日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

ModiChecker : Accessibility Excessiveness Analysis Tool for Java Program

Dotri Quoc　Kazuo Kobori　Norihiro Yoshida
Yoshiki Higo　Katsuro Inoue

In object-oriented programs, access modifiers are used to control the accessibility of fields and methods from other objects. Choosing appropriate access modifiers is one of the key factors for highly secure and easily maintainable programming. In this paper, we propose a novel analysis method named Accessibility Excessiveness (AE) for each field and method in Java program, which is discrepancy between the access modifier declaration and its real usage. We have developed an AE analyzer - ModiChecker which analyzes each field or method of the input Java programs, and reports the excessiveness. We have applied ModiChecker to various Java programs, including several OSS, and have found that this tool is very useful to detect fields and methods with the excessive access modifiers.

1 Introduction

To realize good encapsulation in Java programs, we have to choose appropriate access modifiers of methods and fields in a class, which may be possibly accessed by other objects. However, inexperienced developers tend to set all of the access modifiers `public` or `none` as `default` indiscriminately.

For example, Figure 1 is a case of bad access modifier setting. Suppose that we have 2 methods: *Method A* and *Method B* in class *X*. *Method A* keeps an initialization process for *Method B*. It means *Method A* must be called before *Method B* is called. Otherwise, *Method B* can not work properly. In this case, *Method B* should be always called via *Method A*, and the access modifier of *Method B* should be set `private`. However, a novice developer might set that access modifier `public` without thinking seriously. In a meanwhile, other developer would want to use *Method B* and he/she can directly call it since the access modifier of *Method B* allows direct access to it. This may cause a fault due to lack of the initialization process performed by *Method A*.

In this example, the access modifier of *Method B* is `public`, but the current program accesses Method B from `private` method (*Method A only in this case*) and the access modifier of method B should be `private`. Such discrepancy between declared accessibility and actual usage of each method and field is called *Accessibility Excessiveness(AE)* here.

An AE would cause an unwilling access to a method or

ModiChecker :Java プログラムのアクセス修飾子過剰性分析ツール
Dotri Quoc, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻, Department of Computer Science, Graduate School of Information Science and Technology, Osaka University.
小堀 一雄, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻, Department of Computer Science, Graduate School of Information Science and Technology, Osaka University.
吉田 則裕, 奈良先端科学技術大学院大学 情報科学研究科, Graduate School of Information Science, Nara Institute of Science and Technology.
肥後 芳樹, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻, Department of Computer Science, Graduate School of Information Science and Technology, Osaka University.
井上 克郎, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻, Department of Computer Science, Graduate School of Information Science and Technology, Osaka University.

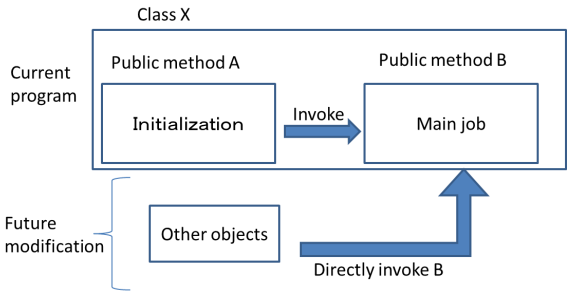

Figure 1 Example of Wrong Access Modifier Declaration

field which should not be accessed by other objects in a latter development or maintenance phase as shown in the example. Also, existence of AE would be an indicator of immaturity of the developer.

Checking AE by hand for each method and field is not a easy task since collecting actual usage of methods and fields from other objects requires a lot of effort.

In this paper, we propose an AE analysis tool named *ModiChecker*, which takes a Java program as input, then analyzes and reports the excessiveness of each access modifier declared for each method and field. ModiChecker is based on static program analysis framework *MASU*[3,4,5], which allows a flexible composition of various analysis tools very easily.

Using ModiChecker, we have analyzed several open source software(OSS) such as Ant and jEdit. Also, MASU itself has been analyzed by ModiChecker. The analysis results show that some OSS contain many AE methods and fields, which should be set to more restrictive access modifiers.

There are little works related to ours. Tai Cohen studied the distribution of the number of each Java access modifier in some sample methods[1].

In the following, we will define AE in Section 2, Section 3 describes ModiChecker and MASU. In Section 4, we will show our experimental results of ModiChecker. Section 5 will conclude our discussions with a few future works.

2 Accessibility Excessiveness Map

Table 1 is called *Accessibility Excessiveness Map*(AE map), which lists all the cases where an AE happens. The vertical column shows the declaration of an access modifier for a method or field in the source code. The horizontal row shows its actual usage from other objects. Each element in AE map is an *AE Identifier(AE id)* which identifies each AE case. For example, if a method has `public` as the declaration of the access modifier, and it is accessed only by the objects of same class, the AE id is “pub3” meaning it could be set to `private`. Note that “`default(none)`” means the case that there is no explicit declaration of the access modifier and it is the same as `package`.

An AE id “`ok-xxx`” means that there is no discrepancy between the declaration and actual usage, and it is an ideal way of secure and quality programming. An AE id “`x`” means that these cases are detected as error at the compilation time and they are out of the scope of the AE analysis. An AE id in shaded cells means that the declaration is excessive one from the actual usage of the access modifier.

Purpose of the AE analysis is to identify an AE id for each method and field in the input source code. Also, we are interested in the statistic measures of AE ids for the input program, which would be important clues of program quality.

3 AE Analysis Tool ModiChecker

3.1 Approach to AE Analysis

To perform the AE analysis, we need to know the declaration of the access modifiers of each method and field of the input program. This is easily done by parsing the program. Also, we have to investigate into the actual usage of each method and field. For this work, we employ a static source-code analysis, which identifies other classes that may possibly access the target method or field. For these purposes, we have used a Java program analysis framework MASU[3,4,5].

Table 1 Accessibility Excessiveness Map

Declaration \ Actual Usage	Public	Protected	Default(None)	Private
Public	ok-pub0	pub1	pub2	pub3
Protected	x	ok-pro0	pro1	pro2
Default(None)	x	x	ok-def0	def1
Private	x	x	x	ok-pri0

MASU has been originally designed to implement pluggable multi-purpose metrics infrastructure, but it is very useful as a Java program analysis framework. MASU transforms the input Java program into an Abstract Syntax Tree (AST), and then it analyzes AST for actual usage of the methods and fields in the input program.

3.2 Overview of ModiChecker Architecture

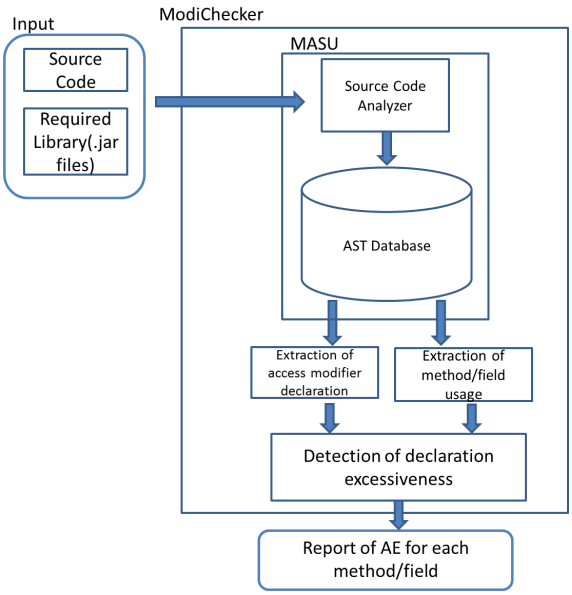

Figure 2 Architecture of ModiChecker

Figure 2 shows the architecture of ModiChecker. Firstly, Modichecker reads source program and all of the required library (normally, the library files are often in .jar files) in Java. The source code is transformed to an AST associated with various static code analysis results.

After analyzing source, we get the access modifier declaration and also usage of each field and method. From the AST database, we can easily know which class may access that method/field.

By comparing the declaration of the access modifier and real usage of the field and method, ModiChecker reports AE for each field and method.

ModiChecker treats some special cases as follow

- ModiChecker does not give any report for methods of abstract classes or interfaces because they are overridden by the method of other classes. One more reason is that an abstract class or interface does not generate any object so that its methods will never be called and those access modifiers do not affect maintenance processes.
- In the case of a method overriding an other method, the overriding method in a subclass must have an access modifier with an equal or more permissive level to the access modifier of the overridden method. ModiChecker detects such an overriding method and reports an AE id between the access modifier of the overridden method and its actual usage. For example, in Figure 3, assume that we have two classes *Class A* and *Class B* with *Method A.C* and *Method B.C* of access modifier `public` for both. *Method B.C* overrides *Method A.C* so ModiChecker do not report `private` for *Method B.C* even if *Method B.C* is actually used inside *Class B* only.

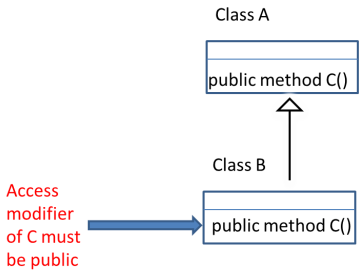

Figure 3 Example of Access Modifier of Overriding Method

4 Experiment and Discussion

4.1 Overview

We have conducted case studies with some open-source code projects to evaluate the AE analysis. In the evaluation, we have focused on the following points.

- The total number of each AE id is measured to evaluate how program is well designed.
- Based on the above result, we have closely investigated the reasons of setting the access modifiers excessively. Sometime an access modifier would be intentionally set excessively by the developers for the future purpose, or sometime they would be set excessively by automatic code generator.

The target software products are the following Java programs.

- MASU itself with 519 source files and 102,000 LOC (41,000 LOC are automatically generated code by ANTLR.)
- Ant 1.8.2 with 1,141 source files and 127,235 LOC.
- jEedit 4.4.1 with 546 source files and 109,479 LOC.

4.2 Experiment Result

4.2.1 MASU

By analyzing MASU, we got the number of detected AE ids for methods and fields as shown in Table 2 and Table 3.

We have found 280 fields with the excessive access modifiers. Out of these 280 fields, 255 fields were identified as automatically generated code by our hand-analysis. We have interviewed the developer of MASU and asked the reason of the excessiveness of other fields. 20 fields have been found that they are intentionally set excessively for future uses. Finally, 5 fields have been identified actually excessive and those access modifiers have been changed to proper ones.

We have also found 253 methods with the excessive access modifier. And by our hand-analysis, 6 methods were found to be automatically generated code. Out of those 253 methods, 181 methods are intentionally set excessively for future uses. Finally, 66 methods have been identified actually excessive and those access modifiers have been changed to proper ones.

4.2.2 Ant 1.8.2

We have investigated into the newest version of Ant 1.8.2 and got the number of detected AE ids for methods and fields as shown in Table 4 and Table 5.

We have found 611 fields and 1520 methods with the excessive access modifiers. By our hand-analysis, we were unable to find any field with excessive access modifier generated by some automatic code generator.

Looking at the ratio of excessive access modifier, the ratio of excessive fields is 18.9%(shown in the shaded cells in Table 4) while ratio of excessive methods is 35.5%(shown in shaded cells in Table 5). Since a standard design strategy might be to make all fields private and to provide public getter/setter methods for them, methods has more probability to be set excessively for future use than fields. That would be the reason why the ratio of excessive methods is higher than ratio of excessive fields.

4.2.3 jEdit 4.4.1

The result of detected AE ids for methods and fields for jEdit 4.4.1 is shown in Table 6 and Table 7

We have found 604 fields and 981 methods with the excessive access modifiers. We were unable to find any field or method with excessive access modifier generated by some automatic code generator by our hand-analysis.

For jEdit 4.4.1, the ratio of excessive fields is

Table 2 Number of Detected AE ids for Fields in MASU(Ratio to the Total)

Declaration \ Actual Usage	Public	Protected	Default(None)	Private
Public	16(6.4%)	0	3(0.4%)	259(33.0%)
Protected	x	0	1(0.1%)	14(1.8%)
Default(None)	x	x	0	3(0.4%)
Private	x	x	x	488(62.2%)

Total fields : 784 Total fields in shaded cells : 280(35.7%)

Table 3 Number of Detected AE ids for Methods in MASU(Ratio to the Total)

Declaration \ Actual Usage	Public	Protected	Default(None)	Private
Public	471(26.9%)	3(0.2%)	90(5.1%)	124(7.1%)
Protected	x	19(1.1%)	0	35(2.0%)
Default(None)	x	x	4(0.2%)	1(0.1%)
Private	x	x	x	1006(57.4%)

Total fields : 1753 Total fields in shaded cells : 253(14.43%)

Table 4 Number of Detected AE ids for Fields in Ant 1.8.2 (Ratio to the Total)

Declaration \ Actual Usage	Public	Protected	Default(None)	Private
Public	157(4.9%)	22(0.7%)	64(2.0%)	285(8.9%)
Protected	x	54(1.7%)	47(1.5%)	135(4.2%)
Default(None)	x	x	21(0.7%)	58(1.8%)
Private	x	x	x	2395(73.8%)

Total fields : 3238 Total fields in shaded cells : 611(18.9%)
Total fields in shaded cells : 611

Table 5 Number of Detected AE ids for Methods in Ant 1.8.2 (Ratio to the Total)

Declaration \ Actual Usage	Public	Protected	Default(None)	Private
Public	1576(36.8%)	100(2.3%)	609(14.2)	454(10.6%)
Protected	x	103(2.4%)	117(2.7%)	217(5.1%)
Default(None)	x	x	52(1.2%)	23(0.5%)
Private	x	x	x	1034(24.1%)

Total fields : 4284 Total fields in shaded cells : 1519(35.5%)

Table 6 Number of Detected AE ids for Fields in jEdit 4.4.1 (Ratio to the Total)

Declaration \ Actual Usage	Public	Protected	Default(None)	Private
Public	228(9.1%)	10(0.4%)	111(4.4%)	163(6.5%)
Protected	x	23(0.9%)	15(0.6%)	51(2.0%)
Default(None)	x	x	126(5.0%)	254(10.1%)
Private	x	x	x	1529(60.9%)

Total fields : 2510 Total fields in shaded cells : 604(24.1%)

Table 7 Number of Detected AE ids for Methods in jEdit 4.4.1 (Ratio to the Total)

Declaration \ Actual Usage	Public	Protected	Default(None)	Private
Public	1224(34.5%)	77(2.4%)	544(16.7)	237(7.3%)
Protected	x	44(1.4%)	14(0.4%)	23(0.7%)
Default(None)	x	x	233(7.2%)	86(2.6%)
Private	x	x	x	874(26.8%)

Total methods : 3223 Total fields in shaded cells : 981(30.4%)

24.5%(shown in the shaded cells in Table 6) while the ratio of excessive methods is 30.4%(shown in the shaded cells in Table 7). Like the case of Ant 1.8.2 shown above, the ratio of excessive fields is lower than the ratio of excessive methods.

4.3 Discussion

To validate the analysis result of ModiChecker, we have changed all the excessive access modifiers of above three programs to suggested access modifiers. All the modified programs have been compiled and executed without any error. This indicates that the output report of ModiChecker is proper one in the sense that the reported excessive access modifiers can be changed to more restrictive access modifiers without causing any error. However, as mentioned before, some fields/methods are intentionally set excessive for future uses. Thus, we need a tool by which a developer can select the excessive access modifiers to change to more restrictive ones.

By using AE analysis, we could propose a quality metrics in the following two ways.

- We set a value called AE index for each AE id and sum up each AE index as metrics value.
- We set a value for each method and field based on the number of other classes accessing those fields and methods. Those values for each method/field are accumulated as this metrics value.

The idea of using access modifier metrics would be related to our previous work[2]. In that paper, the number of each Java access modifier is used as one of the metrics for checking the similarity between Java source codes.

5 Conclusion

In this paper, we have proposed an analysis method named AE for each field and method in Java program, which is discrepancy between an access modifier declaration and the real usage of the field and method. We have also introduced AE Map which lists all of the cases where an AE happens.

We have developed a tool named ModiChecker, which

finds excessive method/field and reports AE id of each excessive method/field. We have also used ModiChecker to analyzed several OSS such as MASU, Ant, jEdit, and found that our system is quite useful to detect fields and methods with the excessive access modifiers.

Since there is no refactoring tool or automatic code generating tool for detecting and optimizing the access modifiers as discussed here, we think ModiChecker will be an important tool to support secure and quality programming in Java.

Currently we are analyzing other Java programs including industrial systems, and are trying to identify the relation between the AE analysis results and other program quality indicators such as bug frequency.

Acknowledgements This work is supported by ISPS, Grant-in-Aid for Scientific Research (A) (No.21240002) and Grant-in-Aid for Exploratory Research (No.23650015). This is also supported by MEXT Stage Project, the Development of Next Generation IT Infrastructure.

References

[1] Tal Cohen, “Self-Calibration of Metrics of Java Methods Towards the Discovery of the Common Programming Practice”, The Senate of the Technion, Israel Institute of Technology, Kislev 5762, Haifa, 2001.

[2] K. Kobori, T. Yamamoto, M. Matsushita , and K. Inoue, “Java Program Similarity Measurement Method Using Token Structure and Execution Control Structure”, Transactions of IEICE , Vol.J90-D No.4 pp. 1158–1160, 2007.

[3] Y.Higo, A. Saito, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, “A Pluggable Tool for Measuring Software Metrics from Source Code”, accepted by The Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, Nov. 2011 (to appear).

[4] A. Saito, G. Yamada , T. Miyake, Y. Higo, S. Kusumoto, K.Inoue, “Development of Plug-in Platform for Metrics Measurement”, International Symposium on Empirical Software Engineering and Measurement, Poster Presentation, Lake Buena Vista, 2009.

[5] MASU, http://sourceforge.net/projects/masu/

[6] ANTLR, http://antlr.org

[7] Ant, http://ant.apache.org

[8] jEdit, http://jedit.org