Using Dependency Models to Manage Complex Software Architecture

Neeraj Sangal, Ev Jordan
Lattix, Inc.
{neeraj.sangal,ev.jordan}@lattix.com

Vineet Sinha, Daniel Jackson
Massachusetts Institute of Technology
{vineet,dnj}@csail.mit.edu

ABSTRACT
An approach to managing the architecture of large software systems is presented. Dependencies are extracted from the code by a conventional static analysis, and shown in a tabular form known as the 'Dependency Structure Matrix' (DSM). A variety of algorithms are available to help organize the matrix in a form that reflects the architecture and highlights patterns and problematic dependencies. A hierarchical structure obtained in part by such algorithms, and in part by input from the user, then becomes the basis for 'design rules' that capture the architect's intent about which dependencies are acceptable. The design rules are applied repeatedly as the system evolves, to identify violations, and keep the code and its architecture in conformance with one another. The analysis has been implemented in a tool called LDM which has been applied in several commercial projects; in this paper, a case study application to Haystack, an information retrieval system, is described.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE).
D.2.9 [**Management**]: Life cycle.
D.2.11 [**Software Architectures**]: Information Hiding.

General Terms
Design, Algorithms, Management.

Keywords
Architecture, Dependency, Model, Matrix, DSM.

1. INTRODUCTION
Excessive inter-module dependencies have long been recognized as an indicator of poor software design. Highly coupled systems, in which modules have unnecessary dependencies, are hard to work with because modules cannot be understood easily in isolation, and changes or extensions to functionality cannot be contained.

This paper describes an approach to managing software systems using dependencies. A tool extracts dependencies from code, and displays them using a scheme that highlights potential problems. The user enters 'design rules' that distinguish dependencies that are problematic because they violate architectural assumptions from dependencies that are expected and reasonable. As the system evolves over time, the rules are checked in subsequent analyses to flag deviations from the architecture, usually introduced unwittingly during ongoing development.

The topic of this paper is the underlying dependency model, and the scheme by which potential problems are highlighted. An experimental application of the approach to a system of about 200,000 lines of code is described, which suggests some of the approach's promise, and also indicates areas needing attention.

The extraction and exploitation of dependencies has been a subject of research since Parnas first formulated the notion of inter-module dependency in his early papers (most notably [5]). The particular representation that we use – a partitioned adjacency matrix – has been widely used in the analysis of manufacturing processes, where it is referred to as the 'dependency structure matrix' or 'design structure matrix' or DSM. The potential significance of the DSM for software was noted by Sullivan et al [7], in the context of evaluating design tradeoffs, and has been applied by Lopes et al [18] in the study of aspect-oriented modularization. MacCormack et al [19] have applied the DSM to analyze the value of modularity in the architectures of Mozilla and Linux. Our approach, however, seems to be the first application of DSM for the explicit management of inter-module dependencies, and the tool Lattix Inc's dependency manager (henceforth LDM) seems to be the first publicly available implementation of DSM analysis for software.

Our paper begins, in Section 2, with a short introduction to the dependency structure matrix, explaining its origins in the design for manufacturing process. Section 3 explains our application of the DSM to software, and why it appears to be well-suited to the problems of large-scale software design. Section 4 describes the key elements of our process for discovering and analyzing the architecture of existing systems. Section 5 reports on the application of the approach to Haystack, an information retrieval system whose codebase has evolved over several years. The paper closes in Section 6 with a discussion of related work, in particular the Reflexion Model Tool of Murphy, Notkin and Sullivan [3].

2. THE DEPENDENCY STRUCTURE MATRIX
The dependency structure matrix (DSM) was invented for optimizing product development processes. Although it has broader applications – including, as we shall see, to software – we

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

shall describe it in its original context to make the discussion as concrete as possible.

In the development of a product, a collection of tasks is performed. These tasks have dependencies on one another, either because of physical objects that must flow from task to task, or because of information that one task requires and which another task provides. The structure of dependencies amongst these tasks is a strong indicator of the efficiency of the process as a whole [1]. If the tasks are tightly coupled, with many cyclic dependencies, the pipeline will stall frequently, and tasks will need to be repeated because of dependencies on tasks that follow them.

The term 'dependency structure matrix' refers both to a particular representation of such dependencies, and to algorithms for reorganizing the dependencies by reordering and clustering tasks. The matrix is a simple adjacency matrix with tasks labeling the horizontal and vertical axes, and a mark in the i^{th} column and j^{th} row when the i^{th} task depends on the j^{th}. Dependencies of tasks on themselves are not considered, so there are never marks along the diagonal. In some applications, the strength of the dependencies is given numerically, but we shall first consider only binary values, writing an 'X' for the presence of a dependency, and nothing for its absence.

		1	2	3	4
Task A	1	.		X	X
Task B	2		.	X	
Task C	3	X		.	X
Task D	4				.

Figure 1: A Simple DSM

		1	2	3	4
Task D	1	▪			
Task A	2	X	▪	X	
Task C	3	X	X	▪	
Task B	4			X	▪

Figure 2: Block Triangular DSM after Partitioning

		1	2	3
Task D	1	.		
Task A-C	2	X	.	
Task B	3		X	.

Figure 3: Lower Triangular DSM

			1	2	3	4
Task D		1	.			
A-C	Task A	2	X	.	X	
	Task C	3	X	X	.	
Task B		4			X	.

Figure 4: Hierarchical DSM

One important criterion that is used to evaluate the matrix is that the dependency relation should be acyclic. This means, in matrix terms, that the tasks can be permuted so that the matrix is *lower triangular* – that is, with no entries above the diagonal.

Figure 1 shows a simple DSM. Examining column 1 we note that task A depends on task C; examining column 3 we note that task C depends on tasks A and B. Because tasks A and C are mutually dependent, the tasks cannot be reordered to make the matrix lower triangular. However, if A and C are regarded as a single composite task, the cycle can be eliminated. This transformation is known as *partitioning*, and its result is shown in Figure 2, with the composite tasks indicated by shading. Such a DSM, which has been rearranged so that all dependencies either fall below the diagonal or within groups, is said to be in *block triangular* form.

The grouping of tasks can be shown in different ways. A new compound task can be formed [2] as in Figure 3; in this case, the matrix becomes lower triangular. Alternatively, the identities of the basic tasks can be retained, by introducing some hierarchical structure, as in Figure 4, in which the grouping of A and C is shown by their indentation.

Algorithms have been developed to optimize the ordering of tasks and their aggregation into groups. Such algorithms are known as *partitioning algorithms*, and include those of Warfield [10] and of Gebala and Eppinger [9]. A different class of algorithms, described by Hartigan [11], and known as *clustering algorithms*, optimizes the ordering and aggregation to reduce the number of off-diagonal dependencies. Their purpose is not merely to eliminate cycles, but to reduce the incidence of any dependencies between task clusters. Clustering has been used for architectural decomposition [13], and to optimize the organization of product development teams [14][15][22].

3 APPLYING THE DSM TO SOFTWARE

The application of the DSM to software, with modules playing the role of tasks, is straightforward and yet appears to have several advantages over more widely used dependency representations:

- The matrix representation itself scales better than box-and-line diagrams; the inclusion of hierarchy, as shown in Figures 4 through 10, is particularly helpful.
- The criteria that motivate partitioning in product development workflow have analogues in the structure of software systems. Parnas discussed the elimination of cyclic dependencies in his early paper [5]. The term 'layered' is often used approvingly of systems in which modules can be partitioned into layers, with each module having dependencies only on modules within the layer or belonging to the layer below [17]. Partitioning finds layers and highlights cycles.
- The partitioning algorithms provide an automatic mechanism for architectural discovery in a large code base. Partitioning eliminates cycles by forming subsystems. The groupings and orderings recommended by these algorithms can be applied straightforwardly to reorganize the code base so that its

inherent structure (evident, in Java for example, in the package namespace hierarchy) matches the desired structure.

3.1 LDM's Dependency Notion

The LDM tool uses a standard notion of dependency, in which a module *A* depends on a module *B* if there are explicit references in *A* to syntactic elements of *B*. Currently, a module is a Java class, but a more fine-grained analysis is possible. This simple but effective notion of dependency works well for understanding design dependencies, in which modifications to one module might affect another. It is less well suited to determining runtime properties (such as how failures can propagate between modules), which require a deeper static analysis.

As in other dependency tools (such the Reflexion Model Tool [3]), the extraction of dependences can be decoupled from their analysis, so the techniques we describe here would apply equally well on top of more sophisticated static analyses.

In the LDM tool, the DSM can be configured to display an 'X' for a dependency or to display a dependency strength representing the number of references between two modules that is responsible for their dependence.

LDM offers DSM algorithms for partitioning; it does not currently offer automatic clustering. The default decomposition used by LDM is based on the code organization: the Java package structure. The matrix presented is hierarchical (as in Figure 4), and the algorithms and manual intervention can be applied at different levels.

Users can edit the systems structure. They can reorder modules and partition by hand, and create, delete, and move subsystems to reflect their understanding of the architecture. Dependencies are automatically recalculated and re-aggregated as the structure is changed.

3.2 Architectural Patterns

A DSM can readily reveal an underlying architectural pattern in an existing system, and highlight deviations from it. For example, Figure 5 shows layering, in which each layer depends on the layers underneath but not on the layers above. Figure 6 shows a strictly layered system [20] in which each layer depends only on the layer immediately below it.

Figure 7 shows a *change propagator*: a subsystem that depends on a large number of subsystems and in turn has many subsystems depending on it. Change propagators make systems brittle because they increase the likelihood that the effect of a change will propagate to a disproportionately large portion of the system. In this case, the propagator is *Project* because, as shown in column 9, it depends on a large number of subsystems, and as shown in row 9, a large number of subsystems depend on it. A change in *Services* could affect nearly the entire system, because *ProjectLoader* depends on *Services*, *Project* depends on *ProjectLoader*, and every subsystem depends on *Project*.

Baldwin and Clark argue that the value of modular systems comes, in large part, from hidden subsystems [2]. Hidden subsystems can be replaced easily, and are easier to maintain because they have a limited and well-defined interface to the system and are therefore unaffected by most modifications.

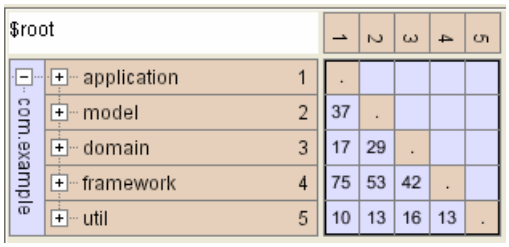

Figure 5: Layered System

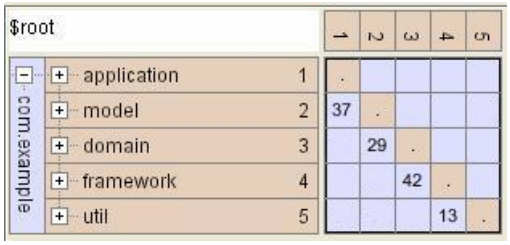

Figure 6: Strictly Layered System

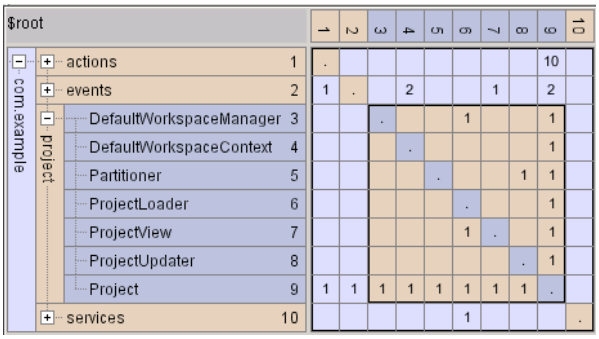

Figure 7: DSM with a Change Propagator

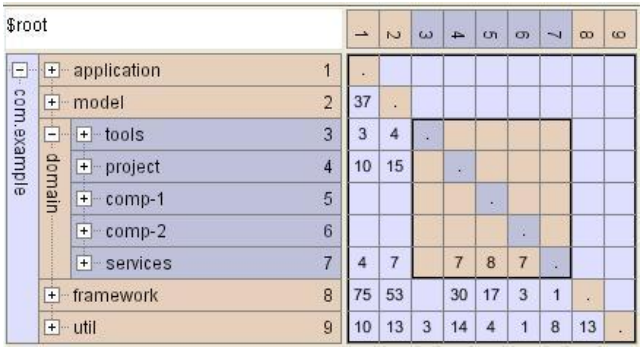

Figure 8: Hidden Subsystems

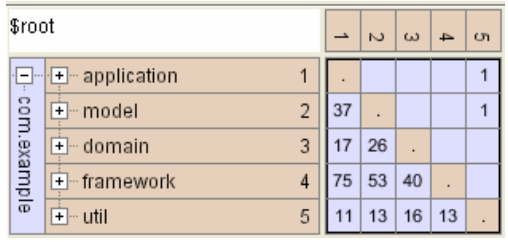

Figure 9: Imperfectly Layered System

Figure 8 shows two subsystems *comp-1* and *comp-2*, which are regarded as hidden within the subsystem *domain*, because no other subsystem depends on them.

A key advantage of matrix over graph representations is that the preponderance of dependencies in the lower triangular part of a matrix makes it easy to see the layering pattern even when the layering is itself imperfectly pattern even when the layering is itself imperfectly implemented. Figure 9 shows a system that is not completely layered because of dependencies in column 5. Module *util* depends on *application* and *model*, but dependency strengths suggest that this dependency is not as strong as the reverse dependency of *application* or *model* on *util*.

Note that an approach based on DSMs does *not* imply that every software architecture should have a layered, acyclic structure. The merit of the DSM approach is simply that it highlights those aspects of the architecture that deviate from these norms. This allows a succinct characterization of a system's architecture in terms of which deviations are acceptable. Some simply represent flaws: violations of the architectural design that can be corrected by modifying the code. Others, however, result from conscious design tradeoffs, in which the software architect has decided with good reason to deviate from a pure layered architecture.

3.3 Examples

Figure 10 shows a DSM for *JUnit version 3.8.1*, an open source regression testing framework. This DSM was created by loading the *JUnit* jar file into LDM and then applying the partitioning algorithm. The DSM shows that *JUnit* is a layered system with clean separation of the user interface layers from the underlying core logic.

Figure 11 shows the DSM (created by the same process) for *jEdit version 4.2*, an open source editor with about 800 classes. Its layering, unlike *JUnit*'s, is not immediately apparent. The large number of dependencies in column 15[1] shows that there are a number of classes in the top level package *jedit* that reference most of the other subsystems. It also shows that most of the other subsystems also reference these classes. This suggests that a refactoring might be in order to reduce the coupling.

3.4 Design Rules

The distinction between acceptable and unacceptable dependencies is expressed using *design rules*, which are provided by the user, and applied to the displayed DSM, in order to highlight the dependencies that violate the intended architectural design. A design rule may require, for example, that a library subsystem has no dependencies on the rest of the application, or that only certain parts of a core subsystem may depend on GUI modules.

Continuous checking of design rules, akin to regression testing, can be used to keep a code base in sync with its design. Architectural creep becomes less of a problem, and flaws (especially those introduced by new team members) are caught as soon as they are introduced.

[1] In the displayed DSM and in design rules, the symbol * refers to immediate components of a package; in design rules, ** refers to all direct or indirect descendants.

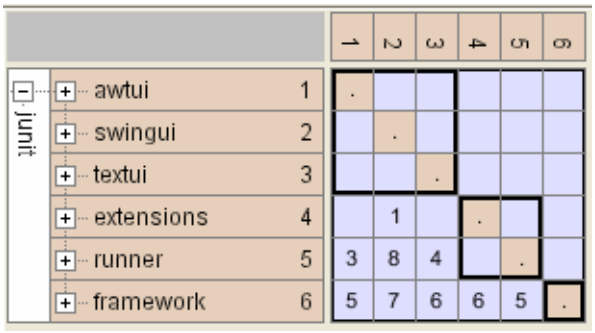

Figure 10: DSM for JUnit

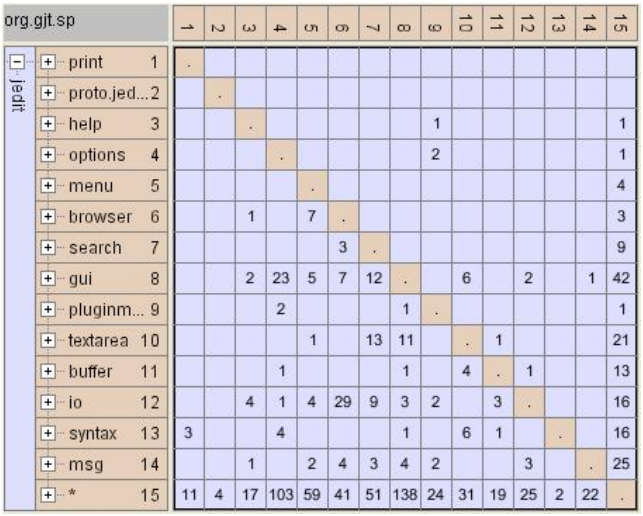

Figure 11: DSM for jEdit

The very expression of design rules has benefits that go beyond a shared articulation of design intent. When current design rules are violated for good reasons, it forces the revision of design rules thereby making architectural evolution explicit.

The DSM itself provides a convenient way to input design rules, by having the user click on cells to identify allowed or forbidden relationships. Design rules exploit the hierarchical structure too, since a rule specified for a subsystem can apply to all its constituents.

3.4.1 Specifying Design Rules

Design rules come in two forms

```
S1 can-use S2
S1 cannot-use S2
```

indicating that S_1 can and cannot depend on S_2. In their simplest form, the specifiers S_1 and S_2 are names drawn from the program's namespace (such as Java packages or classes). More generally, they can be lists of arbitrary path names with wildcards.

The user can also define specifiers to use as an orthogonal classification of program elements; for example, subsystems that access a database might be classified as *persistence*, and subsystems that are web-based servlets or Java server pages as *presentation*, allowing rules such as:

```
presentation cannot-use persistence
```

Classification can be manual, or can be computed automatically (for example, according to which external libraries a subsystem uses).

Rules are by default inherited, so that a rule for a subsystem applies to its components. They are interpreted in order, so that one rule can override another to handle exceptions. Rules can be applied equally to external systems; the rule:

```
S can-use org.apache.**
```

for example, permits dependences on all external libraries whose names start with `org.apache`. This provides a technique to control the proliferation of external library usage.

3.4.2 Using a DSM to Represent Design Rules

In LDM, design rules are shown visually (Figure 12) by marking the corners of a cell with green and black to indicate whether a dependency is permitted. For a can-use rule, the cell has a green triangle in the upper left corner; for a cannot-use rule, the cell has a black triangle in the lower left corner. A dependency in a cell governed by a cannot-use rule is a design rule violation, indicated by a red triangle in the upper right corner.

Note that often we turn off the display of the green triangle. This makes the display easier to read by enabling the user to focus on dependencies which are not permitted. In subsequent figures, a triangle in the lower left-hand corner of a cell shows where a dependency is prohibited, except for Figure 18, in which the triangles highlight violations.

3.4.3 Example of Design Rules for Patterns

Architectural patterns can be expressed with design rules. For example, the following rules, shown visually in Figure 13, express layering (where *$root* refers to the top-level node in the hierarchy):

```
$root can-use $root
model cannot-use application
domain cannot-use application, model
framework cannot use application, model,
       domain
util cannot-use application, model,
       domain, framework
```

A strictly layered system might be specified thus:

```
$root cannot-use $root
application can-use application, model
model can-use model, domain
domain can-use domain, framework
framework can-use framework, util
util can-use util
```

In this case it was more convenient to specify the top-level rule as a *cannot-use*, and override it. The rules are shown visually in Figure 14.

A system with independent components might be specified thus:

```
comp-1 cannot-use comp-2, comp-3
comp-2 cannot-use comp-1, comp-3
comp-3 cannot-use comp-1, comp-2
```

and shown as in Figure 15.

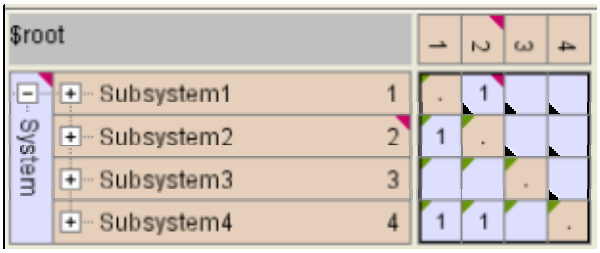

Figure 12: DSM with Rule View

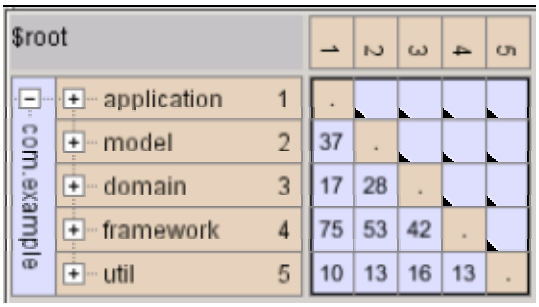

Figure 13: Design Rules for a Layered System

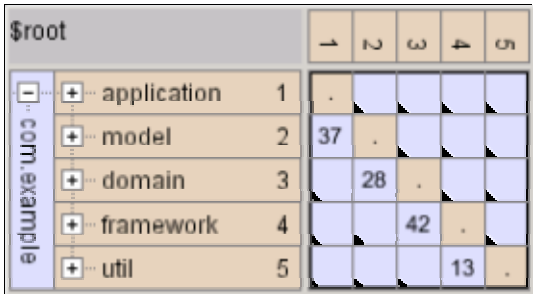

Figure 14: Design Rules for a Strictly Layered System

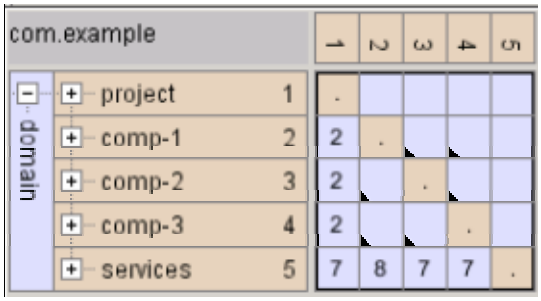

Figure 15: Design Rules for Independent Components

4. AN APPROACH TO MANAGING DEPENDENCIES

In this section, we give a brief overview of how LDM is used in the context of a development project to manage dependencies.

Although the dependencies that underlie the DSM are extracted automatically, establishing the right hierarchical structure – which we call the *conceptual architecture* – relies on guidance from the user. As mentioned above, the tool by default uses the program's

own package structure as an initial hierarchy, but this usually does not reflect the important architectural structure fully.

Not surprisingly, an iterative process seems to work best, combining an understanding of the problem domain and the system itself with information obtained from the tool's DSM analyses.

Once the conceptual architecture has been defined and the corresponding DSM obtained, the design rules are developed.

Our experience so far in using this approach has been as consultants applying the LDM tool to projects for our clients. Typically, we progress as follows:

1. *Understand Application*. We obtain a working knowledge of the function and use of the application, by reading user documentation and, when possible, running the application.

2. *Create Preliminary DSM*. We run LDM to create a preliminary DSM using the hierarchical structure of the code's own namespace.

3. *Create Conceptual Architecture*. We interview the architects and senior developers who have an understanding of how the entire application is structured. We then create a conceptual architecture, usually in diagrammatic form as a directed graph. The DSM is then updated to reflect this hierarchical structure, and we examine the resulting dependencies, as aggregated by this structure. The structure is refined by removing irrelevant subsystems, moving subsystems, and adding new levels in the hierarchy.

5. *Audit Dependencies*. With the hierarchy in place, we now embark on a careful analysis, using LDM, of the dependencies. We identify dependencies that appear to violate the intended layering or modularity.

6. *Define Design Rules*. A set of design rules is then developed to explain the dependencies generated. Dependencies that are considered acceptable even though they violate the overall architectural intent are permitted by creating exception rules. For each rule, a rationale is recorded.

7. *Architectural Remediation.* Once a conceptual architecture has been defined and a corresponding dependency model obtained, architectural violations are highlighted by the tool. Initial remediation generally involves package reorganization so that the package and file hierarchy corresponds to the subsystem hierarchy that was created in the DSM. Other remediation may require more substantive code changes so that the dependencies conform to the design rules, for example, by creating new interfaces, or adopting patterns such as *Factory* and *Listener*.

8. *Ongoing Dependency Management*. By this point, the code has been brought into conformance with the architectural intent, as articulated in the hierarchical structuring and in the design rules. As the code is developed further, LDM is applied to flag deviations from this intent. In most cases, we expect the deviations to represent flaws that should then be fixed, but in some cases, the deviations will represent evolving changes to the architecture which should be accommodated by changes to the hierarchy and design rules themselves.

This process does vary for each application since the quality and quantity of system documentation differs greatly, as does the availability of key architects with critical insights. Steps may be repeated simply to reconcile conflicting information from different sources.

5. A CASE STUDY: A DEPENDENCY MODEL FOR HAYSTACK

In order to determine whether our approach meets its goals, we undertook a case study with the following questions in mind:

- Can the dependency model capture the architecture and scale of a program with significant complexity?
- Can the dependency model help in the management of the program's architecture? Is the dependency model useful for extracting the architecture of the program? Does the model help in reengineering the software architecture?
- Can the dependency model ease future maintenance of the architecture? Can design rules capture the extracted architecture of the program? Are they able to provide support for architectural exceptions?

Our choice for this case study was Haystack [12], a research prototype of a tool for managing personal information. The current incarnation of the tool has been under development for 4 years, with the core application consisting of 196,707 lines of Java code (as measured by Unix wc, which includes comments and whitespace). Being a test-bed for research ideas, the program has portions whose current use is different from the original intent, as well as small portions of code that are not currently used.

Although the code was developed by a team, the original design was the work of a single developer who has since left. There is very little documentation of the architectural structure – not even a high-level architectural diagram. The reengineering task therefore reflected the challenge faced in many software projects, in which the architectural descriptions, if they ever existed, are no longer in sync with the code.

Our plan in conducting this study was to develop a conceptual architecture with the help of LDM, then identify violations, and determine what code changes and refactorings would be needed to make the code conform.

The study was conducted by one of the authors of this paper, a graduate student with 5 years' experience in system programming in Java and C++. He had been a developer of some extensions to Haystack, but was not familiar with the entire codebase, and had not worked with DSM's or LDM prior to the study.

5.1 Defining the Architecture

The code base was loaded into LDM which created a DSM (shown in Figure 16) based on the package structure. This revealed the contrast between previous architectural decisions (embedded in the directory structure) and the exceptions to those decisions (shown by dependencies above the diagonal in the matrix). It quickly became clear that the hierarchical package structure did not reflect the architecture well, although the individual packages grouped classes reasonably well.

Using the approach described in Section 4, the conceptual architecture of Figure 17 was created. The corresponding DSM

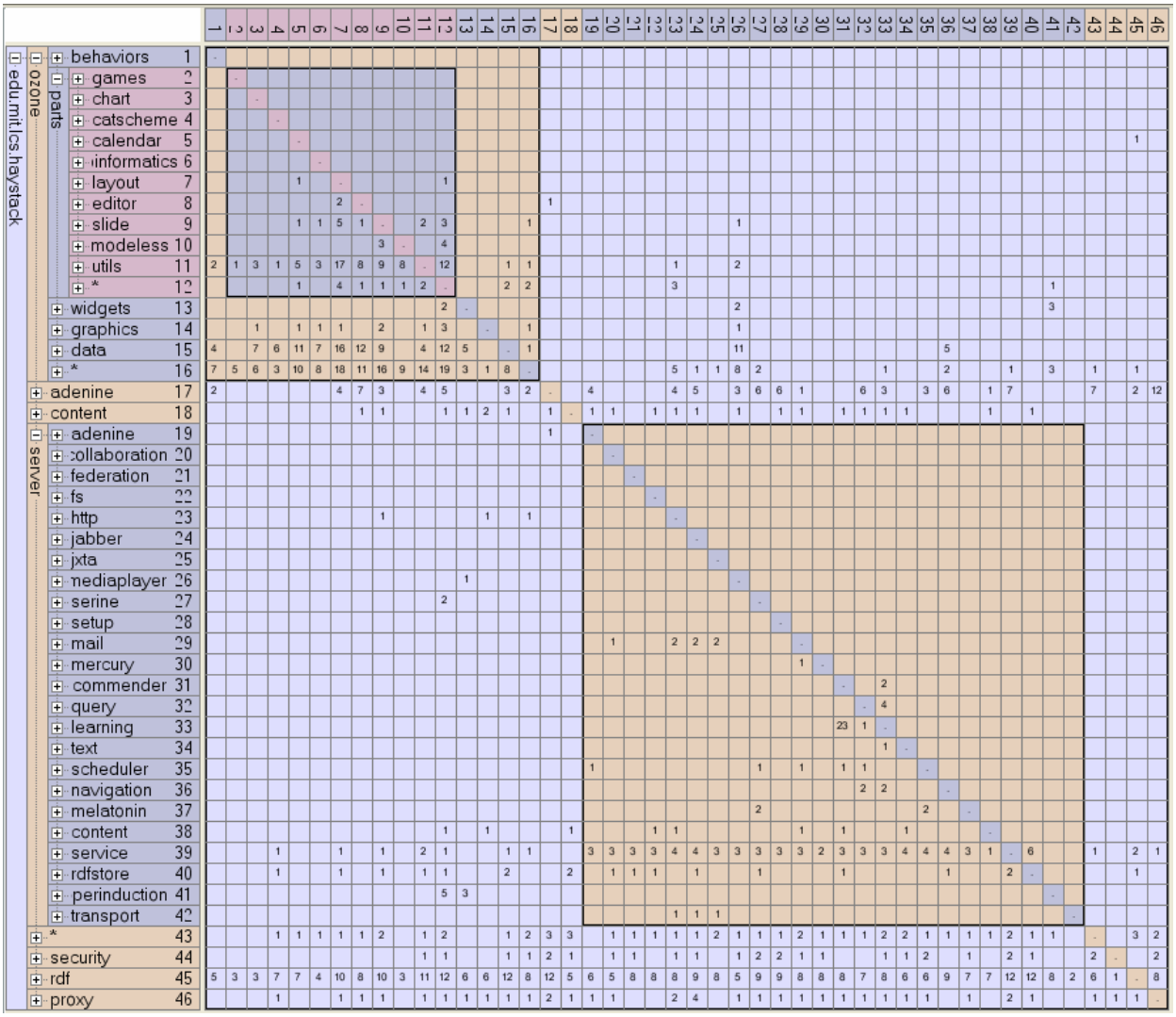

Figure 16: Initial DSM for the Haystack code base

for the conceptual architecture is shown in Figure 17. LDM's partitioning algorithm was a great help in discovering layering. The rapid recomputation of aggregated dependencies as the hierarchy is changed was found to be essential, because it allowed us to experiment with different structurings and examine their consequences in terms of dependencies. The progress we made in understanding the architecture is apparent in a comparison of Figures 16 and 18 – in particular, the elimination of many dependencies in the upper right portion of the matrix.

5.2 Leveraging the Architecture

Having developed a conceptual architecture, we then used LDM to reveal extra dependencies. An unexpected benefit of building a DSM with a cleaner architecture was that by going over the extra dependencies and expanding the hierarchy in the matrix, it was possible to very quickly to find the cause of the extra dependencies. In fact, in the DSM in Figure 18, four regions (numbered 1-4) are supposed to be free of dependencies; regions 1-3 because they are in the top-right diagonal and would represent cycles, and region 4 because (as shown in the architectural diagram) it represents the independence of the UI from the Server. Analysis of the dependencies in these regions suggests the remediation necessary in order to clean up the architecture:

- Region 1 represents dependencies from what is known as the project's data-model. These modules form the building blocks of the entire system and therefore are not supposed to depend on any other modules. Examining the extra dependencies shows that they are caused mainly due to static methods. These dependencies show the practice in Haystack of not using any particular criteria in placing non-instance based helper methods. While these methods should ideally be in another module, they are in the data model for ease in finding and using them.
- Region 2 represents dependencies from the project's inference engines. These engines have resulted from the evolution and extension of two smaller inference engines one primarily used in the server and the other in the UI. Over time, they had evolved to have different strengths and are being used inconsistently in the code. A redesign of this component was already being contemplated. The extra dependencies in the grid highlight this need.

- Region 3 and 4 dependencies represent minor project inconsistencies. Examining these extra dependencies shows that resolving each one needs a small and local design decision, and have likely been caused because of the absence of an architectural diagram in the past.

Beyond helping identify these regions of extra dependencies, the rearchitected DSM yielded significant benefits. Just from the DSM it is apparent that subsystems for server extensions and user interface (ozone) extensions have no major dependencies on the rest of the system. This means that they can be changed or newer extensions added without significant risk to the system. In fact, a large number of these components are now hidden subsystems of the server and the user interface.

Similarly, we were able to create a new abstraction for inference engines (region 2) with the DSM without actually writing code. While we still needed to refactor the code so that the new abstractions would be properly layered and easily available, the DSM allowed us to define the details of the new abstraction by allowing us to examine all dependencies that would arise from creating the abstraction.

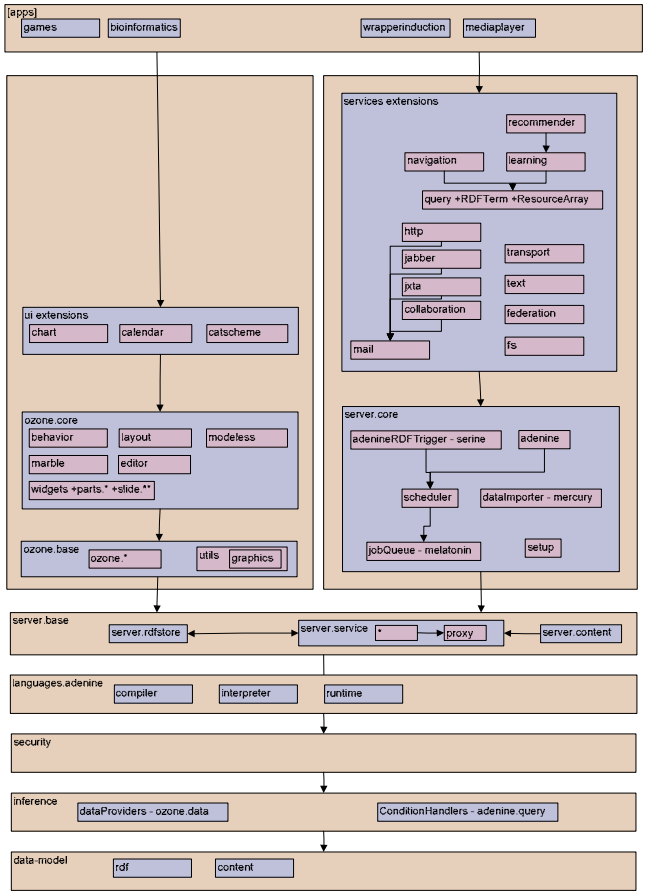

Figure 17: Haystack - Conceptual Architecture

5.3 Maintaining the Architecture

In order to maintain the discovered architecture while re-architecting and for future development, we described the architecture using design rules. Three types of rules were defined:

1. Rules to reflect the layering (as shown by regions 1-3). For example, the rules for region 1 were:

   ```
   data-model cannot-use $root
   data-model can use edu.mit.lcs.haystack.*
   ```

2. Rules to indicate the key design decision of the independence of server and the user-interface::

   ```
   ozone cannot-use server
   ```

3. Rules to characterize the usage of external libraries. This included defining how the base system and extensions use external libraries, and were similar to examples shown before.

The red triangles in the top right corner of cells in Figure 18 highlight the conflicts once the above design rules are entered. These rules allow for the periodic automatic checking of the code base for design violations.

5.4 Evaluating DSMs

By and large, the approach worked well on the case study. Although we have not yet been able to assess the efficacy of design rules in ongoing development, we found the tool very helpful in extracting the architecture, identifying problems and checking the results of refactoring. Our experience confirmed the value of the two key features of DSM's: the hierarchical structure, and the partitioning algorithm. Without these features, we would not have been able to handle a code base even of this size; a box-and-line diagram extracted directly from the code is unintelligible.

We had two kinds of problems. First, matrices, such as the DSM, while inherently more scalable a representation of relations than graphs, are not always easy to read. In particular, finding the cell that corresponds to a relationship between two elements has a greater cognitive overhead, and following the edges of a relation transitively requires going back and forth between row and column indices rather than simply following arrows along a path. With time, however, our proficiency at reading DSM's increased. Second, there were limitations of the underlying dependency model. Haystack's design involves two orthogonal classifications; while we could have constructed two separate DSM's, we would have liked to have been able to use the tool to understand the interplay of dependencies between the two views. Also, we occasionally wanted a more refined notion of dependency, which maps a client class to the class it uses at runtime rather than to the class it declares as a use in its code. There seems to be no fundamental reason why a richer dependency model (such as [6]) could not be incorporated into the tool.

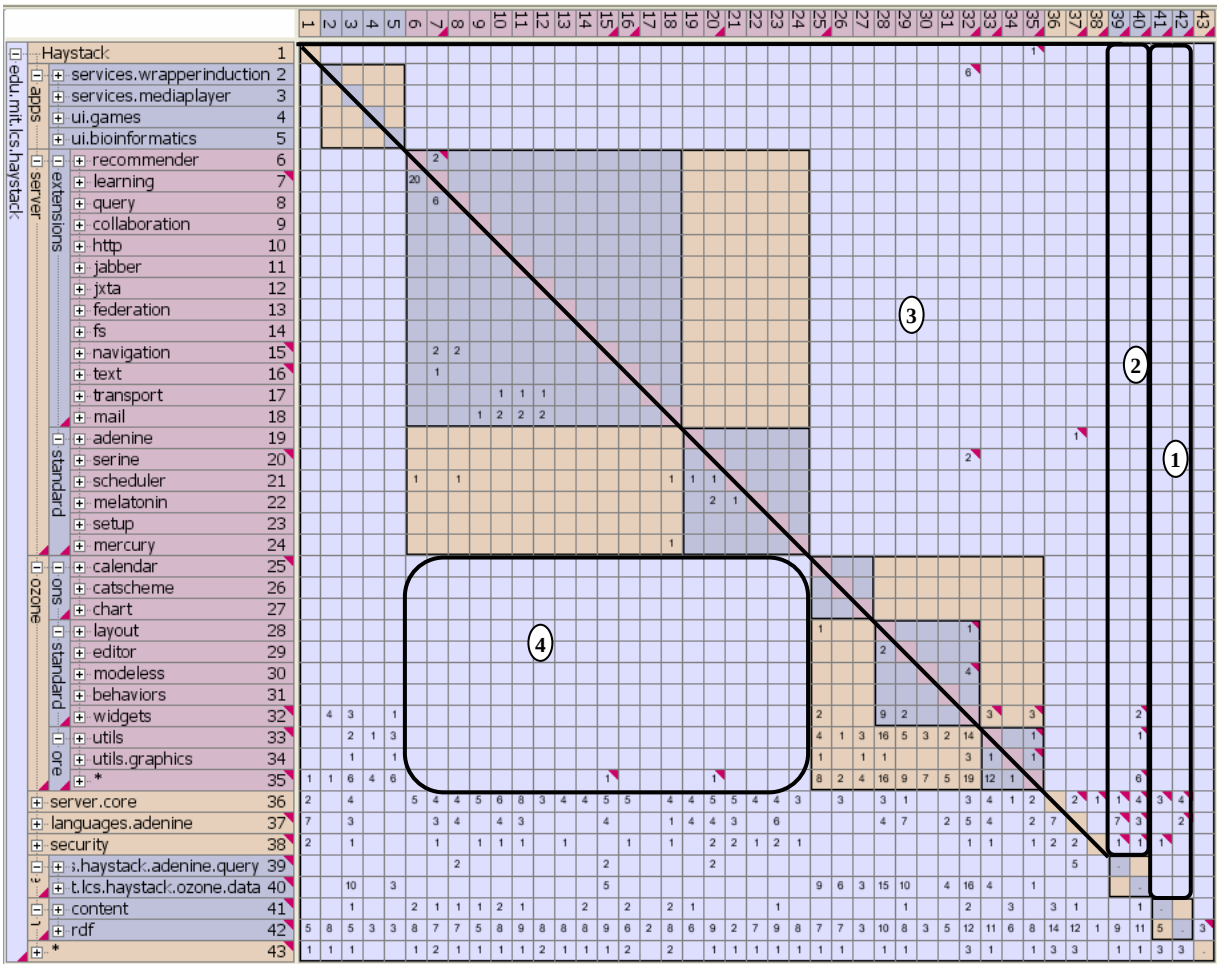

Figure 18: DSM for Haystack after Hierarchy Reorganization (triangles indicate design rules violations)

6. RELATED WORK

The importance of understanding dependencies between modules has also been understood and emphasized by other researchers. Our work is perhaps most similar to the work on the Reflexion Model Tool (RMT)[3]. Our design rules combine the reflexion map and the idealized model. When there are multiple, orthogonal views of the architecture this will make our design rules less succinct than the idealized model of RMT; on the other hand, when a single hierarchy is dominant, our design rules are likely to be simpler to express than the model and map of RMT together. We believe that our hierarchical matrix representation scales better than the graphical representation of RMT. The use of DSM algorithms for architectural discovery is a major benefit of our approach; it would be interesting to see how it might be incorporated into RMT.

Hierarchical representations are, of course, not new. Tran et al. [4] examines systems in terms of their hierarchical decomposition, using Harel's higraphs.

Heuristic algorithms for organizing a system into layers have been investigated in the context of the reverse engineering tool Rigi [16]. Rigi seems to be less flexible than LDM in allowing a mixture of manual and automatic organization, and in terms of its ability to scale. We have not been able to evaluate the effectiveness of Rigi's algorithms in comparison to the DSM algorithms, but such a study would be worthwhile.

A number of tools are available for extracting dependencies from code, such as *sa4j* from IBM, *OptimalJ* from Compuware and *JDepend* from Clarkware Consulting. These do much the same as the frontend of LDM.

Jackson [6] has proposed a more elaborate notion of dependence, in which interfaces are not treated as modules in their own right, but rather mediate dependencies. We plan to explore whether this notion might be useful in our context.

A dependence-based view of software is, of course, only one of several useful views. In Kruchten's "4+1" model of software architecture [8], our representation of the system corresponds roughly to the 'development view'.

7. CONCLUSION

The approach we have described seems to be lightweight enough to be usable in practice, and yet offers benefits that have not been available in previous approaches. It seems to scale well, and provides, with little effort from the user, a view of the system that is valuable, especially during ongoing development, or when reengineering. The approach we have described does not disrupt standard development processes, and seems to offer a notion of

architecture and architectural conformance that is compatible with the intuitions of practicing software engineers.

In future work, we plan to explore more refined notions of dependence and the role they might play. We are also investigating the impact of design rules on the evolution of the architecture of software systems, which we believe will be especially valuable in distributed organizations where architectural intent is harder to communicate and maintain.

REFERENCES

[1] Steven D. Eppinger, "Innovation at the Speed of Information", *Harvard Business Review*, January 2001.

[2] Baldwin, C.Y. and Clark K.B., *The Power of Modularity Volume 1*, MIT Press, Cambridge, MA, 2000.

[3] Murphy, G.C., Notkin D., and Sullivan, K.J., "Software Reflexion Models: Bridging the Gap between Design and Implementation", *IEEE Transactions on Software Engineering*, Vol.27, No. 4, April 2001

[4] Tran, J.B., Godfrey M.W., Lee E.H.S., Holt, R.C., "Architectural Repair of Open Source Software", *Proc. of 2000 Intl. Workshop on Program Comprehension* (IWPC-00), Limerick, Ireland, June 2000.

[5] Parnas, D.L., "Designing Software for Ease of Extension and Contraction", *Transaction on Software Engineering*, SE-5(2), 1979

[6] Jackson, D., "Module Dependences in Software Design", *Post-workshop Proceedings of the 2002 Monterey Workshop: Radical Innovations of Software and Systems Engineering in the Future* (Venice, Italy October 7-11, 2002). Springer Verlag, 2003.

[7] Sullivan K., Cai Y., Hallen B., Griswold W., "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*, 2001

[8] Kruchten, P., "The 4+1 View Model of Architecture", *IEEE Software* 12(6): 42-50, 1995

[9] Gebala, David A. and Eppinger, Steven D., "Methods for Analyzing Design Procedures", *Proceedings of the ASME Third International Conference on Design Theory and Methodology*, pp. 227-233, 1991.

[10] Warfield, John N., "Binary Matrices in System Modeling" *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 3, pp. 441-449, 1973.

[11] Hartigan, John A., "*Clustering Algorithms*," John Wiley & Sons, New York, 1975

[12] The Haystack Project. MIT Computer Science and Artificial Intelligence Laboratory. http://haystack.lcs.mit.edu/.

[13] Browning, T. "*Applying the Design Structure Matrix to System Decomposition and Integration problems: A Review and New Directions*". IEEE Transactions on Engineering management, Vol. 48, No. 3, August 2001.

[14] Pimmler, Thomas U. and Eppinger, Steven D., "Integration Analysis of Product Decompositions", Proceedings of the ASME Sixth International Conference on Design Theory and Methodology, Minneapolis, MN, Sept., 1994.

[15] Fernandez, CIG, "Integration Analysis of Product Architecture to Support Effective Team Co-location", Master's Thesis (ME), MIT 1998.

[16] H. A. Müller, K. Wong, and S. R. Tilley. "Understanding software systems using reverse engineering technology." *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings* (ACFAS 1994).

[17] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[18] Cristina Videira Lopes and Sushil Bajracharya, "An Analysis of Modularity in Aspect-Oriented Design," *Proc. Aspect-Oriented Software Development (AOSD'05)*, Chicago, March 2005.

[19] Alan MacCormack, John Rusnak and Carliss Baldwin, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code", *Harvard Business School Working Paper Number 05-016*.

[20] Clemens Szyperski, "Component Software - Beyond Object-Oriented Programming", *ACM Press/Addison- Wesley*, 1997.

[21] R. Kazman, S. J. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence", *Journal of Automated Software Engineering*, 6:2, April, 1999, 107-138.

[22] Yassine, Ali, "An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure Matrix (DSM) Method", *Quaderni di Management (Italian Management Review), www.quaderni-di-management.it*, No.9, 2004.