that overzealous pursuit of modularity can result in more expense than benefit. Further development of valuation models for modular decomposition may help managers evaluate these tradeoffs.

Suboptimal solutions

Modular strategies in product design and development may lead to suboptimal solutions. As a cause of this phenomenon, researchers have pointed both to cognitive-procedural factors, and to architectural factors.

Brusoni *et al.* (2004) provide a model that suggests that decomposition of a system into modules increases the speed of search for better solutions, by decreasing the size of the unit of experimentation. However, such decomposition also constrains the search space, because possibilities that are based on alternative decompositions are excluded from the search.

Ulrich (1995) provides a similar rationale, in the paradigm of systems architecting. Specifically, he argues that whereas modular architectures provide ample opportunity for optimization of "local performance" within their components, only integral architectures offer opportunities to optimize "global performance," which only emerges at the system level.

These arguments translate well into the language of software architecture. If the span of changes necessary to implement a functional enhancement is limited to a single software module, then the speed of implementation should indeed be faster. Similarly, by specifying interfaces with circumscribed functionality, software architectures impose constraints on what functionality can be achieved within them. Finally, by imposing function call overhead or other indirection between software components which might perform more optimally if they were combined, modular decomposition can lead to lower overall system-level performance.

However, for many software vendors, time to market may trump operational performance. Delays in shipping software lead directly to lost revenues, and increase the engineering cost of the product. Processor speed, in contrast, has grown steadily cheaper. Moreover, if modular decompositions aid design comprehensibility in a way that decreases the incidence of defects, such a benefit may also outweigh a marginal decrease in performance.

Premature definition of modular structure

Modular approaches to software design may cause problems when they are used to develop highly novel software applications, or software applications for problem domains which are poorly understood.

Nguyen *et al.* (2000) have observed that when requirements engineering teams conceptualize a new problem domain, they often follow a repeating "catastrophe cycle" pattern: Their conceptual model of a problem domain progressively increases in complexity until it becomes untenable, and this event is followed by a complete revision to create a new, simplified, and consistent conceptual model. If modular architectures are based on early, incomplete understandings of the problem, erroneous or inconsistent models may result.

Similarly, Baldwin and Clark (2000) warn that, if design rules are chosen hastily, their later revision may force expensive and widespread changes that propagate through dependent modules. If an application or problem domain is characterized by architectural uncertainty, modularizing a system before system-level interactions are well understood could be counterproductive.

Hampering creativity

Conversely, if the modularization and associated design rules are fixed, and not allowed to change in response to changing understanding, it may prevent software designers from effectively using improved understanding to improve software. Such inflexibility could impede creativity within the engineering organization.

Resistance to the processes necessary to create and maintain modularity

Even if a modular architecture is both adequate and efficient as a basis for a product platform or family, software engineers may chafe at a constraint-based approach to maintaining architectural integrity. They may not enjoy being told, "You can use these components, but these others are off limits, even though they may contain functionality which appears useful to you."

Fortunately, the model itself might be useful to mitigate such resistance. DSMs and block architectural diagrams could be used as a tool to communicate details of the architecture, and its strategic and technical rationales. Moreover, such communication of architectural intent may offer engineers a chance to offer architectural insights, potentially critiquing and improving these structures.

Dangers of software reuse

Substitution of and experimentation in software modules is effectively a form of software reuse, because a newly substituted module represents a modified operating environment for the other software components with which the module interfaces. Leveson (2004) has noted that reuse of software carries distinct risks. Reuse increases the risk of system accidents and malfunctions, because undocumented assumptions about the operating environment may not hold true in the changed environment. High-profile accidents that have resulted in this manner from erroneous software reuse include the loss of the Ariane 5 launcher rocket due to the malfunction of reused software (Lions, 1996), and the Therac-25 accidents, in which software reused in a radiation therapy machine killed or injured several patients (Leveson and Turner, 1993).

In the module-substitution scenario, the correspondent danger is that other portions of the software system will be based on assumptions about the operation of the module to be substituted. Such assumptions may be valid for the original module, but if they are not carefully documented, developers writing a new substitute module may be unaware of them.

Leveson (2004) argues that carefully capturing intent and assumptions is critical for safe reuse of software. This suggests that careful and complete specification of interfaces used as design rules in software modularization, including all assumptions about the operating behavior of dependent modules, is important to mitigate the risks associated with module substitution.

Indeed, in the case of Company 2, even though the substitution of a third-party application server was based on the detailed design rules embodied by the J2EE specification, and even though every such application server was required to pass a comprehensive suite of compatibility tests, it was still necessary to test all of Company 2's applications in the modified operating environment. Such testing and debugging should be accounted for when estimating the cost and value of module substitution.

Future directions

The integration of structural and financial or strategic models of software architecture is just beginning. Future research in many directions could help realize its benefits.

Further development of valuation models

The real-options valuation of modular architecture is based on the notion that the expected value of a module is based on a random variable, the realization of the module's technical potential in the market, whose outcome is unknown. Modular architectures create option value, because they allow the value of improvements within these modules to be realized without replacing the whole system (Baldwin and Clark, 2002). One extant problem in appraising the value of modular architectures is estimating the volatility of individual modules' value.

Tools development

Another area of future development in this domain is the implementation of these models as interactive tools that can be used by managers and architects. Such tools might allow users to navigate through a DSM; zoom in and out to different levels of granularity; isolate dependencies that violate certain constraints; and view the source code which causes the dependency. At least some of this functionality has been implemented by a commercial product called Lattix Dependency Manager, which is used by Sangal *et al.* (2005).

Understanding architects' mental models

In addition to making these models available to architects, it would be interesting to know how well these models match architects' current conceptions of software structure and strategy. Do architects consider dependency structure important?

One source of information about architects' intents is their work product. By investigating a wide variety of software architectures, we may determine whether the

model predicts their structures. However, because software structure can change during its evolution, there is no guarantee that the architects' original intentions have been preserved within existing structures.

Another possible means of investigation would be actively to test architect's choices, by asking them to choose among and critique alternative architectures. However, in this case, there is a danger that the means of questioning or presenting alternatives to the architects could itself influence them to frame their responses in terms of a particular model.

Understanding the role of organizational structure in creating modularity

Finally, there is very fertile ground to investigate what kinds of organizational structure and processes are conducive to developing and maintaining modular software. Authors such as MacCormack *et al.* (2005) have investigated whether open source and proprietary software development efforts produce different results.

Conclusion

Professor Carliss Baldwin of Harvard Business School offered an apt comparison of these explorations of design structure to the early efforts of cartographers. The first maps were not very accurate, or very useful. But today, based on these humble beginnings, we have maps that integrate and illustrate tremendous amounts of information – geographical, navigational, commercial, political, and so on. Similarly, we hope that these software DSMs and dependency models will form the basis for a deepening understanding and integration of structural and strategic information about software architecture.

In many ways, the creation of models for software design structure simply makes explicit what good software architects have always known. However, even the full realization of these models will not transform software design into an algorithmic activity. Software design, like any innovative activity, will always require the spark of creativity, and be as much art as science. By integrating notions of design structure and technology strategy, we hope to enrich both managers' and architects' intuitions, and to provide a new means of communicating these understandings.

Appendix A: Computational details of the model

This appendix presents some algorithmic and algebraic details of how the dependency model used for this thesis was implemented. These details may be of interest to anyone who wishes to reproduce the methods used here, but they are not required to understand the results and conclusions of the thesis.

Aggregation of class dependencies in the package hierarchy

Chapter two describes the creation of the dependency graph. Recall, this dependency graph assigns a number $depStrength(class_A, class_B)$ as a strength of the dependency from class A to class B.

As noted, Java packages create a hierarchical namespace. Classes may be contained in any package at any level of this namespace, and any package may have subpackages. The conceptual goal of dependency aggregation in the package hierarchy is to create total package-to-package dependency strengths, based on the strengths of the dependencies between the classes contained within them. An example class dependency graph, along with its package structure, is shown below in Figure 18.

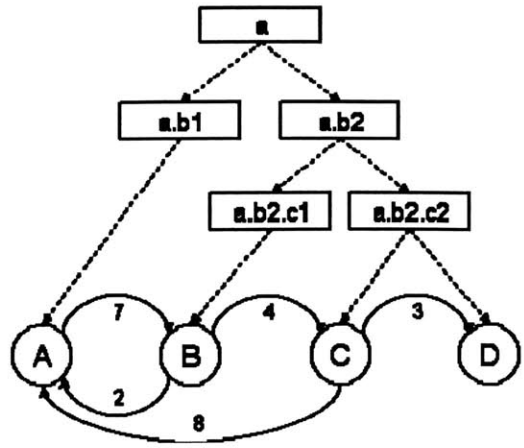

Figure 18: A class dependency graph, in which the classes (A, B, C, and D) are denoted by circles, and dependencies are shown by arcs. The classes are shown as they are contained within the package hierarchy. Packages are shown by rectangles, and their parent-child relationships are shown by dashed lines.

The strength of the dependency between two packages with a common parent packages is simply the sum of the strength of the dependencies that cross from the child classes of one package to the other:

- If $package_a$ and $package_b$ have the same parent package:

$$depStrength(package_a, package_b) = \sum_{\substack{class_A \in package_a, \\ class_B \in package_b}} depStrength(class_A, class_B)$$

- else:

$$depStrength(package_a, package_b) = 0$$

Figure 19 shows the same dependency and package structure, with the package dependency strengths added:

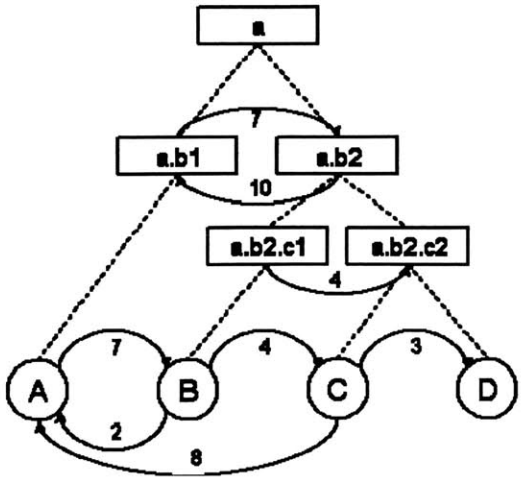

Figure 19: Class and package dependency graph. The class dependencies (denoted by arcs between circular nodes) have been aggregated into package dependencies (denoted by arcs between rectangular nodes).

Hierarchical topological sorting of package and class dependency graphs

Chapter Two presented an example of how sorting the order of classes and packages in a software DSM can increase its comprehensibility and highlight cyclic dependencies. This section presents one algorithm for performing this operation. It is adapted from a standard topological sort algorithm, which extracts a full ordering from an acyclic directed graph (Skiena, 1998). This algorithm performs a heuristic topological sort, using the dependency weights to resolve cyclic dependencies, at each level of the package-and-class hierarchy.

The algorithm follows:

- Starting with the root package $package_{root}$:
  Perform recursive step with package $package_{root}$.
- Definition of recursive step, for package $package_a$:

- Perform topological sort of dependency graph with the dependency graph of the child packages of $package_a$.
- Perform topological sort of dependency graph with the dependency graph of the child classes of $package_a$.
- For each child package $package_{child}$ of $package_a$:
  Perform recursive step with package $package_{child}$.

- Definition of topological sort of dependency graph, for arbitrary dependency graph:
  Let S denote the set of nodes in the dependency graph.
  - Let R denote the result, an ordered list of dependency graph nodes.
  - Let $strength_{xy}$ denote the strength of dependency from each node $node_x$ in set S to each other node $node_y$ in set S.
  - Let $incoming_x$ denote the total strength of incoming dependencies to node $node_x$ from all other nodes in S.
  - While S in nonempty, do the following:
    - Let $node_x$ denote the node in S with the lowest total strength of incoming dependencies $incoming_x$.
    - Remove $node_x$ from set S.
    - Append $node_x$ to the end of the result list R.
    - For each node $node_y$ in set S:
      - Decrement the total incoming dependencies for node $node_y$ as follows: $incoming_y = incoming_y - strength_{xy}$

Computing Propagation Cost: Alternative algorithms

MacCormack *et al.* (2005) note that the DSM representation is very useful for computing propagation cost: The DSM shows the first-order dependency relationships, i.e. classes that are visible to each other along a path of length one. However, when the matrix is raised to the nth power, it will show the visibility along paths of length n. In this manner, by summing (or taking the binary union) of successive powers of the matrix, the transitive closure of the dependency matrix can be derived. The ratio of the number of filled cells to the total number of cells is the average propagation cost.

As an alternative algorithm, it is possible to visit and process only the nodes that are actually in the dependency closure to compute the propagation cost. This can be accomplished using a depth-first search in the graph.

The following algorithm assumes a set G of dependency graph nodes $node_i$ each of which has a set D_i of dependent nodes (nodes that depend on it). We will further associate a set C_i with each $node_i$, initially empty, to represent the dependent closure on $node_i$.

- For each $node_i$ in G:
  - Perform <u>recursive step 1</u> with $node_i$ and closure set C_i.
- Definition of <u>recursive step</u>, for arbitrary node $node_x$ and closure set C_i:
  - For each node $node_y$ in dependent nodes D_x:
    - If $node_y$ is already in the closure set C_i :
      - Do nothing
    - Else:
      - Add $node_y$ to the closure set C_i
      - Perform the <u>recursive step</u> with $node_y$ and the same closure set C_i.

The propagation cost is then simply the average cardinality of the closure sets C_i, divided by the total number of nodes:

$$PropagationCost = \frac{Average(|C_i|)}{NumNodes} = \frac{\sum_{i=1}^{NumNodes} |C_i|}{NumNodes^2}$$

In the worst case, this algorithm will start with each node, and, for each of those nodes, search through the graph to visit all of the other nodes. Thus, for a system with N files, this algorithm requires on the order of N^2 operations.

This algorithm has the interesting property that it performs better on dependency graphs with low propagation cost. That is to say, if the average propagation cost is 17%, then on average only 17% of the nodes in the graph will be visited in any iteration.

Moreover, it is possible to add an optimization to this algorithm, which prunes the search performed on each node. If the dependent closure is known for a node encountered in the depth-first search, it is not necessary to continue that branch of the search beyond that node – the transitive closure beyond that node is already known! This optimization yields the following improved algorithm:

- For each $node_i$ in G:
    - Begin <u>recursive step</u> with $node_i$ and closure set C_i.

- Definition of <u>recursive step</u>, for arbitrary node $node_x$ and closure set C_i:
    - For each node $node_y$ in dependent nodes Dx:
        - If $node_y$ is already in the closure set C_i :
            - Do nothing
        - Else, if C_y, the closure set for $node_y$, is nonempty, then we know we have already computed the closure for $node_y$
            - We simply add $node_y$'s closure set to the current closure set, and search no further on this branch: Let $C_i = C_i \cup C_y$ [todo: rewrite this with equation editor.]
        - Else:
            - Add $node_y$ to the closure set C_i
            - Perform the <u>recursive step</u> with $node_y$ and the same closure set C_i.

Once the operation is complete, the propagation cost is computed from the closure sets in the same manner as above. This pruning optimization should yield a considerable speedup on graphs with high propagation cost.

Appendix B: Dependency Analysis Toolkit Code

Overview

This appendix presents the source code developed in support of this thesis. It implements the model described in Appendix A, as well as computing various metrics based on the model, and performing data input and output. Each source file is described in a separate section below.

Source code

com.mlamantia.deptools.model.DependencyModel

```
package com.mlamantia.deptools.model;

import java.io.*;
import java.util.*;


/**
* Implementation of the global dependency model.
*/
public class DependencyModel {

  ////////////////////////////
  ////////////////////////////
  ////////////////////////////
  // data members

  /**
  * Global (static singleton) instance of this class
  */
  static DependencyModel mGlobalDependencyModel;

  /**
  *  All of the packages seen so far, hashed by QName (fully qualified name).
  */
  GraphNodeList mAllPackages;

  /**
  *  All of the packages seen so far, hashed by QName (fully qualified name).
  */
  GraphNodeList mNonemptyPackages;

  /**
  *  The root package node, a special, empty node.
  *  It is not contained in mAllPackages.
  */
  PackageGraphNode mRootPackageNode;

  /**
  * Packages prefixes that we are interested in.
```

```
* Classes in these packages will be tracked individually.
* Other classes will be clumped by package.
*/
String[] mInterestingPrefixes;

/**
* Total number of classes.
*/
int mCountClasses=0;

/**
* All the class nodes in the model, in arbitrary order, but
* indexed according to their serialId property.
*/
ArrayList mClassNodesBySerialId;

//////////////////////////////
//////////////////////////////
//////////////////////////////
// Constructor
public DependencyModel(String[] pInterestingPrefixes) {
  mAllPackages = new GraphNodeList();
  mNonemptyPackages = new GraphNodeList();
  mRootPackageNode = new PackageGraphNode(null, null);
  mInterestingPrefixes = pInterestingPrefixes;
  mClassNodesBySerialId = new ArrayList();
}

//////////////////////////////
//////////////////////////////
//////////////////////////////
// Global instance

/**
* Initialize the global instance of the model.
*/
public static void initializeGlobalInstance(
    String[] pInterestingPrefixes ) {
  mGlobalDependencyModel = new DependencyModel(pInterestingPrefixes);
}

/**
* Get the global instance of the model.
*/
public static DependencyModel getGlobalDependencyModel() {
  return mGlobalDependencyModel;
}

//////////////////////////////
//////////////////////////////
//////////////////////////////
// Public methdods

/**
*  Returns the root package node, a special, empty PackageGraphNode,
*  which is the parent of all other PackageGraphNodes.
```

```
*/
public PackageGraphNode getRootPackageNode() {
  return mRootPackageNode;
}

/**
*  Returns the ClassGraphNode corresponding to the given class and package
*  names, either by finding the already existing ClassGraphNode, or by
*  creating it,creating all of the necessary package hierchary nodes along
*  the way.
*  @param pPackageName The fully qualified package name.
*  @param pClassName The local (unqualified) name of the class.
*/
public ClassGraphNode findOrCreateClassGraphNode( String pPackageName,
                                                  String pClassName ) {

  /*
  System.out.println("findOrCreateClassGraphNode: pPackageName="
                     + pPackageName
                     + ", pClassName="
                     + pClassName);
  */

  PackageGraphNode pkgNode = (PackageGraphNode)(
    mAllPackages.get(pPackageName) );
  if ( pkgNode == null ) {

    StringTokenizer tok = new StringTokenizer(pPackageName, ".");

    // Find or create the ancestor packages:
    PackageGraphNode currentParentPkg = mRootPackageNode;
    while ( tok.hasMoreTokens() ) {
      String childPackageLName = tok.nextToken();
      PackageGraphNode childPackage =
      currentParentPkg.getChildPackage( childPackageLName );
      if ( childPackage == null) {
        childPackage = new PackageGraphNode( childPackageLName,
                                             currentParentPkg );
        currentParentPkg.addChildPackage( childPackage );
        mAllPackages.add ( childPackage, childPackage.getQName() );
      }
      currentParentPkg = childPackage;
    }  // END while ( tok.hasMoreTokens() )

    pkgNode = currentParentPkg;
  }

  ClassGraphNode classNode = pkgNode.getChildClass( pClassName );
  if ( classNode == null ) {
    classNode = new ClassGraphNode ( pClassName,
                                     pkgNode,
                                     mCountClasses );
    mNonemptyPackages.add ( pkgNode, pkgNode.getQName() );
    pkgNode.addChildClass( classNode );
    mClassNodesBySerialId.add(classNode);
    mCountClasses++;
```

```
  }

  return classNode;
}

/**
*  Increments all appropriate dependency counts for a dependency from
*  pFromNode to pToNode, including all appropriate package node
*  dependenciesIn both the flattened and hierarchical package
*  dependency models.
*/
public void mutuallyIncrementClassDependencies ( ClassGraphNode pFromNode,
                                                 ClassGraphNode pToNode ) {
  // we don't allow any node to depend upon itself.
  if (pFromNode == pToNode) {
    return;
  }

  pFromNode.incrementOutgoingDependency(pToNode);
  pToNode.incrementIncomingDependency(pFromNode);

  if ( pFromNode.getParentPackage() != pToNode.getParentPackage() ) {
    pFromNode.getParentPackage().incrementOutgoingDependency(
      pToNode.getParentPackage() );
    pToNode.getParentPackage().incrementIncomingDependency(
      pFromNode.getParentPackage() );
    mutuallyIncrementSiblingAncestorDependencies(
      pFromNode.getParentPackage(),
      pToNode.getParentPackage() );
  }
  else {
    pFromNode.incrementOutgoingSameParentDependency(pToNode);
    pToNode.incrementIncomingSameParentDependency(pFromNode);
  }
}

/**
* Create an array containing all of the packages in the flattened
* package graph
*/
public Object[] getNonemptyPackageNodesArray() {
  return mNonemptyPackages.toArray();
}

/**
* Gets the GraphNodeList of top-level packages.
*/
public GraphNodeList getNonemptyPackageNodeList() {
  return mNonemptyPackages;
}

/**
*  Injects a new GraphNodeList for the packages.
*  This is somewhat dangerous -- if it doesn't contain the same
*  PackageGraphNodes as the old (current) one, you can corrupt the model.
*/
```

```
public void putNewNonemptyPackageNodeList(GraphNodeList pList) {
  mNonemptyPackages = pList;
}

/**
* Gets the total count of classes in the model.
*/
public int getClassCount() {
  return mCountClasses;
}

/**
* Gets the ClassGraphNode with a given serialId.
* The following invariant should hold:
*   dependencymodel.getClassNodeBySerialId( classnode.getSerialId() )
*     == classnode
*/
public ClassGraphNode getClassNodeBySerialId(int pId) {
  return (ClassGraphNode)( mClassNodesBySerialId.get(pId) );
}

//////////////////////////////
//////////////////////////////
//////////////////////////////
// helper methods

/**
*  Finds the sibling ancestors of pFromNode and pToNode,
*  and increments the same-parent dependency counts from pFromNode
*  to pToNode.
*/
void mutuallyIncrementSiblingAncestorDependencies(
  PackageGraphNode pFromNode,
  PackageGraphNode pToNode) {

  // We find the closest sibling nodes by climbing up the tree to the root
  // node, pushing each ancestor along the way.  We then pop both stacks
  // until we find a different node on each one.

  Stack fromNodeAncestorStack = new Stack();
  GraphNode treeClimber = pFromNode;
  while (treeClimber != null) {
    fromNodeAncestorStack.push(treeClimber);
    treeClimber = treeClimber.getParentNode();
  }

  Stack toNodeAncestorStack = new Stack();
  treeClimber = pToNode;
  while (treeClimber != null) {
    toNodeAncestorStack.push(treeClimber);
    treeClimber = treeClimber.getParentNode();
  }

  GraphNode fromAncestor = null;
  GraphNode toAncestor = null;
```

```
  while ( fromAncestor == toAncestor
          && !toNodeAncestorStack.empty()
          && !fromNodeAncestorStack.empty() )
  {
    fromAncestor = (PackageGraphNode)(fromNodeAncestorStack.pop());
    toAncestor = (PackageGraphNode)(toNodeAncestorStack.pop());
  }

  // assertion:  fromAncestor and toAncestor are now sibling ancestors
  // of pFromNode and pToNode, or else one of them ran out of stack before
  // they became unequal.
  if ( fromAncestor != toAncestor ) {
    fromAncestor.incrementOutgoingSameParentDependency(toAncestor);
    toAncestor.incrementIncomingSameParentDependency(fromAncestor);
  }
}

//////////////////////////////
//////////////////////////////
//////////////////////////////
// test/debug methods

/**
* Prints the global model as a package and class tree, for debugging
* purposes.
*/
public void printModelTree() {
  System.out.println("");
  System.out.println("******  Full model tree ******");
  System.out.println("");
  printPackageRecursive(mRootPackageNode, 0);
}

/**
* Helper method the global model as a package and class tree, for
* debugging purposes.
*/
private void printPackageRecursive(PackageGraphNode pPkgNode,
                                   int pIndentLevel) {

  // first print the name of this class
  if (pPkgNode.getLName() != null) {
    indentPrintln( pIndentLevel,
                   ""
                   + pPkgNode.getLName()
                   + " ["
                   + pPkgNode.getQName()
                   + "], parent="
                   + pPkgNode.getParentNode().getLName() );
  }
  else {
    indentPrintln( pIndentLevel,
                   "(root node)");
  }

  // print the child classes
```

```
  Object[] childClasses = pPkgNode.getClassNodesArray();
  for (int i=0; i<childClasses.length; i++) {
    ClassGraphNode classNode = (ClassGraphNode) ( childClasses[i] );
    indentPrintln( pIndentLevel + 1,
                   ""
                   + classNode.getLName()
                   + " ["
                   + classNode.getQName()
                   + "], parent="
                   + classNode.getParentPackage().getLName() );
  }

  // then print the child packages
  Object[] childPackages = pPkgNode.getPackageNodesArray();
  for (int i=0; i<childPackages.length; i++) {
    printPackageRecursive( (PackageGraphNode) ( childPackages[i]),
                           pIndentLevel + 1 );
  }
}

/**
* Helper method which prints a String indented to the specifed level.
*/
private void indentPrintln(int pIndentLevel, String pString) {
  for (int i=0; i<pIndentLevel; i++) {
    System.out.print("    ");
  }
  System.out.println(pString);
}

/**
* Create an array containing all of the nonempty packages, in the order
* given by the (possibly sorted) package hierarchy.  The traversal
* is preorder, meaning a parent precedes its child classes.
*/
public Object[] getNonemptyPackageNodesArrayInHierarchyOrder() {
  ArrayList list = new ArrayList(mNonemptyPackages.size());
  getNonemptyPackagesInHierarchyOrderRecursiveHelper( list,
                                                      mRootPackageNode );
  return list.toArray();
}

/**
* Recursive helper function for
* getNonemptyPackageNodesArrayInHierarchyOrder
*/
private void getNonemptyPackagesInHierarchyOrderRecursiveHelper(
  ArrayList pList,
  PackageGraphNode pNode) {
  // First add this PackageGraphNode if appropriate:
  if ( pNode.getChildClassCount() > 0 ) {
    pList.add( pNode );
  }

  // Then recurse into the child packages and do the same:
  Object[] childPackages = pNode.getPackageNodesArray();
```

```
    for (int i=0; i<childPackages.length; i++) {
      PackageGraphNode child = (PackageGraphNode)(childPackages[i]);
      getNonemptyPackagesInHierarchyOrderRecursiveHelper(pList, child);
    }
  }

}
```

com.mlamantia.deptools.model.GraphNode

```
package com.mlamantia.deptools.model;

import java.io.*;
import java.util.*;


/**
* Generic node of a dependency graph; may correspond to a
* package, a class, or another entity.
*/

abstract public class GraphNode {

  ////////////////////////////////
  ////////////////////////////////
  // data members

  /**
  * Default size for dependency tables.
  */
  public static final int mDefaultTableSize = 40;

  /**
  * name of this entity
  */
  protected String mLName;

  /**
  * parent node of this node.
  */
  protected GraphNode mParentNode;

  /**
  * The incoming dependencies -- nodes which depend ON
  * this node.
  * The nodes are the keys, and the strength of the dependency
  * is the corresponding value, as type Integer.
  */
  HashMap mIncomingDependencies;

  /**
  * Total strength of all incoming dependencies.
  */
  int mTotalIncomingDependencies = 0;
```

```
/**
* The outgoing dependencies -- nodes which on which
* THIS NODE DEPENDS.
* The nodes are the keys, and the strength of the dependency
* is the corresponding value, as type Integer.
*/
HashMap mOutgoingDependencies;

/**
* Total strength of all outgoing dependencies.
*/
int mTotalOutgoingDependencies = 0;

/**
* The same-parent incoming dependencies -- nodes which depend ON
* this node and also have the same parent node.
* The nodes are the keys, and the strength of the dependency
* is the corresponding value, as type Integer.
*/
HashMap mIncomingSameParentDependencies;

/**
* Total strength of all same-parent incoming dependencies.
*/
int mTotalIncomingSameParentDependencies = 0;


/**
* The outgoing same-parent dependencies -- nodes which on which
* THIS NODE DEPENDS, and which have the same parent node.
* The nodes are the keys, and the strength of the dependency
* is the corresponding value, as type Integer.
*/
HashMap mOutgoingSameParentDependencies;

/**
* Total strength of all same-parent outgoing dependencies.
*/
int mTotalOutgoingSameParentDependencies = 0;

//////////////////////////////
//////////////////////////////
// constructor

public GraphNode (String pLName, GraphNode pParentNode) {
  mLName = pLName;
  mParentNode = pParentNode;
  mIncomingDependencies = new HashMap(mDefaultTableSize);
  mOutgoingDependencies = new HashMap(mDefaultTableSize);
  mIncomingSameParentDependencies = new HashMap(mDefaultTableSize);
  mOutgoingSameParentDependencies = new HashMap(mDefaultTableSize);
}

//////////////////////////////
//////////////////////////////
```

```
// public interface

/**
* Returns the local (unqualified) name of this node.
*/
public String getLName() {
  return mLName;
}


/**
* Returns the qualified name of this node.
*/
abstract public String getQName();


/**
* Returns the parent node of this graph node.
*/
public GraphNode getParentNode() {
  return mParentNode;
}

//////////////////////////
// incoming dependencies

/**
* Returns true iff this pNode depends on this node.
*/
public boolean hasIncomingDependency(GraphNode pNode) {
  return mIncomingDependencies.containsKey(pNode);
}

/**
* If pNode depends on this node, returns the strength of the same-parent
* dependency.  Otherwise returns zero.
*/
public int getIncomingDependencyStrength(GraphNode pNode) {
  if ( hasIncomingDependency(pNode) ) {
    return ((Integer)(mIncomingDependencies.get(pNode))).intValue();
  }
  return 0;
}

/**
* If pNode depends on this node, increments the strength of the dependency.
* Otherwise creates a new incoming dependency with strength one.
*/
public void incrementIncomingDependency(GraphNode pNode) {
  if ( hasIncomingDependency(pNode) ) {
    int newVal = ((Integer)(mIncomingDependencies.get(pNode))).intValue()
                 + 1;
    mIncomingDependencies.put( pNode, new Integer(newVal) );
    mTotalIncomingDependencies++;
  }
  else {
```

```
    mIncomingDependencies.put( pNode, new Integer(1) );
    mTotalIncomingDependencies++;
  }
}

/**
* Returns the Set of all nodes that satisfy the condition:
* {pNode depends on this node}.
*/
public Set getAllIncomingDependencies() {
  return mIncomingDependencies.keySet();
}

/**
* Returns the total strength of incoming dependencies on this node.
*/
public int getTotalIncomingDependencyStrength() {
  return mTotalIncomingDependencies;
}


//////////////////////////
// outgoing dependencies

/**
* Returns true iff this this node depends on pNode.
*/
public boolean hasOutgoingDependency(GraphNode pNode) {
  return mOutgoingDependencies.containsKey(pNode);
}

/**
* If this node depends on pNode, returns the strength of the dependency.
* Otherwise returns zero.
*/
public int getOutgoingDependencyStrength(GraphNode pNode) {
  if ( hasOutgoingDependency(pNode) ) {
    return ((Integer)(mOutgoingDependencies.get(pNode))).intValue();
  }
  return 0;
}

/**
* If this node depends on pNode, increments the strength of the dependency.
* Otherwise creates a new incoming dependency with strength one.
*/
public void incrementOutgoingDependency(GraphNode pNode) {
  if ( hasOutgoingDependency(pNode) ) {
    int newVal = ((Integer)(mOutgoingDependencies.get(pNode))).intValue()
                 + 1;
    mOutgoingDependencies.put( pNode, new Integer(newVal) );
    mTotalOutgoingDependencies++;
  }
  else {
    mOutgoingDependencies.put( pNode, new Integer(1) );
    mTotalOutgoingDependencies++;
```

```
  }
}

/**
* Returns the Set of all nodes that satisfy the condition:
* {this node depends on pNode}.
*/
public Set getAllOutgoingDependencies() {
  return mOutgoingDependencies.keySet();
}


/**
* Returns the total strength of outgoing dependencies on this node...
*/
public int getTotalOutgoingDependencyStrength() {
  return mTotalOutgoingDependencies;
}

//////////////////////////
// incoming same-parent dependencies

/**
* Returns true iff this pNode depends on this node, and this node
* and pNode have the same parent node.
*/
public boolean hasIncomingSameParentDependency(GraphNode pNode) {
  return mIncomingSameParentDependencies.containsKey(pNode);
}

/**
* If pNode depends on this node, and this node, and pNode have
* the same parent node, returns the strength of the dependency.
* Otherwise returns zero.
*/
public int getIncomingSameParentDependencyStrength(GraphNode pNode) {
  if ( hasIncomingSameParentDependency(pNode) ) {
    return
    ((Integer)(mIncomingSameParentDependencies.get(pNode))).intValue();
  }
  return 0;
}

/**
* If pNode depends on this node, and this node
* and pNode have the same parent node, increments the strength of the same-
* parent dependency.  Otherwise creates a new incoming same-parent
* dependency with strength one.
*/
public void incrementIncomingSameParentDependency(GraphNode pNode) {
  if ( hasIncomingSameParentDependency(pNode) ) {
    int newVal =
      ((Integer)(mIncomingSameParentDependencies.get(pNode))).intValue()
      + 1;
    mIncomingSameParentDependencies.put( pNode, new Integer(newVal) );
    mTotalIncomingSameParentDependencies++;
```

```
  }
  else {
    mIncomingSameParentDependencies.put( pNode, new Integer(1) );
    mTotalIncomingSameParentDependencies++;
  }
}

/**
* Returns the Set of all nodes that satisfy the condition:
* {pNode depends on this node AND this node
  * and pNode have the same parent node}.
*/
public Set getAllIncomingSameParentDependencies() {
  return mIncomingSameParentDependencies.keySet();
}

/**
* Returns the total strength of incoming same-parent dependencies on
* this node.
*/
public int getTotalIncomingSameParentDependencyStrength() {
  return mTotalIncomingSameParentDependencies;
}


///////////////////////////
// outgoing dependencies

/**
* Returns true iff this node depends on pNode, and this node
* and pNode have the same parent node.
*/
public boolean hasOutgoingSameParentDependency(GraphNode pNode) {
  return mOutgoingSameParentDependencies.containsKey(pNode);
}

/**
* If this node depends on pNode, and this node
* and pNode have the same parent node, returns the strength of the
* same-parent dependency.  Otherwise returns zero.
*/
public int getOutgoingSameParentDependencyStrength(GraphNode pNode) {
  if ( hasOutgoingSameParentDependency(pNode) ) {
    return
      ((Integer)(mOutgoingSameParentDependencies.get(pNode))).intValue();
  }
  return 0;
}

/**
* If this node depends on pNode, and this node
* and pNode have the same parent node, increments the strength of the same-
* parent dependency.  Otherwise creates a new incoming same-parent
* dependency with strength one.
*/
public void incrementOutgoingSameParentDependency(GraphNode pNode) {
```

```
    if ( hasOutgoingSameParentDependency(pNode) ) {
      int newVal =
        ((Integer)(mOutgoingSameParentDependencies.get(pNode))).intValue()
         + 1;
      mOutgoingSameParentDependencies.put( pNode, new Integer(newVal) );
      mTotalOutgoingSameParentDependencies++;
    }
    else {
      mOutgoingSameParentDependencies.put( pNode, new Integer(1) );
      mTotalOutgoingSameParentDependencies++;
    }
  }

  /**
  * Returns the Set of all nodes that satisfy the condition:
  * {this node depends on pNode AND this node and pNode have the
  * same parent}.
  */
  public Set getAllOutgoingSameParentDependencies() {
    return mOutgoingSameParentDependencies.keySet();
  }

  /**
  * Returns the total strength of outgoing same-parent dependencies on
  * this node...
  */
  public int getTotalOutgoingSameParentDependencyStrength() {
    return mTotalOutgoingSameParentDependencies;
  }


}
```

com.mlamantia.deptools.model.GraphNodeList

```
package com.mlamantia.deptools.model;

import java.io.*;
import java.util.*;

/**
* Stores a set of GraphNodes
*
*/
public class GraphNodeList {

  ////////////////////////////
  ////////////////////////////
  // data members

  /**
  * Ordered list of child nodes
  */
  ArrayList mList;
```

```
  /**
  * HashSet of child nodes, for fast access
  */
  HashMap mHash;

  ////////////////////////////
  ////////////////////////////
  // Constructor
  public GraphNodeList() {
    mList = new ArrayList();
    mHash = new HashMap();
  }

  ////////////////////////////
  ////////////////////////////
  // Public methdods

  public GraphNode get(String pName) {
    return (GraphNode) ( mHash.get(pName) );
  }

  public void add(GraphNode pNode, String pName) {
    if ( mHash.containsKey( pName ) ) {
      return;
    }
    mHash.put( pName, pNode );
    mList.add( pNode );
  }

  public int size() {
    return mList.size();
  }

  public Object[] toArray() {
    return mList.toArray();
  }

  public Object[] toArray(Object[] pArg) {
    return mList.toArray(pArg);
  }

}
```

com.mlamantia.deptools.model.PackageGraphNode

```
package com.mlamantia.deptools.model;

import java.io.*;
import java.util.*;


/**
* Generic node of a dependency graph, corresponding specifically
```

```
* to a class.
*/
public class PackageGraphNode
extends GraphNode {

  ////////////////////////////
  ////////////////////////////
  // data members

  /**
  * The fully-qualified name of this package.
  */
  String mQName;

  /**
  * Contains all the known classes in this package, hashed by LName.
  */
  GraphNodeList mChildClasses;

  /**
  * Contains all the known subpackages of this package, hashed by LName.
  */
  GraphNodeList mChildPackages;

  /**
  * Contains the total number of child classes in this PackageGraphNode.
  */
  int mCountChildClasses;

  ////////////////////////////
  ////////////////////////////
  // constructor
  /**
  *  Creates a new PackageGraphNode.
  *  @argument pName The fully qualified package name.  If this is
  *      the root package, this should be null.
  *  @argument pParent The PackageGraphNode for the parent package
  *      for this package.  If this is the root package, this
  *      should be null.
  */
  public PackageGraphNode(String pLName, PackageGraphNode pParent) {
    super(pLName, pParent);
    mChildClasses = new GraphNodeList();
    mChildPackages = new GraphNodeList();
    if (pParent != null) {
      if ( pParent.getQName() != null ) {
        mQName = pParent.getQName() + "." + pLName;
      }
      else {
        mQName = pLName;
      }
    }
    else {
      mQName = null;
    }
  }
```

```
//////////////////////////////
//////////////////////////////
// public interface

/**
* Returns the ClassGraphNode corresponding to the given name, if
* it exists, or null if it does not exist.
*/
ClassGraphNode getChildClass(String pName) {
  return (ClassGraphNode) ( mChildClasses.get(pName) );
}

/**
* Adds the given ClassGraphNode as a child of this package, if it
* isn't already a child.  If it is already a child, does nothing.
*/
void addChildClass(ClassGraphNode pNode) {
  if ( mChildClasses.get( pNode.getLName() ) == null ) {
    mCountChildClasses++;
    mChildClasses.add(pNode, pNode.getLName());
  }
}

/**
* Gets the list of classes in this package, as an array of Objects.
*/
public Object[] getClassNodesArray() {
  return mChildClasses.toArray();
}


/**
* Gets the total number of child classes in this PackageGraphNode.
*/
public int getChildClassCount() {
  return mCountChildClasses;
}


/**
* Gets the child classes as a GraphNodeList.
*/
public GraphNodeList getChildClassesGraphNodeList() {
  return mChildClasses;
}

/**
* Replaces the GraphNodeList of child classes.  Danger:  If the
* old and the new GraphNodeList do not contain exactly the same
* GraphNodes (only differing in order) then this can corrupt the model.
*/
public void replaceChildClassesGraphNodeList(GraphNodeList pChildClasses) {
  mChildClasses = pChildClasses;
}
```

```
/**
* Returns the PackageGraphNode corresponding to the given name, if
* it exists, or null if it does not exist.
*/
PackageGraphNode getChildPackage(String pName) {
  return (PackageGraphNode) ( mChildPackages.get(pName) );
}

/**
* Adds the given PackageGraphNode as a child of this package, if it
* isn't already a child.  If it is already a child, does nothing.
*/
void addChildPackage(PackageGraphNode pNode) {
  if ( mChildPackages.get( pNode.getLName() ) == null ) {
    mChildPackages.add(pNode, pNode.getLName());
  }
}

/**
* Gets the list of subpackages in this package, as an array of Objects.
*/
public Object[] getPackageNodesArray() {
  return mChildPackages.toArray();
}

/**
* Gets the child packages as a GraphNodeList.
*/
public GraphNodeList getChildPackagesGraphNodeList() {
  return mChildPackages;
}

/**
* Replaces the GraphNodeList of child packages.  Danger:  If the
* old and the new GraphNodeList do not contain exactly the same
* GraphNodes (only differing in order) then this can corrupt the model.
*/
public void replaceChildPackagesGraphNodeList(GraphNodeList pChildPackages)
{
  mChildPackages = pChildPackages;
}

/**
* Gets the total number of child classes in this PackageGraphNode.
*/
public int getChildPackageCount() {
  return mChildPackages.size();
}

/**
* Get the fully qualified name of this package.
*/
public String getQName() {
  return mQName;
```

```
  }

}
```

com.mlamantia.deptools.model.ClassGraphNode

```
package com.mlamantia.deptools.model;

import java.io.*;
import java.util.*;


/**
* Generic node of a dependency graph, corresponding specifically
* to a class.
*/

public class ClassGraphNode
extends GraphNode {

  ////////////////////////////
  ////////////////////////////
  // data members
  PackageGraphNode mParentPackage;

  int mSerialId;

  ////////////////////////////
  ////////////////////////////
  // constructor
  public ClassGraphNode(String pName,
                        PackageGraphNode pParent,
                        int pSerialId) {
    super(pName, pParent);
    mParentPackage = pParent;
    mSerialId = pSerialId;
  }

  ////////////////////////////
  ////////////////////////////
  // public interface
  /**
  * Returns the PackageGraphNode that contains this ClassGraphNode.
  */
  public PackageGraphNode getParentPackage() {
    return mParentPackage;
  }

  /**
  * Returns the serialId of this ClassGraphNode.
  */
  public int getSerialId() {
    return mSerialId;
  }
```

```
  /**
  * If pNode depends on this node, increments the strength of the dependency.
  * Otherwise creates a new incoming dependency with strength one.
  */
  public void incrementIncomingDependency(ClassGraphNode pNode) {
    super.incrementIncomingDependency(pNode);
  }

  /**
  * If this node depends on pNode, increments the strength of the dependency.
  * Otherwise creates a new outgoing dependency with strength one.
  */
  public void incrementOutgoingDependency(ClassGraphNode pNode) {
    super.incrementOutgoingDependency(pNode);
  }


  /**
  * Gets the fully qualified (package + class) name for this ClassGraphNode.
  */
  public String getQName() {
    return mParentPackage.getQName() + "." + getLName();
  }

}
```

com.mlamantia.deptools.controller.TopologicalSort

```
package com.mlamantia.deptools.controller;

import com.mlamantia.deptools.model.*;
import java.io.*;
import java.util.*;


/**
* Methods which take a GraphNodeList, and heuristically reorder the nodes
* so that they are (hopefully) in a reasonably sorted order.
*/
public class TopologicalSort {

  public static final boolean mbDebug = true;

  public static double mBonusPointsDiscount = 0.0;

  public static double mBonusPointsDecayFactor = 0.85;

  /**
  * Performs a "bottom-up" heuristic topological sort on the graph in
  * GraphNodeList pNodeListIn, starting with the least dependent
  * (bottom-level) and going to the most dependent.
  * @returns A new GraphNodeList, topologically sorted.
  * @argument pNodeListIn The GraphNodeList to sort
  * @pbHierarchical If true, this method uses the "Same-parent" or
  * sibling-node dependency graph, instead of the flattened dependency graph.
```

```
*/
public static GraphNodeList topSortLeastToMostDependent(
                                GraphNodeList pNodeListIn,
                                boolean pbHierarchical) {

  // placeholder for result
  GraphNodeList result = new GraphNodeList();

  // get typed array containing the original ordering from pNodeListIn
  int numNodes = pNodeListIn.size();
  GraphNode[] originalNodes = new GraphNode[ numNodes ];
  originalNodes = (GraphNode[]) ( pNodeListIn.toArray(originalNodes) );

  // Create the HashMaps which keep track of the outgoing dependency
  // strength for the nodes, and the bonus points for the nodes...
  ArrayList nodesToBeProcessed = new ArrayList(numNodes);
  HashMap nodesToOutgoingDepWeights = new HashMap(numNodes);
  HashMap nodesToBonusPoints = new HashMap(numNodes);
  for (int i=0; i<originalNodes.length; i++) {
    GraphNode curNode = originalNodes[i];
    nodesToBeProcessed.add( curNode );
    if (pbHierarchical) {
      nodesToOutgoingDepWeights.put(
        curNode,
        new Double(
          curNode.getTotalOutgoingSameParentDependencyStrength() ) );
    }
    else {
      nodesToOutgoingDepWeights.put(
        curNode,
        new Double(curNode.getTotalOutgoingDependencyStrength() ) );
    }
    nodesToBonusPoints.put( curNode,
                            new Double (0) );
  }


  // The body of the topsort algorithm:
  while ( nodesToBeProcessed.size() > 0 ) {

    // find best node to choose next:
    GraphNode bestNode = null;
    int bestNodeIndex = -1;
    double bestScore = Double.MAX_VALUE;
    for (int i=0; i<nodesToBeProcessed.size(); i++) {
      GraphNode curNode = (GraphNode) (nodesToBeProcessed.get(i));
      double outgoingWeight =
        ((Double)(nodesToOutgoingDepWeights.get(curNode))).doubleValue();
      double bonusPoints =
        ((Double)(nodesToOutgoingDepWeights.get(curNode))).doubleValue();
      double score = outgoingWeight - (mBonusPointsDiscount * bonusPoints);

      if ( bestNode == null || score < bestScore ) {
        bestNode = curNode;
        bestNodeIndex = i;
        bestScore = score;
```

```
      }
    }  // END finding best node


    // decay all the bonus points:
    Iterator iter = nodesToBonusPoints.keySet().iterator();
    while (iter.hasNext()) {
      GraphNode curNode = (GraphNode) (iter.next());
      double bonus =
        ((Double)(nodesToBonusPoints.get(curNode))).doubleValue();
      bonus = bonus * mBonusPointsDecayFactor;
      nodesToBonusPoints.put( curNode, new Double(bonus) );
    }  // END decaying bonus points

    // remove the best node from the graph and update strengths on
    // remaining nodes...
    nodesToBeProcessed.remove(bestNodeIndex);
    Iterator incomingDepsIter = null;
    if ( pbHierarchical ) {
      incomingDepsIter =
        bestNode.getAllIncomingSameParentDependencies().iterator();
    }
    else {
      incomingDepsIter = bestNode.getAllIncomingDependencies().iterator();
    }
    while (incomingDepsIter.hasNext()) {
      GraphNode dependentNode = (GraphNode) (incomingDepsIter.next());

      int dependencyStrength = 0;
      if ( pbHierarchical ) {
        dependencyStrength =
          bestNode.getIncomingSameParentDependencyStrength(dependentNode);
      }
      else {
        dependencyStrength =
          bestNode.getIncomingDependencyStrength(dependentNode);
      }
      double dependentNodeOutgoingStrength =
        ((Double)(nodesToOutgoingDepWeights.get(dependentNode)))
        .doubleValue();
      dependentNodeOutgoingStrength =
        dependentNodeOutgoingStrength - dependencyStrength;
      nodesToOutgoingDepWeights.put(
        dependentNode,
        new Double( dependentNodeOutgoingStrength ) );
    }  //  END remove the best node from the graph and update strengths
       //  on remaining nodes.

    // add the best node to the result:
    result.add(bestNode, bestNode.getQName());

  }  // END while ( nodesToBeProcessed.size() > 0 )

  return result;

}  // END function topSortLeastToMostDependent(...)
```

```
/**
* Performs a "bottom-up" heuristic topological sort on the flat package-
* dependency graph in the global depenendency model.
*/
public static void topSortGlobalFlatPackageGraph() {

  DependencyModel model = DependencyModel.getGlobalDependencyModel();
  GraphNodeList packages = model.getNonemptyPackageNodeList();
  GraphNodeList newPackageOrder =
    TopologicalSort.topSortLeastToMostDependent(packages, false);
  model.putNewNonemptyPackageNodeList(newPackageOrder);
}

/**
* Recursively performs a "bottom-up" heuristic topological sort on the
* sibling dependency graph at each level of the hierchical model.
*/
public static void topSortGlobalHierarchicalModel() {
  DependencyModel model = DependencyModel.getGlobalDependencyModel();
  visitAndTopSortRecursively( model.getRootPackageNode() );
}

/**
*  Helper method to topSortGlobalFlatPackageGraph.
*/
private static void visitAndTopSortRecursively(
                        PackageGraphNode pPackageNode ) {

  if (mbDebug) {
    System.out.println("TopologicalSort recursive on package: "
                       + pPackageNode.getQName() );
  }

  // First topsort the direct child classes:
  GraphNodeList newChildClassesOrder = topSortLeastToMostDependent
    ( pPackageNode.getChildClassesGraphNodeList(),
      true );
  pPackageNode.replaceChildClassesGraphNodeList(newChildClassesOrder);

  // Then topsort the child packages:

  GraphNodeList newChildPackagesOrder = topSortLeastToMostDependent
  ( pPackageNode.getChildPackagesGraphNodeList(),
    true );
  pPackageNode.replaceChildPackagesGraphNodeList(newChildPackagesOrder);

  // Finally recurse into the child packages and topsort them:
  Object[] childPackages = pPackageNode.getPackageNodesArray();
  for (int i=0; i<childPackages.length; i++) {
    PackageGraphNode child = (PackageGraphNode)(childPackages[i]);
    visitAndTopSortRecursively(child);
  }

}
```

```
}
```

com.mlamantia.deptools.controller.Metrics

```
package com.mlamantia.deptools.controller;

import com.mlamantia.deptools.model.*;
import java.io.*;
import java.util.*;


/**
* Methods which calculate metrics on a dependency graph...
*/
public class Metrics {

  /**
  * Whether to output debug information.
  */
  public static final boolean mDebug = true;

  /**
  * Compute the average propagation cost for the whole dependency graph.
  * The propagation cost is the average number of nodes in the transitive
  * closure divided by the total number of nodes in the graph.
  */
  public static double computePropagationCost() {

    DependencyModel globalModel = DependencyModel.getGlobalDependencyModel();

    int classCount = globalModel.getClassCount();

    // We will need a bitfield for each class node, to store its
    // dependency closure.
    // We will stored them hashed by ClassGraphNode.

    double sumOfPropagationRatios = 0.0;

    HashMap classNodesToBitSets = new HashMap(classCount);

    for (int i=0; i<classCount; i++) {
      ClassGraphNode curClassNode = globalModel.getClassNodeBySerialId(i);

      BitSet curBitSet = new BitSet(classCount);
      classNodesToBitSets.put(curClassNode, curBitSet);

      doDepthFirstSearch(curClassNode, curBitSet, classNodesToBitSets);

      int closureCount = 0;
      for (int j=0; j<classCount; j++) {
        if ( curBitSet.get(j) ) {
          closureCount++;
        }
      }
```

```
    double propagationCostForThisClass = ( (double)(closureCount)
                                          / (double)(classCount) );
    sumOfPropagationRatios = (sumOfPropagationRatios
                              + propagationCostForThisClass);

    System.out.println("" + curClassNode.getQName()
                + ": "
                + propagationCostForThisClass);

  }   // END for (int i=0; i<classCount; i++)

  return sumOfPropagationRatios / (double)(classCount);

}   // END function computePropagationCost


/**
*  Recursive body of the DFS for computing propagation cost.
*/
private static void doDepthFirstSearch(ClassGraphNode pCurClassNode,
                                       BitSet pCurBitSet,
                                       HashMap pClassNodesToBitSets) {

  // First add the current ClassGraphNode to the current BitSet
  pCurBitSet.set(pCurClassNode.getSerialId());

  // Now examine its dependent nodes:
  Iterator dependentNodesIter =
    pCurClassNode.getAllIncomingDependencies().iterator();

  while ( dependentNodesIter.hasNext() ) {

    ClassGraphNode dependentClassNode =
      (ClassGraphNode) ( dependentNodesIter.next() );

    // If we already have the dependent node in the closure, do nothing:
    if ( pCurBitSet.get( dependentClassNode.getSerialId() ) ) {
      ;
    }
    // If the dependent node already has a BitSet for its closure, we just
    // "OR" that BitSet into the current BitSet, and there's no need to
    // recurse:
    else if ( pClassNodesToBitSets.containsKey( dependentClassNode ) ) {
                pCurBitSet.or( (BitSet)
                    ( pClassNodesToBitSets.get( dependentClassNode ) ) );
    }
    // Else, recurse on the dependent node and continue the DFS...
    else {
      doDepthFirstSearch( dependentClassNode,
                          pCurBitSet,
                          pClassNodesToBitSets );
    }  // END if ... else if ... else

  } // END while ( dependentNodesIter.hasNext() )

}  // END function doDepthFirstSearch
```

```
/**
*  Calculates the clustered cost based on the current package-level
*  clustering.
*/
public static long computeClusteredCost(long pLambda) {

  DependencyModel globalModel = DependencyModel.getGlobalDependencyModel();
  int classCount = globalModel.getClassCount();
  long cacheCrossPackageCost = longPower(classCount, pLambda);

  long result = 0;

  for (int i=0; i<classCount; i++) {
    ClassGraphNode curClassNode = globalModel.getClassNodeBySerialId(i);
    PackageGraphNode curPackage = curClassNode.getParentPackage();

    long cacheIntraPackageCost = longPower(curPackage.getChildClassCount(),
                                           pLambda);

    Iterator dependentNodesIter =
      curClassNode.getAllOutgoingDependencies().iterator();
    while ( dependentNodesIter.hasNext() ) {
      ClassGraphNode dependentClassNode =
        (ClassGraphNode) ( dependentNodesIter.next() );
      PackageGraphNode dependentPackageNode =
        dependentClassNode.getParentPackage();

      if ( curPackage == dependentPackageNode ) {
        result = result + cacheIntraPackageCost;
      }
      else {
        result = result + cacheCrossPackageCost;
      }
    }  // END iteration over dependent nodes

  } // END iteration over all classes

  return result;

}


/**
* Raises first arg (pBase) to the power of second argument (pPower).
*/
static long longPower(long pBase, long pPower) {
  long result = 1;
  for (int i=0; i<pPower; i++) {
    result = result * pBase;
  }
  return result;
}
```

```
  /**
  *  Calculates the total number of class-to-class dependencies in the graph.
  */
  public static long computeNumberOfDependencies() {
    DependencyModel globalModel = DependencyModel.getGlobalDependencyModel();
    int classCount = globalModel.getClassCount();

    long totalNumberOfDeps = 0;

    for (int i=0; i<classCount; i++) {
      ClassGraphNode curClassNode = globalModel.getClassNodeBySerialId(i);

      Iterator dependentNodesIter =
        curClassNode.getAllOutgoingDependencies().iterator();
      while ( dependentNodesIter.hasNext() ) {
        dependentNodesIter.next();
        totalNumberOfDeps++;
      }  // END iteration over dependent nodes

    } // END iteration over all classes -- for (int i=0; i<classCount; i++)

    return totalNumberOfDeps;
  }

}
```

com.mlamantia.deptools.io.DependencyParserBase

```
package com.mlamantia.deptools.io;

import java.io.*;
import java.util.*;
import com.mlamantia.deptools.model.*;


abstract public class DependencyParserBase {

  ////////////////////////////
  ////////////////////////////
  // properties

  /**
  *  Turns on debugging output.
  */
  public static final boolean mbDebug = false;

  String[] mIgnorePrefixes = new String[] { "java." };
  /**
  * Sets the package prefixes for the parser to ignore.
  */
  public void setIgnorePrefixes(String[] pPrefixes) {
    mIgnorePrefixes = pPrefixes;
  }
```

```
  String[] mInterestingPrefixes = new String[] {"org.apache" } ;
  /**
  * Sets the package prefixes for the parser to ignore.
  */
  public void setInterestingPrefixes(String[] pPrefixes) {
    mInterestingPrefixes = pPrefixes;
  }

  boolean mbOnlyAddInterestingPrefixes = true;
  /**
  * Determines whether only packages considered "interesting" should be
  * added to the DependencyModel.
  */
  public void setOnlyAddInterestingPrefixes(boolean pOnlyInteresting) {
    mbOnlyAddInterestingPrefixes = pOnlyInteresting;
  }

  File mPackageListFile = null;
  /**
  * Sets a file which will receive a list of packages whose
  * dependencies are being modeled.
  */
  public void setPackageListFile(File pFile) {
    mPackageListFile = pFile;
  }

  File mIgnoredListFile = null;

  /**
  * Sets a file which will receive a list of packages whose dependencies
  * are being ignored.
  */
  public void setIgnoredListFile(File pFile) {
    mIgnoredListFile = pFile;
  }

  ////////////////////////////////
  ////////////////////////////////
  // public methods

  /**
  * Parse a file in the format output from DependencyFinder's
  * dependentsToText or hideIncomingDependenciesToTest xsl.
  */
  abstract public void parseFile(File pFile)
  throws IOException;

}
```

com.mlamantia.deptools.io.DepFinderXmlParser

```
package com.mlamantia.deptools.io;

import java.io.*;
```

```
import java.util.*;
import com.mlamantia.deptools.model.*;

import org.xml.sax.*;
import javax.xml.parsers.*;


public class DepFinderXMLParser
extends DependencyParserBase
implements ContentHandler
{

  ////////////////////////////////
  ////////////////////////////////
  // Data members

  /**
  * total count of dependencies, for progress output
  */
  long mCountDeps = 0;

  /**
  * SAX parser instance
  */
  XMLReader mXmlReader;

  /**
  * The current parser state
  */
  int mParseState;

  /**
  * Parser state that says we are in the root node
  */
  private static final int IN_ROOT=1;

  /**
  * Parser state that says we are in the package node
  */
  private static final int IN_PACKAGE=2;

  /**
  * Parser state that says we are in the class node
  */
  private static final int IN_CLASS=3;

  /**
  * Parser state that says we are in the feature node
  */
  private static final int IN_FEATURE=4;

  /**
  * Current package name
  */
  private String mPackageName = null;
```

```
/**
* Whether the current package is one that we will ignore...
*/
private boolean mbIgnorePackage = false;

/**
* Current class name
*/
private String mClassName = null;

/**
* Current feature name
*/
private String mFeatureName = null;

/**
* Current class ClassGraphNode
*/
private ClassGraphNode mCurrentClassNode = null;

/**
* Current dependency type
*/
private int mDependencyType = -1;

/**
* Dependency type for a feature dependency
*/
private static final int DEPENDENCY_TYPE_FEATURE = 1;

/**
* Dependency type for a class dependency
*/
private static final int DEPENDENCY_TYPE_CLASS = 2;

/**
* Whether we are currently in a "name" element:
*/
private boolean mbInNameElement = false;

/**
* Whether we are currently in a "outbound" element:
*/
private boolean mbInOutboundElement = false;

/**
* the text content of an outbound dependency node
*/
private String mOutboundContent = null;

/**
* the text content of a "name" node
*/
private String mNameContent = null;

/**
```

```
* Set of packages that are being modeled (as String).
*/
private TreeSet mPackagesModeledSet;

/**
* Set of packages that are being Ignored (as String).
*/
private TreeSet mPackagesIgnoredSet;

////////////////////////////
////////////////////////////
// Constructor

public DepFinderXMLParser() {

  mPackagesModeledSet = new TreeSet();
  mPackagesIgnoredSet = new TreeSet();

  try {
    SAXParserFactory factory = SAXParserFactory.newInstance ();
    factory.setNamespaceAware (true);
    SAXParser parser = factory.newSAXParser ();
    mXmlReader = parser.getXMLReader ();
  }
  catch (ParserConfigurationException exc) {
    throw new RuntimeException (exc.toString ());
  }
  catch (SAXException exc) {
    throw new RuntimeException (exc.toString ());
  }

  mXmlReader.setContentHandler(this);

  // we deactivate validation because we trust the output from
  // DependencyFinder.
  try {
    mXmlReader.setFeature("http://xml.org/sax/features/validation",
                false);
  } catch (SAXException e) {
    ;
  }


}  // END constructor DepFinderXMLParser()

////////////////////////////
////////////////////////////
// public methods

/**
* Parse a file in the format output from DependencyFinder's
* dependentsToText or hideIncomingDependenciesToText xsl.
*/
public void parseFile(File pFile)
throws IOException {
```

```
  DependencyModel.initializeGlobalInstance( mInterestingPrefixes );

  FileInputStream fis = new FileInputStream(pFile);
  InputSource inSource = new InputSource(fis);

  try {
    mXmlReader.parse(inSource);
  }
  catch (SAXException exc) {
    throw new RuntimeException(
      "An exception was encountered while parsing the XML"
      + " dependencies file " + pFile + ":  "
      + exc.toString());
  }
  finally {
    if (fis != null) {
      try {
        fis.close ();
      }
      catch (IOException exc) {}
    }
  }

  if ( mPackageListFile != null ) {
    FileOutputStream fos = new FileOutputStream(mPackageListFile);
    PrintWriter writer = new PrintWriter(fos);

    Iterator iter = mPackagesModeledSet.iterator();
    while (iter.hasNext()) {
      writer.println( (String)(iter.next()) );
    }  // END while (iter.hasNext())

    writer.close();
    fos.close();
  }  // if ( mPackageListFile != null )

  if ( mIgnoredListFile != null ) {
    FileOutputStream fos = new FileOutputStream(mIgnoredListFile);
    PrintWriter writer = new PrintWriter(fos);

    Iterator iter = mPackagesIgnoredSet.iterator();
    while (iter.hasNext()) {
      writer.println( (String)(iter.next()) );
    }  // END while (iter.hasNext())

    writer.close();
    fos.close();
  }  // if ( mIgnoredListFile != null )

  if (mbDebug) {
    DependencyModel.getGlobalDependencyModel().printModelTree();
  }
}

//////////////////////////////
```

```
//////////////////////////////
// Implementation of Sax parser interface.

/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
public void startElement (String namespaceURI, String localName,
              String qName, Attributes atts)
throws SAXException

{

  if (localName.equals("dependencies")) {
    // this is the root node; requires no action
    mParseState = IN_ROOT;

  }
  else if (localName.equals("package")) {
    mParseState = IN_PACKAGE;
    assert mPackageName == null;

  }
  else if (localName.equals("class")) {
    mParseState = IN_CLASS;
    assert mClassName == null;

  }
  else if (localName.equals("feature")) {
    mParseState = IN_FEATURE;
    assert mFeatureName == null;
  }
  else if (localName.equals("outbound")) {
    String typeAttr = atts.getValue("type");
    assert typeAttr != null;

    if (typeAttr.equals("feature")) {
      mDependencyType = DEPENDENCY_TYPE_FEATURE;
    }
    else if (typeAttr.equals("class")) {
      mDependencyType = DEPENDENCY_TYPE_CLASS;
    }
    else {
      throw new RuntimeException("Unexpected dependency type");
    }
    mbInOutboundElement = true;
    mOutboundContent = "";
  }
  else if (localName.equals("inbound")) {
    // we are ignoring inbound dependencies, so do nothing
  }
  else if (localName.equals("name")) {
    mbInNameElement = true;
    mNameContent = "";
  }

}  // END method startElement
```

```
/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
public void endElement (String namespaceURI, String localName,
                        String qName)
throws SAXException
{
  if (localName.equals("dependencies")) {
    // this is the root node; requires no action
    mParseState = IN_ROOT;

  }
  else if (localName.equals("package")) {
    assert mParseState == IN_PACKAGE;
    mParseState = IN_ROOT;
    mPackageName = null;

  }
  else if (localName.equals("class")) {
    assert mParseState == IN_CLASS;
    mParseState = IN_PACKAGE;
    mClassName = null;

  }
  else if (localName.equals("feature")) {
    assert mParseState == IN_FEATURE;
    mParseState = IN_CLASS;
    mFeatureName = null;

  }
  else if (localName.equals("outbound")) {

    if ( !mbIgnorePackage ) {

      String dependeePkgName = null;
      String dependeeClassName = null;
      String qualifiedClassName = null;

      if (mbDebug) {
        System.out.println(
          "  processing outbound " +
          ( mDependencyType == DEPENDENCY_TYPE_CLASS
            ? "class "
            : "feature " )
          + "dep to \"" + mOutboundContent + "\"");
      }


      if ( mDependencyType == DEPENDENCY_TYPE_CLASS ) {
        qualifiedClassName = mOutboundContent;
      }
      else if ( mDependencyType == DEPENDENCY_TYPE_FEATURE ) {
        // If the dependency points to a function, trim that off the
        // arguments:
```

```
  if ( mOutboundContent.indexOf("(") != -1 ) {
    mOutboundContent = mOutboundContent.substring(
      0, mOutboundContent.indexOf("("));
  }
  qualifiedClassName = mOutboundContent.substring(
    0, mOutboundContent.lastIndexOf('.') );
}

StringTokenizer tok = new StringTokenizer(qualifiedClassName, ".");
// the unqualified class name is the very last token after a '.'
while (tok.hasMoreTokens()) {
  String token = tok.nextToken();
  if (tok.hasMoreTokens()) {
    if (dependeePkgName == null) {
      dependeePkgName = token;
    }
    else {
      dependeePkgName = dependeePkgName + "." + token;
    }
  }
  else {
    dependeeClassName = token;
  }
}  // END while (tok.hasMoreTokens())

boolean bIgnoreDependee = false;
if ( mIgnorePrefixes != null ) {
  for (int i=0; i<mIgnorePrefixes.length; i++) {
    if ( dependeePkgName.startsWith(mIgnorePrefixes[i]) ) {
      bIgnoreDependee = true;
      break;
    }
  }
}  // END if ( mIgnorePrefixes != null )
if ( !bIgnoreDependee && mbOnlyAddInterestingPrefixes ) {
  bIgnoreDependee = true;
  for (int i=0; i<mInterestingPrefixes.length; i++) {
    if ( dependeePkgName.startsWith(mInterestingPrefixes[i]) ) {
      bIgnoreDependee = false;
      break;
    }
  }
}  // END if ( !bIgnoreDependee && mbOnlyAddInterestingPrefixes )


if (bIgnoreDependee) {
  mPackagesIgnoredSet.add(dependeePkgName);
}
else {
  DependencyModel model = DependencyModel.getGlobalDependencyModel();
  mPackagesModeledSet.add(dependeePkgName);
  ClassGraphNode dependeeClassNode = model.
  findOrCreateClassGraphNode(dependeePkgName, dependeeClassName);

  model.mutuallyIncrementClassDependencies( mCurrentClassNode,
                                            dependeeClassNode );
```

```
    }  // END if (bIgnoreDependee) ... else ...

    if (mbDebug) {
      System.out.println("  --> "
                 + ( bIgnoreDependee ? "IGNORED " : "ACCEPTED " )
                 + "package='"
                 + dependeePkgName
                 + "', class='"
                 + dependeeClassName
                 + "'.");
    }  // END if (mbDebug)

  } // END if ( !mbIgnorePackage )

  if ( mCountDeps++ % 10000 == 0 ) {
    System.out.println(" ... " + mCountDeps);
  }

  mbInOutboundElement = false;

}
else if (localName.equals("inbound")) {
  // we are ignoring inbound dependencies, so do nothing
}
else if (localName.equals("name")) {
  mbInNameElement = false;


  if ( mParseState == IN_FEATURE ) {
    mFeatureName = mNameContent;
  }
  else if ( mParseState == IN_CLASS ) {
    StringTokenizer tok = new StringTokenizer(mNameContent, ".");
    // the unqualified class name is the very last token after a '.'
    while (tok.hasMoreTokens()) {
      mClassName = tok.nextToken();
    }

    if ( mbIgnorePackage ) {
      mCurrentClassNode = null;
      if (mbDebug) {
        System.out.println("IGNORE class='"
                   + mNameContent);

      }  // END if (mbDebug)
    }
    else {
      mCurrentClassNode = DependencyModel.getGlobalDependencyModel().
      findOrCreateClassGraphNode(mPackageName, mClassName);
      if (mbDebug) {
        System.out.println(
          "package='"
          + mCurrentClassNode.getParentPackage().getQName()
          + "', packageLocal='"
          + mCurrentClassNode.getParentPackage().getLName()
```

```
            + "', class='"
            + mCurrentClassNode.getLName()
            + "'.");
        }  // END if (mbDebug)
      } // END if (mbIgnorePackage) ... else ...
    }
    else if ( mParseState == IN_PACKAGE ) {
      // check if this package is being ignored:
      mbIgnorePackage = false;
      if ( mIgnorePrefixes != null ) {
        for (int i=0; i<mIgnorePrefixes.length; i++) {
          if ( mNameContent.startsWith(mIgnorePrefixes[i]) ) {
            mbIgnorePackage = true;
            break;
          }
        }
      }  // END if ( mIgnorePrefixes != null )
      if ( !mbIgnorePackage && mbOnlyAddInterestingPrefixes ) {
        mbIgnorePackage = true;
        for (int i=0; i<mInterestingPrefixes.length; i++) {
          if ( mNameContent.startsWith(mInterestingPrefixes[i]) ) {
            mbIgnorePackage = false;
            break;
          }
        }
      }  // END if ( !mbIgnorePackage && mbOnlyAddInterestingPrefixes )

      if ( mbIgnorePackage ) {
        mPackagesIgnoredSet.add(mNameContent);
        mPackageName = null;
        if (mbDebug) {
          System.out.println("IGNORE package='"
                     + mNameContent);

        }  // END if (mbDebug)
      }
      else {
        mPackagesModeledSet.add(mNameContent);
        mPackageName = mNameContent;
        if (mbDebug) {
          System.out.println("BEGIN package='"
                     + mNameContent);

        }  // END if (mbDebug)
      }

    }  // END if (mParseState == ...)


  }  // END else if (localName.equals("name"))

}  // END method endElement
```

```
/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
public void characters (char ch[], int start, int length)
throws SAXException
{
  String content = new String(ch, start, length);
  content = content.trim();

  if ( mbInNameElement ) {
    mNameContent = mNameContent + content;
  }  // END if (mbInNameElement) ...
  else if ( mbInOutboundElement ) {
    mOutboundContent = mOutboundContent + content;
  }

}



/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
public void startDocument()
throws SAXException {
  ;
}

/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
public void endDocument()
throws SAXException {
  ;
}

/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
public void startPrefixMapping(String prefix,
                  String uri)
throws SAXException {
  ;
}

/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
public void endPrefixMapping(String prefix)
throws SAXException {
  ;
}

/**
* implementation of org.xml.sax.ContentHandler interface method.
*/
```

```
  public void ignorableWhitespace(char[] ch,
                  int start,
                  int length)
  throws SAXException {
    ;
  }

  /**
  * implementation of org.xml.sax.ContentHandler interface method.
  */
  public void processingInstruction(String target,
                    String data)
  throws SAXException {
    ;
  }

  /**
  * implementation of org.xml.sax.ContentHandler interface method.
  */
  public void setDocumentLocator(Locator locator) {
    ;
  }

  /**
  * implementation of org.xml.sax.ContentHandler interface method.
  */
  public void skippedEntity(String name)
  throws SAXException {
    ;
  }

}
```

com.mlamantia.deptools.io.DSASFormatWriter

```
package com.mlamantia.deptools.io;

import java.io.*;
import java.util.*;
import com.mlamantia.deptools.model.*;


public class DSASFormatWriter {

  ////////////////////////////
  ////////////////////////////
  // data members

  /**
  * Whether to output debugging info.
  */
  public static final boolean mbDebug = false;

  /**
```

```
*  File to write the output to.
*/
File mFilesOutputFile;

/**
*  File to write the output to.
*/
File mDepsOutputFile;

//////////////////////////////
//////////////////////////////
// constructor
public DSASFormatWriter() {
}

//////////////////////////////
//////////////////////////////
// public methods

/**
* Sets file to write to
*/
public void setFilesOutputFile (File pFile) {
  mFilesOutputFile = pFile;
}

/**
* Gets file to write to.
*/
public File getFilesOutputFile () {
  return mFilesOutputFile;
}

/**
* Sets file to write to
*/
public void setDepsOutputFile (File pFile) {
  mDepsOutputFile = pFile;
}

/**
* Gets file to write to.
*/
public File getDepsOutputFile () {
  return mDepsOutputFile;
}

/**
* Writes the packages in the global DependencyModel as a DSM
* importable by Excel (comma-delimited).
* Assumes the model has already been initialized and populated.
*/
public void run( boolean pbHierarchical, boolean pbAliasNames )
throws IOException {

  if (mbDebug) {
```

```
  System.out.println("Writing DSAS format dependencies to file "
            + mFilesOutputFile);
}

FileOutputStream filesFos = new FileOutputStream(mFilesOutputFile);
PrintWriter filesWriter = new PrintWriter(filesFos);

FileOutputStream depsFos = new FileOutputStream(mDepsOutputFile);
PrintWriter depsWriter = new PrintWriter(depsFos);

DependencyModel globalModel = DependencyModel.getGlobalDependencyModel();
Object[] packageArray = null;
if ( pbHierarchical ) {
  packageArray =
    globalModel.getNonemptyPackageNodesArrayInHierarchyOrder();
}
else {
  packageArray = globalModel.getNonemptyPackageNodesArray();
}

HashMap namesToAliasesMap = null;
if ( pbAliasNames ) {
  namesToAliasesMap = createNamesToAliasesMap(globalModel);
}

// Create an array of all the classes in all the packages
ArrayList classNodesList = new ArrayList(1000);
HashMap classNodesToIndexMap = new HashMap(1000);
int cumulativeClassIndex = 0;

for (int i=0; i<packageArray.length; i++) {
  PackageGraphNode curPackage = (PackageGraphNode)(packageArray[i]);

  Object[] classesInPkgArray = curPackage.getClassNodesArray();

  for (int j=0; j<classesInPkgArray.length; j++) {
    classNodesList.add(classesInPkgArray[j]);
    classNodesToIndexMap.put(classesInPkgArray[j],
                 new Integer(cumulativeClassIndex));

    if (mbDebug) {
      System.out.println(""
                 + cumulativeClassIndex
                 + ": "
                 +((ClassGraphNode)(classesInPkgArray[j])).getQName());
    }
    cumulativeClassIndex++;
  }  // END for j...

} // END for i...

// Now iterate through all classes, and output both the files list and
// the dependencies/strengths
for (int i=0; i<classNodesList.size(); i++) {

  ClassGraphNode curClassNode = (ClassGraphNode) (classNodesList.get(i));
```

```
    String fullyQualifiedName;
    if ( pbAliasNames ) {
      fullyQualifiedName = (String)
        (namesToAliasesMap.get(curClassNode.getQName()));
    }
    else {
      fullyQualifiedName = curClassNode.getQName();
    }
    fullyQualifiedName = fullyQualifiedName.replace('.', '/');
    filesWriter.println(fullyQualifiedName);

    if (mbDebug) {
      System.out.println(""
                  + i
                  + ": "
                  + curClassNode.getQName()
                  + " ... ");
    }

    Set allOutgoingDependencies=curClassNode.getAllOutgoingDependencies();
    Iterator iter = allOutgoingDependencies.iterator();

    while (iter.hasNext()) {
      ClassGraphNode dependeeClass = (ClassGraphNode) ( iter.next() );
      if (mbDebug) {
        System.out.println("  --> "
                    + dependeeClass.getQName() );

      }

      int dependeeIndex =
        ( ((Integer)(classNodesToIndexMap.get(
                        dependeeClass))).intValue() );
      int dependencyStrength =
        curClassNode.getOutgoingDependencyStrength( dependeeClass );
      depsWriter.println(""
                  + (i+1)
                  + ","
                  + (dependeeIndex+1)
                  + ","
                  + dependencyStrength);
    }

  }

  filesWriter.close();
  filesFos.close();
  depsWriter.close();
  depsFos.close();
}


private HashMap createNamesToAliasesMap(DependencyModel pModel) {
  HashMap result = new HashMap();
  createNamesToAliasesRecursive(pModel.getRootPackageNode(),
```

```
                                    result, null, 0);
    return result;
  }


  private void createNamesToAliasesRecursive(PackageGraphNode pPackageNode,
                          HashMap pMap,
                          String pCurrentName,
                          int pDepth) {

    // First create aliases for this package's child classes, if any...

    Object[] childClasses = pPackageNode.getClassNodesArray();
    for (int i=0; i<childClasses.length; i++) {
      ClassGraphNode child = (ClassGraphNode)(childClasses[i]);
      String classAliasLName = ( "c"
                      + ( i < 100 ? "0" : "" )
                      + ( i < 10 ? "0" : "" )
                      + i );

      String aliasQName = ( ""
                  + ( pCurrentName==null ? "" : pCurrentName + "." )
                  + classAliasLName );

      pMap.put( child.getQName(), aliasQName );

    }


    // Finally recurse into the child packages and topsort them:
    Object[] childPackages = pPackageNode.getPackageNodesArray();
    for (int i=0; i<childPackages.length; i++) {
      PackageGraphNode child = (PackageGraphNode)(childPackages[i]);
      String packageAliasLName = ( "l"
                      + ( pDepth < 100 ? "0" : "" )
                      + ( pDepth < 10 ? "0" : "" )
                      + pDepth
                      + "p"
                      + ( i < 100 ? "0" : "" )
                      + ( i < 10 ? "0" : "" )
                      + i );
      String aliasQName = ( ""
                  + ( pCurrentName==null ? "" : pCurrentName + "." )
                  + packageAliasLName );

      createNamesToAliasesRecursive( child,
                        pMap,
                        aliasQName,
                        pDepth + 1 );
    }
  }

}
```

com.mlamantia.deptools.ant.MetricsTask

```
package com.mlamantia.deptools.ant;

import java.io.*;
import java.util.*;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

import com.mlamantia.deptools.model.*;
import com.mlamantia.deptools.io.*;
import com.mlamantia.deptools.controller.*;


public class MetricsTask extends Task {

  ////////////////////////////
  ////////////////////////////
  // Constructor

  public MetricsTask() {
    ;
  }

  ////////////////////////////
  ////////////////////////////
  // property setters

  File mFile;
  /**
  * Determines the file to be created
  */
  public void setOutputFile(File pFile) {
    mFile = pFile;
  }

  long mLambda = 2;
  /**
  * Determines the "lambda" value for computing clustered cost.
  * Default value = 2.
  */
  public void setLambda(long pLambda) {
    mLambda = pLambda;
  }

  ////////////////////////////
  ////////////////////////////
  // execute method

  public void execute() throws BuildException {
    try {
      System.out.println("Writing metrics to file " + mFile);

      FileOutputStream fos = new FileOutputStream(mFile);
      PrintWriter writer = new PrintWriter(fos);
```

```
      // ** number of classes
      DependencyModel globalModel =
        DependencyModel.getGlobalDependencyModel();
      int classCount = globalModel.getClassCount();
      writer.print("number of classes\t" + classCount);
      writer.println();

      // ** number of dependencies
      System.out.println("Computing total number of dependencies... " );
      long numDeps = Metrics.computeNumberOfDependencies();
      System.out.println("Finished computing totalNumberOfDeps" );
      writer.print("number of dependencies\t" + numDeps);
      writer.println();

      // ** dependency density
      double depDensity = ( (double)numDeps * (double)1000 )
      / ( (double)classCount * (double)(classCount-1) );
      writer.print("dependency density (per 1000 pairs)\t" + depDensity);
      writer.println();

      // ** propagation cost
      System.out.println("Computing propagation cost... " );
      double propCost = Metrics.computePropagationCost();
      System.out.println("Finished computing propagation cost." );

      writer.print("propagation cost\t" + propCost);
      writer.println();

      // ** clustered cost
      System.out.println("Computing clustered cost... " );
      long clusteredCost = Metrics.computeClusteredCost(mLambda);
      System.out.println("Finished computing clustered cost." );

      writer.print("package-clustered cost\t" + clusteredCost);
      writer.println();
      writer.close();

      fos.close();
      System.out.println("Finished writing metrics to file " + mFile);

    }
    catch (IOException ioex) {
      System.out.println("Error writing metrics graph to file " + mFile);
      throw new BuildException("Error writing metrics to file " + mFile,
                  ioex);
    }
  }

}
```

com.mlamantia.deptools.ant.ParseDepFinderXMLTask

```
package com.mlamantia.deptools.ant;
```

```
import java.io.*;
import java.util.*;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

import com.mlamantia.deptools.model.*;
import com.mlamantia.deptools.io.*;


public class ParseDepFinderXMLTask extends Task {

  private DepFinderXMLParser mParser = null;

  ////////////////////////////
  ////////////////////////////
  // Constructor

  public ParseDepFinderXMLTask() {
    mParser = new DepFinderXMLParser();
  }

  ////////////////////////////
  ////////////////////////////
  // property setters

  /**
  * Sets the package prefixes for the parser to ignore.  Comma-separated, as
  * String.
  */
  public void setIgnorePrefixes(String pPrefixes) {
    StringTokenizer tok = new StringTokenizer(pPrefixes, ",");
    int nTokens = tok.countTokens();
    String[] prefixesArray = new String[nTokens];

    for (int i=0; i<nTokens; i++) {
      prefixesArray[i] = tok.nextToken();
    }
    mParser.setIgnorePrefixes(prefixesArray);
  }

  /**
  * Sets the package prefixes for the parser to consider "interesting."
  * Comma-separated, as String.
  */
  public void setInterestingPrefixes(String pPrefixes) {
    StringTokenizer tok = new StringTokenizer(pPrefixes, ",");
    int nTokens = tok.countTokens();
    String[] prefixesArray = new String[nTokens];

    for (int i=0; i<nTokens; i++) {
      prefixesArray[i] = tok.nextToken();
    }
    mParser.setInterestingPrefixes(prefixesArray);
  }
```

```
  /**
  * Determines whether only packages considered "interesting" should be added
  * to the DependencyModel.
  */
  public void setOnlyAddInterestingPrefixes(boolean pOnlyInteresting) {
    mParser.setOnlyAddInterestingPrefixes(pOnlyInteresting);
  }

  File mFile;
  /**
  * Determines the file to be parsed
  */
  public void setFile(File pFile) {
    mFile = pFile;
  }

  /**
  * Determines the file to output a list of packages modeled.
  */
  public void setPackageListFile(File pFile) {
    mParser.setPackageListFile(pFile);
  }

  /**
  * Determines the file to output a list of packages ignored.
  */
  public void setIgnoredListFile(File pFile) {
    mParser.setIgnoredListFile(pFile);
  }


  ////////////////////////////////
  ////////////////////////////////
  // execute method

  public void execute() throws BuildException {
    System.out.println("Parsing DependencyFinder output from file " + mFile);
    try {
      mParser.parseFile(mFile);
      System.out.println(
        "Finished parsing DependencyFinder output from file "
        + mFile);
    }
    catch (IOException ioex) {
      System.out.println("Error parsing DependencyFinder output from file "
                + mFile);
      throw new BuildException(
         "Error parsing DependencyFinder output from file " + mFile,
          ioex);
    }
  }

}
```

com.mlamantia.deptools.ant.TopologicalSortTask

```
package com.mlamantia.deptools.ant;

import java.io.*;
import java.util.*;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

import com.mlamantia.deptools.model.*;
import com.mlamantia.deptools.io.*;
import com.mlamantia.deptools.controller.*;


public class TopologicalSortTask extends Task {

  ////////////////////////////
  ////////////////////////////
  // Constructor

  public TopologicalSortTask() {
    ;
  }

  ////////////////////////////
  ////////////////////////////
  // property setters

  String mType = "hierarchical";

  /**
  *  Whether the ordering used should be the flattened graph ("flat")
  *  or the package hierarchy ("hierarchical").  Default is "hierarchical".
  */
  public void setType(String pType) {
    mType = pType;
  }

  ////////////////////////////
  ////////////////////////////
  // execute method

  public void execute() throws BuildException {
    System.out.println("Topologically sorting dependency graph.");
    try {

      if ( mType.equals("flat") ) {
        TopologicalSort.topSortGlobalFlatPackageGraph();
      }
      else if ( mType.equals("hierarchical") ) {
        TopologicalSort.topSortGlobalHierarchicalModel();
      }
      else {
```

```
        throw new BuildException(
          "Invalid type parameter to TopologicalSortTask: "
          + mType
          + " -- valid types are 'flat' and 'hierarchical'.");
      }

    }
    catch (Throwable thr) {
      thr.printStackTrace(System.out);
      throw new BuildException(thr);
    }

    System.out.println("Finished topologically sorting dependency graph.");
  }

}
```

com.mlamantia.deptools.ant.WriteDSASFormatTask

```
package com.mlamantia.deptools.ant;

import java.io.*;
import java.util.*;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

import com.mlamantia.deptools.model.*;
import com.mlamantia.deptools.io.*;


public class WriteDSASFormatTask extends Task {

  ////////////////////////////////
  ////////////////////////////////
  // Constructor

  public WriteDSASFormatTask() {
    ;
  }

  ////////////////////////////////
  ////////////////////////////////
  // property setters

  File mFile;
  /**
  * Determines the file to be created
  */
  public void setBaseFile(File pFile) {
    mFile = pFile;
  }

  String mType = "flat";
```

```
/**
*  Whether the ordering used should be the flattened graph ("flat")
*  or the package hierarchy ("hierarchical").  Default is "flat".
*/
public void setType(String pType) {
  mType = pType;
}

boolean mbAlias = false;
/**
* Whether package and class names should be aliased when written,
* so that, when sorted alphabetically, they appear in the order of their
* (possibly sorted) package hierarchy traversal.
*/
public void setAlias(boolean pbAlias) {
  mbAlias = pbAlias;
}

//////////////////////////////
//////////////////////////////
// execute method

public void execute() throws BuildException {
  System.out.println("Writing dependency graph formatted for DSAS to file "
              + mFile);
  try {
    DSASFormatWriter writer = new DSASFormatWriter();
    String baseName = mFile.getCanonicalPath();
    File depsOutputFile = new File (baseName + ".deps");
    File filesOutputFile = new File (baseName + ".files");
    writer.setDepsOutputFile(depsOutputFile);
    writer.setFilesOutputFile(filesOutputFile);
    if ( mType.equals("flat") ) {
      writer.run(false, mbAlias);
    }
    else if ( mType.equals("hierarchical") ) {
      writer.run(true, mbAlias);
    }
    else {
      throw new BuildException(
        "Invalid type parameter to WritePackageDSMTask: "
        + mType
        + " -- valid types are 'flat' and 'hierarchical'.");
    }
    System.out.println(
      "Finished writing dependency graph formatted for DSAS to file "
      + mFile);
  }
  catch (IOException ioex) {
    System.out.println(
      "Error writing dependency graph formatted for DSAS to file "
      + mFile);
    throw new BuildException(
      "Error writing dependency graph formatted for DSAS to file " + mFile,
      ioex);
  }
```

```
    }
}
```

References

Apache Software Foundation, "Apache Tomcat – Heritage." 30 April, 2006. <http://tomcat.apache.org/heritage.html>.

Baldwin, Carliss Y. and Kim B. Clark, *Design Rules, Volume 1: The Power of Modularity.* Cambridge: MIT Press, 2000.

Baldwin, Carliss Y. and Kim B. Clark. "The Option Value of Modularity in Design." (Draft dated May 16, 2002). 30 April, 2006. <http://www.people.hbs.edu/cbaldwin/DR2/DR1Option.pdf>.

Banker, Rajiv D.; Srikant M. Datar; and Chris F. Kemerer. "Factors Affecting Software Maintenance Productivity: An Exploratory Study." *Proceedings of the Eighth International Conference on Information Systems*, New York: ACM (1987): 160-175.

Banker, Rajiv D. and Sandra A. Slaughter. "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement." *Information Systems Research* 11.3 (September, 2000): 219-240.

Brusoni, Stefano; Luigi Marengo; Andre Prencipe; and Marco Valente. "The Value and Costs of Modularity: A Cognitive Perspective." *SPRU Electronic Working Paper Series*, Paper No. 123, (August, 2004). 30 April, 2006. <http://www.sussex.ac.uk/spru/documents/sewp123.pdf>.

Byous, John. "Java Community Process 2.6: More Ways to Get Involved." (August, 2003). 1 May, 2006. <http://java.sun.com/developer/technicalArticles/jcp>.

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

Eick, Stephen G.; Todd L Graves; Alan F. Karr; and J.S. Marron. "Does Code Decay? Assessing the Evidence from Change Management Data." *IEEE Transactions on Software Engineering* 27.1 (January, 2001): 1-12.

Eppinger, Steven D.; Daniel E. Whitney; Robert P. Smith; and David A. Gebala. "A Model-Based Method for Organizing Tasks in Product Development." *Research in Engineering Design* 6 (1994): 1-13.

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

Guth, Robert 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>.

Leveson, Nancy G. and Clark S. Turner. "An Investigation of the Therac-25 Accidents." *IEEE Computer* 26. 7 (July, 1993): 18-41.

Leveson, Nancy G. and Kathryn A. Weiss "Making embedded software reuse practical and safe." *Proceedings of the 12th ACM SIGSOFT Twelfth international Symposium on Foundations of Software Engineering.* New York: ACM Press (Newport Beach, 31 October – 6 November, 2004): 171-178. 27 April, 2006. <http://doi.acm.org/10.1145/1029894.1029897>.

Lions, J. L. "Ariane 501 Failure: Report by the Inquiry Board." (19 July, 1996). 30 April, 2006. <http://www.dccia.ua.es/dccia/inf/asignaturas/PSTR/Teoria/Ariane5.pdf>.

MacCormack, Alan; John Rusnak; and Carliss Y. Baldwin, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. (Draft dated 9 June, 2005). 20 April, 2006. <http://opensource.mit.edu/papers/maccormackrusnakbaldwin2.pdf>. To be published in *Management Science.*

Mazzocchi, Stefano. Personal interview. (24 February, 2006).

Meyer, Marc H. and Seliger, Robert. "Product Platforms in Software Development," *Sloan Management Review* 40.1 (Fall, 1998): 61-75.

Meyer, Marc and Peter Webb, "Modular, Layered Architecture: The Necessary Foundation for Effective Mass Customization in Software," *International Journal of Mass Customization* 1.1 (2005): 14-36.

Nguyen, L.; Carroll, J.; Swatman, P.A., "Supporting and monitoring the creativity of IS personnel during the requirements engineering process," *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences* 1 (4-7 January, 2000): 1-9.

Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM* 15.12 (December, 1972): 1053-1058.

Parnas, David L. "Designing Software for Ease of Extension and Contraction." *Proceedings of the 3rd International Conference on Software Engineering.* Piscataway: IEEE Press, 1978.

Rusnak, John. *The Design Structure Analysis System: A Tool to Analyze Software Architecture*, Ph.D. Thesis, Harvard University, 2005.

Sangal, Neeraj; Ev Jordan; Vineet Sinha; and Daniel Jackson. "Using Dependency Models to Manage Complex Software Architecture." In Richard P. Gabriel, editor, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, *OOPSLA 2005.* San Diego: ACM, October 2005.

Skiena, Steven S. *The Algorithm Design Manual.* New York: Springer, 1998.

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

Sullivan, Kevin J.; William G. Griswold; Yuanfang Cai; and Ben Hallen. "The Structure and Value of Modularity in Software Design." *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering.* (Vienna, Austria, September 10 - 14, 2001). New York: ACM Press (2001): 99-108. 30 April, 2006. <http://doi.acm.org/10.1145/503209.503224>.

Sun Microsystems. *Java™ 2 Platform, Enterprise Edition (J2EE™) Specification, Version 1.4.* (24 November, 2003). 30 April, 2006. <http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf>.

Sun Microsystems (2006a), "1.3 Compatible Implementations," 29 March, 2006. <http://java.sun.com/j2ee/compatibility_1.3.html>.

Sun Microsystems (2006b), "1.4 Compatible Implementations," 29 March, 2006. <http://java.sun.com/j2ee/compatibility.html>.

Ulrich, Karl. "The role of product architecture in the manufacturing firm." *Research Policy* 24 (1995): 419-440.

Waters, Richard. "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>.