

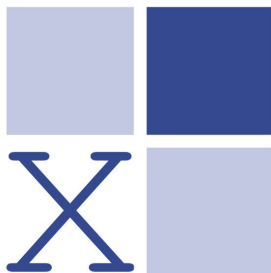


# **Creating a Lattix Dependency Model**

## **The Process**

**Whitepaper**  
**January 2005**

Copyright © 2005-7 Lattix, Inc. All rights reserved



## The Lattix Dependency Model

The Lattix LDM™ solution employs a unique and powerful strategy for communicating, visualizing, analyzing, and managing a software system's architecture. The software architecture is formalized in terms of a Lattix Dependency Model (LDM). An LDM allows the architect to formalize and enforce the overall architecture of a system through design rules, and to leverage DSM as a key component of the approach for controlling the complexity of that system. Furthermore, the LDM serves as a blueprint of the architecture that is easy for managers and developers to understand, which makes it easier for business issues to be addressed in the development process.

The process of creating a Dependency Model involves the use of LDM and leveraging the knowledge that architects already have about their system. This process is iteratively applied to refine and update the architecture over the lifecycle of the product.

We suggest that you should read the Lattix White Paper "DSM for Software Architecture" before reading this paper. "DSM for Software Architecture" explains describes DSMs and how they can be used to manage a Software System's Architecture. We also recommend that users familiarize themselves with LDM. The LDM tutorial helps them use LDM to understand and observe the evolution of the architecture of Ant, an Open Source Java Build Utility.

### Creating an LDM

The process of creating an LDM requires a hierarchical decomposition of the system. The architect's intentions of how these subsystems are expected to interact with each other can then be encoded in the form of design rules.

We suggest the following steps:

1. **Understand the Application Use** – Establish a working knowledge of the function and use of the Application being analyzed and managed. For parties that did not develop the application, some suggestions including reading user documentation and running the application to get an understanding of what the application does.
2. **Create a preliminary DSM based on the code structure** – The code organization is frequently a good starting point because developers naturally tend to group related functionality. Also, as an introduction to the DSM, using familiar structure helps developers understand how to read the DSM. LDM creates an initial DSM based on the current structure. For Java, LDM uses the package structure; for .NET LDM uses the name space to create the ; for C/C++ LDM uses the file/directory structure; for Oracle LDM uses the schema/table organization.
3. **Interview the Architects** – The architects and senior developers have a deep understanding of how the application is structured. If the application is very large it may be necessary to interview several architects each of whom is responsible for different parts of the system. A **conceptual architecture** diagram is then created and a DSM to reflect that conceptual architecture can then be presented to the architects for their comments.
4. **Refine the DSM** – Based on comments received, changes can be made in the subsystem organization to reflect the conceptual architecture. This could involve moving subsystems, adding new levels in the hierarchy and even removing subsystems that are not relevant. At this time, the abstractions represented by the various subsystems should be clearly understood.

5. **Examine the Dependency Structure** – Examine the dependencies to see the ones which appear to violate modularity based on the knowledge of the abstractions represented by the subsystems. One of the big advantages of a DSM is that from amongst the large number of dependencies it lets you focus your attention on those dependencies that appear to violate architectural principles.
6. **Define the Design Rules** – Define the design rules based on the understanding of subsystem abstractions and their inter-relationships. Document design rationale for rules as part of constructing the rules. The dependencies that need to be fixed are marked as exceptions at the appropriate level. If necessary, propose a short-term and long-term architecture remediation plan.

Note that the ordering of the steps is merely suggestive; it is frequently changed because the every individual situation is different. Some systems have much better documentation; at other places, there are key architects who possess the critical insights about how the system is designed. Note that it may also be necessary to repeat some of the steps. Repetition occurs during step wise refinement; it also occurs because the documentation is outdated or because the information provided by the architect does not match the actual system implementation.

## Analyzing the Architecture

There are a number of activities that can be performed to achieve key insights into the architecture when creating the Lattix Dependency Model. For consistency with the Lattix whitepaper, "DSM for Managing Software Architecture", ANT is used to provide an example for selected activities.

### ***Initially Focus on the Essential***

Start by eliminating what you know is not an essential part of the architecture. When you initially load in a large number of files, it becomes hard to understand which of the subsystems are important for the analysis. This is especially true if you are looking at files that you know very little about.

Some of the files that we eliminate when we begin to create a DSM include:

- Files that only contain tests
- Files that are auto generated such as messages, state machines etc.
- External libraries such as java and apache

### ***Identify Independent File Groupings***

If your system consists of multiple containers (jars in the case of Java, dlls in the case of .NET) make sure that you select the option "Create Subsystem for Source File". Then, Partition the DSM. If you see that the DSM partitions into mutually independent groupings of jar files, you can safely look at the independent groupings separately.

		1	2	3	4	5
+ mqjms.jar	1	.	1			
+ mq.jar	2	26	.			
+ commons-resources.jar	3			.		
+ commons-services.jar	4				.	
+ commons-digester.jar	5			2	3	.

This figure shows that the jar files *mqjms.jar* and *mq.jar* are part of one independent system while *commons-resource.jar*, *commons-services.jar*, *commons-digester.jar* are part of another independent system.

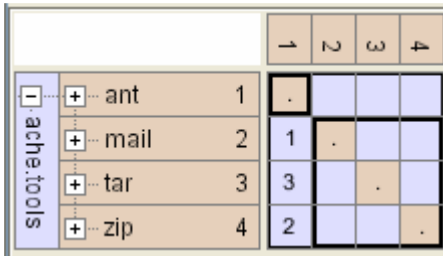
Note that once you have identified the independent file groupings then you can look at them separately or create a new abstraction which contains them. For instance you could create a new partition called 'mq' which contains *mqjms.jar* and *mq.jar*. Similarly you can create a new partition called *commons* which contains *commons-resource.jar*, *commons-services.jar* and *commons-digester.jar*.

## Identify the High-level and Low-level Subsystems

High level subsystems are those parts of the system which are consumers of services that are provided by the lower parts. An application's user interface is an example of a higher level layer.

Lower level subsystems are those parts of the system that provide services to the higher layer. A data access layer is generally an example of a lower level layer.

In well architected systems, lower layers tend not to depend on higher level layers and high level layers do not have other subsystems that depend on them. You can frequently figure out lower level and higher level layers simply by applying DSM Partitioning. For systems that are architected in such a way that the layers are well defined and there are no cycles, DSM Partitioning will order the subsystems correctly.



The image shows a DSM Partitioning matrix for the subsystems under the package `org.apache.tools`. The matrix is a 4x4 grid with columns labeled 1, 2, 3, and 4. The rows represent subsystems: ant (1), mail (2), tar (3), and zip (4). The matrix shows dependencies: ant depends on mail, tar, and zip; mail depends on tar; tar depends on zip; and zip has no dependencies.

	1	2	3	4
ant (1)	.			
mail (2)	1	.		
tar (3)	3		.	
zip (4)	2			.

### DSM for ANT when First Loaded

DSM Partitioning shows us that the subsystems `org.apache.tools.mail`, `org.apache.tools.tar` and `org.apache.tools.zip` are low-level subsystems. `org.apache.tools.ant` is a high-level subsystem which uses the services provided by the low level subsystems.

For complex or poorly structured systems, you might find that the matrix is densely populated and therefore it is not clear which subsystems are low level and which are high level. Examining the dependencies on external libraries can sometimes be very helpful. If the application contains a user interface, simply by seeing which subsystems depend on `javax.swing` or `java.awt` can be help identify the higher layers. Similarly, the subsystems which interface with the database can sometimes be identified by their dependency on `java.sql` or some other proprietary data access library.

## Examine Dependency Strengths

Dependency Strengths are extremely useful in deciding which dependencies are “strong” and which are “weak”. Frequently, subsystems depend on each other and it is unclear if these subsystems are layered or at the same level. The strength of the dependency between subsystems is a strong hint about which subsystem is at a lower level and which is at a higher level. Low Strength of Dependency also frequently reflects a dependency between subsystems that could be eliminated with greater ease.

## Create a Conceptual Architecture and Refine

One of the most important milestones in coming up with the Dependency Model is to define a conceptual architecture for the system. A conceptual architecture is a high level decomposition of the system into its subsystems. We suggest that you come up with a conceptual decomposition as soon as you have a rough understanding of the system. As you proceed with the analysis, the decomposition will change.

The conceptual architecture should be confirmed with architects and senior developers to see if it matches their mental model. In some cases, parts of the conceptual architecture can also be extracted from "marketecture diagrams" that organizations create for explaining the architecture to their customers or to their user communities.

A conceptual architecture has a number of elements:

- The conceptual architecture is hierarchical. It is not uncommon to have a hierarchy which is several levels deep. However, initially it is only necessary to define one or two of the top levels of the hierarchy. Deeper decomposition of the subsystems can be done in subsequent refinements.
- Each of the subsystems should represent a clear named abstraction. Ideally, the package name should name the abstraction. Lattix Dependency Model allows you to give a logical name that is different from the actual package name.
- The nature of dependencies between the subsystems comes from the nature of the abstractions represented by the subsystems. Unexpected dependencies generally indicate violation of the intended subsystem abstraction and ultimately require either fixing the code or a revising the architecture.

If we examine the DSM for ANT Version 1.4.1, we notice that the code organization indicates that ANT is composed of two parts.

			1	2	3	4	5	6	7	8
org.apache.tools	ant	+ taskdefs	1	.						
		+ listener	2		.					
		+ *	3	17	5	.	8	3		
		+ types	4	19		6	.	1		
	+ util	5	7		2	1	.			
	Util	+ mail	6	1					.	
		+ tar	7	3						.
		+ zip	8	2						

One part is a lower layer that we have called "Util" for the utilities (*tar*, *mail*, *zip*) that are shipped with Ant. When we partition the subsystem *org.apache.tools.ant*, we discover that the subsystem called *org.apache.tools.ant.taskdefs* is a high-level subsystem that uses the services provided by the rest of the subsystems.

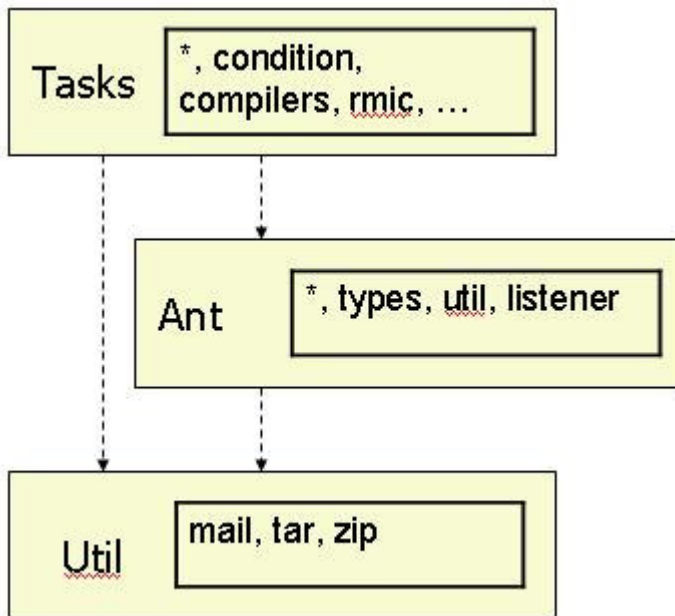
In Ant, the relative independence of the Tasks from the rest of the Ant engine is a key design decision. It allowed independent developers to create a very large number of useful Ant Tasks on a common infra-structure. Now, this decision is not documented. The package naming structure does not reflect this decision either. We arrived at this

conclusion by examining the DSM and by reading published books on Ant and reading the documentation.

We decomposed Ant into three main subsystems:

- Tasks
- Ant Engine
- Util

We draw a conceptual diagram using a typical box and arrow diagram. The arrows show the intended direction of dependencies between the subsystems.



### Conceptual Architecture for Ant

A DSM corresponding to the conceptual architecture is now easy to create. This will require categorizing every single package into the appropriate high level subsystem of the conceptual architecture. The architecture of each of the top level components can then be defined in a similar incremental refinement step.

		1	2	3	4	5	6	7	8	9	10	11	12
org.apache.tools	ant:taskdefs												
	+ compilers	1	.			2							
	+ condition	2		.		2							
	+ optional	3			.								
	+ rmic	4				.							
	+ *	5	5	2	1	2	.						
ant	+ listener	6					.						
	+ *	7	5	4		7	17	5	.	8	3		
	+ types	8	4			5	19		6	.	1		
	+ util	9				2	7		2	1	.		
Util	+ mail	10					1				.		
	+ tar	11					3					.	
	+ zip	12					2						.

Note that it is not uncommon to see unintended dependencies. Many of the dependencies can be cleared by moving subsystems or splitting up subsystems. Sometimes it is necessary to change the actual code in order to fix the dependencies.

## Creating Hierarchy in Subsystems with many Children

For subsystem with a very large number of children, DSM Partitioning will offer you suggestions for possible organizations. Coupled with the knowledge of that subsystem, you can quickly come up with an organization that is reasonable and makes architectural sense.

## Identify Components

Components are an important part of modular architectures. They can frequently be added incrementally. Developers of components generally need only a limited view of the overall subsystem. They rely on a few well documented APIs for accessing the services of the system. On the other hand, developers of the infra-structure that is used by the components should not need actual knowledge of the components; they only need to understand the abstraction of the services that the components provide.

As part of identifying the components, identify exactly what parts of the system the components are allowed to use and identify the interfaces that the components implement. It is also important to identify the inter-relationship between components and whether these relationships should be allowed or not. This issue is particularly important in software product lines where products are flexibly created from an underlying set of software assets.

## Identify Key Classes

As part of architectural analysis it is frequently useful to pinpoint the key classes in a software system. These classes represent important abstractions and as result a large number of other classes tend to depend on them. When these classes themselves depend on many other classes, they become 'change propagators'. When a system is modified, these classes are affected and in turn they affect a large number of other classes.

Stable systems have stable key classes. This implies that most changes to the system do not require changes to key classes. This localizes change and thereby reduces the risk to the system from enhancements.

## Make Adjustments

Many of the dependencies can be resolved by moving one or more subsystems from one subsystem to another.

		1	2	3	4	5
ant:taskdefs	+ compilers 1	.				2
	+ condition 2		.			2
	+ optional 3			.		
	+ rmic 4				.	2
	+ * 5	5	2	1	2	.

		1	2	3	4	5
ant:taskdefs	+ compilers 1	.				2
	+ condition 2		.			2
	+ optional 3			.		
	+ rmic 4				.	
	+ * 5	5	2	1	2	.

Note that the dependency between `org.apache.tools.ant.taskdefs.*` and `org.apache.tools.ant.taskdefs.rmic` was removed by moving the class `org.apache.tools.ant.taskdefs.Rmic` from `org.apache.tools.ant.taskdefs.*` to `org.apache.tools.ant.taskdefs.rmic`. There are many such small adjustments that can improve the DSM and make the architecture much cleaner.

## Split up Packages

Some packages lose meaningful abstraction over time. These packages may have names such as `misc`, `common`, `util` or `*`. Architecture begins to degrade over time when programmers create classes but are unsure which package they belong in. As a result, classes which are really part of different subsystems end up in some of these miscellaneous packages. These classes inside these packages need to be moved to the appropriate package. By examining the dependencies of these with other subsystems is the way to determine where these classes should be moved.

## Fixing and Maintaining the Architecture

### ***Rename Packages***

After you have completed building the dependency model, we suggest that you rename your packages and classes to reflect the actual system decomposition. This makes it easy and intuitive for developers to understand and maintain the architecture. It also increases the likelihood that new modules will be put in the correct subsystem.

### ***Select Dependencies that Require Refactoring***

Ultimately, architectures are all about dependencies between various subsystems. In general there are millions of possible combinations between subsystems for most reasonable sized subsystems. Of all these possible, Dependency Models enable you to focus on the small number of dependencies that require attention. Generally, these are dependencies which violate your intended design rules. For layered systems, these are dependencies in the upper triangular part of the DSM.

### ***Split up Key Classes***

If key classes are found to be change propagators, it is useful to consider splitting those classes into multiple classes and interfaces. For instance, the "Project" class in Ant is a change propagator. As a result, the ANT engine as a monolith subsystem. Developers making changes to this subsystem need to understand a large part of the engine. This adds to the risk when the engine is updated. Ultimately, when the Ant engine becomes too large then change will become increasingly difficult. Breaking up Project into separate subsystems would help in restructuring the Ant engine. Classes can be split up into separate independent classes and interfaces. Classes can also be split up into super classes and sub classes.

### ***Use Design Patterns to Invert or Limit Dependencies***

There are design patterns which can be employed to hide or to invert dependencies. The *Listener Design Pattern* is one of the most commonly used design pattern and is used to invert a dependency. *Listener* is used by a subsystem to signal another subsystem. The subsystem which is waiting for the event registers itself as a listener and receives an event on the Listener Interface. This eliminates the need for the signaling system to know anything about the listening system. Other patterns such as a *Facade* or a *Visitor* can be used to limit the knowledge that one subsystem needs to know about another.

### ***Specify Design Rules***

Once you have created a Dependency Model for your Software System, you can specify design rules to communicate and enforce the architecture of your system. Note that Dependency Model evolves as your software changes. Design Rules allow you to manage this evolution explicitly.

You can specify design rules to specify:

- The dependency between subsystems and external libraries
- The dependency between subsystems

Please read Lattix White Papers "Design Rules" and "Manage Use of External Libraries" for additional detail.