Locating Reusable Classes Using Dependency in Object-Oriented Software

Young Lee and Jeong Yang

Department of Electrical Engineering and Computer Science, Texas A&M University-Kingsville, Kingsville, TX

***Abstract*—With automated measurement tool, a user can locate reusable classes, connected classes and independent classes. This paper describes how an automated tool can guide a programmer through measuring dependency of a program for software reuse. Automated identification of reusable software components based on dependency is explored. The case study demonstrates identifying the reusable units for software reuse and connected units for software package.**

***Index Terms*— Fan-in coupling, Fan-out coupling, Automated measurement tool, Software reuse, Dependency**

1. Introduction

There are valuable open source programs written by other programmers, but it is desirable to have a tool that retrieves reusable software components.

This paper describes the automated modularization technique implemented and integrated in *JamTool* (Java Measurement Tool) [1, 2]. With this automated measurement tool, a user can learn about the dependency information specifically for locating reusable codes.

1.1 Software Reuse

The primary motivation to reuse software components is efficiency. It is achieved by reducing the time, effort and/or cost required to build software systems. The quality and reliability of software systems are enhanced by reusing software components, which also means reducing the time, effort and cost required to maintain software systems.

When a reusable code is written, the intended users should be somewhat identified. If a code is to include the functionality that every user would want, the resulting code would be too expensive to produce and too difficult to use. Code reuse has been common in practice. But, many difficulties are associated with code reuse:

1. Code identification: It is difficult to identify a piece of reusable code. Many times, programmers reuse only a small fraction of their own or their colleagues' code.

2. Code validation and verification: There is usually little assurance that the reused code is correct.

3. Code dependency: It is a nontrivial task to separate a desired piece of code from an entangled chunk of software with complex dependency.

4. Code modification: In addition to the necessary changes, the reused code may implicitly conflict with the new context.

5. Execution environment: The reused code might assume things that are not true in the new environment. This may result in degraded performance.

With careful planning and implementation, many of these difficulties can be avoided. A static analysis of a source code in any stage of development can provide instant feedback to the programmer, the quality of the code in the sense of reusability. This would encourage programmers to ensure that the completed code provides good reusability before it is discovered too late. The measurement can also allow the manager of a software project to evaluate the quality and reward the programmers accordingly.

1.2 Fan-in/Fan-out Coupling

Fan-out coupling measures the degree to which a class has knowledge of, uses, or depends on other classes [3]. To reuse a class with high fan-out coupling in a new context, all the required services must also be understood and reused together. Therefore, high fan-out coupling can decrease the reusability of a class.

Fan-in coupling measures the degree to which a class is used by, depended upon, by other elements [3]. Therefore high fan-in coupling can represent the how many other classes have reused the class.

Coupling connections cause dependencies among classes, which, in turn, have an impact on reusability (to reuse a class may require reuse connected classes together). Thus, coupling metrics also greatly help identify problematic classes to be reused.

DOI: 10.5176_2010-2283_2.1.140

2. Automated Measurement Tool

Java Measurement Tool (JamTool) is a software measurement environment to analyze program source code for software reuse [2]. It is especially designed for object-oriented software. This tool measures attributes from Java source code, collects the measured data, computes various object-oriented software metrics, and presents the measurement results in a tabular form. The tabular interface of the tool provides software developers the capabilities of inspecting software systems, and makes it easy for the developers to collect the metric data and to use them for improving software reusability. By browsing dependency of class a developer can learn how to reuse certain software entity and how to locate problematic parts. The application of this easy-to-use tool significantly improves a developer's ability to identify and analyze reusability of an object-oriented software system.

The acceptance of Java as the programming language of choice for industrial and academic software development is clearly evident. The overall system architecture of the *JamTool* is shown in Figure 1, in which solid arrows indicate information flow. The key components of the architecture are: 1) User Interface, 2) Java Code analyzer, 3) Internal Measurement Tree, 4) Measurement Data Generator, and 5) Measurement Table Generator.

Each key component works as a subsystem of overall system. The Java Code Analyzer syntactically analyzes source code and builds an Internal Measurement Tree (IMT) that is a low level representation of classes, attributes, methods, and relationships of the source code. Then the Measurement Data Generator takes the IMT as an input, collects the measurement data, and generates the size, complexity, coupling, and cohesion metrics of classes in the original source code. Those measurement results as well as the other metrics are displayed in a tabular representation through the Measurement Table Generator subsystem. With this interface of tabular form, software developers can easily analyze the characteristics of their own program.

3. Locating Reusable Classes

Connected unit is defined as the classes that are coupled together. In a connected unit table, all classes directly and indirectly coupled together are displayed in the same column, thus a set of classes in the same column is a connected unit. A connected unit is likely to be of interest to the user in finding software units

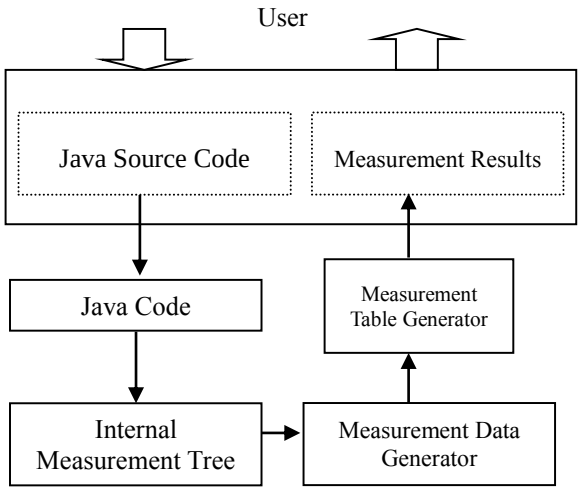
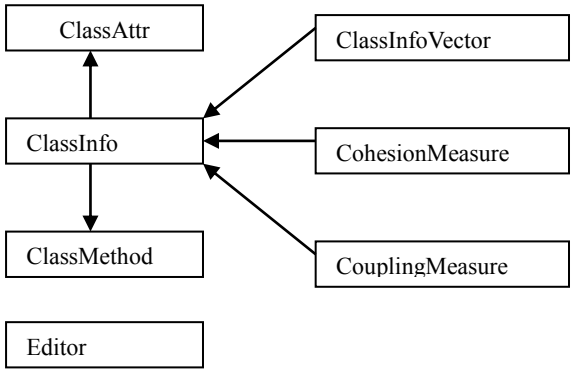
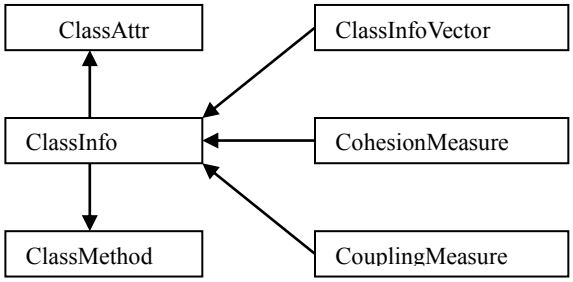


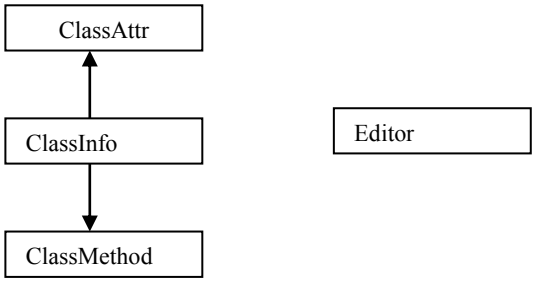
Fig. 1. Architecture of JamTool



(a) Dependency among Classes



(b) Connected Classes



(c) Reusable Classes and Independent Class

Fig. 2. Classes for Software Reuse

that can be reused. Connected units are coupled classes in the coupling metrics and the connected attributes and methods in the cohesion metrics. A user should consider reusing the connected classes together in a new application. In that sense, the connected classes are a reusable unit.

Reusable unit is a collection of a target class and its related classes should be reused together. Identifying reusable components means that each class has its own reusable unit with other classes that the class depends on. The identification of a reusable unit of classes requires an understanding of the relation of classes in a software system. Reusable unit is necessary to understand software structure and more importantly, to serve as a source of information for reuse.

Some important tables are reusable unit and connected unit tables. In a reusable unit table, each class in the first column depends on classes in other columns since the class uses the others. Thus the classes in the same row make a special reusable unit. Connected unit table identifies coupled classes in coupling metrics and connected attributes and methods in cohesion metrics. In this way of representation, programmer could easily recognize which classes need more/less effort when they are needed for reuse, modify, update or fix. This could definitely help programmer in developing and maintaining a program.

Connected unit is defined as the classes that are coupled together. In a connected unit table, all classes coupled together are displayed in the same column. A connected unit is likely to be of interest to the user in finding software units that can be reused. Connected units are built by identifying coupled classes in the coupling metrics and the connected attributes and methods in the cohesion metrics. A user should consider reusing the connected classes together in a new application. In that sense, the connected classes are a reusable unit. Figure 2 shows the retrieved connected unit and the result of applying Connected Unit Search Algorithm to the class-to-class coupling. The connected unit search algorithm, shown in Figure 3, computes a set of coupled classes (i.e., connected unit) and their position in a connected unit table based on a class-to-class coupling table.

```
Input: Class-to-class coupling measurement;
Output: Connected units;

Let classNames = all class names from a
class-to-class table;
foreach class in classNames do
   Let targetClass = a class in classNames
      that has not been searched yet;
   if targetClass is empty then
      return connectUnitsWithPosition;
   Search class-to-class table and let
   connectUnit = coupled classes to
   targetClass;
   Update connectedUnitsWithPosition with the
   connectUnit;
end for
```

Fig. 3. *Connected Unit Search* Algorithm

4. Case Study

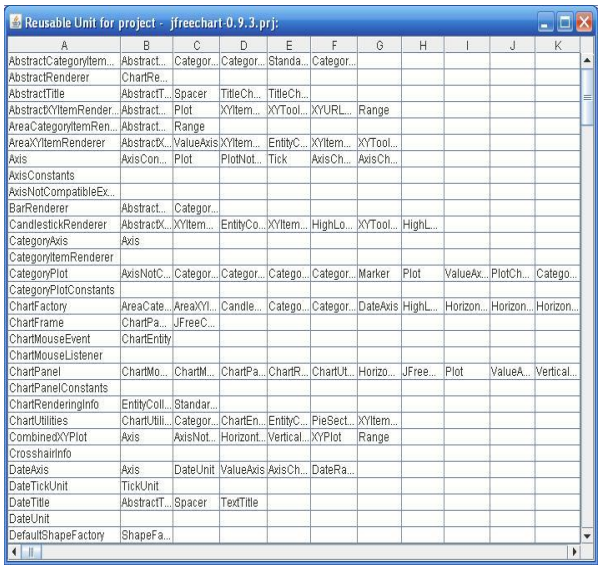
A case study investigates if the reusable components can be used to capture the difference between two consecutive versions on the evolution of *JFreeChart* [4]. According to the empirical study reported in the previous paper [1], there was a big change of coupling metric values from 0.9.3 to 0.9.4. These two versions are chosen to compare with metrics and the measurement result tables obtained by *JamTool*.

A. Reusable Unit Table

Reusable unit table is to present how much a class depends on other classes. In Figure 2 (a), the first column, A, displays all classes in the selected project. A class in column A uses the classes in columns to its right. The classes in the same row make a special reusable unit.

For instant, in the second and third rows, class *AbstractRenderer* depends on a class *ChartRe…*, and class *AbstractTitle* depends on four classes *AbstractT…, Spacer, TitleCh…, and TitleCh….* This dependency means that, for example, if programmer wants to use a certain class (*AbstractTitle*), then he/she must use the other classes in the reusable unit (*AbstractT…, Spacer, TitleCh…, and TitleCh…*) since they are used by the certain class (*AbstractTitle*).

Therefore, if a class depends on too many other classes, it is obvious that such a class is difficult to be reused.



(a) Reusable unit in version 0.9.3

(b) Reusable unit in version 0.9.4

Fig. 4. Reusable unit

Figure 4 shows the reusable units in versions 0.9.3 and 0.9.4. From these unit tables, progression of the reusable units is captured. Some classes like *CategoryPlot*, *Chartfactory*, and *ChartPanel* have too many classes in their reusable units in both versions and some have changed. For example, class *AbstractCategoryItemRender* depends on five classes in version 0.9.3, but seven classes in version 0.9.4.

B. Connected Unit

Each class is displayed in a connected unit table according to its position and its coupling strength is displayed in the connected unit strength table in Figure 5. For instance in Figure 5 (a), only three classes, *StandardToolTipsCollection*, *ToolTip*, and *ToolTipsCollection*, in column D are coupled to each other. Class *DatasetChangeListener* in column E could be a dead code because there is no relation to other classes in the project. By observing connected units, user may also discover connection patterns. For example, if a project is composed of an application program and libraries, an investigation of the connected unit will tell how the application program uses a library function. In that sense, this type of connection pattern is a use pattern.

Figure 5 shows part of connected units of *JFreeChart* in two versions. From these connected units, version 0.9.3 establishes a main connected unit which has 224 classes out of a total of 257 classes as shown in column A in Figure 5 (a), and a minor connected unit with 3 classes in column D of Figure 5 (a). Three classes (*StandardToolTipsCollection*, *ToolTip*, and *ToolTipsCollection*) belong to the same package named "com.jrefinery.chart.tooltips". There are also 11 independent classes, e.g., *DatasetChangeListener* in column E, which have no relation to other classes.

We also found that version 0.9.4 has a main connected unit with 254 classes out of a total of 275 as shown column A in Figure 5 (b), and a minor connected unit with 3 classes in column K of Figure 5 (b).

These three classes (*Function2D*, *LineFunction2D*, *PowerFunction2D*) belong to the same package named "com.jrefinery.data ". There are 18 independent classes, which have no relation to other classes in Figure 5 (b). The independent classes for both versions are listed in Table 1.

C. Summary

The goal of this case study is to compare and analyze two versions of *JFreeChart* at class level. Specifically, it aims to answer the following questions:

- How can the differences between them be compared and detected in terms of reusable units?

Table I. Independent classes in two versions

0.9.3 (11 classes)	0.9.4 (18 classes)
JFreeChartInfo, PlotException, DatasetChangeListener, Values, XisSymbolic,YisSymbolic, DataPackageResources, DataPackageResources_de, DataPackageResources-es, DataPackageResources_fr, DataPackageResources_pl	DataUnit, JFreeChartInfo, PlotException, ChartChangeListener, LegendChangeListener, lotChangeListener, TitleChangeListener, JFreeChartResource, DatasetChangeListener, Regression, Values, XisSymbolic, YisSymbolic, DataPackageResources, DataPackageResources_de, DataPackageResources-es, DataPackageResources_fr, DataPackageResources_pl

- How can the huge information of source code be filtered and compared in the context of software reuse?

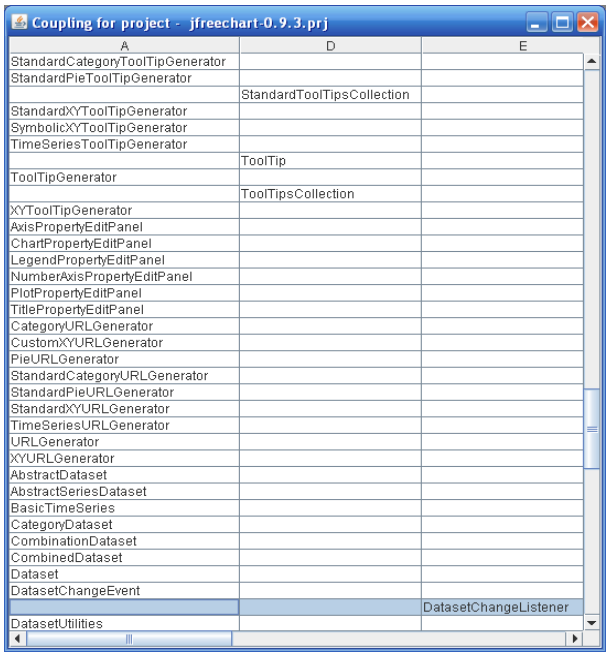
In this case study, we analyzed the differences between the metrics of two versions using *JamTool* and found overall trend of metrics of *JFreeChart* in versions 0.9.3 and 0.9.4. Followings are findings from the comparison and analysis of two versions of *JFreeChart*:

- By comparing reusable units in version 0.9.3 and version 0.9.4, we found newly added or removed classes to the reusable unit.
- By analyzing connected unit, we found that most classes are directly or indirectly related to each other and they form one main connected unit.
- 11 and 18 independent classes that have no relations to other classes in versions 0.9.3 and 0.9.4, respectively.

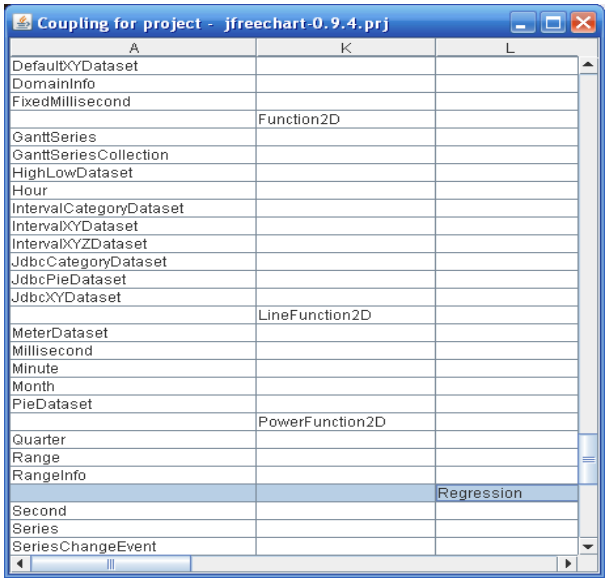
5. Conclusion

This paper presented an automated measurement tool, *JamTool*, for software reuse. The primary benefit of this tool is its ability to automatically capture the dependency among the classes and give informative feedback on software reuse by reusable units.

We believe that various tabular techniques provide structural information for software reuse (e.g., reusable unit and maintainable unit). By inspecting these tables, software developers are able to detect reusable software components from libraries and existing open sources by using library documents or inspecting the source code. To reuse software components from exiting application source code, a user should learn the source code before using it. Measuring relationship of



(a) Connected unit in 0.9.3



(b) Connected unit in 0.9.4

Fig. 5. Connected unit

the software components is useful to overview the software and to locate the reusable software components. From the measurement results, the user may decide whether or not he/she should reuse the software.

By browsing reusable units, a developer can learn how to reuse certain software entity and how to locate problematic parts. The application of this easy-to-use tool significantly improves a developer's ability to identify and analyze quality characteristics of an object-oriented software system.

Measurement Result Tables produced by *JamTool* can be used in the following tasks:

- To locate reusable units that should be reused together.
- To locate connected units that should be packaged together.

In the future, authors consider an empirical test to extract reusable units from the professional library programs (e.g., Java Foundation Class) and to determine whether these features can be used to classify the reusable software.

REFERENCES

[1] Lee, Young, Yang, Jeong, and Chang, Kai H. "Quality Measurement in Open Source Software Evolution", The Seventh International Conference on Quality Software, 2007, Portland, Oregon

[2] Lee, Young, "Automated Source Code Measurement Environment For Software Quality", Doctorial Dissertation, Auburn University, 2007

[3] Briand, L., Daly, J., and Wust, J., "A Unified Framework for Coupling Measurement in object-oriented Systems," IEEE Trans. on software eng., vol. 25, no. 1, 1999

[4] http://www.jfree.org/jfreechart/

Jeong Yang is a lecturer in the department of electrical engineering and computer science at Texas A&M University-Kingsville. She received her M.S. degree in computer science and software engineering from Auburn University.

Young Lee is an assistant professor in the department of electrical engineering and computer science at Texas A&M University-Kingsville. He received his Ph.D. in computer science and software engineering from Auburn University.