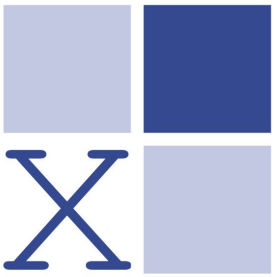
The Lattix Approach

Design Rules to Manage Software Architecture

Whitepaper

December 2004-7



Introduction

The Lattix approach uses a DSM to represent the architecture of software systems. This approach leads to a hierarchical decomposition of the software system and the use of a DSM grid to represent the dependencies between subsystems. In the past, DSMs have been used to capture and represent architectures; Lattix Dependency Model has extended this usage to allow architectural enforcement. This is done through Design Rules.

Design Rules allow us to tackle one of the thorniest problems in software. It has frequently been noted that software begins to degrade over successive revisions. A key reason behind this common phenomenon is the inability to communicate and enforce architectural intent. Over successive revisions, changes to software no longer adhere to the original architecture. Often, new developers change things in unintended ways. These changes are necessary for the evolution of the software system to support new capabilities. However, the changes made to accomplish this are made without either a clear understanding of the current architecture or a clear understanding of how the architecture should evolve to support those changes.

As will be seen in this paper, the DSM grid itself provides a powerful representation for setting and visualizing design rules. Further, the grid makes it easy to pinpoint violations of design rules and to understand where the violations come from.

What are Design Rules?

Design Rules are a way to specify the allowed nature of the relationships between various subsystems. This specification is an important part of the Lattix Dependency Model for formalizing the architecture of a software system. Once these rules have been codified, newer versions of the software can be tested to enforce these rules. These rules serve two important purposes:

1. They flag architectural errors that developers might make during routine development. It is these errors which erode the integrity of the architecture over time. Frequently these errors are a result of changes made to software systems for routine bug fixes and minor improvements.
2. They capture critical changes to the architecture that might necessitate changes to the system decomposition or to how subsystems interact with each other. By forcing the architect to come up with new design rules when such changes become necessary they make architectural evolution explicit.

Lattix LDM is an application that allows architects and developers to specify design rules and then to monitor the evolution of the system with respect to conformance to those rules.

Specifying Design Rules

Rules are Inherited

Suppose there is a System *S*, which has been decomposed into the following subsystems:

$$S = S_1 + S_2 + \ldots + S_N$$

Consider the following rule:

$$S_2 \textit{ can-use } S_1$$

This rule is applied to the subsystem S_2. It says that the subsystem S_2 is allowed to depend on the subsystem S_1. When rules are applied to a subsystem, those rules are generally inherited by the children of that subsystem. Therefore, all subsystems which compose S_2 are allowed to depend on subsystem S_1. Said another way, all descendents of S_2 are allowed to depend on all descendents of S_1.

A design rule normally consists of three parts: (1) source, (2) verb, and (3) target. In the preceding example, S_2 is the source, *can-use* is the verb, and S_1 is the target.

Now consider another rule:

$$S_1 \textit{ cannot-use } S_2$$

The rule says that the subsystem S_1 is not allowed to depend on the subsystem S_2. This means that no subsystem within the S_1 subsystem tree is allowed to depend on S_2.

This technique allows a simple enforcement of a common design paradigm for software systems: A software system is typically decomposed into subsystems which are layered. For instance, S_1 might represent the framework of the application while S_2 might represent the application business logic. These rules would then represent the common intuition that the application's framework should not depend on its business logic while the business logic is certainly expected to depend on the services provided by the framework. Enforcement of these rules allows multiple business applications to use a common application infra-structure. Such layering also simplifies testing by enabling independent testing of lower layers.

Rules are Evaluated in Sequence

Rules are evaluated in sequence and can be over-ridden. Assume that the subsystem *S1* and *S2* can be further decomposed as follows:

$$S_1 = S_{11} + S_{12} + \ldots + S_{1N1}$$

$$S_2 = S_{21} + S_{22} + \ldots + S_{2N2}$$

Based on our first rule, we know that S_{11}, S_{12}, .. S_{1N1} are not allowed to use S_2. Upon examining the current architecture, the architect finds that S_{11} actually does depends on S_{21}. This violates the architectural intent. In software, like most other pragmatic domains, it is entirely possible that the cost of change is excessive; it is also not hard to conceive that this dependency is actually necessary for reasons of performance or simplicity. The architect can then specify the following additional rule:

S_{11} *can-use* S_{21}

This rule explicitly over-rides a part of the rule that S_{11} inherited from S_1. LDM applies rules in sequence. When the subsystem S_{11} is evaluated, it is evaluated for the following rules:

S_{11} *cannot-use* S_2 (this rule is inherited from S_1)

S_{11} *can-use* S_{21}

By evaluating the rules in sequence, parts of the rule can be over-ridden. Rules can be marked as **exceptions**. This allows architects to indicate rules created to accept architectural violations which might exist for reasons which could be historical, or related to performance or to scheduling. The architect may also attach a rationale for rules.

Rules can be applied to External Systems

The Lattix Dependency Model also allows design rules to specify the external libraries that subsystems can use. Consider the following example:

S_1 *can-use org.apache.***

This rule applies to all subsystems within the S_1 subsystem tree. All of them are allowed to use external types whose names start with *org.apache*. This provides a technique to control the proliferation of external library usage. From a theoretical perspective, this is not very different from rules between subsystems that were previously just described. However, from a practical standpoint it has great benefits as it is neither useful nor practical to build a DSM which includes the system being analyzed and all its supporting external libraries and its operating environment.

Rules can be qualified

The Lattix Dependency Model also allows rules to be qualified by *dependency kind* and *atom kind*. Each Dependency is of a specific kind based on the type of project. For instance, Java has dependency kinds of type inheritance (extends, implements), method invocation, data member reference etc while .NET has additional dependency kinds associated with .NET constructs such as Events and Properties.

Each leaf node of the DSM tends to be associated with an atom which corresponds to the types of atoms associated with a project. For instance, the atoms in a Java project are of kind: classes, interfaces, methods and data members. This means that it is possible to create rules such as enforcing access from one subsystem to another through interfaces.

Using a DSM Grid to Represent Design Rules

The DSM grid provides a powerful way to visually represent the rules. We use green and yellow triangles at different vertices of the cell to indicate whether a dependency is permitted. For a *can-use* rule, the cell has a green triangle in the upper left vertex; and, for a *cannot-use* rule, the cell has a yellow triangle in the lower left vertex. If there is a dependency in a cell governed by a *cannot-use* rule, we show it with a red triangle on yet another vertex of the cell (upper right).

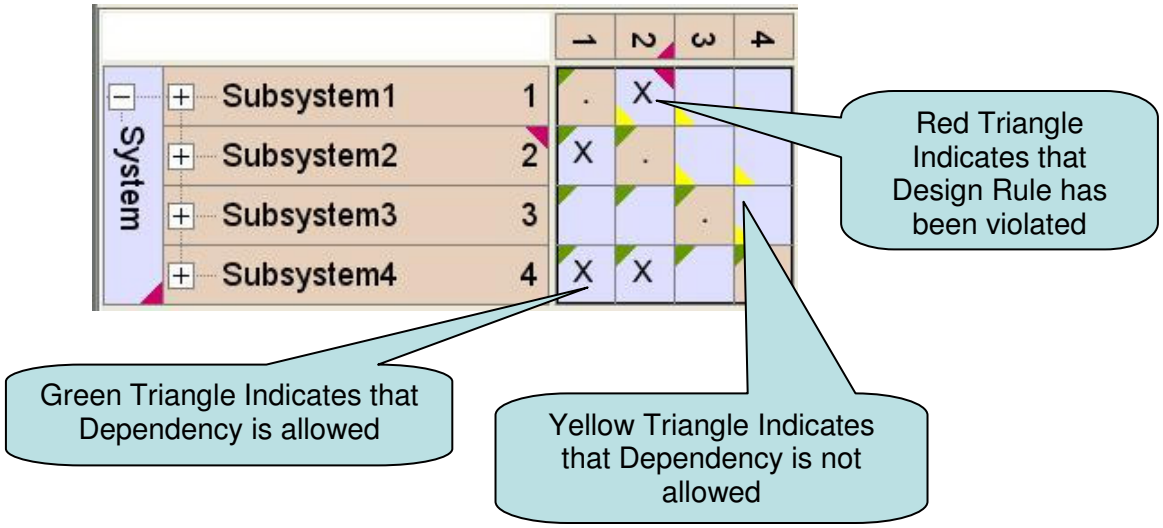


Figure 1: Dependencies and Rules in the DSM Grid

The use of a DSM for representing design rules illustrates yet another benefit of the DSM grid. Every element of the grid represents design intent. The traditional representation which relies on directed graphs becomes cluttered and incomprehensible when used just for showing dependencies. Using a directed graph to show design intent would be even more difficult as a line segment would be required between every subsystem to every other subsystem.

Lattix further improves upon this view by allowing users to click on any cell to see the actual dependency, rule and rule violation. The grid navigation is simple and intuitive. It is simple for users to drill down into any subsystem to identify exactly which subsystem is responsible for violating the design rule.

Example: Applying Design Rules to Apache ANT

Ant is a one of the most popular build utility. It allows development teams to automate the build process for activities such as compiling, building jar files, unit testing etc. The architecture of Ant has been specifically developed so that Ant tasks are components of the Ant infrastructure. This has permitted a large number of disparate developers to work in parallel to create the wide variety of things (tasks) that Ant can do. The clean separation between the infrastructure and tasks has also added to the robustness of Ant because bugs in the tasks have a minimal affect the rest of the system.

A Dependency Model was constructed for Ant Version 1.4.1. First the system decomposition of the Ant application was done. This was done by noting that a key design decision behind Ant's architecture was to separate the Ant framework from Ant's tasks. Tasks depend upon the framework but the framework does not depend on tasks. This allows tasks to be added and tested independently. It also reduces the risk to the entire application because of bugs that might be introduced in newly added or modified tasks.

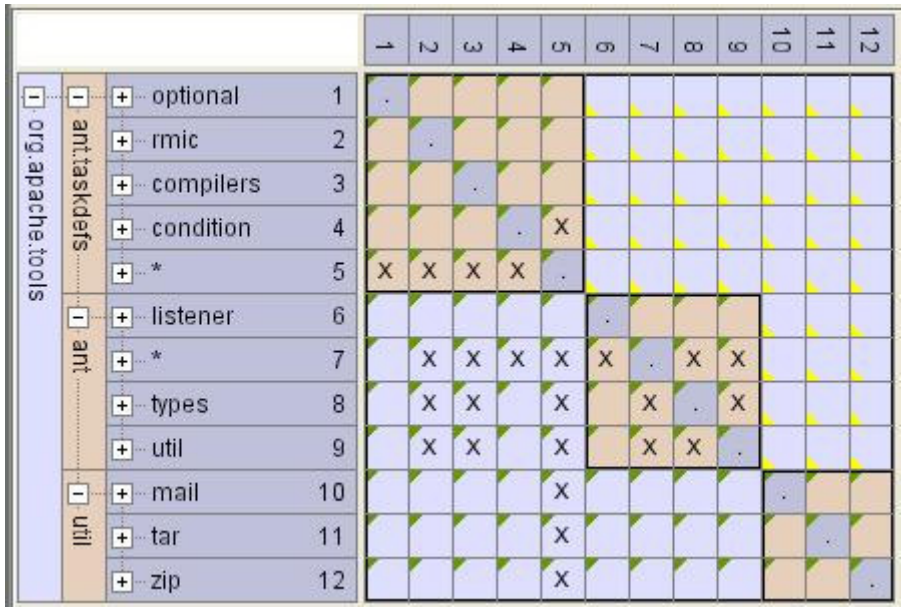


Figure 2: Ant Version 1.4.1 with dependencies and rules

Figure 2 shows the DSM for Ant Version 1.4.1. It has been decomposed into three subsystems – *util*, *ant*, *taskdefs*. The dependencies between various subsystems reflect the hierarchical layering that actually exists within the implementation. We have added rules to enforce this intended layering. The cells with yellow triangles indicate areas where dependencies are not permitted.

The following rules were added (from an initial state where every subsystem is allowed to use any other subsystem):

util cannot-use ant

util cannot-use taskdefs

ant cannot-use taskdefs

Note that these 3 rules now govern the cells above the block diagonal, and those cell all show a yellow indicator in the lower left corner, signifying a cannot use rule applies.

As a next step we applied this dependency model to Ant Version 1.5.1.

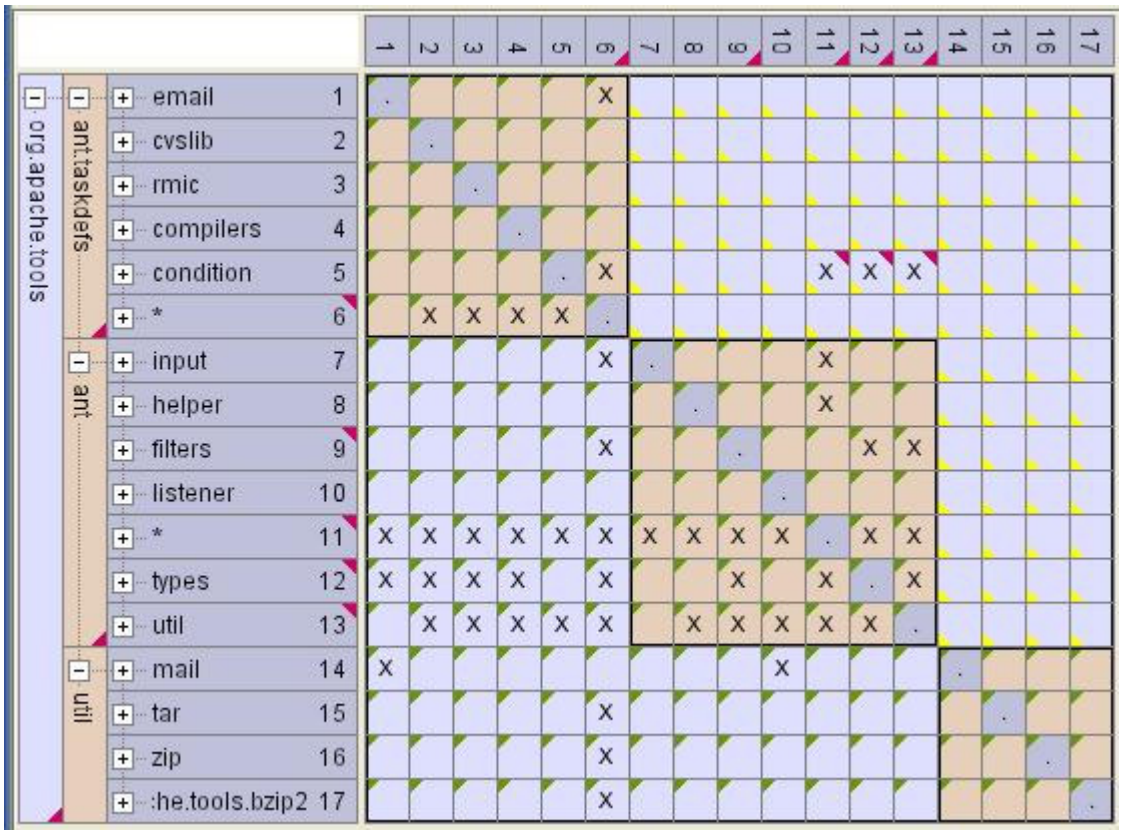


Figure 3: Ant Version 1.5.1 (has Rule Violations)

Notice that the architecture is largely intact. However, the ant framework now has dependencies on the condition subsystem in *taskdefs*; these dependencies are clearly identified by the cells in the grid with the red triangles in the upper right corner.

LDM allows you to move subsystems from one place to another. We moved the condition task from *taskdefs* to *ant*. This removed the current violations but introduced a new violation from the *condition* tasks to *taskdefs*. This points to the need for further refactoring to maintain the architectural intent.

We also applied the dependency model to Ant Version 1.6.1. This now showed additional violations of the intended layering. They illustrate how a design begins to degrade over time. Ant is a popular application which is scrutinized by hundreds of developers. Most applications that are written today will never receive that same

scrutiny. They are likely to degrade much more quickly unless the architectural intent is clearly codified and enforced with design rules.

