特別研究報告

題目

Case Studies of Accessibility Excessiveness Analysis for Java Programs

指導教員
井上克郎 教授

報告者
Quoc DoTri

平成 24 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 23 年度 特別研究報告

Case Studies of Accessibility Excessiveness Analysis for Java Programs

Quoc DoTri

Abstract

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 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.

Keywords

Access Modifier
Accessibility Excessiveness
ModiChecker

目 次

1 Introduction

1.1 Four principles of Object Oriented Programming(OOP)

Object-orientd programs are often said to have four major properties [1] as following:

1. Encapsulation : In object-oriented programming, objects interact with each other by messages. The only thing that an object knows about another object is the object's interface. Each object's data and logic is hidden from other objects. In other words, the interface encapsulates the object's code and data.

   This allow the developer to separate an object's implementation from its behavior. This separation creates a "black-box" affect where the user is isolated from implementation changes. As long as the interface remains the same, any changes to the internal implementation is transparent to the user. Encapsulation is necessary because if we make all data and logic of a class directly visible to external users (for example in Java, make all fields public), we lose control over what they do to the fields. They might modify the fields in such a way as to break the intended functionality of the object (give the fields inappropriate values, let ’s say).

2. Data abstraction : Data abstraction is the simplest of principles to understand. Data abstraction and encapsulation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details. Abstraction denotes a model, a view, or some other focused representation for an actual item. It is the development of a software object to represent an object we can find in the real world. Encapsulation hides the details of that implementation.

   Data abstraction is used to manage complexity. Software developers use abstraction to decompose complex systems into smaller components. As development progress, programmers know the functionality they can expect from as yet undeveloped subsystems. Thus, programmers are not burdened by considering the ways in which the implementation of later subsystems will affect the design of earlier development.

3. Inheritance Objects can relate to each other with either a “ has a ”, “ uses a ” or an “ is a ” relationship. Figure 1 is one example of inheritance. In this example, we can say graduate student and undergraduate student are students. Inheritance allows a class to have the same behavior as another class and extend or tailor that behavior to provide special action for

specific needs. here we can see both Graduate class and Undergraduate class have similar behavior such as managing a name, an address, and a GPA. Rather than put this behavior in both of these classes, the behavior is placed in a new class called Student. Both Graduate and Undergraduate become subclass of the Student class, and both inherit the Student behavior. We can say both Graduate and Undergraduate students are Student

Both Graduate and Undergraduate classes can then add additional behavior that is unique to them. For example, Graduate can be either Master's program or PhD program. On the other hand, Undergraduate class might want to keep track of either the student is Freshman, Sophmore, Junior or Senior. Classes that inherit from a class are called subclasses. The class a subclass inherits from are called superclass. In the example, Student is a superclass for Graduate and Undergraduate. Graduate and Undergraduate are subclasses of Student.

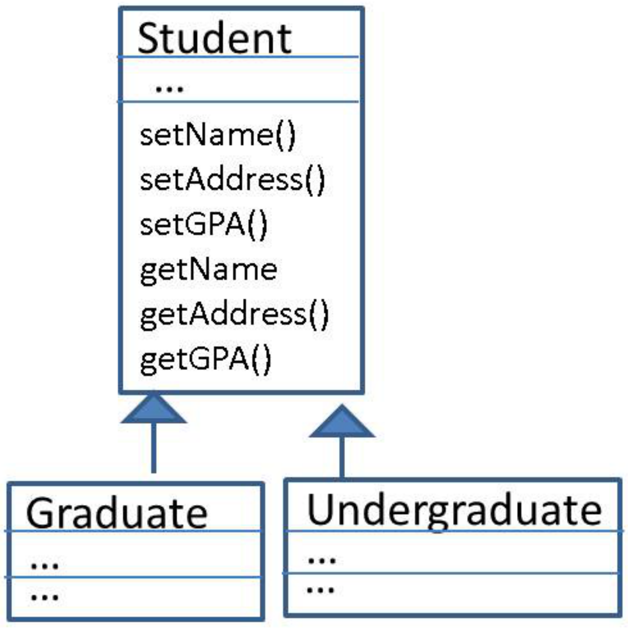


図 1: Example of Inheritance

4. Polymorphism: Polymorphism means one name, many forms. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality. Many Java programmers are familiar with interface polymorphism. We are only going to introduce polymorphism from the point of view of inheritance because this is the part that is new to many people. Because of this, it can be difficult to fully grasp the full potential of polymorphism until you get some practice with it and see exactly what happens under

different scenarios. We are only going to talk about polymorphism at the basic level.

There are 2 basic types of polymorphism. Overriding, also called run-time polymorphism, and overloading, which is referred to as compile-time polymorphism. This difference is, for method overloading, the compiler determines which method will be executed, and this decision is made when the code gets compiled. Which method will be used for method overriding is determined at runtime based on the dynamic type of an object.

In this paper, we are mostly investigated into Encapsulation and developed tool to solve problems we are interested in.

1.2 Encapsulation and Access Modifier Problem

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.

Before showing one example of improper access modifier setting, we would like to introduce you access modifier system in Java. In this paper, we only focus on the access modifier of fields/methods of classes in Java so I would like to give some brief introduction of access modifier of fields/methods . In java, access modifier of a field/method control the access of other classes to that field/method. In other word, access modifier determines how fields/methods are accessible from other classes. The Table 1 below show the access level of access modifier in Java language.

表 1: Access level of access modifier in Java

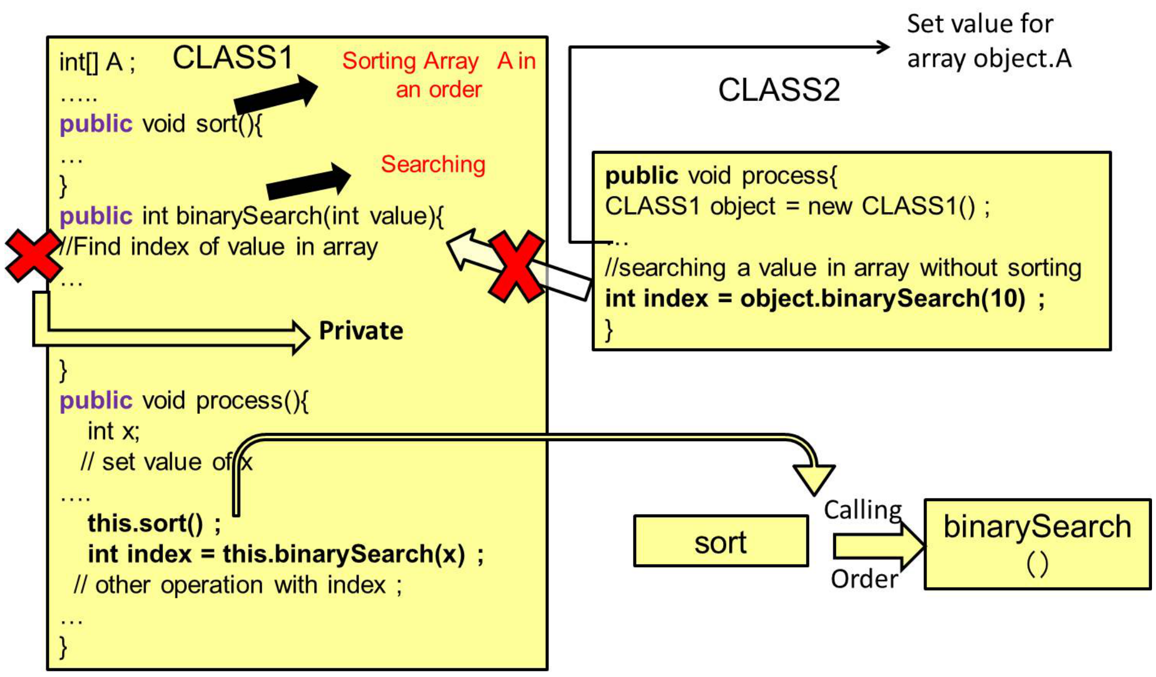
Access Modifier	Class	Package	Subclass	Any Classes
Public	OK	OK	OK	OK
Protected	OK	OK	OK	-
Default	OK	OK	-	-
Private	OK	-	-	-

Look at the Table 1, we can see that how access modifier determines the accessible range of fields/methods in Java. The fields/methods which has access modifier as `public` can be accessed from everywhere. The fields/method which has access modifier as `protected` can be accessed only from their owner class, the classes in the same package with their owner class. The fields/methods which has access modifier as `default` (no declaration for access modifier) can be accessed only from their owner class and the classes in the same package with their owner class. The fields/methods which has access modifier as private can be accessed only from their owner class.

Figure 2 is a case of inappropriate Access Modifier Declaration. This example shows the implementation of binary search. Most of the programmers should be familiar with binary search - one kind of searching method. To search for a key value (comparable type such as int , double...) in an array, we first have to sort that array in an order(ascending or descending). In this case, assume that we sort the array in ascending order. Here, we have the array as current array. Then the process of searching can be described as following:

- Step1 : If length of the current array is 0 then we stop searching and state that the key value does not exist in the array.
- Step2 : Compare the key value to middle element of the array.
- Step3-1 : If value of the middle element is equal to the key value, we stop the searching process and state that the value we want to search for is the middle element of the array
- Step3-2 : If the value of the middle element is smaller than the key value, we understand that if the key value exists in the array it must be in the right side of the middle element. So we make a new array with the elements start from the element right after the middle element to the last element of the current array.
- Step 3-3 : The value of the middle element is bigger than the key value, we understand that if the key value exists in the array it must be in the left side of the middle element. So we make a new array with the elements start from the first element to the element right before the middle element .
- Step 4 : Assign current array to new array then jump to Step 1

In the Figure 2, we have class *CLASS1* which implements the binary search. Class *CLASS1* has array *A* as a field, method *sort()* keeps the role of sorting array *A* to an order (ascending or descending), method *binarySearch()* is for searching key value in the sorted array *A*. The whole searching process is implemented in method *process()*. Here value of variable *X* is set, then method *binarySearch(x)* is called after calling method *sort()*. Note that, method *binarySearch* only can be called after method *sort()* is called once. Otherwise, method *binarySearch()* can not work properly. In this case, method *binarySearch()* should be always called via method *sort()*, and the access modifier of method *binarySearch()* shout 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 *binarySearch()* and he/she can directly call it since the access modifier of method *binarySearch()* allows direct access to it like in the class *CLASS2*. In class *CLASS2*, after object of class CLASS1 is set, we assume that value of object A is defined. Here, this developer doesn't know about the calling order and he/she could call method *object.binarySearch(10)* directly without calling method *object.sort()* and it could cause a fault here.



```
int[] A ;
…..
public void sort(){
…
}
public int binarySearch(int value){
//Find index of value in array
…
}
public void process(){
   int x;
  // set value of x
….
   this.sort() ;
   int index = this.binarySearch(x) ;
 // other operation with index ;
…
}
```

```
public void process{
CLASS1 object = new CLASS1() ;
…
//searching a value in array without sorting
int index = object.binarySearch(10) ;
}
```

図 2: Inappropriate Access Modifier Declaration

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

When investigated into the actual usage of fields/methods from the other classes, we also found that there are some fields/methods exist in the java program which was declared but was unused in the whole program. And we also treat such fields/methods as one kind of AE in this paper.

Existence of AE would be a bad smell of program, and it would indicate various issues on the designs and developments of program as follows.

1. Immature Design and Programming Issue: An AE would cause unwilling access to a method or field which should not be accessed by other objects in a latter development or maintenance phases as shown in the example. This is an issue of design and development processes from the view point of encapsulation [2]. This problem shows the immaturity and carelessness of the designer and developer.

2. Maintenance Issue: Sometimes developer intentionally set field or method excessive for future use or for the purpose of being called from outsiders. It is not easy to distinguish whether AE is intentionally set by developer or it is a case of Issue 1, so that the maintenance of such program is not straightforward and complicated.

3. Security Vulnerability Issue: A program with AE has potential vulnerability of its security in the sense that an attacker may access an AE field and/or method against the intention of the program designer and developer [3].

In this paper, we discuss on an AE analysis method mostly focusing on its application to Issue 1 and 2, and Issue 3 will be a further research topic.

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* [7] [8] [9] , 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 some previous works related to ours. Tai Cohen studied the distribution of the number of each Java access modifier in some sample methods [4]. Security vulnerability analysis has been studied using static analysis approaches [5]. Among these researches, an issue of access modifier declaration has been discussed by Viega et al. [3], where a prototype system Jslint has been presented without any detailed explanation of its internal algorithm and architecture. Also, Jslint only gives warning for the fields/methods which are undeclared private, while our tool supports all kinds of access modifier declarations based on analyzing actual usage.

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. Section 5 will conclude our discussions with a few future works.

2 Accessibility Excessiveness Map

表 2: **Accessibility Excessiveness Map**

Declaration \ Actual Usage	Public	Protected	Default	Private	Unused
Public	ok-pub0	pub1	pub2	pub3	pub4
Protected	-	ok-pro0	pro1	pro2	pro3
Default	-	-	ok-def0	def1	def2
Private	-	-	-	ok-pri0	pri1

Table 2 is called *Accessibility Excessiveness Map* (AE map), which lists all the cases where an AE happens. The horizontal row shows the declaration of an access modifier for a method or field in the source code. The vertical column shows its actual usage from other objects. The last column "`Unused` " means that field/method was not actually accessed by any 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` " means the case that there is no explicit declaration of the access modifier and it is the same as `package`. Or if a field has `public` as the declaration of the access modifier, and it is not accessed by any object, the AE id is "pub4" meaning it could be deleted.

An AE id "`ok-xxx`" means that there is no discrepancy between the declaration and actual usage, and it is an ideal way of 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 Overview of ModiChecker Architecture

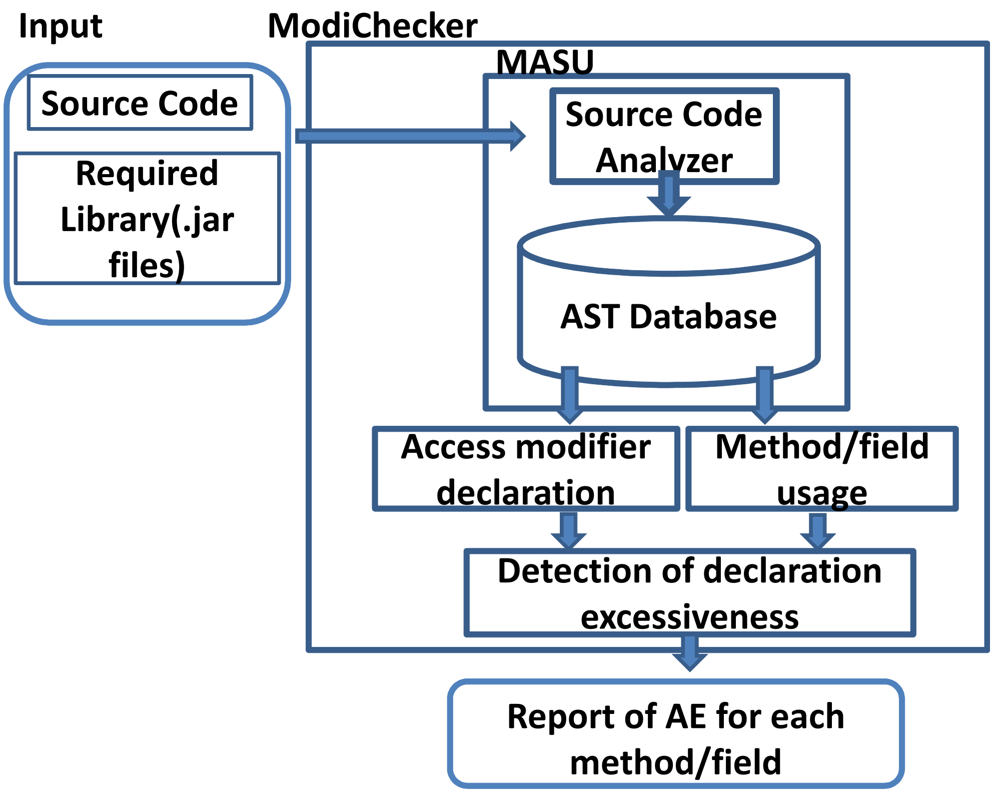

Input
ModiChecker
MASU
Source Code
Required Library(.jar files)
Source Code Analyzer
AST Database
Access modifier declaration
Method/field usage
Detection of declaration excessiveness
Report of AE for each method/field

図 3: Architecture of ModiChecker

Figure 3 shows the architecture of ModiChecker. Firstly, ModiChecker reads source program and all of the required library files (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 another 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 does not report `private` for method *B.C* even if method *B.C* is actually used inside class *B* only.

  In dynamic binding cases, if a class accesses a method of a superclass, ModiChecker will consider that class also accesses the method of all subclasses of the superclass. By this way, dynamic binding cases should not bring about any bad effect to ModiChecker analysis result.

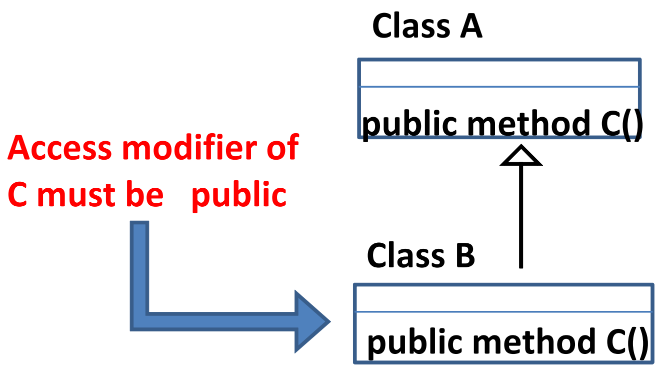


図 4: Access Modifier of Overriding Method

3.2 GUI of ModiChecker

When the program starts, the window is shown as Figure 5 below.

Part 1 is the input label area. This part shows the input target of the project. Users can click to the input label then the folder selection window will appear as Figure 6 to let users choose target project input. Note that before analyzing the source code, ModiChecker needs to analyze the dependent libraries. I suppose that the library files should always be inside a subfolder named *lib* of the target project folder. So in order to let ModiChecker analyze dependent libraries, users should make a subfolder named *lib* inside target project folder and put all of the library files(.jar

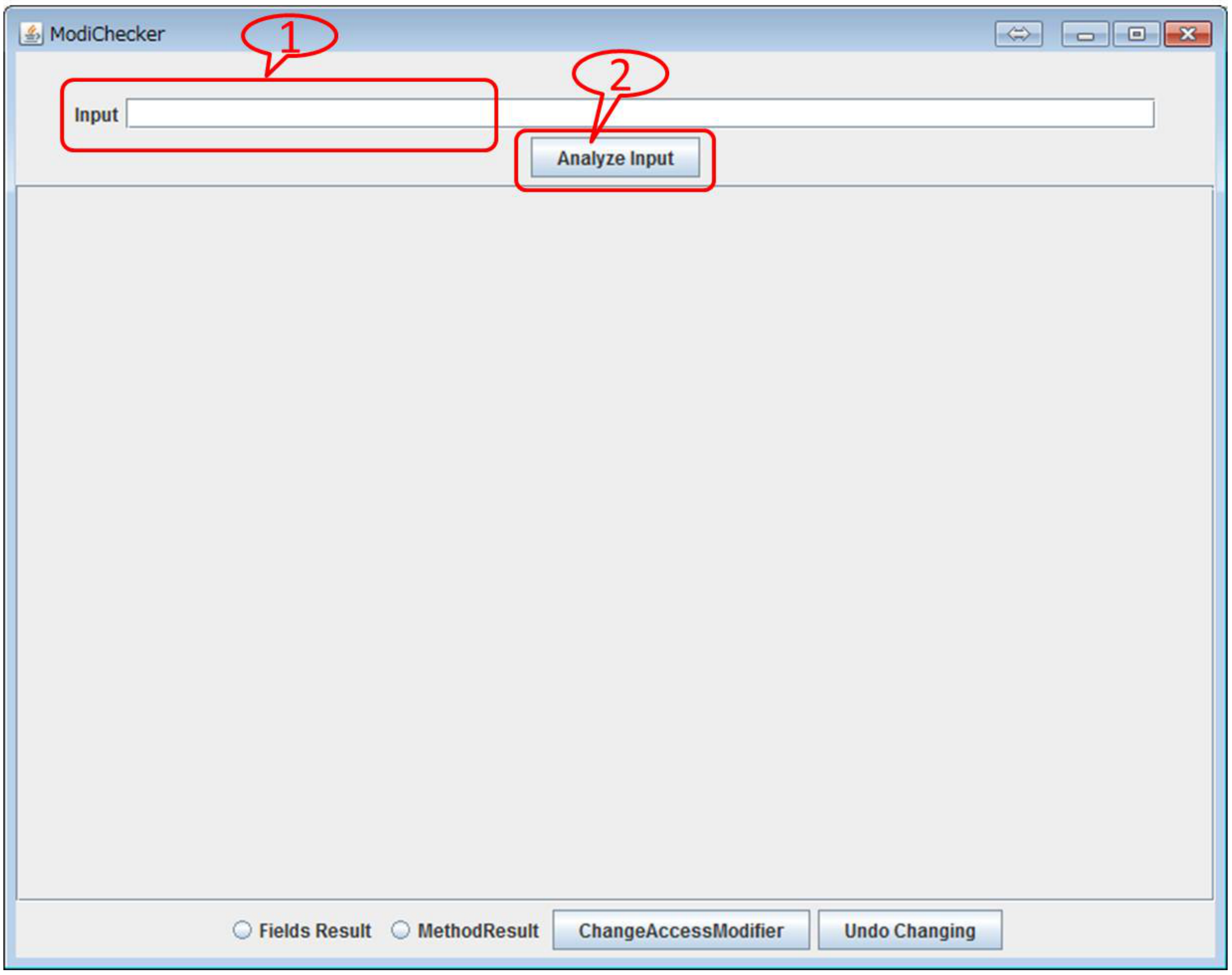


図 5: Start Window of ModiChecker

files) into folder *lib* .

Part2 is Analyze button. After designating the target input folder, users can click this button to start analyzing the target project. ModiChecker will analyze the target folder and give the result in the following file :

- field.csv and method.csv give the list of the fields and methods which have the excessive access modifier declaration compared with the actual usage of these fields/method (AE Id as pub2, pub3, pro2...). These files store the information of each field/method such as field/method's name, ownerclass' name (full qualified name), current access modifier(declared access modifier), recommended access modifier (the range which field/method is actual used). If the recommend access modifier of a field/method is `default`, ModiChecker

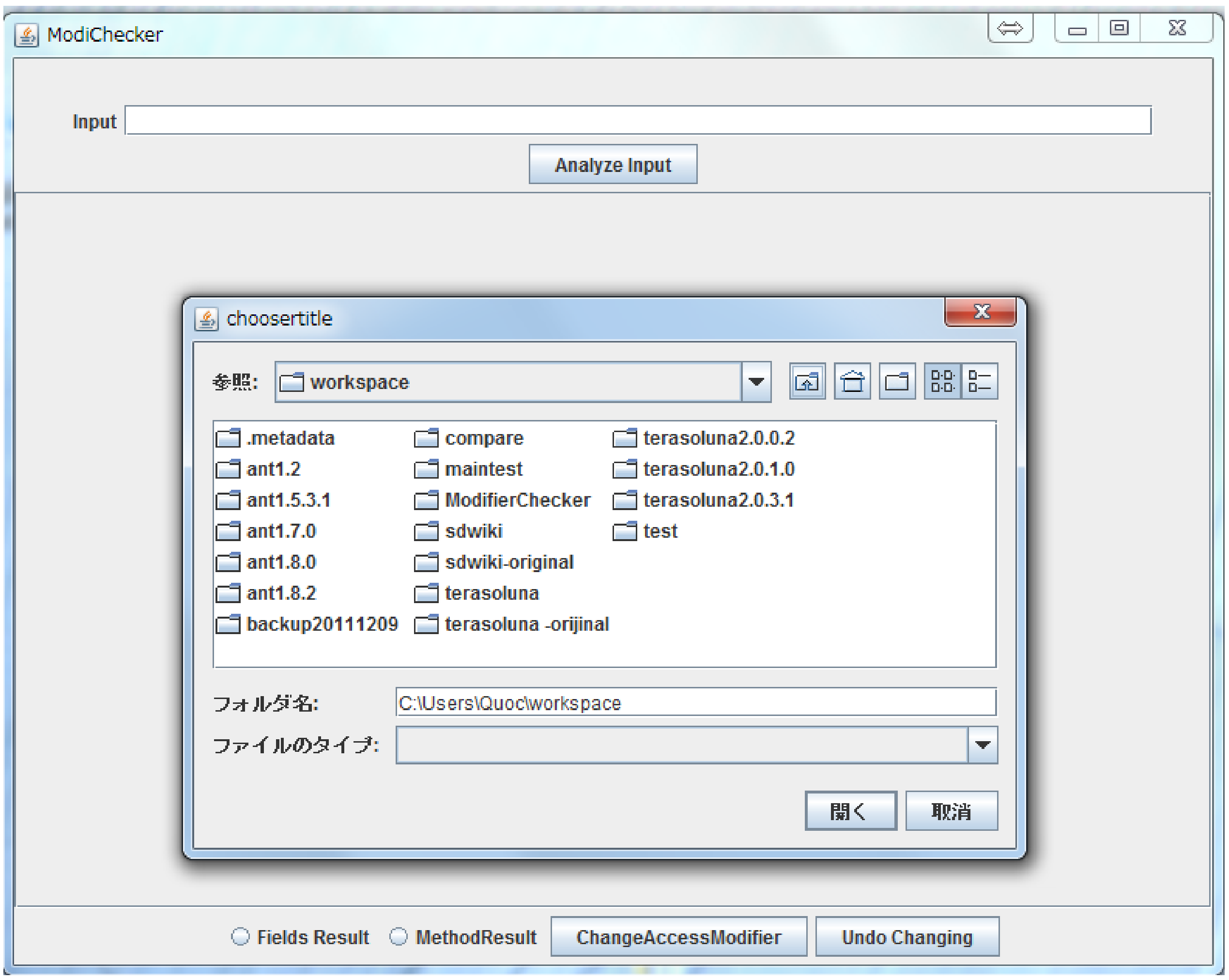
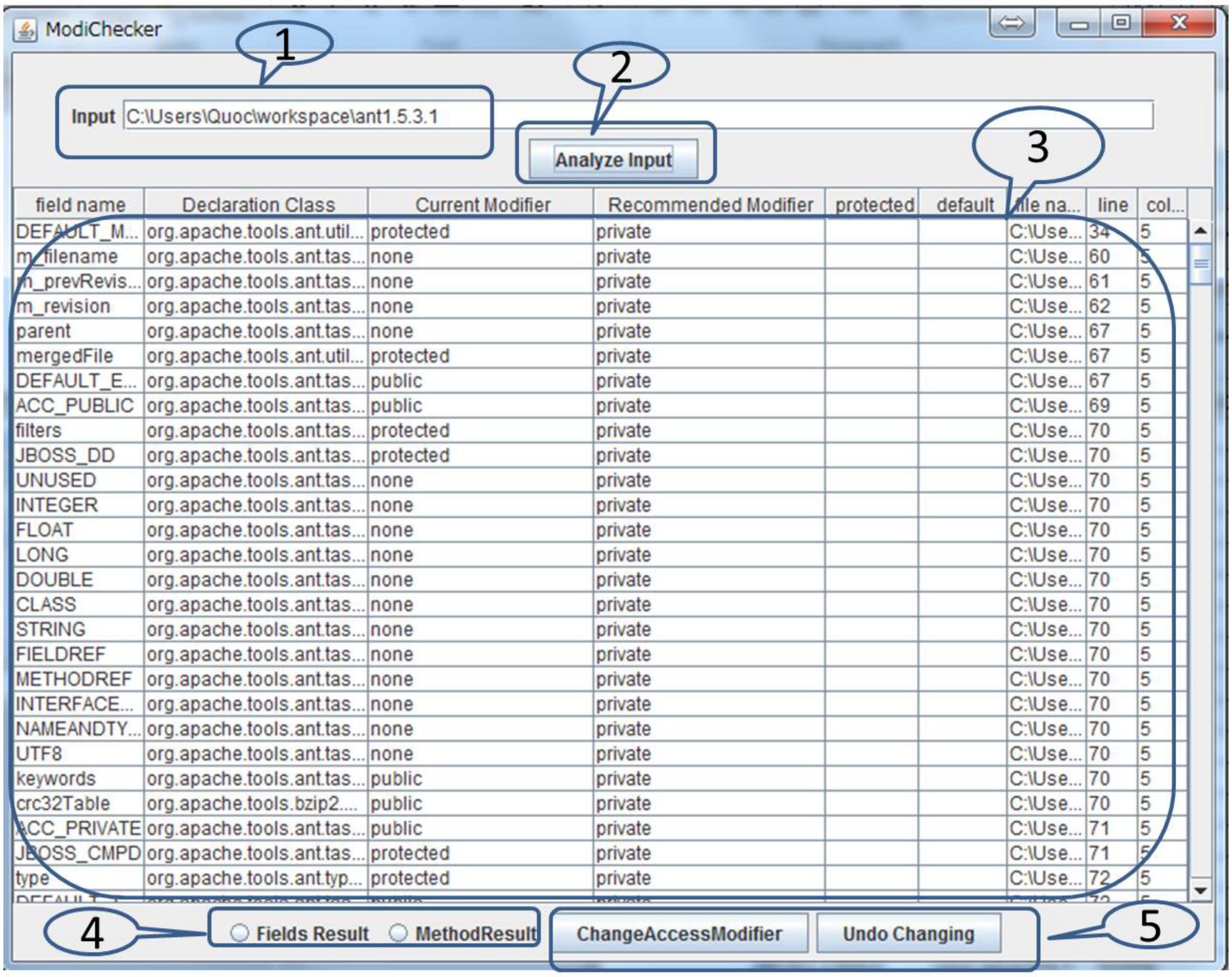


図 6: Target Project Input Chooser Window

also shows one example of a class which is in the same package with owner class and accessed to that field/method (It is a proof to show users that the access modifier of this field/method can not be set private). If the recommended access modifier is `protected`, ModiChecker shows one example of a class which is a subclass but not in the same package with owner class and accessed to that field/method (It is a proof to show users that the access modifier of this field/method can not be set `default` or `private`). Because the fields/methods which have access modifier as public will not appear in these files and the access modifier `private` is the most restrictive one, there is no need to give any proof the cases that the recommended access modifiers are `public` or `private`. Besides, these files also store the absolute path the file with the line and column where field/method is declared.

All of this information in these files will be read and show in result Table (Part 3) of the Figure 7.

- fieldnoAccess.csv and methodnoAccess.csv give the list of unused fields and methods. These files store the information of each field/method such as field/method's name, owner-class name(full qualified name),current access modifier(declared access modifier), absolute path the file with the line and column where field/method is declared
- statistic.cvs : by each case of AE, the total number of fields or methods which have the same AE Id is calculated and that number is shown in this file.



Input C:\Users\Quoc\workspace\ant1.5.3.1

field name	Declaration Class	Current Modifier	Recommended Modifier	protected	default	file na...	line	col...
DEFAULT_M...	org.apache.tools.ant.util...	protected	private			C:\Use...	34	5
m_filename	org.apache.tools.ant.tas...	none	private			C:\Use...	60	5
m_prevRevis...	org.apache.tools.ant.tas...	none	private			C:\Use...	61	5
m_revision	org.apache.tools.ant.tas...	none	private			C:\Use...	62	5
parent	org.apache.tools.ant.tas...	none	private			C:\Use...	67	5
mergedFile	org.apache.tools.ant.util...	protected	private			C:\Use...	67	5
DEFAULT_E...	org.apache.tools.ant.tas...	public	private			C:\Use...	67	5
ACC_PUBLIC	org.apache.tools.ant.tas...	public	private			C:\Use...	69	5
filters	org.apache.tools.ant.tas...	protected	private			C:\Use...	70	5
JBOSS_DD	org.apache.tools.ant.tas...	protected	private			C:\Use...	70	5
UNUSED	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
INTEGER	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
FLOAT	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
LONG	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
DOUBLE	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
CLASS	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
STRING	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
FIELDREF	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
METHODREF	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
INTERFACE...	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
NAMEANDTY...	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
UTF8	org.apache.tools.ant.tas...	none	private			C:\Use...	70	5
keywords	org.apache.tools.ant.tas...	public	private			C:\Use...	70	5
crc32Table	org.apache.tools.bzip2....	public	private			C:\Use...	70	5
ACC_PRIVATE	org.apache.tools.ant.tas...	public	private			C:\Use...	71	5
JBOSS_CMPD	org.apache.tools.ant.tas...	protected	private			C:\Use...	71	5
type	org.apache.tools.ant.typ...	protected	private			C:\Use...	72	5

図 7: GUI of ModiChecker

Part 3 is the result Table which show the result of excessive access modifier for fields or methods (the information is load from the field.csv or method.csv). User can choose to display fields or methods by clicking the radio button in part4. As shown in 7 , the table has the following columns:

- field/method name : name of field or method
- Declaration Class : owner class of the field/method. The full name(includes package name) is displayed.
- Current Modifier : current access modifier of the field/method (access modifier declared in the source code)
- Recommended Modifier : the access modifier recommended by the program(or the actual usage of field/method from other class)
- *protected* and *default* column : if the recommended access modifier is protected or default, this column show the class in the same package or subclass of the owner class of field/method which accessed that field/method.
- file name : the absolute path name of the file where field/method is declared
- *line* and *column* : the line and the column where field/method is declared

Part 4 is the two radio buttons area : "Fields Result" and " Method Result". Theses radio button is for choosing to display excessive fields or excessive methods list in the Result Table (Part 3). User click "Fields Result" to show the excessive fields in Result Table(Part 3) and click " Method Result" to show the excessive methods in Result Table(Part 3). From the starting window in Figure 5, after designate target project input folder, user can choose to view excessive fields/methods by these two radio button without clicking analyze button(It is really time-consuming to analyze one project). At that time users will be asked whether she/he wants to view the result without analyzing the target project source code.

Part 5 is "Change Access Modifier " and " Undo Changing " button : User can choose fields/methods in the Result Table (Part3) and change their current access modifier to the recommended access modifier by click the " Change Access Modifier" and they even can change the access modifier to the original one by clicking "Undo Changing" button.

4 Experiments and Discussions

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 for setting the access modifiers excessively and the reasons for not using fields and methods which were declared. The reason for the excessive/unused fields/methods can be devided in to the following categories.

  1. The unused/excessive fields/methods are sometimes set by developers' carelessness or immaturity. Sometimes the developers declared fields/methods and forgot to implement them, or sometimes they set all of the access modifier public or default (they don't set access modifier for fields/methods).
  2. The unused/excessive fields/methods are set for future use. For example, in some framework, there are some fields and methods which are designated to be accessed from outside but they are not accessed from inside that framework(unused cases) or they are only accessed from inside the owner class or owner package(excessive cases).
  3. The unused/excessive fields/methods are sometime created by other programs such as automatic code generator or code refactoring tools. In such cases, the access modifiers are mostly set public
  4. When the program uses some framework, there are some special setting that the access modifier of fields/methods must be set public (even they are only used inside the owner class or even they are not used inside the program) in order to be accessed by the used framework (for example : Java bean).

For these purpose, we conducted the study cases on the following target software products :

- MASU
- Ant 1.8.2
- jEdit 4.4.1

表 3: **Size and Running Time of each Experiment**

Program	Size	Running Time (seconds)
MASU	519 Java files/102,000 LOC	108.6
jEdit	546 Java files/109,479 LOC	150.2
Industrial Software	341 Java files/ 64455 LOC	46.9
Ant 1.8.2	838 Java files/ 103642 LOC	188.9

- Industrial software : The subject of this case study was a web-application software implemented in Java. It is 110KLOC across 296 files. This software is developed and used by NEC Japan.

While investigated into the access modifier ,we were quite interested in the Unused fields/methods and we also perform some study case about the changes of Unused fields/methods by each version of a specified software. For this purposes, we peformed study cases on 20 versions of Ant.

The target software products are the following Java programs. We did these experiments on a PC workstation with the following specification.

- OS: Windows 7 Professional 64bit
- CPU: Intel Xeon 5620(2.40 GHZ , 4 processors , 8 threads)
- Memory: 16.0 GB

Table 3 shows the size and running time of each experiment.

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 4 and Table 5.

We have found 280 fields with the excessive access modifiers and 13 fields which are unused one. Out of these excessive and unused fields, 255 excessive fields and 1 unused one were identified as automatically generated code by our hand-analysis. For the unused fields, we found 10 fields named serialVersionUID which is necessary for the classes which implement “java.io.Serializable” and they are necessary even they are not accessed by any classed. We have

表 4: Number of Detected AE ids for Fields in MASU

Declaration \ Actual Usage	Public	Protected	Default	Private	Unused
Public	16	0	3	259	1
Protected	x	0	1	14	0
Default	x	x	0	3	0
Private	x	x	x	488	12

Total number of fields: 797

Total number of fields in shaded cells : 280

Total number of unused fields : 13

表 5: Number of Detected AE ids for Methods in MASU

Declaration \ Actual Usage	Public	Protected	Default(None)	Private	Unused
Public	471	3	90	124	171
Protected	x	19	0	35	9
Default(None)	x	x	4	1	3
Private	x	x	x	1006	0

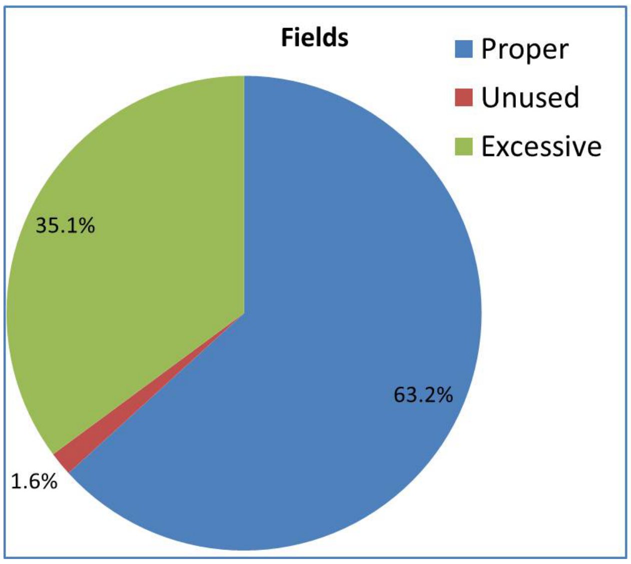
Total number of methods: 3288

Total number of methods in shaded cells : 252

Total number of unused methods : 183

interviewed the developer of MASU and asked the reason of the excessiveness and the unimplementation of other fields. 20 excessive and 1 unused fields have been found that they are intentionally set excessively for future uses. Finally, 5 excessive and 1 unused fields were found to be set by developers' carelessness and immatuarity. Those access modifiers of the excessive fields have been changed to proper ones and the unnecessary field was also deleted.

We have also found 253 methods with the excessive access modifier and 189 fields which are unused one. And by our hand-analysis, 6 excessive and 24 unused methods were found to be



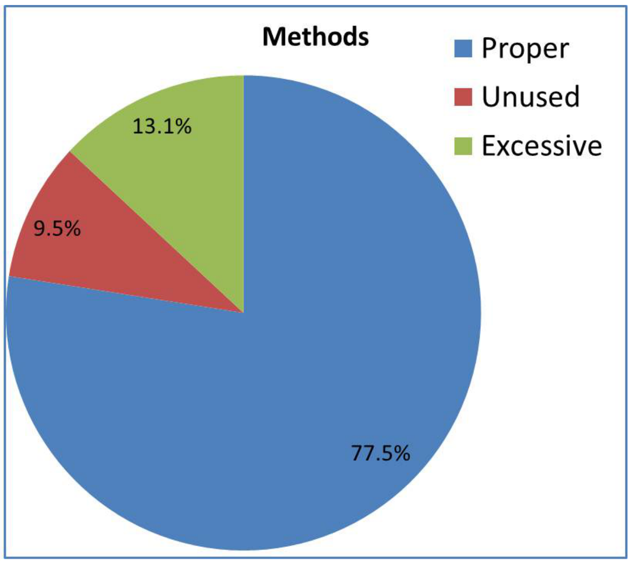


図 8: Ratio of Proper, Excessive and Unused field/method in MASU

automatically generated code. Out of those excessive and unused methods, 181 excessive methods and 158 unused one are intentionally set excessively for future uses. Finally, 66 excessive methods and 7 unused one have been identified to be set by developers' carelessness or immaturity. The access modifiers of the actually excessive methods have been changed to proper ones and the unnecessary methods were also deleted.

Looking at the ratio chart at Figure 8, the ratio of excessive fields is 35.1% the ratio of unused fields is 1.6% while ratio of excessive methods is 13.1% and the ratio of the unused methods is 9.5%. Normally 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 for future use than fields. But in this case, there are a large number of excessive fields which was created by automatic generated code and it could be the main reason for the high ratio of excessive fields(The ratio of the excessive fields is quite high compared with the ratio of the excessive methods).

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 fields and methods as shown in Table 6 and Table 7.

We have found 611 fields and 1520 methods with the excessive access modifiers. Also, 54 fields and 2395 methods were found unused. By our hand-analysis, we found that there were 8 unused fields named serialVersionUID which is necessary for the classes which implement

表 6: **Number of Detected AE ids for Fields in Ant 1.8.2**

Declaration \ Actual Usage	Public	Protected	Default	Private	Unused
Public	151	6	49	282	28
Protected	-	44	45	132	9
Default	-	-	18	58	1
Private	-	-	-	2348	16

Total number of fields: 3187

Total number of fields in shaded cells : 672

Total number of unused fields : 54

表 7: **Number of Detected AE ids for Methods in Ant 1.8.2**

Declaration \ Actual Usage	Public	Protected	Default	Private	Unused
Public	1576	100	609	454	2294
Protected	-	103	117	217	66
Default	-	-	52	23	19
Private	-	-	-	1034	16

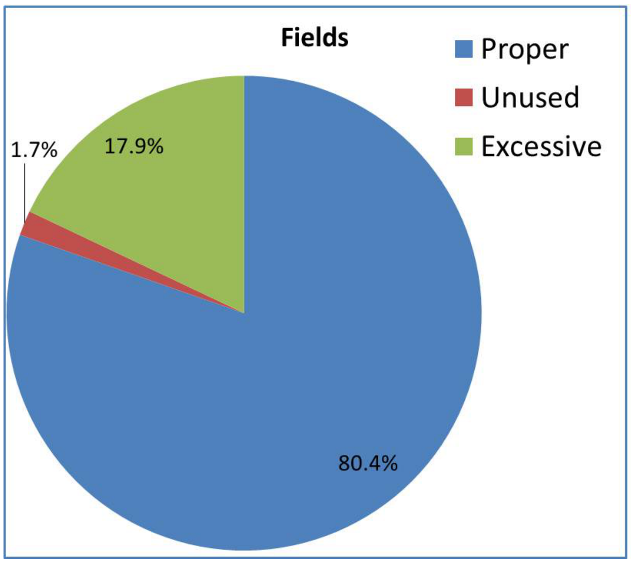
Total number of methods: 6680

Total number of methods in shaded cells : 1520

Total number of unused methods : 2395

“java.io.Serializable” and they are necessary even they are not accessed by any classed. For the unused and excessive fields and methods, were unable to find any field and method which were created by other tools(automatic code generator, refactoring tool...) or created to be access by some designated programs.

Since we could not interviewed developers to ask the reasons for the excessive and unused fields/methods, the number of fields/methods which were intentionally set for future used and the number of fields/methods which were set by developers’ immaturity and carelessness is unknown.



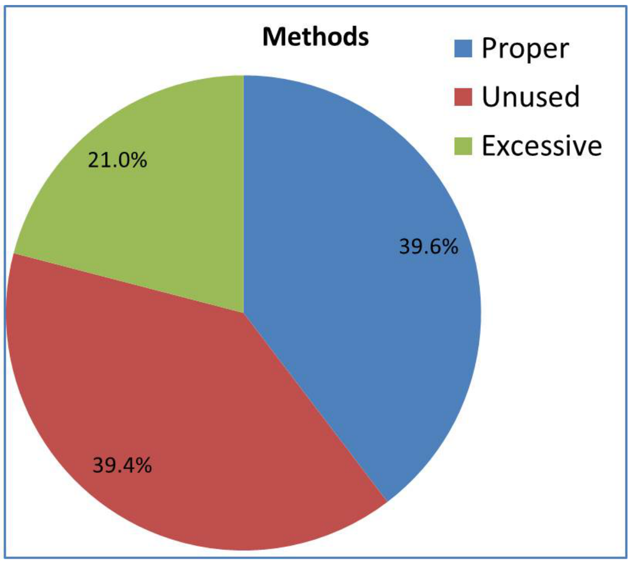


図 9: Inappropriate Access Modifier Declaration

Looking at the ratio chart at Figure 9, the ratio of excessive fields is 17.9% the ratio of unused fields is 1.7% while ratio of excessive methods is 21.0% and the ratio of the unused methods is 39.4%. 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 for future use than fields. That would be the reason why the ratio of excessive and unused methods is higher than ratio of excessive fields.

4.2.3 jEdit 4.4.1

The result of detected AE ids for fields and methods for jEdit 4.4.1 is shown in Table 8 and Table 9.

We have found 532 fields and 981 methods with the excessive access modifiers. Also, 80 fields and 654 methods were found unused.

By our hand-analysis, we found that there were 4 unused fields named serialVersionUID which is necessary for the classes which implement "java.io.Serializable" and they are necessary even they are not accessed by any classed. For the unused and excessive fields/methods, were unable to find any field/method which were created by other tools(automatic code generator, refactoring tool...) or created to be access by some designated programs.

Since we could not interviewed developers to ask the reasons for the excessive and unused fields/methods, the number of fields/methods which were intentionally set for future used and the

表 8: Number of Detected AE ids for Fields in jEdit 4.4.1

Actual Usage Declaration	Public	Protected	Default(None)	Private	Unused
Public	228(9.1%)	10	102	117	55
Protected	x	23	15	47	4
Default(None)	x	x	126	241	13
Private	x	x	x	1515	14

Total number of fields: 2510

Total number of fields in shaded cells : 532

Total number of unused fields: 86

表 9: Number of Detected AE ids for Methods in jEdit 4.4.1

Actual Usage Declaration	Public	Protected	Default(None)	Private	Unused
Public	1224	77	544	237	624
Protected	x	44	14	23	6
Default(None)	x	x	233	86	14
Private	x	x	x	874	10

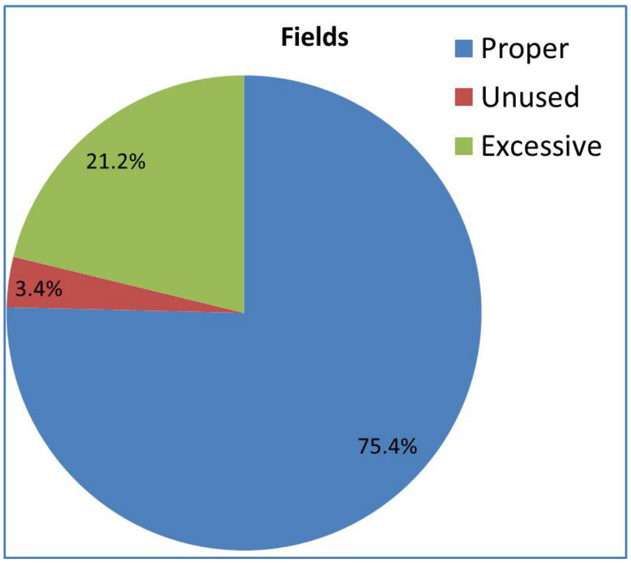
Total number of fields: 3910

Total number of fields in shaded cells : 981

Total number of unused fields: 86

number of fields/methods which were set by developers' immaturity and carelessness is unknown.

Looking at the ratio chart at Figure 10, the ratio of excessive fields is 17.9% the ratio of unused fields is 1.7% while ratio of excessive methods is 21.0% and the ratio of the unused methods is 39.4%. 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 for future use than fields. That would be the reason why the ratio of excessive and unused methods is higher than ratio of excessive fields.



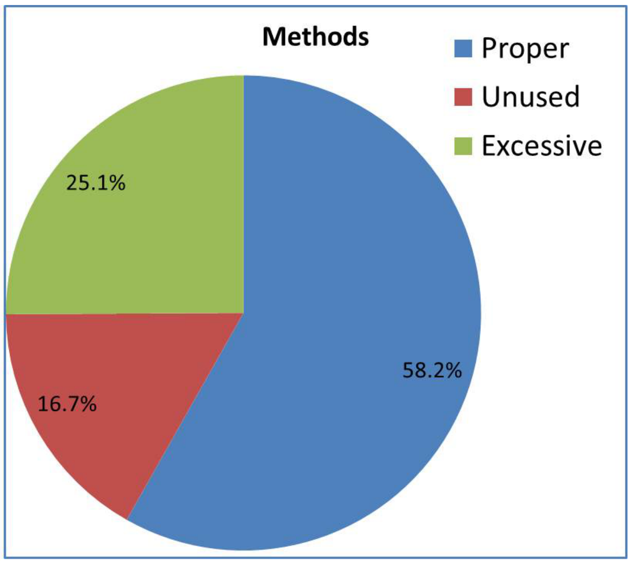


図 10: Inappropriate Access Modifier Declaration

4.2.4 Industrial Software

The result of this case study is shown in Figure 10 and Figure 11.

表 10: Number of Detected AE ids for Fields in Industrial Software

Declaration \ Actual Usage	Public	Protected	Default	Private	Unused
Public	207	0	59	936	33
Protected	x	0	9	18	0
Default	x	x	4	5	2
Private	x	x	x	1123	5

Total number of fields: 2401

Total number of fields in shaded cells : 1027

Total number of unused fields: 40

For this industrial software, we found 1027 excessive fields and 40 unused fields. By our hand analysis, we found out that out of the 40 unused fields, there are 5 fields named serialVersionUID which is necessary for the classes which implement "java.io.Serializable" and they are necessary even they are not accessed by any classed. For methods, we found 512 excessive methods and

表 11: Number of Detected AE ids for Methods in industrial software

Declaration \ Actual Usage	Public	Protected	Default	Private	Unused
Public	816	▨ 14	▨ 230	▨ 190	1005
Protected	x	13	▨ 36	▨ 48	9
Default	x	x	0	▨ 3	0
Private	x	x	x	488	4

Total number of methods: 2864

Total number of methods in shaded cells : 512

Total number of unused methods: 1018

1018 unused methods. We worked with developers and found that this software employed java Spring framework [13], and there are 299 methods which were intentionally set to be accessed by java beans. Out of these 299 methods, there are 24 excessive methods and 85 unused methods. We also found that this software use Direct Web Remoting [14] tectnique and there are also 5 methods were set public to be accessed by JavaScript. After ignoring all of the fields/methods accessed or set by other programs above, there are 35 fields and 904 methods were found unused, 1027 fields and 498 methods were found excessive. Because of the large number of the exccessive/unused fields/methods, we have not checked the reason for the exccessive/unused fields/methods yet, we only could investigated into the unused fields. Developers said that out of the 35 unused fields, there are 8 fields they intended to used in the future, 27 fields are said to be deleted because they are unnecessary code. For those 27 fields, there are 6 fields are said to contain potential bug of the program. Currently, developers are working with other unused/excessive fields/methods to figure out how many of them are really excessive/unused fields/methods and after identifying those fields/methods, developers should change/delete those fields/methods to increase the quality of the software.

4.3 Changes of unused fields/methods through 20 version of Ant

After using ModiChecker to investigate into the changes of unused fields/methods through 20 version of Ant, we got the result shown in the Table 12.

Looking at the Table 12, we can see that after each time of new main version released, there

表 12: **Changes of Unused fields/methods through 20 version of Ant**

		Unused Fields		Unused Methods	
No	Version	Number of disappeared Fields	Number of New Fields	Number of Disappear Methods	Number of New Methods
1	Ant 1.2				
2	Ant 1.3	2	8	46	256
3	Ant 1.4	2	19	16	373
4	Ant 1.4.1	0	0	1	5
5	Ant 1.5	15	48	119	508
6	Ant 1.5.1	4	0	5	8
7	Ant 1.5.2	2	1	16	16
8	Ant 1.5.3.1	0	0	1	2
9	Ant 1.5.4	1	0	0	0
10	Ant 1.6.0	35	7	57	256
11	Ant 1.6.1	0	0	8	44
12	Ant 1.6.2	3	5	8	74
13	Ant 1.6.3	3	3	45	84
14	Ant 1.6.4	0	0	0	0
15	Ant 1.6.5	0	0	0	0
16	Ant 1.7	14	17	167	375
17	Ant 1.7.1	5	12	11	45
18	Ant 1.8.0	6	8	148	268
19	Ant1.8.1	1	5	5	13
20	Ant 1.8.2	1	2	2	23

was a big change in the number of unused fields/methods compare with the changes in one main version. For example, from version 1.4.1 to version 1.5, there are 15 unused fields, 119 unused methods disappeared from version 1.4.1 and there are 48 new unused fields, 508 unused methods added to version1.5 while from version 1.4 to version 1.4.1 there are really few unused fields, methods disappear from version 1.4 and also there are few unused fields,methods added to version 1.4.1. To investigate into detailed reasons for the changes of unused fields/methods, we need to take a look at the source code but because of the large number of change and the limitation of time, we haven’t finished this work and it should be interesting research topic in the future.

4.4 Discussions

To validate the analysis result of ModiChecker, we have changed all the excessive access modifiers of above three programs to suggested access modifiers. Because of the large amount number of unused fields/methods, We also could try to delete some unused fields/methods (We have not develop deleting unused fields/methods function for your tool so we have to deleted them by hand). All the modified programs have been compiled and executed without any error. This indicates that the output report for of ModiChecker is proper one in the sense that the reported excessive access modifiers can be changed to more restrictive access modifiers and unused fields/methods can be deleted without causing any error.

As mentioned before, our tool gives the developer the AE analysis result for each field/method in the current target program, but it still can not make sure that some of them were intentionally set for future use. Only designer and developer can identify that those fields/methods are really unused/excessive or not. Thus, we highly suggest user use our tool to take a look into detailed fields/methods and change the access modifier of fields/methods which are thought to be really exccessive. The tool for deleting the unused fields/methods will be developed in the future.

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

These metrics would indicate bad smell of program such as immaturity of design and implementation, maintainability, security vulnerability, and so on. We need further experiments for the validation of effectiveness of such metrics.

The idea of using access modifier metrics would be related to our previous work [6]. 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.

To recognize intentional AE fields/methods for future use without interviewing developers is not easy. As a simple estimation method, we propose the following approach which can figure out some part of the excessive fields/methods for future use, using test files associated with the target program.

The first step is checking the source files without test files and we get the result of ModiChecker for the target program itself. Then we add the test files and check them together without counting the fields/methods in test files. The excessive fields/methods found in the first step but not in the second step could be the fields/methods for future use, since they were actually accessed by the objects of test files. The test designer anticipated that those fields/methods should be access from outside the program in the future.

In this experiment we have not counted for the fields/methods which are not accessed by any objects (we would like to call them Unused fields/methods). The reason of Unused fields/methods might be developers' carelessness and intentional future use. Investigating into Unused fields/methods would be an interesting future research topic.

5 Conclusions

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 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.

Acknowledgments

First and foremost, I would like to thank Professor Katsuro Inoue for giving me the opportunity to work with him. He provided advices and important direction to my thesis work. Special thanks to Associate Professor Makoto Matsushita in Department of Computer Science, Osaka University and Assistant Professor Yoshiki Higo for their valuable comments and guidance especially in technical matters. I would like to express my gratitude to Assistant Professor Norihiro Yoshida in Nara Institute of Science and Technology for his comments, criticism and advice on this thesis. I am very grateful to Mr Kobori in NTT Data company and staffs in NEC company for giving me alot of ideas and technique advices on this thesis. I would like to express my gratitude to all members of Department of Computer Science for their guidance. Thanks are also due to many friends in Department of Computer Science, especially students in Inoue Laboratory.

参考文献

[1] K. Khor , Nathaniel L. Chavis , S.M. Lovett , and D. C. White. “Welcome to IBM Smalltalk Tutorial” , 1995

[2] G. Booch, R.A. Maksimchuk, M.W. Engel, B.J. Young, J. Conallen and K.A. Houston, “Object-Oriented Analysis and Design with Applications”, Addision Wesly, 2007.

[3] J. Viega, G. McGraw, T. Mutdosch and E. Felten, “Statically Scanning Java Code: Finding Security Vulnerabilities”, IEEE software, Vol.17 No. 5 pp. 68-74, Sep/Oct 2000.

[4] 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.

[5] D. Evans, and D. Larochells, “Improving Security Using Extensible Lightweight Static Analysis”, IEEE software, vol.19, No.1, pp. 42-51, Jan/Feb 2002.

[6] 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.

[7] 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).

[8] 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.

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

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

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

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

[13] Spring doc, http://static.springsource.org/spring/docs/1.2.9/reference/beans.html

[14] Direct Web Remoting, http://directwebremoting.org/dwr/index.html