IMPROVING JAVA SOFTWARE THROUGH PACKAGE STRUCTURE ANALYSIS

Edwin Hautus
Compuware Europe
P.O. Box 12933
The Netherlands
edwin.hautus@nl.compuware.com

Abstract

Packages are an important mechanism to decompose Java programs. However, because packages are defined implicitly, it is not easy to develop a large application with a proper package structure. This article presents a tool that assists the programmer in developing a proper package structure through analysis and visualization. The tool indicates weak areas in package structures and allows human assisted refactoring of the source code based on the analysis. The article also introduces a new metric that is an indicator for the quality of the package architecture.

Keywords

Visualization, Software Re-engineering, Software Metrics, Refactoring, Component Based Development

1. Introduction

Decomposing software systems into modules has been a hot topic from the early days of computer science. The benefits of a good modular decomposition were already recognized in the early seventies: product flexibility, comprehensibility and reduced development time [1].

The Java language allows for decomposition by means of the package construct. Every Java class is part of a package, and packages are hierarchically structured in a package tree. Packages can therefore be considered to be the modules of a Java program [2].

This paper presents the Package Structure Analysis Tool (PASTA). PASTA analyzes the modular structure of Java programs. The focus is on two particular aspects: avoiding cycles in the dependency graph and layering. Some of the problems that are found during analysis can be fixed immediately through refactoring of the source code from within PASTA.

2. Related Work

Many researchers have investigated the analysis and visualization of programs for the purpose of comprehension. Recent efforts in the area of visualizing object-oriented programs include [3,4,5,6].

In [3], object oriented programming metrics are visualized in graphs using node color, positioning and size. In [4], software is visualized in 3D to assist refactoring. In [5], a Java exploration environment based on JavaBeans is presented. In [6], UML class diagrams are drawn based on class metrics.

All of these efforts focus on class or method level analysis. Perhaps because a lot of research was originally focused on C++ and Smalltalk, Java packages have been largely ignored. My work on the contrary focuses specifically on packages. I think that packages are particularly important because they are best suited to define the high level architecture of java programs.

Headway Review and *SmallWorlds* are two commercial products that allow dependencies in software to be analyzed. These tools provide a lot of information regarding dependencies, but they do not focus on layering, nor do they allow for refactoring of the code.

3. Package Structure Analysis

3.1 Acyclic Dependency Principle

An important principle in object-oriented design is the Acyclic Dependency Principle, ADP [7]. It states that:

The dependencies between packages must form no cycles.

Whether or not a program should always confirm to this rule is a matter of purity. However, fact is that package structures with many cycles are in general more difficult to understand and to maintain than those that confirm to the ADP.

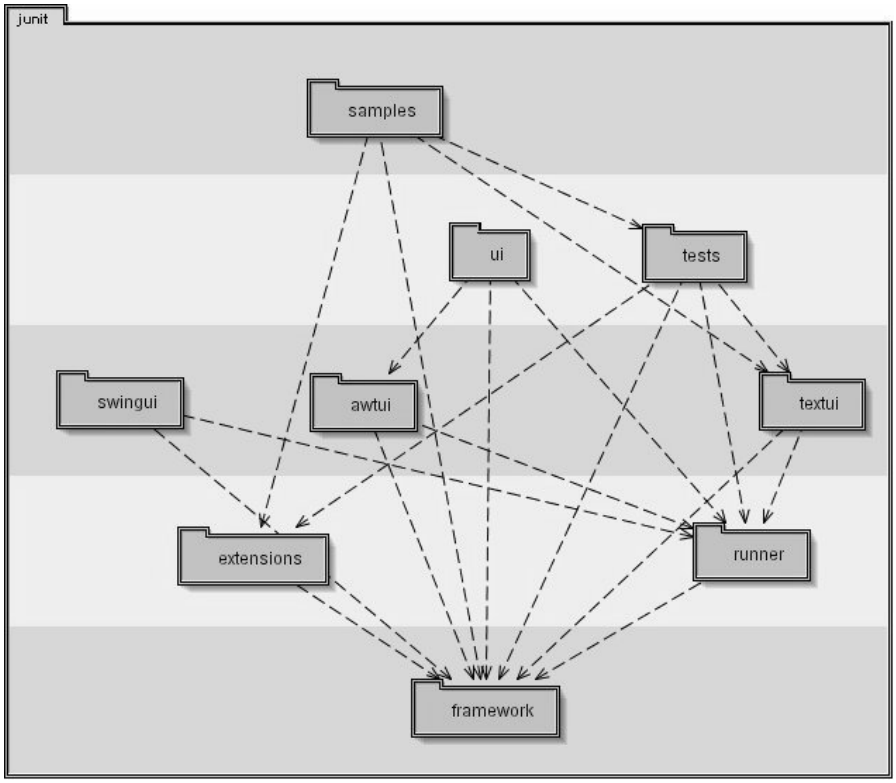

Figure 1 Layering for `junit` package

3.2 Layering

A second well-known principle of system architecture is the layered architecture [2].

Szyperski makes a distinction between strict layering and non-strict layering [8]. In strict layering, the implementation of one layer may only depend on the layer directly below. In non-strict layering, any lower layer may be used. In Java programming practice, strict layering is uncommon.

3.3 Re-engineering ADP Packages

If a package conforms to the APD, it is possible to reverse engineer a layering from the dependencies, using the following definition:

The layer of a package is the maximum length of a dependency path to a package with no dependencies.

This definition finds a layering with a maximum numbers of layers. See Figure 1 for an example of a layout generated by PASTA using this layering definition.

3.4 Re-engineering non-ADP Packages

Unlike the `junit` example shown in Figure 1, most Java programs do not confirm to the ADP. In such cases, it can be difficult to comprehend the intended package structure.

A simple way to deal with cyclic package structures is to simply ignore dependencies that are part of a cycle. This approach will be called simple layering.

A more advanced algorithm that has been implemented with PASTA is smart layering. Smart layering is a heuristic algorithm to determine a layering that best represents the intended architecture.

The smart layering of a package is defined as the simple layering corresponding to the package graph with *undesirable* dependencies removed.

A set of undesirable dependencies is chosen in such a way that the remaining dependencies form an acyclic graph. There are of course many possible sets of undesirable dependencies that lead to an acyclic graph. The advanced layering chooses a set, which has a minimal total weight of the undesirable dependencies.

The weight of a dependency is the number of references from one package to another. The weight is an indicator for the amount of work that is necessary to remove this dependency. Therefore, the smart layering algorithm finds an estimate for the amount of work that would be necessary to make a package structure acyclic.

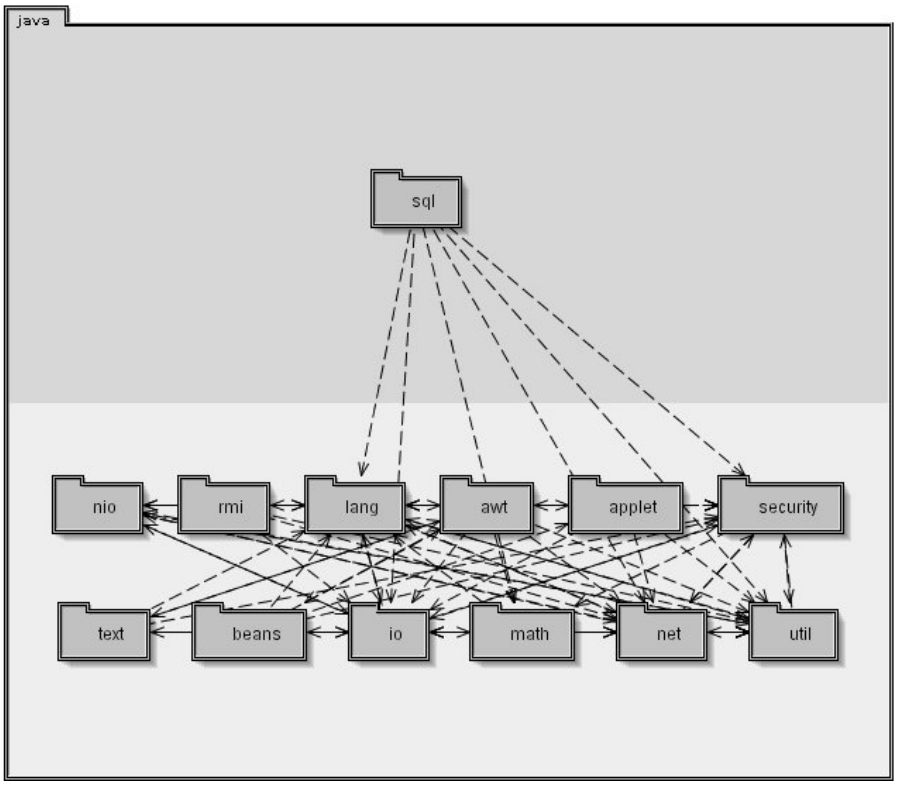

Figure 2 `java` Simple Layering

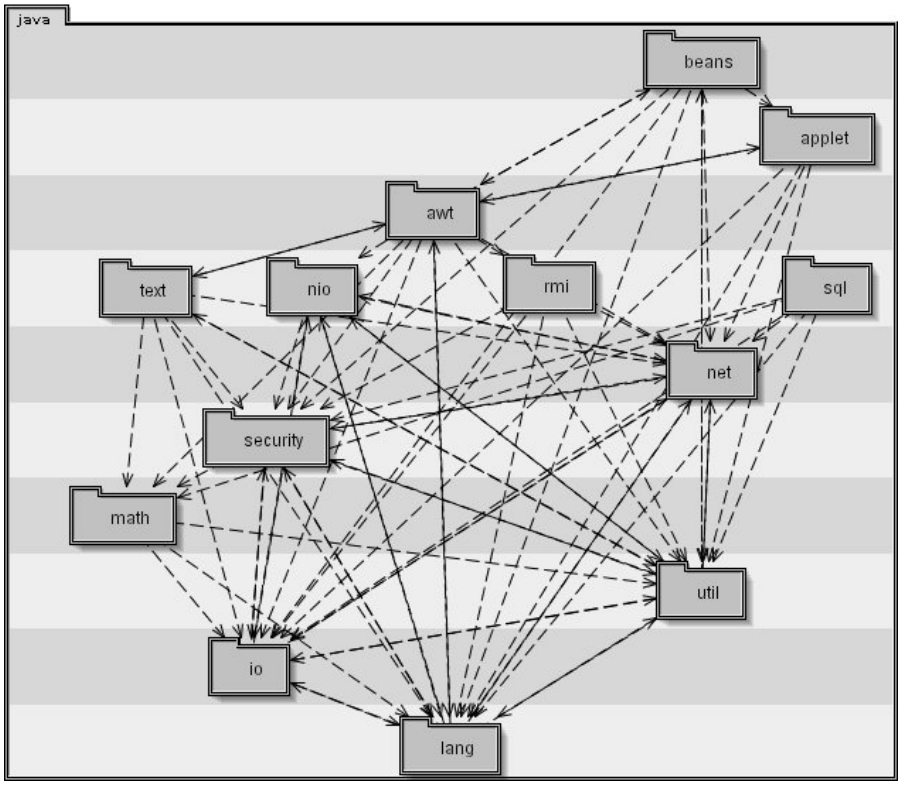

Figure 3 `java` Smart Layering

3.5 Collapsed Classes Packages

It is quite common for packages to contain both sub packages and types (classes or interfaces). If packages and types appear in the same package, PASTA combines the types into a *collapsed classes package*. Collapsed classes packages are treated like any other sub package: they are placed in a layer and are shown in the diagram.

3.6 The PASTA Metric

In this section, we introduce a new metric for evaluating the quality of package structures. Packages structures with a lot of cycles are not as good as those that conform to the ADP. However, in some cases, cycles can be removed more easily than in others.

We therefore use the results of the smart-layering algorithm and define the PASTA metric for a package as:
the weight of the undesirable dependencies between the sub packages divided by the total weight of the dependencies between the sub packages.

The PASTA metric is an indicator for the percentage of the software that would need to be changed in order to make a package structure acyclic. Lower percentages are therefore an indication for a better modular structure.

The PASTA metric for a package tree is defined as:
the weight of all desirable dependencies in all packages divided by the total weight of the dependencies in all packages.

An alternative definition for a package tree is to simply take the average of the PASTA metrics for all packages. However, the chosen definition has the advantage that high level packages have a higher weight, which corresponds to the idea that high level architecture is more important than low level architecture.

I have applied the PASTA metric to a number of freely available sources and the results are shown in Table 1.

Package	PASTA Metric
`junit`	0%
`org.apache.batik`	0%
`org.apache.tools.ant`	1%
`java`	5%
`org.apache.jmeter`	6%
`javax.swing`	10%
`org.jboss`	11%
`org.gjt.sp.jedit`	18%
`java.awt`	20%

Table 1: PASTA metric applied

3.7 Refactoring

PASTA visualizes package structures, but also allows changes to be made to the structure by drag and drop of types and sub packages from one package to another. Results are immediately shown in an updated UML diagram, allowing for 'what if' analysis. A future version will also allow dependencies to be reversed through the use of interfaces and abstract factories as described in [7].

When the user is satisfied with the resulting dependency structure, the changes can be applied to the source code.

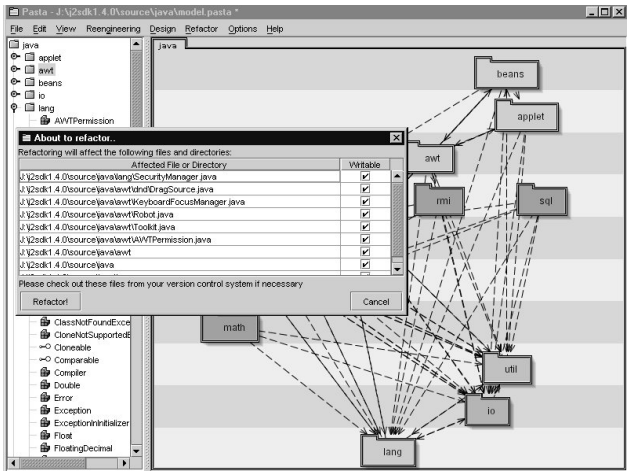

Figure 4: the PASTA application

3.8 Case Study

I illustrate the use of PASTA by examining the JDK1.4 `java` package, the core package of the Java language class library.

From Figure 2, it is clear that the `java` package structure does not have a strict layering. In fact, the only package that is not used by other packages is the `sql` package. It is not possible to determine any intended layering from Figure 2.

In Figure 3, smart layering has been applied, and it reveals the intended layering. The arrows that point up in Figure 3 are the undesirable dependencies that according to the PASTA should be fixed.

The `java.lang` package is at the bottom. Other low level packages are `java.io` and `java.util` The higher level packages include `java.beans`, `java.applet` and `java.awt`. This corresponds to the idea that higher layers represent higher levels of abstraction.

The analysis reveals some unexpected dependencies. For example, `java.lang` depends on `java.awt`. Further analysis with PASTA shows that this is the result of two classes: the `SecurityManager` in `lang` depends on the

`AWTPermission` in `awt`. Since `AWTPermission` is only used `in awt` and `lang`, moving `AWTPermission` to `lang` would solve this problem. `AWTPermission` should of course be renamed appropriately.

Making `lang` independent from `awt` is important when users of `java` want to replace `awt` with their own user interface library. Note however that this fix does not solve the problem completely, as `lang` still depends on `awt` indirectly via other packages. Further analysis would be necessary to achieve this, and simply moving classes from one package to another can not solve all problems.

4. Conclusions

I have presented a tool that can assist programmers and software architects with improving the modular structure of their Java software.

Software architects usually have a layered architecture in mind when designing applications. However, without proper tool support, package structures tend to become polluted with cyclic dependencies. I have presented an algorithm that can recover intended layering from polluted structures. The recovered layering also suggests which dependencies should be fixed. Fixing can be done partly using the tool itself with its refactoring capabilities.

The presented algorithm can also be used to calculate a new metric for evaluating the high level modular structure of large Java programs. The metric provides a way to quickly evaluate the internal quality of large software products based on their source code.

Future work includes more advanced refactoring proposals, support for other design principles, and integration with other tools. The tool is downloadable from the Compuware website.

References

[1] D.L. Parnas, Carnegie-Mellon University, On the criteria to be used in decomposing systems into modules, *Communications of the ACM, Vol. 15, No. 12,* 1972, pp. 1053 - 1058

[2] M.Fowler, Reducing Coupling*, IEEE Software July August, 2001*, pp. 102-105

[3] S. Demeyer, S. Ducasse and M. Lanza,, A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization, *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering)*, IEEE, October, 1999

[4] F. Steinbrückner, C. Lewerentz, Metrics Based Refactoring, *Proc.of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001),* IEEE Computer Society Press, March 2001, pages 30 - 38

[5] M.-A. D. Storey, C. Best , J. Michaud, SHriMP Views: An Interactive and Customizable Environment for Software Exploration, *Proc. of International Workshop on Program Comprehension (IWPC '2001)*, May 2001.

[6] R. Kollman, M. Gogolla, Metric-Based Selective Representation of UML Diagrams, *Proc. 6th European Conf. Software Maintenance and Reengineering (CSMR 2002)*. IEEE, Los Alamitos, 2002.

[7] R.C. Martin, *Design Principles and Design Patterns*, http://www.objectmentor.com, 2000.

[8] C. Szyperski, *Component Software* (New York: Addison-Wesley, New York, 1998)