Automatically Detecting Detects on Class Implementation in Object Oriented Program on the Basis of the Law of Demeter: Focusing on the Dependency between Packages

RYOTA CHIBA, HIROAKI HASHIURA, SEIICHI KOMIYA

Abstract— In an object-oriented software development project, one of the design methods such as OOSE and OMT is typically employed. However, a common weak point is recognized for these design methods, that is, they do not provide an easy approach for designing a class. In this study, we propose a method to support designing a class by focusing on modularization to reduce the degree of coupling. We extended the Law of Demeter, which is one of the laws to reduce the degree of coupling, so that it may be applied to Java. In addition, in Java, since modules developed by somebody are usually reused as packages rather than as classes, we decided to apply the Law to a set of packages. We developed a tool that can automatically detect a violation of the Law as a plug-in of Eclipse. We have conducted an experiment to prove that the tool can automatically detect a violation of the Law of Demeter and point out the location of violation Then, we enter the violated portion of source code to our tool to prove that the problem can be corrected.

Keywords— Class Design, Object Oriented Software Development, Law of Demeter, Eclipse

I. Introduction

IN today's highly information-oriented society, software development efforts are coming to be increasingly larger and complicated. As a result, wider and deeper knowledge is required for software development.

In particular, although the object-orientated approach is commonly used in recent years and it includes typical design methods such as the OOSE (Object-Oriented Software Engineering) and OMT (Object Modeling Technique) methods, its design methods have common difficulties in designing classes. In addition, even if a designer has acquired the ability to use one of the modeling languages such as object-oriented programming languages or UML, he/she cannot take advantage of its merits such as robustness and expandability when he/she uses it without the knowledge of background principles.

Therefore, the industry expects universities to grow well-trained students who have the knowledge and skill of software development. Based on the above understanding, we have adopted PBL (Project-Based Learning) as the exercise lessons of software development to promote training of practical object-orientated software design and development technology in our university.

To develop human resources with deep knowledge and excellent skills of software development, it is necessary for students to learn how to design proper software (software design methods). For this purpose, it is necessary to develop tools that can provide support for understanding how to design software.

Based on the above discussion, we decided to use coupling [1] as the metrics to measure properness of developed software designs in this study. Coupling represents the strength of interactions among modules, and weaker coupling among modules indicates that they are better-designed. To reduce coupling among modules, we focused on the interdependent relationship among modules, and placed additional restrictions on the destinations of messages.

We adopted the Law of Demeter as the method to reduce coupling. We used this method to develop a tool that can closely scan the occurrences of interdependent relationship and point out the locations at which the destination of a message should be modified. This tool can be used to detect and clearly specify the points where modification is needed to provide information useful for future program modification (including modification of the specification).

This paper contains the following sections. Chapter 2 shows the position of this study. Chapter 3 discusses the Law of Demeter and the enhancement developed in this study. Chapter 4 describes an outline of our tool and its implementation method. Chapter 5 shows that our tool can be used to detect points violating the Law of Demeter. Chapter 6 shows the effectiveness of our tool based on the comparison with the related tools applied to open source code. Chapter 7 describes the conclusion.

II. Purpose of this Study

According to Myers[1], proper modularization is one of the essential factors to develop well-defined software. One of the techniques to perform modularization is to split a system into modules in view of the independency of each module.

Ryota Chiba, Shibaura Institute of Technology Graduate School of Engineering, 3-7-5 Toyosu, Koutou-ku, Tokyo, Japan,
chiba@komiya.ise.shibaura-it.ac.jp
Hiroaki Hashiura, Shibaura Institute of Technology Graduate School of Engineering, 3-7-5 Toyosu, Koutou-ku, Tokyo, Japan
hashiura@komiya.ise.shibaura-it.ac.jp
Seiichi Komiya, Shibaura Institute of Technology Graduate School of Engineering, 3-7-5 Toyosu, Koutou-ku, Tokyo, Japan
skomiya@shibaura-it.ac.jp.

Myers[1] proposed the concepts of module cohesion (module strength) and module coupling as the metrics to measure independency of program modules. It states that higher cohesion and lower coupling are essential for modules to achieve a high degree of independency. It also states that the module maintainability is increased when the modules satisfy these conditions.

The maintainability of software is defined in ISO/IEC 9126-1[2], stating that maintainability is essential to enhance the quality of software.

To quantitatively evaluate the independency of these modules, Chidamber and Kemerer[3] introduced the CK metrics. It is important for an actual implementation to keep the value of metrics within an appropriate predefined range based on the information provided with the metrics. However, neither practical procedure nor effective means or method has been provided to keep the value within an appropriate range. As a result, the developer's experiences and intuition has played a major role to achieve an appropriate value. As a technique to cope with the issue, the Refactoring [4] can be used to improve the quality of software while keeping its external behavior as it is. The effectiveness of refactoring has been proved in traditional software development projects. Various studies have been performed to achieve appropriate values for the metrics using this technique. [5]-[7] In these studies, source code is modified based on the results of source code analysis.

However, modifying source code based only on the values of metrics causes another problem that it is difficult to understand how the corresponding design is modified. As a result, the original design has been modified in such a way that it is difficult to track the modification made through refactoring.

Therefore, in order to solve this problem, the mechanism is needed, which can point out the interdependencies among portions of source code by analyzing the source code. With consideration of approaches based on use of tools, our tool adopted an approach in which implementation flaws are pointed out but not corrected automatically, although there may be other approaches in which implementation flaws are corrected automatically. Although minimum interdependencies are required for programs, we think that the necessity of interdependencies should be judged according to the intention of the designer or programmer. In addition, since our aim is to provide support for learning a software design method, automatically correcting a design flaw does not serve this purpose.

III. THE LAW OF DEMETER

A. What is the Law of Demeter?

The Law of Demeter was proposed by Lieberherr [8][9] of Northeastern University in 1987 as a rule for producing the design proposal which may be set for object-oriented software. The Law of Demeter, as shown in Fig. 1 provides a rule with which "a message can be sent from a standard object exclusively to the objects directly connected to the standard object, and cannot be sent to other objects." According to Lieberherr, et al.[8], an object which can directly communicate with the standard object is called "friend." They defined friend objects for C++ programs.

This was also shown by Larman [7] as "Low Coupling Pattern." That is, in other words, the Law of Demeter provides a mechanism to clarify the portions for which no dependency relationship (coupling) is required.

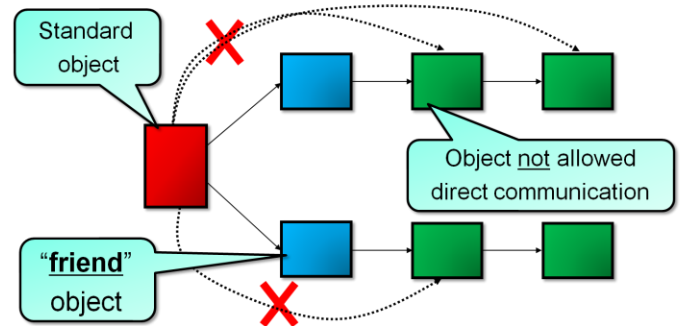

Fig. 1. The Law of Demeter

B. The Law of Demeter Extended for Java

The Law of Demeter provides the way to develop programs for implementing objects with higher independency and lower coupling defined by Myers. The concept of "friend" was defined in 1988 for the object-oriented language C++ [8] (at that time JAVA did not exist). Therefore, we define "friend" as shown in Fig. 2 in this study so that it may be used in JAVA.

> With a program written in Java, every method of an object should be allowed to send a message to a target which is restricted by the following criteria.
>
> (i) The object itself.
> (ii) An object passed as an argument to the object itself.
> (iii) An object owned by the object itself as an attribute.
> (iv) An object generated with a method of the object
> (v) The embedded class of JAVA and its base type

Fig. 2. The Law of Demeter for Java

Messages are allowed to be passed only between those objects that satisfy one of the above conditions (the interdependent relationship between classes is not restricted).

In particular, Definition (v) is added to the Law of Demeter so that the embedded classes and data types used in Java may be treated as part of OS and may not to be regarded as part of the dependency relationship. If the embedded classes and data types of Java are not treated as "friend," a violation of the Law of Demeter might be detected even when writing to the standard output or calling a method to perform string conversion.

C. Extending the Scope of the Law of Demeter

The Law of Demeter aims to reduce coupling between classes

by focusing on the interdependent relationship between classes to restrict the address of a message. However, applying the Law of Demeter with the smallest granularity of class, the number of objects (number of couplings) with the friend relationship for each object dramatically grows as the scale of software to be developed becomes large. As a result, even when the relationship between two objects is "friend," it is difficult to judge whether message passing is really required between these objects or not. In this study, we propose to use not only a class but also a package that consists of more than one class as the unit. The reason is that, in Java, reuse of a module is often performed in the unit of package rather than the unit of class. When reusing a package, it is usually used without checking all of the classes in it. In addition, checking internal components of a package indicates violation of the principle of information hiding. Therefore, checking internal components of a package is out of the scope of our tool. Fig. 3 illustrates the above discussion. As shown in this figure, communication between two classes that are located beyond more than one class within a package is allowed.

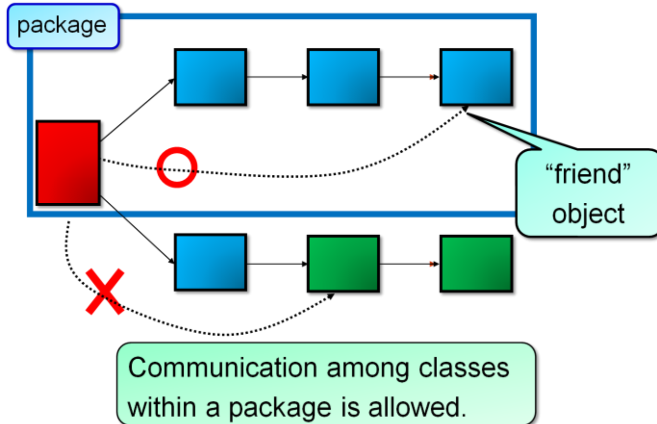

Fig. 3. The Law of Demeter applied to more than one package

IV. Class Design Support with Tool

In this study, we developed a tool that can be used to analyze source code to detect the location where a violation of the Law of Demeter occurs. The following sections describe an outline of the tool and its necessity, and also describe how to analyze source code.

A. The necessity of our tool

The degree of coupling between packages is measured by tracking the destination addresses of messages in source code based on the definitions of five friends shown in Fig. 2. For example, Object A can call any routine of Object B if Object A has instantiated Object B. When a routine of an object provided by Object B is called from Object B, just investigating the source code of Object A does not clarify to which class the message is sent. In other words, it is necessary to track the contents of more than one class to examine one class. It is possible to track destinations manually if the volume of source program is small, but it becomes difficult to examine the forwarding addresses of messages when the program volume is large. In addition, it is necessary to use fully-qualified name to distinguish two or more classes with the same name. Therefore, it is necessary to use a tool to automatically extract the destination addresses of all message defined in the package, and based on the extraction results, the destination addresses are checked if they are the same or not per a package and type to check the dependency relationship between packages automatically.

In addition, since the knowledge of module independency is required to find and identify the dependency relationship, it is difficult for a novice programmer whom we are going to provide support for making a proper decision. Therefore, it is impossible for an instructor to manually detect and identify all occurrences of dependency, since there are as many programs as the number of student groups.

B. Outline of our tool

Eclipse [12] is a free Integrated Development Environment (IDE). IDE provides standardized user interfaces that can be used to consistently handle programming tools such as an editor, compiler, and debugger. Eclipse has been used in various development projects with Java in many companies as its proven history.

Based on such a background, we decided to implement our tool as a plug-in of Eclipse. As a plug-in of Eclipse, our tool can provide suitable advices to novice programmers at any time when they write programs.

Fig. 4 illustrates an outline of development task using this tool. The tool analyzes source code within the specified package on the specified directory, finds classes with any violation of the Law of Demeter, and identifies the source code line corresponding the detected violation, according to the friend definition in Section 3.B.

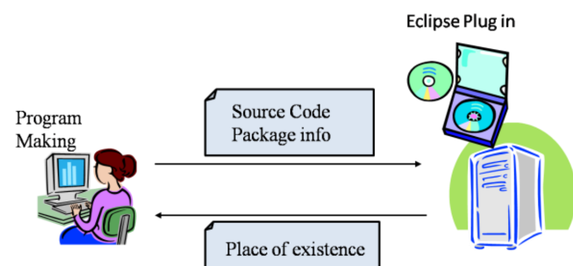

Fig. 4. Development tasks using our tool

C. How to analyze source code

To find defects in the implementation based on the Law of Demeter, it is necessary to extract the package name and the class names from the source code of the program. For this purpose, it is necessary to develop source code models. Source code modeling is often performed using Java element API and DOM/AST API. With Java element API, the signature information of the method constructer defined in the source code can be obtained, but the information of source code written in the method cannot be obtained. On the other hand, DOM/AST API included in the package

org.eclipse.jdt.core.dom can be used to obtain the information inside the method because it provides more detailed models of Java source code compared with the Java element. In addition, since it can handle Abstract syntax Tree (AST) and incomplete or incorrect source code, it is possible to analyze source code under development. Therefore, we decided to use DOM/AST API in this study to enable analysis of any kind of source code.

DOM/AST API performs configuration analysis to generate an abstract syntax tree when source code is entered to the tool. Then, the Visitor pattern that is one of the design patterns proposed by Gamma et al. [13] is used to traverse this abstract syntax tree to extract the package name and class names. The Visitor pattern is a general-purpose mechanism that can be used to traverse an object-oriented hierarchy structure and add new functions without making changes in the component nodes that compose the hierarchy structure. For this reason, we can add a procedure that extracts the package name and class names without making changes in an abstract syntax tree.

Summarizing the above, the tool can perform analysis following the five steps written below.

(i) Input source code.
(ii) Generate an abstract syntax tree.
(iii) Traverse an abstract syntax tree by using Visitor.
(iv) Identify the location in a package from which a message is sent to another package.
(v) Check whether the target object is a friend object or not, and generate a warning message if the object is not a friend object.

Fig. 5. Analysis steps performed by the tool

D. Functionality of the tool

This tool provides the function to examine the classes in the specified package to detect a class that violates the Law of Demeter. If the tool finds that a message is sent directly to a class other than the one defined as a friend in Section 3.B, it lists the name of the class in the package, the number of related lines of the source code in that class, and the destination class name of the message. This detection result can be used to find specific destinations of the messages that violate the Law of Demeter and advise the developer to modify the program to reduce coupling between modules. In addition, the number of message destinations between packages can be regarded as the degree of coupling between packages based on our definition.

V. EXPERIMENT OF THE DETECTION ABILITY OF THE TOOL

A. Purpose of our experiment

In this experiment, we develop source code of a sample program. The source code contains a portion that violates the Law of Demeter. We use the tool to analyze the source code to find the violated portion.

The detection result is examined to identify the information of location in the source code where a violation is detected and the destinations of messages sent from the source code. Then, we modify the detected portion of source code, use the tool to analyze the modified source code again, and show that no violation is found.

B. Experimental procedure

1) Development of sample source code

For the sample source code, we assume a case in which two points are located on a plane of two-dimensional coordinate system.

The following describes an outline of the source code.

The Line and Point classes are defined in the model package.

The Point class uses the x and y coordinates to represent a specific point with the two-dimensional coordinate system. It has a method to set a point and another method to return the values of the x and y coordinates of the point.

The Line class is used to represent a line on the two-dimensional coordinate system. Two points are required to represent a line with the two-dimensional coordinate. Therefore, two points are located on the field. It also provides a method to set two Points used to create a line and another method to return a Point class for each Point.

The lod package contains the AntiLoD class that is necessary to use the model package. The AntiLoD class is used from the Line class to directly get the addresses in the Point class and actually draw a line. For example, to call the first x coordinate from the AntiLoD class, write the code “line.getP1.getX().”

Fig. 6 illustrates the above description with class diagrams.

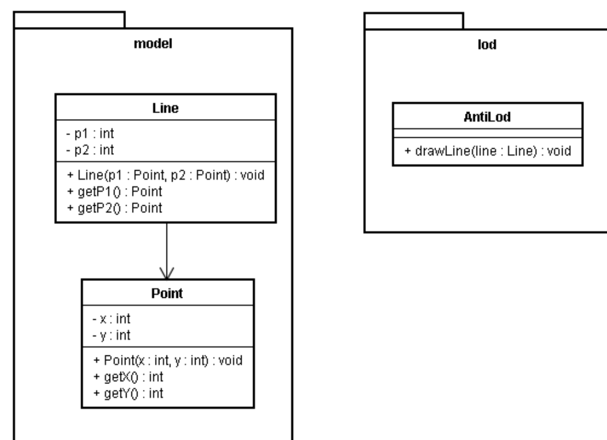

Fig. 6. The class diagrams of generated source code

2) Using the tool to detect violation

Using the source code developed in Section 5.B.1) as input to the tool has generated the results shown in Tables 1 and 2.

In this case, the cause of violation is a call issued from the AntiLod class to call the getP1() and getP2() methods in the Line class and return the Point class. ((1) in Fig. 7) Calling the getX() and getY() methods in the Point class ((2) in Fig.7) violates the Law of Demeter because it performs an inter-package communication to get the coordinate values.

As for the AntiLoD class, a value of 1 is added to the number of counts since the message is sent exclusively to model.Point.

In this study, when more than one message is sent to the same class, they are counted as one as a whole.

The above discussion has proved that the tool works as defined in this study to detect violations.

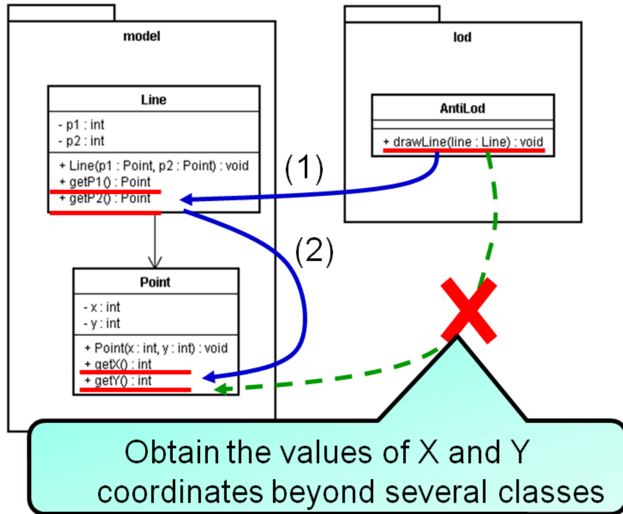

Fig. 7. The cause of violation

Table 1. Detection of the AntiLod class

lod.AntiLoD

Line number	Destination of message
11	model.Point
12	model.Point
13	model.Point
14	model.Point

Table 2. The number of violations in AntiLod

Class name	Number of counts
lod.AntiLoD	1

3) Correction of violation

Lieberherr, et al. proposed to create a wrapper class as an approach to dissolve such a dependency relationship. In this approach, the dependency relationship is dissolved by allowing a class to send a message to a class located beyond several classes. We adopt this approach in this study.

According to the above discussion, it is necessary to hide the instances of the Point and Line classes. We have created the LineWrpper class as a wrapper class and also created the methods getP1X(), getP1Y(), getP2X(), and getP2Y() in the LineWrpper class. The dependency relationship is dissolved by changing the message destination in the called instance.

In addition, the LoD class is added that is used to make a call. The LoD class obtains the values of a coordinate in the Point class from the LineWrapper class. As a result, line.getP1X() can be used to obtain the value of the first x coordinate from the LoD class.

Fig. 8 shows the new class diagram corrected according to the result of detection.

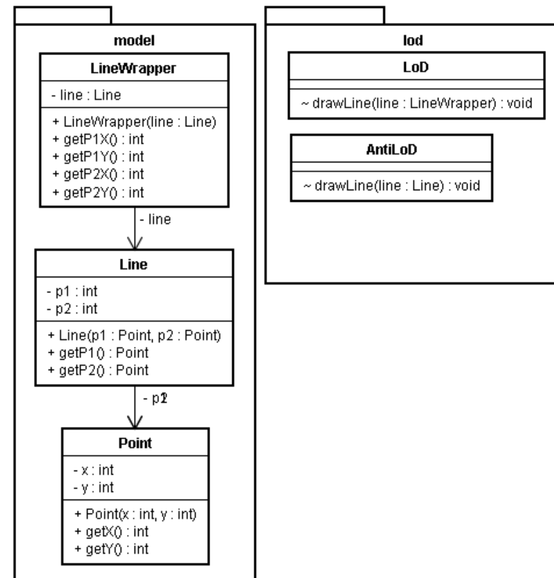

Fig. 8. The corrected class diagram

4) Using the tool to detect violation after correction

Using the source code corrected in Section 5.B.3) as input to the tool has detected no violation.

The LineWrpper class calls the getP1() and getP2() methods in the Line class and returns the Point class. The getX() and getY() methods in the Point class are called. The Point class resides in the same package as the LineWrpper class. As a result, no violation occurs with regard to the Law of Demeter discussed in this study. The experiment has proved that the tool works as expected since no dependency relationship has been detected within the package.

C. Summary of experiment

Since the LoD class takes LineWrpper as its argument, it can call a method of the LineWrpper class. Calling a method of the Point class from the LineWrpper class to obtain the coordinate values does not violate the Law of Demeter defined in this study since the called method resides in the same package.

However, the AntiLoD class takes only the Line argument. As a result, it is possible to call a method of the Line class, but it is not possible to call a method of the Point class. Therefore, obtaining the coordinate values violates the Law of Demeter defined in this study.

The above discussion has validated the Law of Demeter proposed in this study and the tool developed based on it by detecting a message passing beyond more than one package as a violation and allowing a message passing within a package.

Fig. 9 shows the experimental result and summary.

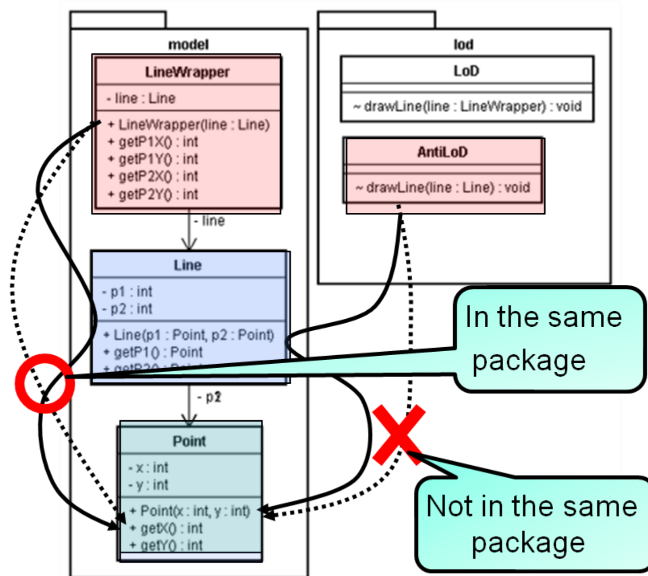

Fig. 9. Comparison of class diagrams before and after correction

D. Consideration on the result of experiment

In Section 5.B.3), the measure taken to dissolve the violation of the Law of Demeter is creating a wrapper class of the Point class rather than creating a wrapper class of the Line class. Complying with the Law of Demeter class by class causes another problem in which many wrapper classes are required to be created to send a message to a class beyond several classes. Lieberherr proposed to use Aspect to resolve such a problem[18]. On the contrary, this study proved that creation of many wrapper classes can be prevented by applying the Law of Demeter to a package rather than a class.

As described in Section 3.C, our approach aims to enhance the reusability of packages by regarding a package as a virtual object and providing the user interface for the object. For example, if a method name is altered in the Point or Line class, the AntiLod class is affected in the case of the class diagram in Fig. 6, however in the case of the class diagram in Fig.8, only the classes in the model package are required to be modified and no other class in other packages is required to be modified.

In addition, with this tool, the class diagram can be used to identify the portion of source code in which a violation is detected, or a message is forwarded. Therefore, this tool can be used not only to modify source code to dissolve dependency relationships but also to clarify how to alter the destination of a message. That is, reviewing the design allows the designer or implementer to decide that the destination of a message should be changed or not.

VI. Related Work

This section discusses the effectiveness of our tool based on comparisons with existing tools, such as Eclipse Metrics Plugin [3] and REV-SA of Cooper et al. [5]

A. An outline of Eclipse Metrics Plugin

To measure the behavior of IDE (Integrated Development Environment) based on various metrics, several tools are available, including Metrics plugin for Eclipse[4], Eclipse Metrics Plugin [3], etc.

Metrics plugin for Eclipse allows the user to visually display the types of dependency relationships observed among packages. Eclipse Metrics Plugin can be used to represent the degree of coupling in a package or for each Type as values.

Using these tools allows the developer to monitor the metrics in real time even under development. Thus, useful information to determine the dependency of modules can be obtained during development.

B. An outline of REV-SA

According to Cooper, et al. [15], in order for an implementation of a system to exactly satisfy the specification, it must closely adhere to the design documents. Even if an implementation satisfies the requirements, they pointed out that a system implemented not closely adhering to the documents may lacks maintainability and support and maintenance of such a system may become very difficult. For this reason, the system REV-SA is generated by applying XMI (XML Metadata Interchange) to the class diagrams created at the time of designing and those created by reverse engineering of source code and by comparing them automatically. Then, we use the system REV-SA to evaluate whether the system is implemented adhering to the original design.

C. The purpose and methodology of experiment

In this experiment, we compared the degree of coupling among classes detected by Eclipse Metrics Plugin with the degree of coupling among packages detected based on our definition, and showed that the number of objects that must be recognized decreases when focusing on the packages. It also showed that the number of objects that must be recognized decreases when focusing on packages.

We applied this tool to each package of org.apache.catalina.* selected from the source code of Version 5.5.20 of Apache Tomcat[17]. An outline of this product is shown in Table 3.

Table 3. An outline of Apache Tomcat

Number of classes	380
Number of packages	18
Programming language	Java
Target package	org.apache.catalina.*
Version	5.5.20

D. Experimental result and consideration

We performed two types of measurement by analyzing the source code of Tomcat with this tool and with Eclipse Metrics Plugin, respectively. Table 4 shows the numbers of message transferred between packages. The results in Table 4 show that

every package has dependent portions with other packages.

Eclipse Metrics Plugin uses the number of classes, or Ce (Efferent Couplings), referenced from the class being measured as the metrics to represent the degree of coupling.

Eclipse Metrics Plugin shows smaller values for some packages such as org.apache.catalina.core since it returns the value of the class which has the maximum Ce value among classes in the target package. (For example, as for the package org.apache.catalina.core, a value of 82 for the StandardContext class is output.) On the other hand, our tool counts all destinations of messages sent from classes in a package, which depend on classes outside the package, and eliminates duplication to generate the total value.

The result shows that, from the viewpoint of the whole program, the number of dependencies based on our definition is smaller, and the number of objects that the designer is required to recognize decreases.

However, showing the number of dependencies alone does not differentiate valid dependencies from invalid ones. The difference can be identified only by reading the source code. In addition, since the number of dependencies may be large depending on a package, it takes a long time and effort to find all of them manually.

Table 4. Comparison of the degree of coupling between our tool and Eclipse Metrics Plugin

The name of package	The degree of coupling measured with our tool	Eclipse Metrics Plugin
org.apache.catalina.ant	7	14
org.apache.catalina.authenticator	27	35
org.apache.catalina.connector	48	55
org.apache.catalina.core	103	82
org.apache.catalina.deploy	1	16
org.apache.catalina.launcher	2	5
org.apache.catalina.loader	13	62
org.apache.catalina.mbeans	52	39
org.apache.catalina.realm	25	41
org.apache.catalina.security	5	18
org.apache.catalina.servlets	21	50
org.apache.catalina.session	24	39
org.apache.catalina.ssi	14	27
org.apache.catalina.startup	39	46
org.apache.catalina.users	8	26
org.apache.catalina.util	19	24
org.apache.catalina.valves	26	28

Thus, the tool can generate output that shows if a specific class in a package has dependency relationships with other classes in other packages.

Eclipse Metrics Plugin allows the user to specify the upper limit of allowable values for metrics. The default upper limit of allowable values is set to 25. In the above case, if the default allowable value is applied, twelve packages out of seventeen packages have a value exceeding the upper limit, which results in generating a warning message.

A possible cause of the result is the size of org.apache.catalina.core that consists of 28 classes and has about 600 KB of source code. On the contrary, org.apache.catalina.launcher consists of only one class and its file size is nothing more than 4 KB. Considering the difference in size, it is natural to exceed the threshold value. It is possible but not easy to specify the threshold value per package.

Therefore, our tool does not aim to judge the validity of a package according to the threshold value, but it aim to present the information that shows which class sends a message to which class. Fig. 8 shows the dependency relationships among packages based on the result generated by the tool for the package org.apache.catalina.users that consists of nine classes and has about 55 KB.

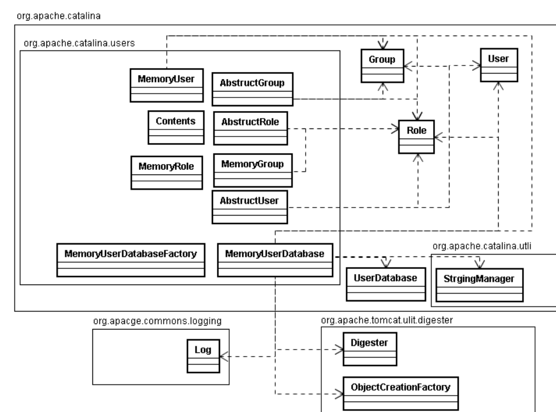

Fig. 10. The dependency relationship of org.apache.catalina.users

As shown in this figure, output can be generated, which shows if a specific class in a package has dependency relationships with other classes in other packages.

REV-SA advises the user to modify the design document by confirming the user if the implementation satisfies the user's requirements or not, focusing on aggregation and multiplicity.

However, just reverse-engineering the source code may cause a problem that the class diagram becomes complicated. Therefore, to reduce the complexity, our tool presents only the dependency relationships among packages.

Taking advantage of the result makes it possible to clarify the dependency relationships among packages, comprehend the dependency relationships without reading the source code, and present the information of which classes need to be redesigned.

E. Summary of consideration

In this study, we assume that indispensable dependency relationships should be intentionally selected by the designer or implementer. For this purpose, we proved that the number of target objects to be recognized can be reduced by focusing on the dependency relationships among packages. In addition, since it is difficult to comprehend the overall structure of a program only by pointing out violated portions, we present the

dependency relationships among packages to promote an understanding of the overall structure. If the implementation is consistent with the design, it is easy to modify programs since the portion that requires correction can be easily understood from the viewpoint of overall program structure.

As a result, such implementation makes it easier to do future program modification, including changes in the specification.

VII. Conclusion and Future Direction

In terms of proper software design, since it is essential to perform proper modularization, we decided to focus on the module independency which is one of the metrics of modularization. Because the Law of Demeter was proposed in the time of C++, we extended its definition to include "friend" so that it may be applicable to Java. Furthermore, we proposed a measure to reduce the complexity in detecting dependency relationships by applying the Law of Demeter to a package rather than a class.

We implemented our tool as a plug-in of Eclipse to discover dependency relationships among packages based on the definition discussed in this paper.

As a result, the tool makes it possible to scan the dependency relationships among packages and detect violations.

Then, the list of message destinations is generated to clarify the position of the modified portion from the viewpoint of the overall program structure and allow the designer and implementer to select the indispensable dependency relationships.

As a result, such implementation makes it easier to do future program modification, including changes in the specification.

It is not possible to use the tool properly if the packages are not modularized adequately. Therefore, it is necessary for novice programmers to learn how to adequately modularize packages in advance. It is possible to naturally modularize packages by referring to the coupling information of each package and by modularizing the package with this tool.

In addition, in this study we applied the Law of Demeter to packages. However, when the same Law of Demeter is applied to classes, since the number of target objects becomes bigger, it is necessary to develop the rule to reduce messages by identifying the role of the message between objects, otherwise it is not possible to comprehend the relationship of messages to achieve an appropriate degree of coupling.

For future direction, we plan to handle the case in which a system that is implemented as specified in the design document causes a violation of the Law of Demeter. In such a case, we are going to prepare several measures, including exclusion of violation portion according to the intention of the designer. Another interesting issue is to generate a warning message when the system is deviated from the design on the way of implementation by comparing the design with the implementation in the XMI format. For this purpose, the design documents such as class diagrams need to be converted to the XMI format in advance.

Appendix

The following source code used in Chapter 5

Model.Point

```
 1:  package model;
 2:
 3:  public class Point {
 4:
 5:      private int x, y;
 6:
 7:      public Point(int x, int y) {
 8:          this.x = x;
 9:          this.y = y;
10:      }
11:
12:      public int getX() {
13:          return x;
14:      }
15:
16:      public int getY() {
17:          return y;
18:      }
19:  }
```

Model.Line

```
 1:  package model;
 2:
 3:  public class Line {
 4:
 5:      private Point p1, p2;
 6:
 7:      public Line(Point p1, Point p2) {
 8:          this.p1 = p1;
 9:          this.p2 = p2;
10:      }
11:
12:      public Point getP1() {
13:          return p1;
14:      }
15:
16:      public Point getP2() {
17:          return p2;
18:      }
19:  }
```

Lod.AntiLod

```
 1:  package lod;
 2:
 3:  import model.Line;
 4:
 5:  public class AntiLoD {
 6:
 7:      java.awt.Graphics g;
 8:
 9:      void drawLine(Line line) {
10:          g.drawLine(
11:              line.getP1().getX(), // violation
12:              line.getP1().getY(), // violation
13:              line.getP2().getX(), // violation
14:              line.getP2().getY()); // violation
15:      }
16:  }
```

Model. LineWrapper

```
 1:  package model;
 2:
 3:  public class LineWrapper {
 4:
 5:      private Line line;
 6:
 7:      public LineWrapper(Line line) {
 8:          this.line = line;
 9:      }
10:
11:      public int getP1X() {
12:          return line.getP1().getX();
13:      }
14:
15:      public int getP1Y() {
16:          return line.getP1().getY();
17:      }
18:
19:      public int getP2X() {
20:          return line.getP2().getX();
21:      }
22:
23:      public int getP2Y() {
24:          return line.getP2().getY();
25:      }
26:  }
```

Lod.Lod

```
 1:  package lod;
 2:
 3:  import model.LineWrapper;
 4:
 5:  public class class LoD {
 6:
 7:      java.awt.Graphics g;
 8:
 9:      void drawLine(LineWrapper line) { // LineWrapper is
friend
10:          g.drawLine(
11:              line.getP1X(),
12:              line.getP1Y(),
13:              line.getP2X(),
14:              line.getP2Y());
15:      }
16:  }
```

REFERENCES

[1] G.J Myers, Composite / Structured Design. Van Nostrand, Reinhold, 1978.

[2] ISO/IEC9126-1. Software engineering { Product quality { Part1: Quality model. ISO, 2001.

[3] Shyam R, Chidamber and Chris F. Kemerer, A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, Vol.20, No.6, June 1994, pp.476-493.

[4] Martin Fowler. Refactoring: Improving The Design of Existing Code.Addison-Wesley, 1999.

[5] C.Lewerentz F.Simon, F. Steinbruckner. Metrics based refactoring. InProc. European Conf. Software Maintenance and Reengineering, pp. 30–38, 2001.

[6] K.Kontogiannis L.Tahvildari. A metric based approach to enhance design quality through meta-pattern transformations. In Proceedings of7th European Conference on Software Maintenance and Reengineering, pp.183‐192, March 2003.

[7] Katsuhiko Hatano, Yoshinari Nomura, Hideo Taniguchi , Kazuo Ushijima. Development Environments and Automated Technologies) A Mechanism to Support Automated Refactoring Process Using Software Metrics <Special Issue> Object-Oriented Technologies.(in Japanease) Transactions of Information Processing Society of Japan, Vol.44, No.6, pp. 1548–1557,20030615.

[8] K. Lieberherr, I. Holland and A. Riel, Object-oriented programming: an objective sense of style. Conference proceedings on Object-oriented programming systems, languages and applications, San Diego California United States, September, 1988, pp.323-334.

[9] K. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method. PWS Pub., Boston, 1996.

[10] Larman, C., Applying UML and patterns: an introduction to object-oriented analysis and design. Prentice Hall, 1998.

[11] D. L. Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules,” CACM, Vol.15, No.12, pp.1053-1058, Dec. 1972.

[12] Eclipse. http://www.eclipse.org/ (2010/10/30)

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns. Addison-Wesley, 1994.

[14] Eclipse Metrics Plugin. http://www.stateofflow.com/projects/16/eclipsemetrics (2010/10/30)

[15] David Cooper, Benjamin Khoo, Brian R. von Konsky and Michael Robey, Java implementation verification using reverse engineering. ACM International Conference Proceeding Series Vol. 56, Proceedings of the 27th Australasian conference on Computer science, Vol. 26, 2004, pp. 203–211.

[16] Metrics plugin for Eclipse. http://metrics.sourceforge.net/ (2010/10/30)

[17] Apache Foundation, Apache Tomcat. http://tomcat.apache.org/ (2010/10/30)

[18] Karl J. Lieberherr. Adaptive Object-Oriented Software:The DemeterMethod with Propagation Patterns. PWS Publishing Company, Boston,1996. ISBN 0-534-94602-X.

Ryota Chiba He received a bachelor's degree in engineering from Shibaura Institute of Technology, Japan in 2007. He received a master's degree from Graduate School of Engineering, Shibaura Institute of Technology, Japan, in 2009.At present, he is working in Japan National Statistics Center.

Hiroaki Hashiura He received a bachelor's degree in engineering from Shibaura Institute of Technology, Japan in 2002. He received a professional degree in engineering management from Graduate School of Engineering Management, Shibaura Institute of Technology, Japan in 2005. He received Doctorate in engineering from Graduate School of Engineering, Shibaura Institute of Technology, Japan, in 2008. At present, he is postdoctoral fellow at Shibaura Institute of Technology.

Seiichi Komiya He received the degree of the B. S(C), from Saitama University, Japan, 1969. He received Dr. Eng. from Shinshu University in March 2000. He was been working for Hitachi Ltd. as a software engineer during 1970-2001, and has been on loan from Hitachi Ltd. to Information-technology Promotion Agency Japan (IPA), a substructure of MITI, during 1984-1999. He has studied the frameworks to construct many kinds of CASE tools (e.g. an automatic programming system, a software collaborative distributed development environment, etc.), software specification/design process, CAI and Intelligent CAI at IPA. In IPA, he has been a principal researcher of Software Technology Center during 1988-1999, and was also an assistant to director general of Laboratory for New Software Architectures during 1991-1998. He works for Shibaura Institute of Technology as a full-time professor since April 2001. He was also a visiting professor of Tokushima University in 1993. He was also a visiting lecturer of Chiba University during 1995- 2009, and a visiting professor of a graduate school of Shibaura Institute of Technology 1997-2001. He was a manager/a vice-chairman/a chairman of SIG-KBSE/IEICE during 1992-1994/1994/1996/1996-1998. He was also a member/a manager of editorial committee of IEICE transactions 1994-1999/1998-1999. He was given a title of “IEICE fellow” from IEICE in 2010.