

Lattix LDM

Analysis of Software Artifacts

CMU 17-654/17-754

**Kenichi Nakao
Rikuo Kittaka
Radhika Bansal
Karoon Phatangjaijing
Thomsun Sriburadej**

Mar 25, 2008

Table of Content

- A. PROJECT SUMMARY 4**
- B. PROJECT MEMBERS 4**
- C. PROJECT SUMMARY 4**
 - A. EVALUATION TOOL..... 4
 - B. ARTIFACTS AGAINST WHICH WE TESTED THE TOOL 4
- D. QUALITATIVE/QUANTITATIVE DATA ANALYZED..... 4**
 - A. FUNCTIONALITY 4
 - i. System decomposition..... 4
 - ii. Design Structure Matrix (DSM) 5
 - iii. Analyzing architecture design 5
 - iv. Analyzing 3rd party library..... 7
 - v. Metrics Data 9
 - vi. Design Rules..... 12
 - B. USABILITY..... 15
 - i. Benefits..... 15
 - ii. Limitations 16
 - iii. What practices are good to learn the tool quickly 16
 - iv. Learn-ability of document and tutorial 17
 - v. How easily we can find the most critical changes in the system 17
 - vi. How much amount of customization is required for a project 17
 - C. MODIFIABILITY (CUSTOMIZABILITY) 18
 - i. What kinds of options are available? What kinds of option sets are recommended? 18
 - ii. What kinds of options are preferable for our studio project? 21
 - D. PERFORMANCE 21
- E. TOOL STRENGTHS AND WEAKNESSES 22**
 - A. STRENGTH 22
 - B. WEAKNESS 22
- F. APPENDIX 1: FILTER DEPENDENCIES 22**
- G. REFERENCES 24**

Table of Figures

Figure 1: Pipe-n-Filter System	5
Figure 2 :Call-Return System.....	6
Figure 3: Lattix reports wrong class names 1	6
Figure 4: Implicit-Invocation System.....	7
Figure 5: Lattix reports wrong class names 2	7
Figure 6: The result before partitioning	8
Figure 7: The result after automatic partitioning.....	8
Figure 8: The result after manually grouping.....	8
Figure 9: Metrics comparison.....	12
Figure 10: The system that has design rules.....	13
Figure 11: The system that violates design rules.....	13
Figure 12: The tool reports that our libraries need those three packages.....	14
Figure 13: All rule violations have gone after setting design rules	14
Figure 14: Basic User Interface.....	16
Figure 15: Screen for designing a rule	18
Figure 16: Filter Dependencies.....	22

A. Project summary

Evaluation of Lattix LDM

B. Project members

Radhika Bansal, Rikuo Kittaka, Thomsun Sriburadej, Karoon Phatangjaijing, Kenichi Nakao

C. Project summary

Assess the strengths and weaknesses of the Lattix LDM in both quantitative and qualitative terms. We would be especially, focusing on applicability and adoptability to the project including following points:

- The efficiency of the tool for helping inexperienced users to take benefit from the tool
- The suitability of the tool for various kinds of architecture
- Any other benefit that the tool would give besides analyzing the architecture

a. Evaluation Tool

Lattix LDM 4.05 for Java 1.4 and above

b. Artifacts against which we tested the tool

The artifacts that we used belonged to one of the following categories:

- Source code
- Architecture Design

Following are some of the artifacts that we tested the tool against:

1. JEdit : Programmer's text editor written in Java
2. Ant : Java build utility
3. A1-A4 : Assignments from Architecture class
4. Sample code from Design pattern book

D. Qualitative/Quantitative data analyzed.

a. Functionality

i. System decomposition

Lattix uses "design structure matrix" (DSM) to present the system. This technique allows users to understand the system easily by decomposing the system into subsystems. Users can use this feature to adjust a proper level of abstraction that makes system understandable. Moreover, the information associated with each subsystem is also visible in every level of composition. Users can easily comprehend the overview of the system by looking at single table [1].

This feature is useful in comparing with UML class diagram. For understanding the overall system, UML class diagram would have several levels of presentation to match the size of the project and the need of user. Comparing with Lattix, the tool provides a single matrix that allows user navigate back and forth between abstract level and concrete level.

ii. Design Structure Matrix (DSM)

DSM technique is the result of a research from MIT, Harvard, and University of Illinois that uses matrix, rather than boxes and lines diagram, to show dependency among subsystems. This technique allows users to realize architecture of the system, and identify a place of the system that has high or low coupling.

The tool also provides the “DSM Partitioning” feature. This feature automatically partitions subsystems into groups so that users can see pattern of the system quickly. Users can partition the system manually as well. Logical subsystem can be created to group subsystems together. Subsystems can be moved up and down in the matrix. In the product’s website, Lattix shows a sample analysis with ANT, which has layered structure. We try Lattix with other kinds of architectural patterns to see if the tool could help us to realize and analyze the architecture design from the source code.

iii. Analyzing architecture design

○ **Pipe-n-Filter Pattern**

For pipe-n-filter pattern, we use source code of A1, the first assignment from our architecture class, as a test case. Since one of intent of this pattern is to decouple each filter, each filter in this pattern will not depend on each other. Therefore, low coupling was the expected result from the tool.

The result was in line with the expectation. From Lattix, we can see that the filters do not have any dependencies within themselves; they are just dependent on the “SystemMain” class, which is the start up class.

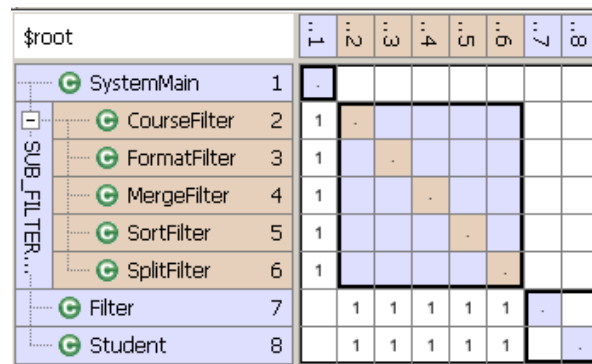


Figure 1: Pipe-n-Filter System

○ **Call-Return Pattern**

The second pattern is call-return pattern. We tested a 3-tier system that uses RMI connection for calling services, and expected a result of high coupling from Lattix. The result that we got was quite surprising. After both automatically and manually partitioning, we had found dependencies only at the interface area. On the other hand, no dependency appears among the concrete tier classes, which are Data, Logic, LogicWithLog, and Client.

The reason is that the system uses interface class for communication among tiers, and thus decouple the classes of each tier. In this case, the tool gives us more insight

about architecture.

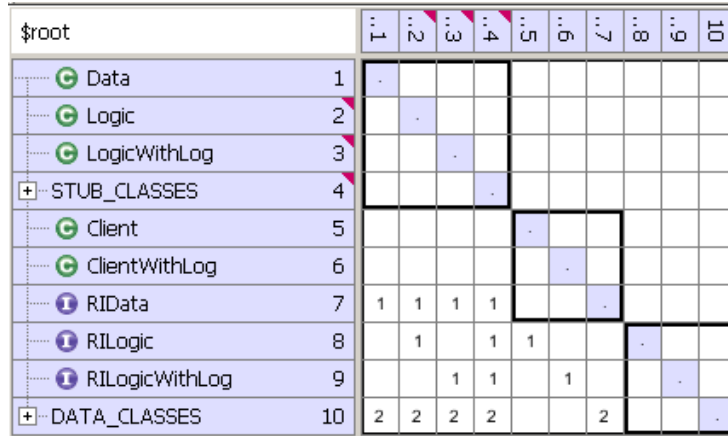


Figure 2 :Call-Return System

However, we have found a bug in data in 'Rules'. This system does not use any other library so that the system should not break any rule by default. Lattix wrongly determines the class name in the system. Therefore, it reported some violations from this system.

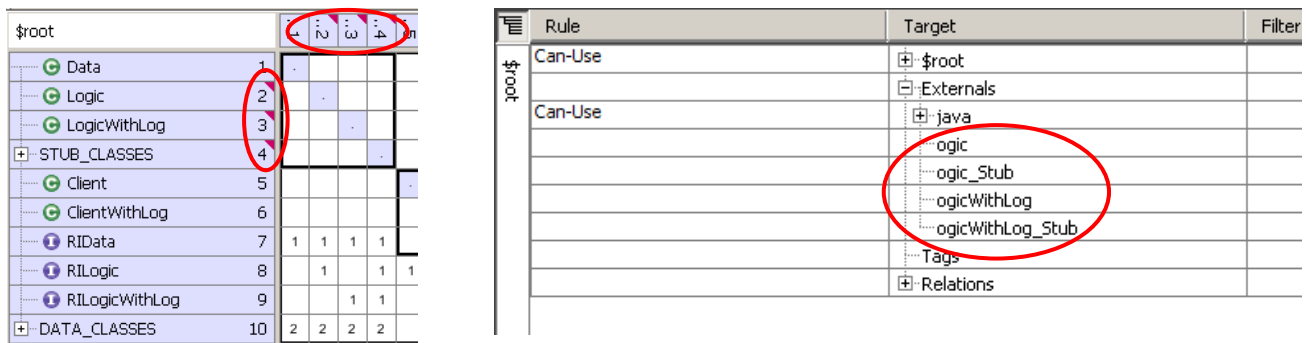


Figure 3: Lattix reports wrong class names 1

○ **Implicit-Invocation Pattern**

We expected to see lowest dependency from this pattern. We use architecture assignment 3 to evaluate the result. After partitioning, the result was not bad. Lattix can show that no concrete event handler classes depend on each other.

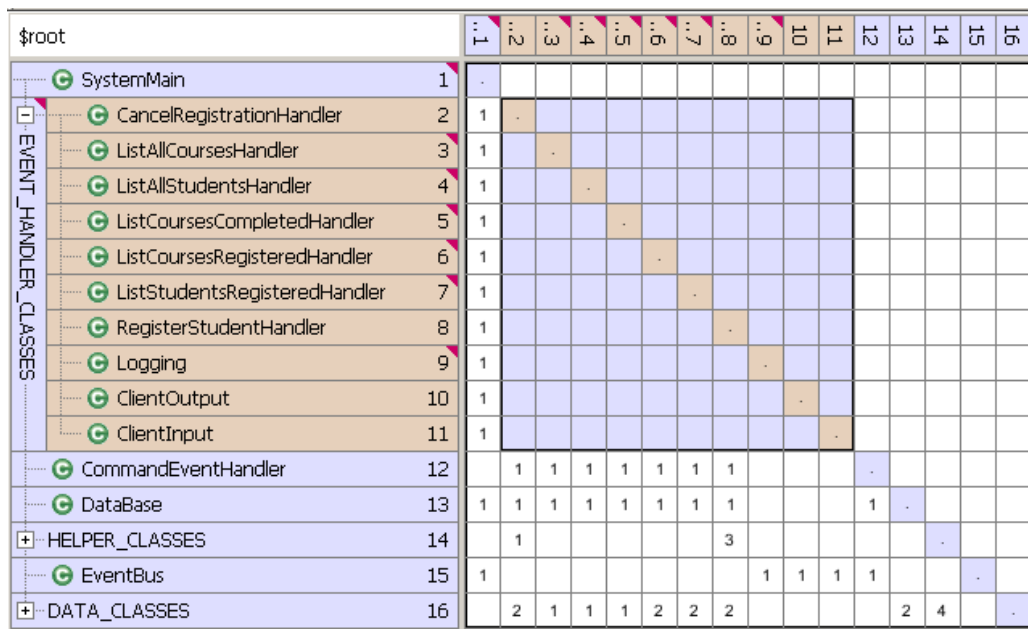


Figure 4: Implicit-Invocation System

Similar to previous example, the tool still report wrong result in the design rule.

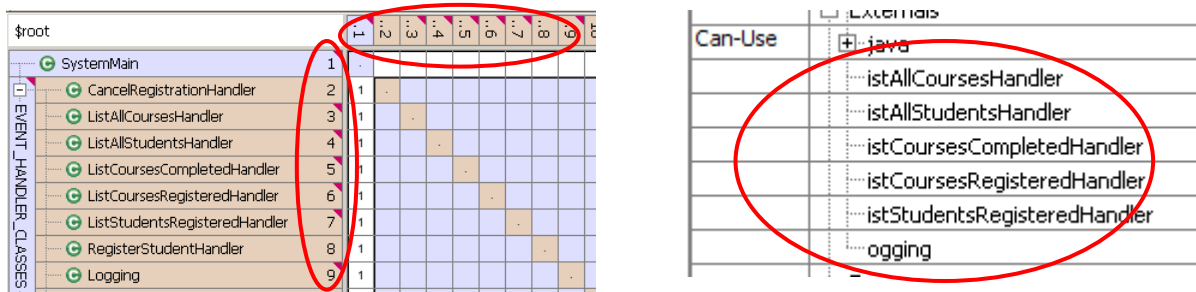


Figure 5: Lattix reports wrong class names 2

iv. Analyzing 3rd party library

Most of Java application today leverage some form of 3rd party libraries. However, one component may require a set of components, and these dependencies become messy and hard to manage. Not only does the tool can show dependencies among the source code, but it also helps user to understand dependencies among 3rd party libraries. We examine this feature by having the tool analyze “jelly” framework, which is an open source framework that we use in “experimentation phase” of our studio project.

This framework use another set of 3rd party library. When Lattix loads all required libraries, it shows DSM as following.

\$root		←	↶	↷	→	↵	↷
+ jelly-tags-swt.jar	1	.	18	52	10	11	
+ jelly.jar	2		.				
+ commons-beanutils-1.6.jar	3	40	.				
+ commons-jexl-1.0.jar	4	5	32	.			
+ commons-collections-2.1.jar	5		4		.		
+ dom4j-1.5.2.jar	6		6	5		.	
	7		10				.

Figure 6: The result before partitioning

First, we use “DSM Partitioning” feature to partition the system automatically

\$root		←	↶	↷	→	↵	↷
+ jelly-tags-swt.jar	1	.					
+ jelly.jar	2	40	.				
+ commons-beanutils-1.6.jar	3	5	32	.			
+ commons-jexl-1.0.jar	4		4		.		
+ commons-logging-1.0.3.jar	5	18	52	10	11	.	
+ commons-collections-2.1.jar	6		6	5			.
+ dom4j-1.5.2.jar	7		10				.

Figure 7: The result after automatic partitioning

After having the tool partitioning, the result is quite easy to interpret. The ‘jelly-tags-swt’ is built on top of ‘jelly’ library. Additionally, the ‘jelly’ library is built on top of other libraries. According to the type of library, the ‘commons-beanutils-1.6’ and ‘commons-collections-2.1’ could be grouped together because they both are dealing with JavaBean. Similarly, the ‘dom4j-1.5.2’ and ‘commons-jexl-1.0’ that are dealing with XML could also be grouped together. We manually group them as ‘BEAN_UTIL_GROUP’ and ‘XML_UTIL_GROUP’ respectively.

\$root		←	↶	↷	→	↵	↷
+ jelly-tags-swt.jar	1	.					
+ jelly.jar	2	40	.				
+ XML_UTIL_GROUP	3		14	.			
+ BEAN_UTIL_GROUP	4	5	38		.		
+ commons-logging-1.0.3.jar	5	18	52	11	10	.	

Figure 8: The result after manually grouping

Now we have a good picture. Logging feature is the base library that all libraries require. The classes in ‘jelly’ use XML and Bean features, as well as logging feature. The ‘jelly-swt,’ which is built as an tag extension of ‘jelly,’ lets ‘jelly’ handle XML parsing, but it can handle object bean and logging itself.

v. Metrics Data

Lattix LDM provides matrixes to measure architecture of a system:

- **System Stability:** A measure of how sensitive a system is when the system is changed.
- **Average Impact:** A measure of how much it affects when the class is changed.
- **Instability:** How concrete a class is. The more concrete implementations are used along with high dependency, the more Instable a system will become.
- **Abstractness:** A measure of how abstract is an implementation.
- **Distance:** A measure of abstractness sum with instability.
- **Outgoing Dependencies:** The number of classes outside a subsystem depended upon by classes in side that subsystem.
- **Incoming Dependencies:** The number of classes outside a subsystem that depend on classes inside that subsystem.
- **Atom Count:** The number of classes in a system.
- **Abstract Atom Count:** The number of abstract classes in a system.

Our group will compare metrics obtained from the tool with the metrics obtained by manual calculations, to check correctness of the tool.

Average Impact

Class	Dependency Count	Classes Depended Upon
CheesePizza	4	SimplePizzaFactory, PizzaHomeStore, PizzaMenu, PizzaStore
ClamPizza	4	SimplePizzaFactory, PizzaHomeStore, PizzaMenu, PizzaStore
PepperoniPizza	4	SimplePizzaFactory, PizzaHomeStore, PizzaMenu, PizzaStore
VeggiePizza	4	SimplePizzaFactory, PizzaHomeStore, PizzaMenu, PizzaStore
Pizza	8	All other classes
PizzaHomeStore	0	
PizzaMenu	0	
PizzaStore	0	
SimplePizzaFactory	3	PizzaHomeStore, PizzaMenu, PizzaStore

$$\begin{aligned}
 \text{Average Impact} &= \text{Sum of Dependency Count} / \text{Atom Count} \\
 &= 27 / 9 \\
 &= 3
 \end{aligned}$$

System Stability

$$\begin{aligned}
 \text{System Stability} &= 100\% - (\text{Average Impact} / \text{Atom Count}) * 100 \\
 &= 100\% - (3/9) * 100 \\
 &= 66.67\%
 \end{aligned}$$

Outgoing Dependency

Outgoing Dependency is the number of classes outside a subsystem depended upon by classes inside that subsystem.

Class	Outgoing Dependency Count	Outgoing Classes Depended Upon
-------	---------------------------	--------------------------------

CheesePizza	1	Java.util.*
ClamPizza	1	Java.util.*
PepperoniPizza	1	Java.util.*
VeggiePizza	1	Java.util.*
Pizza	3	Java.util.*, Java.io.*, Java.lang.*
PizzaHomeStore	1	Java.lang.*
PizzaMenu	1	Java.lang.*
PizzaStore	1	Java.lang.*
SimplePizzaFactory	1	Java.lang.*

$$\begin{aligned} \text{Total outgoing dependency} &= \text{Sum of Outgoing Dependency Count} \\ &= 11 \end{aligned}$$

Incoming Dependency

Incoming dependency is the number of classes outside a subsystem that depend on classes inside that subsystem. Our subsystem don't have a call from the outside classes so

$$\begin{aligned} \text{Total incoming dependency} &= \text{Sum of Incoming Dependency Count} \\ &= 0 \end{aligned}$$

Instability

Instability = $\text{Outgoing Dependencies} / (\text{Outgoing Dependencies} + \text{Incoming Dependencies})$

Class	Outgoing Dependency Count	Incoming Dependency Count	Instability
CheesePizza	1	0	1
ClamPizza	1	0	1
PepperoniPizza	1	0	1
VeggiePizza	1	0	1
Pizza	3	0	1
PizzaHomeStore	1	0	1
PizzaMenu	1	0	1
PizzaStore	1	0	1
SimplePizzaFactory	1	0	1

$$\begin{aligned} \text{Total instability} &= \text{Sum of Instability / Atom Count} \\ &= 9 / 9 \\ &= 1 \end{aligned}$$

Abstractness

Class Pizza is the only abstract class in the subsystem.

$$\begin{aligned} \text{Abstractness} &= \text{Abstract Atom Count / Atom Count} \\ &= 1 / 9 \\ &= 0.111 \end{aligned}$$

Distance

$$\begin{aligned} \text{Distance} &= | \text{Abstractness} + \text{Instability} - 1 | \\ &= | 0.111 + 1 - 1 | \\ &= 0.111 \end{aligned}$$

Compare with tool result

Metrics	Manually Count	Tool Count	Correctness
System Stability	66.67 %	66.67 %	<input checked="" type="checkbox"/>
Average Impact	3	3	<input checked="" type="checkbox"/>
Instability	1	1	<input checked="" type="checkbox"/>
Abstractness	0.111	0.111	<input checked="" type="checkbox"/>
Distance	0.111	0.111	<input checked="" type="checkbox"/>
Outgoing Dependencies	11	11	<input checked="" type="checkbox"/>
Incoming Dependencies	0	0	<input checked="" type="checkbox"/>
Atom Count	9	9	<input checked="" type="checkbox"/>
Abstract Atom Count	1	1	<input checked="" type="checkbox"/>

Significant of the metrics

Our group experimented on the significant of the metrics provided by the tool by running the tool on two test subject: High coupled code, Low coupled code. The High coupled code is the code that has high dependency between classes. The original code is borrowed from 'Head First Design Pattern' book (the factory method pattern). The low coupled code is improved version of the high coupled code. We applied *Factory Method pattern* to reduce dependency between classes.

Metrics	Factory Pattern	High Coupled Pattern	Significant
System Stability	66.67 %	70.31%	<input type="checkbox"/>
Average Impact	3	2.375	<input type="checkbox"/>
Instability	1	1	<input checked="" type="checkbox"/>
Abstractness	0.111	0.125	<input checked="" type="checkbox"/>
Distance	0.111	0.125	<input checked="" type="checkbox"/>
Outgoing Dependencies	11	10	<input checked="" type="checkbox"/>
Incoming Dependencies	0	0	<input checked="" type="checkbox"/>
Atom Count	9	8	<input checked="" type="checkbox"/>
Abstract Atom Count	1	1	<input checked="" type="checkbox"/>

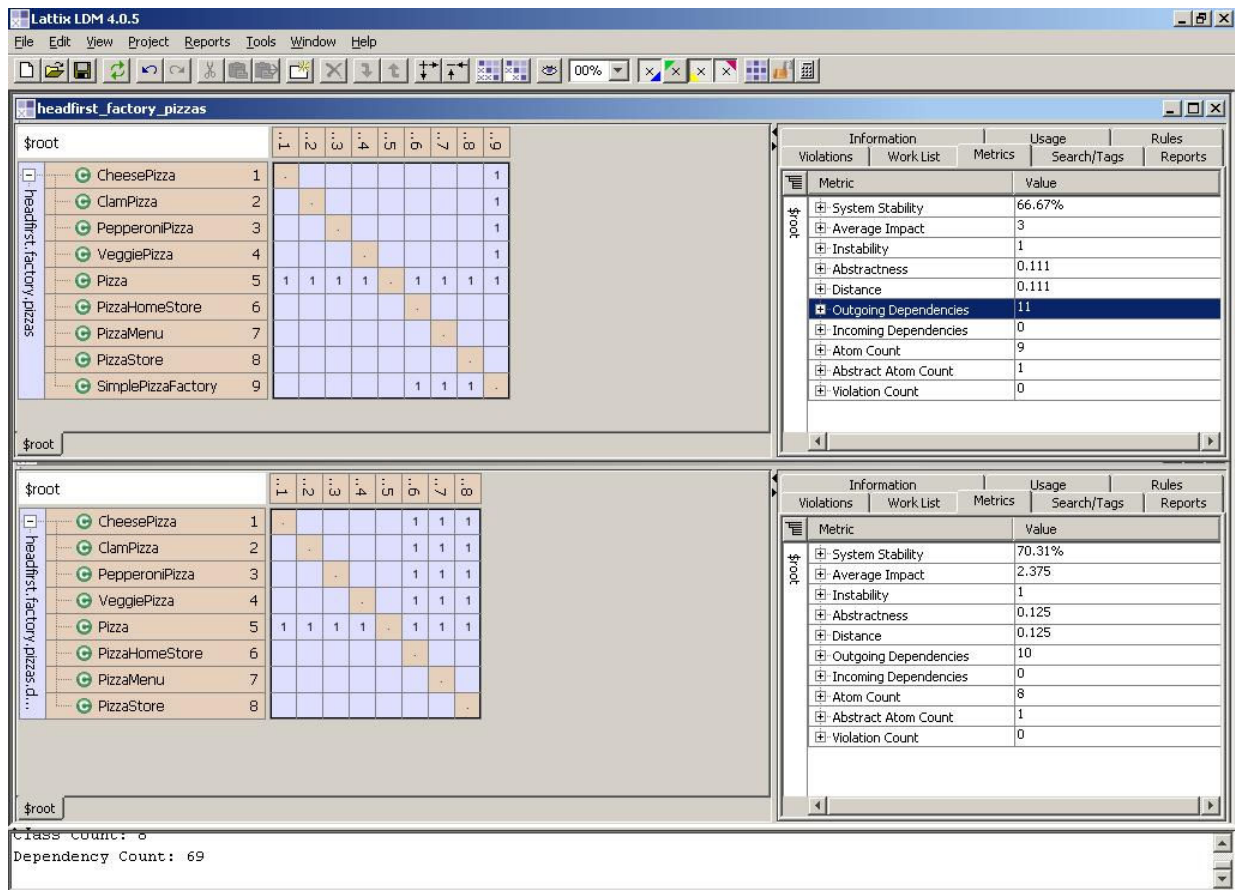


Figure 9: Metrics comparison

Results

The System Stability and Average Impact metrics from the tool show negative result (difference from expectation). The high coupled code has higher System Stability and has lower Average Impact. The reason of this is because the tool counts dependency, in a transitive manner. For example, if class A depends on B, and B depends on C, Both A and B depend on C. As a result, the tool can't recognize the utility of the Factory class.

vi. Design Rules

Design Rules are a way to specify the allowed nature of the relationships between various subsystems [2]. During development time, changes to software might not adhere to the desired architecture. Although those changes are important to the system capability, those changes must be done carefully and intentionally. The purpose of this feature is to enforce the change to the system even without clear understanding of the current architecture or effect of change.

User can set 'Can-use' or 'Cannot-use' rules, and apply those rules to subsystems. They could apply at as low level as method level. User can use this feature to restrict subsystem from using a specific component.

Design Rules with source code

We use the A1 code to set up design rules and try to break them. A1 is pipe-n-filter system that all filters will communicate only over filter. Therefore, no filter is allowed to use any service from other filters.

We set up these rules by selecting package or subsystem we want to set the rules to. Open the 'Rules' tab at the right hand side, and specify 'Cannot-Use' to the package or subsystem that we will not allow. The rules will be shown in the matrix as a small yellow annotation at the corner.

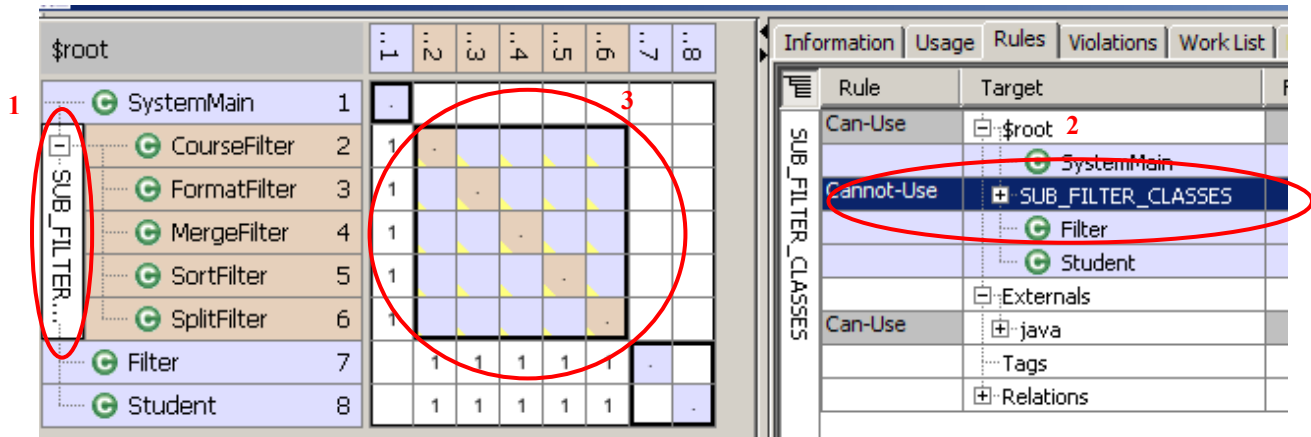


Figure 10: The system that has design rules

Next, we break the rules by adding the following code into SplitFilter.

```
// Break the rule!!!!//
new CourseFilter("Name", pInput, pOutput1, 0);
```

After updating source file, Lattix shows us a violation that takes place in the system. We are also able to the summary of violation at the 'Violations' tab.

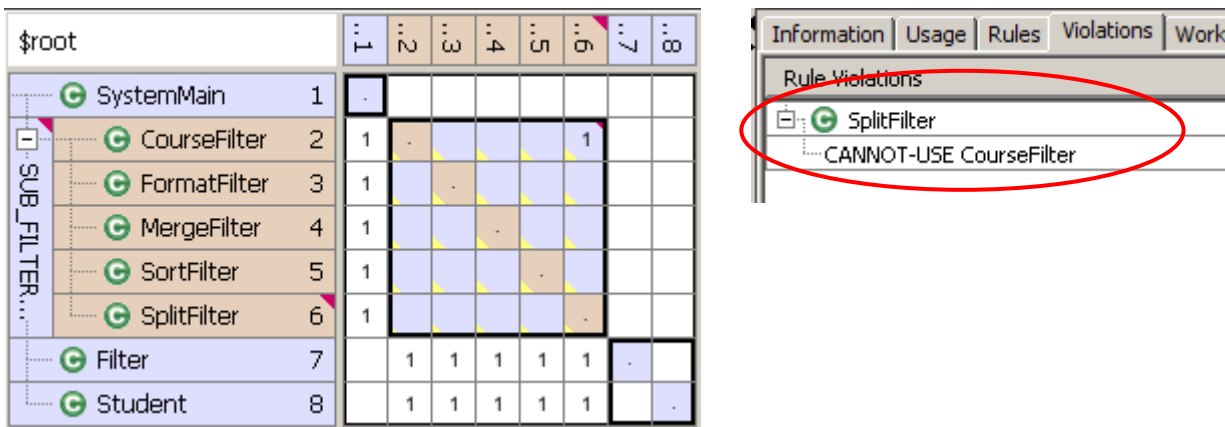


Figure 11: The system that violates design rules

Design Rules with 3rd party library

The design rules feature of Lattix LDM can also apply to 3rd party for managing and controlling. User can set 'Can-use' or 'Cannot-use' rules, and apply those rules to the library. They could apply the rules at either as high as the package

level or as low as method level. User can use this feature to restrict some packages from using a specific library.

According to the whitepaper [3], the tool also provides a command line program that can also validate design rule, called LDC. LDC could be plugged into tightly build process so that the violation result could be passed to the library administrator. LDC also provides other metrics, such as number of classes and dependencies, and classes that have been added or removed. This feature notifies user of any violation and status of the source code early in the development phase. Unfortunately, we could not analyze this feature because of license issue.

However, this feature could be misleading. The previous example of “jelly” framework shows a set of 3rd party library for a small program. The tool reports violation for all other external libraries that our libraries are referencing; even though they are not necessary for running the application. The following picture highlights the package name that our libraries are referencing, but they are not allowed to use yet.

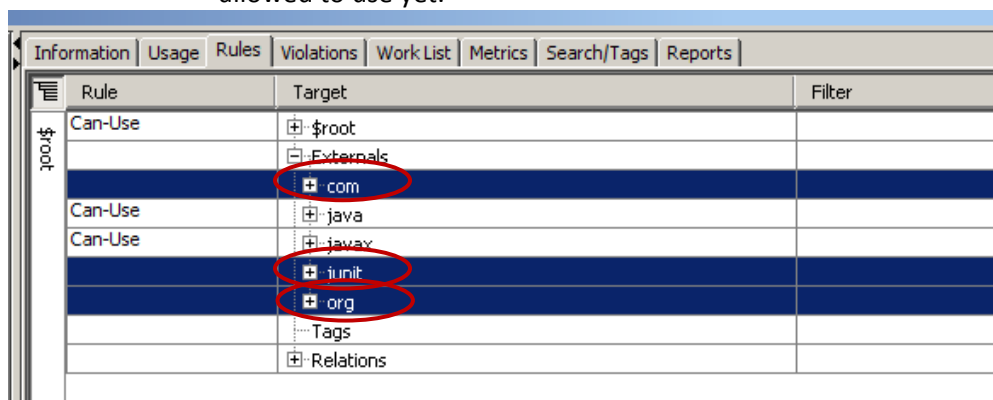


Figure 12: The tool reports that our libraries need those three packages

When we allow our library to use those packages name, the warning is gone.

Dependency	Count
jelly-tags-swt.jar	1	.			
jelly.jar	2	67	.		
XML_UTIL_GROUP	3		36	.	
BEAN_UTIL_GROUP	4	5	72		.
commons-logging-1.0.3.jar	5	22	122	18	29

Figure 13: All rule violations have gone after setting design rules

The tool counts all dependencies, including the one that we don’t really use. For example, package ‘com.sun.msv.datatype’ is used by several libraries here, but this package is not necessary for the application.

In the case that we don’t know what library file is missing, it is hard to identify from the “Rules” tab that what package is the missing or the violation rules shown by the tool is just unnecessary reference. Therefore, this feature is good

for having a better understanding relationship among 3rd party library, but it could not help user to identify what library is missing.

b. Usability

i. Benefits

- The tool provides a simple and intuitive user interface so that we can generate the analysis result (DSM: Dependency Structure Matrix) without complex operation. Figure14 shows a basic user interface and representation. To see the initial analysis result, the user only needs to open the jar, class, or zip files without any configuration. It did not take a long time (about 1hour) to understand the representation of the tool and key features.
- The tool supports undo function for most of operations and its depth is sufficient; we tested by 50 depth
- The tool provides a command-line application that allows us to automate the process of checking and updating the dependency model. Specifically, it support:
 - Create a new model or update a current model
 - Generate reports in a variety of formats
 - Publish reports using a web server

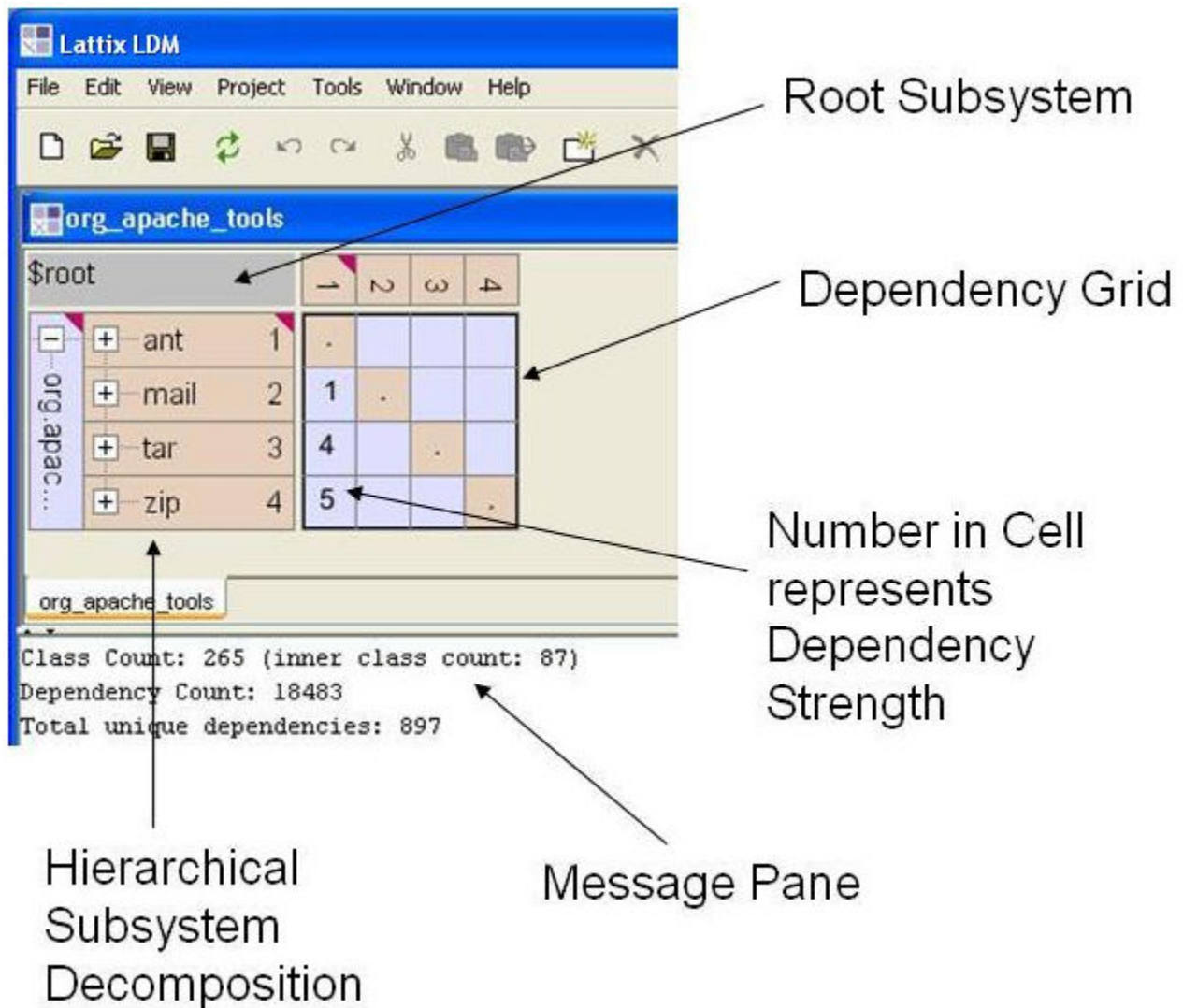


Figure 14: Basic User Interface

ii. Limitations

One of our group members had a trouble to start up the application on his PC. The cause was the environment variable PATH that includes a path for QuickTime application ("C:\Program Files\QuickTime\QTSystem\"). Though this is trivial issue, it took a long time for users to figure out the problem since there is no information about this problem on FAQ page of the tool site. *We have already informed this issue to Lattix, Inc's support representative.

iii. What practices are good to learn the tool quickly

The most effective way to understand the tool is to follow the process the Lattix Inc. provides on the site and apply our own application to the tool. The company also provides a tour which will walk you through the tutorial using a Macromedia Flash demo. 1 hour was enough to learn the big picture of the tool.

iv. Learn-ability of document and tutorial

The Lattix Inc. provides us enough document and tutorial to learn the tool and the way to apply to our application. Specifically, they provide us:

- Basic instruction on the site
- Demo using a Macromedia Flash on the site
- Tutorial included in the tool under the Help menu
- Technical Paper on the site (to dig deeper into the background)

It took 4 hours to go through from 1 to 3 materials including the time to play around with the tool. The documents were well structured so that it was not demanding work for users to learn the tool using the same.

v. How easily we can find the most critical changes in the system

Since the Lattix provides a feature that allows users to follow the impact chain to see how changes propagate, we can easily understand the impact of changes on specific module.

On the other hand, it is not easy to find the most critical changes in the system because dependency strength in the DSM tells us only the number of direct dependency (level 1). If the tool showed the sum of the dependency strength on the change impact chain in a DSM, it would help us find the most critical changes based on the quantitative data.

vi. How much amount of customization is required for a project

We need to have three customization steps to make the use applicable to our project as follows:

- Organize the DSM to reflect the Intended Architecture
- Design Rules: Specify External Library Usage
- Design Rules: Specify Application Interdependencies

Each step can be done through the simple GUI and using small options (move subsystems around, create/delete new abstractions, and select rules). Figure15 shows the screen for designing a rule. If we have a clear understanding of the architecture of the application that we are analyzing, the amount of customization is not so many. In addition, once we create the rules, we can save the rule and make rule checking a part of the build using command-line application.

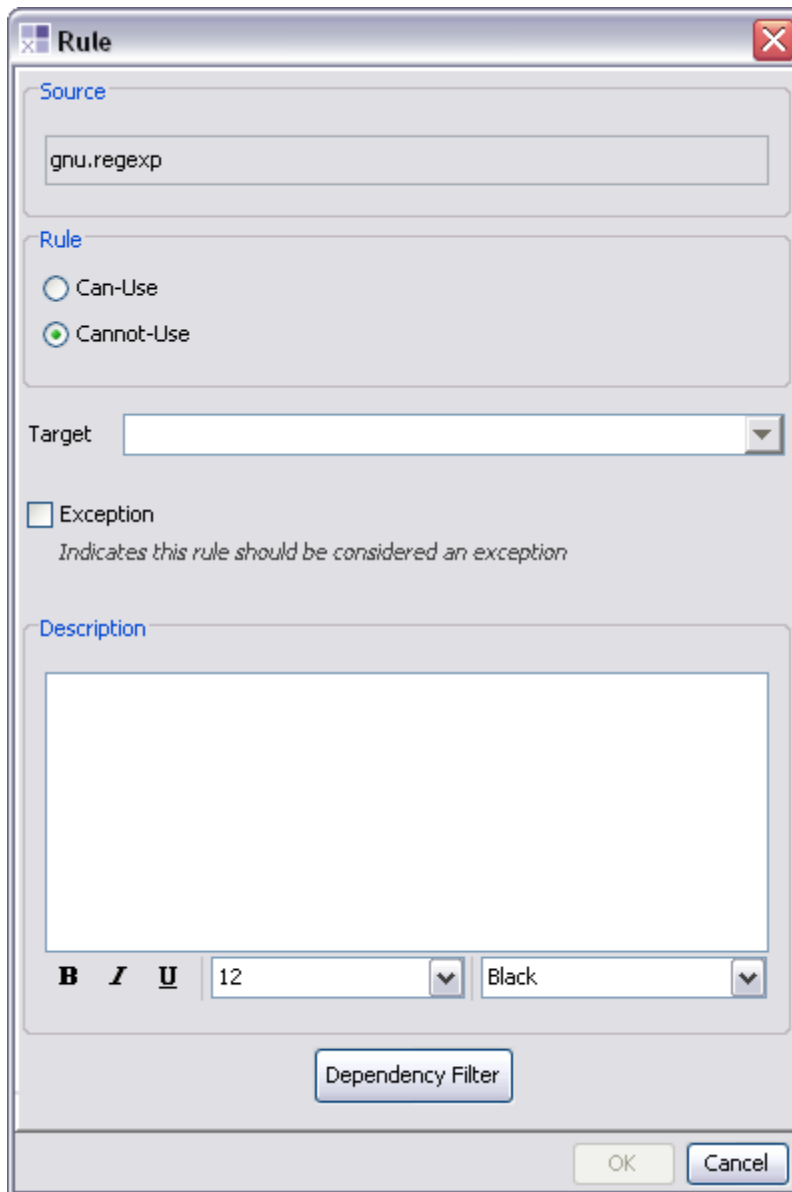


Figure 15: Screen for designing a rule

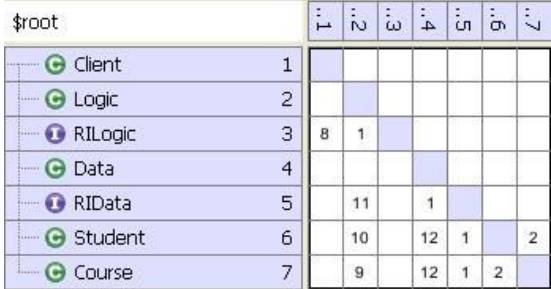
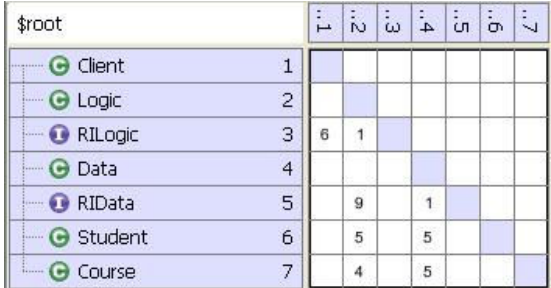
c. Modifiability (Customizability)

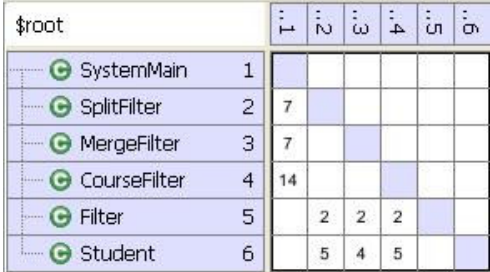
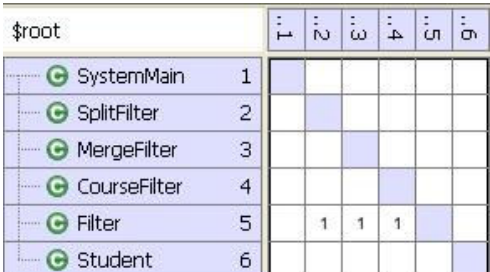
i. What kinds of options are available? What kinds of option sets are recommended?

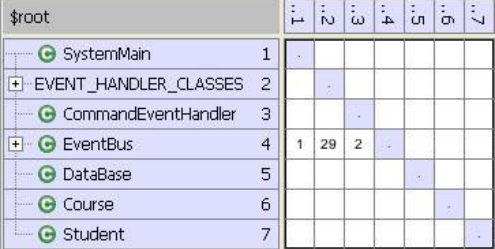
Lattix LDM provides a feature of filtering dependencies that enables users to analyze different kinds of dependencies among modules. It is called “Filter Dependencies.” The dependency kinds the Lattix LDM supports are shown in Appendix X. By the combination of the dependency kinds, users can see a clearer picture of the system structure.

Some of the recommended sets of options are as follows:

	Settings	Purpose	Evaluation

<p>Option Set 1: All kinds of dependencies</p>	<p>Check all settings</p>	<p>To see the most exhaustive dependencies</p>	<p>To see the trend of the dependencies of the system, this setting provides a good start. However, since this setting counts any kinds of syntactic dependencies including inheritance and class reference, we cannot identify what kinds of dependencies exist between the modules. Therefore, the other sets of options discussed below would be required for further dependency analysis.</p>
<p>Option Set 2: Strong dependencies</p>	<p>Check all settings except "Class Reference" and "Constructs - Null".</p>	<p>To find stronger dependencies excluding weak references such as class reference (no method calls) and construct with no arguments.</p>	<p>Lattix LDM could show only the strong dependency correctly. In the case of A2, the figure below shows the case of Option Set 1 "All kinds of dependencies".</p>  <p>The figure below is in the case of the Option Set 2 "Strong dependencies".</p>  <p>We could see that the RIData has only weak references to Student and Course class so that the change of the member methods of the Students and Courses class do not affect the RIData.</p>

<p>Option Set 3: Generalization</p>	<p>Check “Inherits” and check off the others</p>	<p>To find subclasses and implemented classes that has generalization architecture/design style.</p>	<p>In the case of A1, there is a “Extends” relation between the Filter class and Sprit/Merge/CourseFilters. By using the this option setting 3 “Generalization”, Lattix LDM could identify the dependency.</p> <p>The figure below is in the case of Option Set 1 “All kinds of dependencies”.</p>  <p>The figure below is in the case of Option Set3 “Generalization.”</p> 
<p>Option Set 4: Member object reference</p>	<p>Check only “Data member Reference”</p>	<p>To find violations of data encapsulation (direct access to class members outside the class.)</p>	<p>Lattix LDM could identify the data member reference. In the case of A1, A2, and A3, Lattix LDM found zero dependencies between the class modules, correctly.</p> <p>In the JEdit case, we found 8 occurrences. One of the 8 is the access to a class member that is declared as “public”. This might be corrected by using Getter/Setter. The others are declared as “public static.”</p>

<p>Option Set 5: Static Invocation</p>	<p>Check only “Invokes – Static Invocation”</p>	<p>To identify static method invocation for architecture dependency refinement. Because of the static nature, they are built to stand alone, and operate the same, even when they are moved from their original class to a new class home.</p>	<p>In the case of A3, the static method invocation was concentrated to the Event Bus class.</p>  <p>Therefore we could say the designer grouped the static method well.</p>
--	---	--	---

ii. What kinds of options are preferable for our studio project?

For the Studio project (Tool support for economic driven architecting (EDA Tool)), one of the key quality attributes of the system is modifiability of the economic analysis methods, calculation methods, and calculation formulae. The goal of the quality requirement is to provide users a capability to add and modify the methods and formulae by configuration file with a minimum code modification. To achieve the quality response measure goal, fewer strong dependencies between the modules is required. Therefore, the recommended set of options would be the checking the counts and distribution of the “Strong dependencies”, as well as “All kinds of dependencies”. The “Generalization” option might help us to analyze the architecture strategy to achieve the modifiability.

d. Performance

Machine Spec: Intel core 2 CPU T7200 @ 2GHz

Source	file type	interval	1st	2nd	3rd	4th	5th
A1	class	file open -> show DSM	1"36	0"55	0"61	0"48	0"55
A2	class	file open -> show DSM	0"77	0"57	0"55	0"59	0"49
A3	class	file open -> show DSM	0"75	0"56	0"60	0"64	0"66
jEdit	class	create project -> show DSM	2"25	1"95	2"00	2"15	2"40
apache-ant-1.7.0	jar	create project -> show DSM	N/A element over (our licence is limited by 1000 elements)				

E. Tool Strengths and weaknesses

a. Strength

- Fast loading even when opening a huge set of source code
- Matrix is good to present a big system
- Support several file types, comparing with other reverse engineering tool
- It seems like this tool could be used for any architectural styles, since any style should be related to static view one way or another.

b. Weakness

- Lack of semantic dependency (Explicitly know from the general information)
- Helps user realize only one perspective of system quality (modifiability), while other perspective, such as performance, availability, or security, are left out.
- Some bugs with 'Rules'
- Design rule could not help users to find missing 3rd party lib, as expected.
- Cannot disable transitive dependency count.

F. Appendix 1: Filter Dependencies

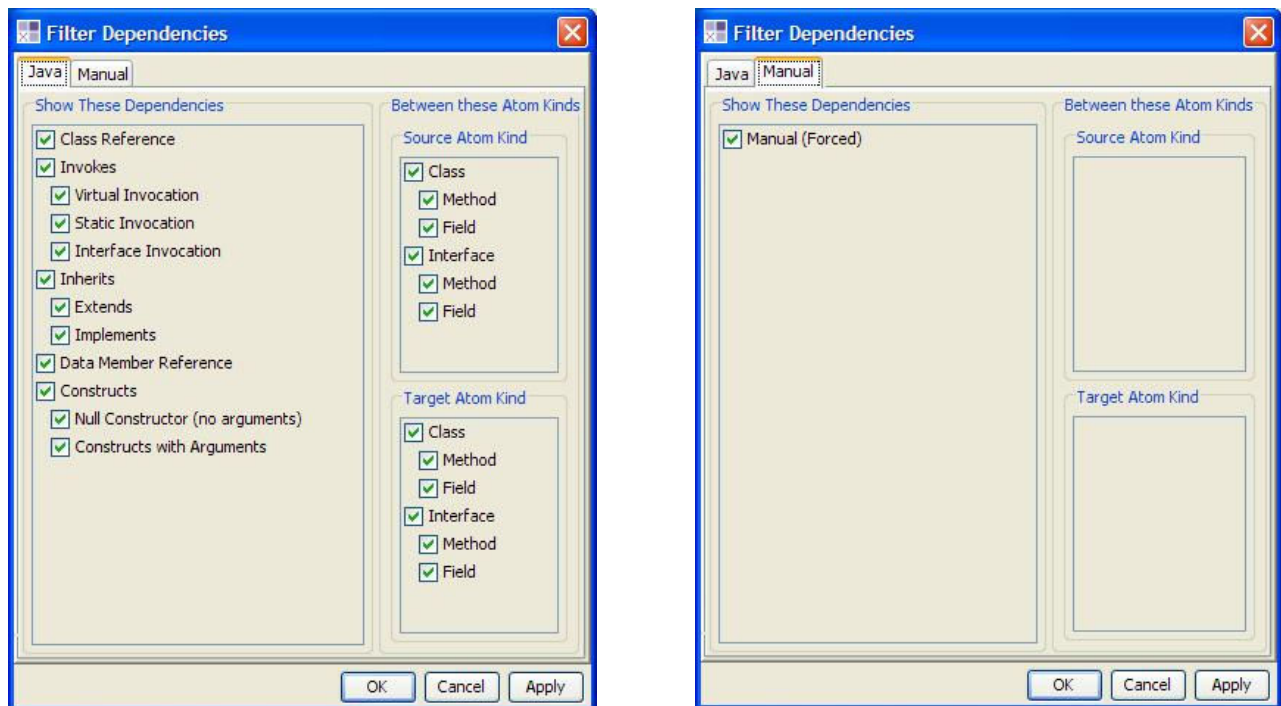


Figure 16: Filter Dependencies

Dependencies		Description
Class Reference		A reference to a class or object, but not to any of the objects members

Invokes	Virtual Invocation	Calling a non-static method (not on an interface)
	Static Invocation	Calling a static method
	Interface Invocation	Calling a method on an interface
Inherits	Extends	Is a subclass
	Implements	Implements an interface
Data Member Reference		A reference to a variable of a class / object
Constructs	Null (no arguments)	Constructs an object, but requires little or no design knowledge of that object
	With Arguments	Constructs an object, and specifying arguments implies some design knowledge of that object is required
Manual (Forced)		Manually assigned dependencies by the user

G. References

- [1] Lattix, Inc. *DSM for Managing Software Architecture*. Whitepaper. November, 2004
- [2] Lattix, Inc. *Design Rules to Manage Software Architecture*. Whitepaper. December, 2004-7
- [3] Lattix, Inc. *Using Lattix LDM to enforce your 3rd Party library adoption process*. Whitepaper. January, 2005