Evolution Analysis for Accessibility Excessiveness in Java

Kazuo Kobori
NTT DATA Corporation
3-3-3, Toyosu, Koto
Tokyo 135-6033, Japan

Makoto Matsushita
Osaka University
1-5 Yamadaoka, Suita
Osaka 565-0871, Japan

Katsuro Inoue
Osaka University
1-5 Yamadaoka, Suita
Osaka 565-0871, Japan

***Abstract*—In Java 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 to improve program quality and to reduce potential vulnerability. In our previous work, we presented a static analysis method named Accessibility Excessiveness (AE) detection for each field and method in Java program. We have also developed an AE analysis tool named ModiChecker that analyzes each field and method of the input Java programs, and reports their excessiveness. In this paper, we have applied ModiChecker to several OSS repositories to investigate the evolution of AE over versions, and identified transition of AE status and the difference in the amount of AE change between major version releases and minor ones. Also we propose when to evaluate source code with AE analysis.**

I. Introduction

Access modifiers in Java [27] are access level directions for classes, constructors, methods, and fields. There are four access levels in Java:

- `public` is visible to all classes everywhere.
- `default` is visible only within its own package.
- `private` can only be accessed in its own class.
- `protected` can only be accessed within its own package and, in addition, by a subclass of its class in another package.

Proper use of access modifiers and controlling visibility are essential to well-encapsulated Java programming [2], [3].

To realize good encapsulation in Java programming, we have to carefully choose appropriate access modifiers for fields and methods in a class, which are accessed by many other objects. However, inexperienced developers might simply set all of the access modifiers `public` indiscriminately.

For example, Fig. 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` has an initialization process for `Method B`. It means that `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 then the access modifier of `Method B` should be `private`. However, a novice developer might set such access modifier `public` without thinking seriously. In a meanwhile, other developer would want to use `Method B` and he/she would 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`.

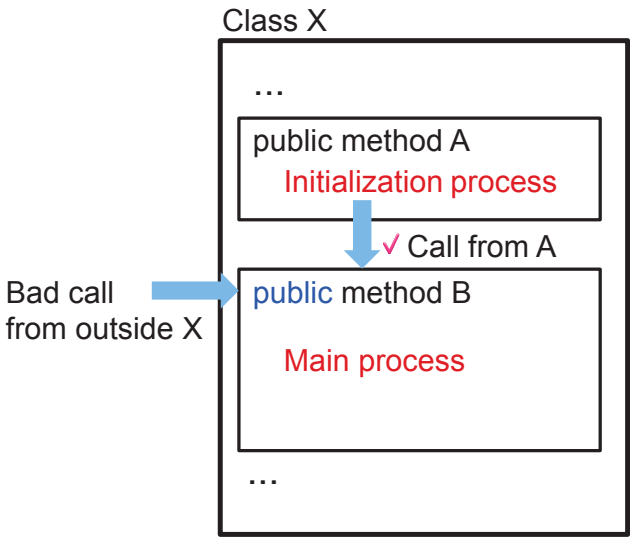

Fig. 1. Inappropriate Access Modifier Declaration

In this example, the access modifier of `Method B` is declared as `public`, but the current program calls `Method B` only from a `private` method (`Method A` in this case), and then the access modifier of `Method B` should be `private`.

A discrepancy between the declared accessibility and actual usage of a field or method is called *Accessibility Excessiveness (AE)* here, and AE can be a cause of program faults due to unintended use of the fields or methods.

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 Issues: An AE would cause unwilling access to a field or method which should not be accessed by other objects in a later development or maintenance phases as shown in the example. This is an issue of design and development processes from the view point of encapsulation [3]. This problem shows the immaturity and carelessness of the designer and developer.
2) Maintenance Issues: Sometimes the developer intentionally set field or method excessive for future use or for the purpose of being called from other programs. It is

not easy to distinguish whether AE is intentionally set by the developer or it is a case of Issue 1, so that the maintenance of such program is complicated and is not straightforward.

3) Security Vulnerability Issues: 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 [21].

In this paper, we will discuss on an AE analysis method mostly focusing on its application to Issue 1), and also mention Issue 2) in our tool implementation. Issue 3) will be a further research topic. If the maintenance issue would be resolved, we could identify unintentionally set fields and methods easily, and fix them with appropriate accessibility modifiers. That might improve the quality of the software.

We have developed an AE analysis tool *ModiChecker* [12], which takes a Java program as input, then analyzes and reports the excessiveness of each access modifier declared at fields and methods. Also it can change the detected access modifiers to proper ones under the consultation of the developer.

ModiChecker is based on a static program analysis framework *MASU* [8], [16], [31], which allows a flexible composition of various analysis tools very easily.

Using ModiChecker, we have analyzed several open source software (OSS) systems to investigate the transition of AE state from a version to a next version, and the evolution of AE analysis results over multiple versions of these systems. As a result, we obtained some observation for the relation of evolution and AE.

The contributions of this paper are as follows.

(i) We propose an AE classification model for the fields and methods of Java program. Based on this model, we also define AE metrics, which would be an indicator of bad smell of the target program.

(ii) Empirical results of AE analyses for various OSS systems have been presented, including transition patterns and evolution pattern of AE metrics.

In the following, we will show AE classification model in Section 2. Section 3 will describe an AE analysis method and ModiChecker with Java analysis framework MASU. In Section 4, we will show our experiments of using ModiChecker with several research questions. Section 5 will discuss our approach and results associated with related works. Section 6 will conclude our discussions with a few future works.

II. Accessibility Excessiveness Classification Model

Table I shows *Accessibility Excessiveness Classification* (AE classification), which lists all the combinations of the declarations and actual usages. The vertical column shows the declaration of an access modifier for a field or method in the source code. The horizontal row shows its actual usage from other objects. Each element in AE classification is an *Accessibility Excessive 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 the same class, the AE id is pub-pri meaning that it could be set to `private`.

TABLE I
Accessibility Excessiveness Classification

Declaration \ Actual Usage	Public	Protected	Default	Private	No Access
Public	pub-pub	**pub-pro**	**pub-def**	**pub-pri**	pub-na
Protected	–	pro-pro	**pro-def**	**pro-pri**	pro-na
Default	–	–	def-def	**def-pri**	def-na
Private	–	–	–	pri-pri	pri-na

We also define AE ids for special cases. AE ids pub-pub, pro-pro, def-def, and pri-pri are the cases that there is no discrepancy between the declaration and actual usage, so those are proper cases for quality programming. AE ids pub-na, pro-na, def-na, and pri-na indicate that those fields or methods are declared but they are never used. These two cases are technically not AE in the sense those are not ordinary discrepancy between the declaration and actual usage, but we include them for better understanding of access modifiers usage. In total, we will use 14 AE ids here. The cases with "–" means that those are detected as error at the compilation time, so that those are not executable program and they are out of the scope of the AE analysis.

The purpose of the AE analysis is to determine an AE id for each field and method in the input program.

Also, we are interested in the statistic measures of AE ids for the input program, which would be important clues of program quality. Here, we define *AE field metric* and *AE method metric* for a program. The AE field metric is a set of the metric values for all 14 AE ids: $\{|pub-pub|, |pub-pro|, ..., |def-na|, |pri-pri|, |pri-na|\}$, where $|AEid|$ means the number of the fields having that AE id in the input program. In the same way the AE method metric is also defined. We may simply call *AE metric* to mean either AE field metric or AE method metric.

III. AE Analysis Tool ModiChecker

A. Approach to AE Analysis

To perform the AE analysis, we need to know the declaration of the access modifiers of each field or method of the input program. This is easily done by parsing the program. Also, we have to investigate into the actual usage of each field or method. For this work, we employ a static source-code analysis, which identifies other classes that may possibly access the target field or method. For these purposes, we have used a Java program analysis framework MASU [8], [16], [31].

MASU has been originally designed to implement pluggable multi-linguistic metric infrastructure, but it is very useful as a Java program analysis framework. MASU transforms the input Java program into a language-independent Abstract Syntax Tree (AST). For this AST, various analyses are performed on class level, method level, and variable level. Caller-callee relations and variable usage are obtained by these analyses.

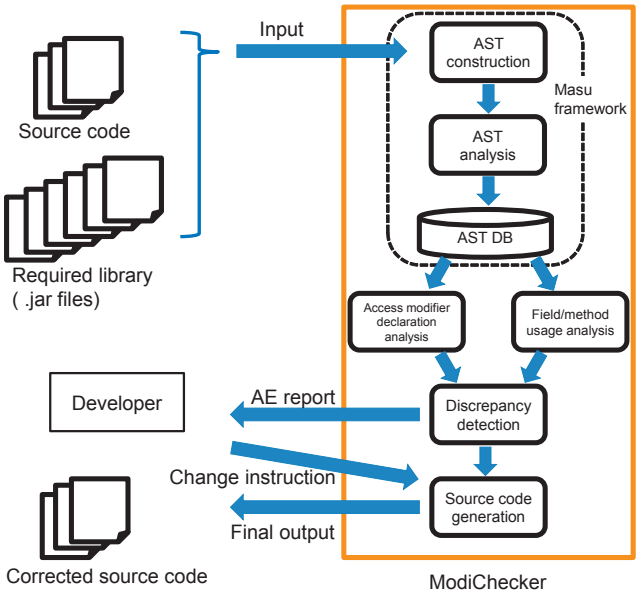

Fig. 2. Architecture of ModiChecker

B. Overview of ModiChecker Architecture

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

After analyzing the AST, we get the access modifier declaration and also usage of each field and method in AST database. From the AST database, we can easily obtain the declaration of the access modifier for each field or method. Also, actual usage of each field and method, i.e., information of who may access each field or method, is also obtained from the AST database.

By comparing the declaration and actual usage of the field and method, ModiChecker reports AE information to the developer.

ModiChecker is a system with 521 source files and 102,250 LOC in Java, developed based on MASU framework as mentioned above. MASU framework itself accounts for 519 source files and 102,000 LOC in Java (41,000 LOC are automatically generated code by ANTLR [24]) .

C. Handling Special Cases

ModiChecker treats some special cases as follows.

Abstract Class and Interface: First, in the case of the abstract method declared in abstract class and interface, they are not called from any instances. ModiChecker detects such an abstract method and an interface, and reports No Access for them. Second, in the case of the non-abstract method defined in an abstract class, ModiChecker reports AE in the same way as a method defined in a non-abstract class.

Overriding Methods: In the case of the method overriding another method, that overriding method in a subclass must have an access modifier with an equal or more permissive level

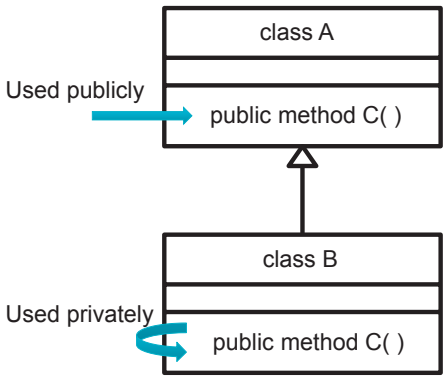

Fig. 3. Access Modifier of Overriding Method

to the access modifier of the overridden method. ModiChecker detects such an overriding method and reports AE between the access modifier of the overridden method and its actual usage.

For example, in Fig. 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 recommend `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.

IV. Experiments

In our previous work [12], we have applied ModiChecker to a single version of source code, and found some attributes of AE including what kind of AEs are found, how many AEs are in the code itself. In this paper we would like to step forward to analyze the evolution of source code, see when is the best timing for developers to check AEs.

A. Research Questions and Targets

We have conducted several experiments with some open source software (OSS) systems. The objectives of these experiments are to know the following research questions.

RQ1: Are the AE fields and methods are fixed in the next versions? What is the AE state transition in the versions?

RQ2: Do the AE metric values change over OSS evolution? If so, what is the evolutional pattern?

We have performed these experiments on a PC workstation with dual Intel Xeon 5160(3.00 GHz) processors and 8 GB memory under Windows 7 Enterprise.

B. RQ1: Transition of AE State

We are interested in knowing if AE state of a field or method is corrected immediately in the next version or not. Life cycle of AE is an intriguing research topic.

TABLE II
SUBJECT SOURCEFORGE PROJECTS

Name	Versions Investigated	Total Versions	Number of Transitions (Fields)	Number of Transitions (Methods)	Years
Apache Ant	1.1 – 1.8.4	23	80920	185156	2003 – 2012
Areca Backup	5.0 – 7.2.17	66	131170	258748	2007 – 2012
ArgoUML	0.10.1 – 0.34	19	85038	252130	2002 – 2011
FreeMind	0.0.2 – 0.9.0	16	8676	30048	2000 – 2011
JDT Core	2.0.1 – 3.7	16	134374	240726	2002 – 2012
jEdit	3.0 – 4.5.2	21	50626	99008	2000 – 2012
Apache Struts	1.0.2 – 2.3.7	34	104218	274271	2002 – 2012

TABLE III
STATE TRANSITION RATIO: FIELDS (%)

		Ant	Areca	ArgoUML	FreeMind	JDT Core	jEdit	Struts
a	Proper → Proper	0.02	0.01	0.03	0.12	0.02	0.02	0.01
b	**AE → Proper**	**0.16**	**0.01**	**0.42**	**0.31**	**0.30**	**0.15**	**0.28**
c	No Access → Proper	0.02	0.01	0.05	0.07	0.07	0.05	0.01
d	Proper → AE	0.04	0.04	0.04	0.38	0.22	0.09	0.14
e	AE → AE	0.07	0.01	0.04	0.17	0.12	0.02	0.07
f	No Access → AE	0.00	0.01	0.02	0.01	0.01	0.02	0.01
g	Proper → No Access	0.03	0.02	0.19	0.21	0.06	0.07	0.05
h	AE → No Access	0.03	0.01	0.07	0.02	0.05	0.03	0.03
i	No Access → No Access	0.00	0.01	0.05	0.00	0.00	0.00	0.00
j	**No Existence → Proper**	**6.41**	**1.45**	**6.84**	**22.59**	**4.08**	**6.16**	**5.57**
k	No Existence → AE	1.41	0.55	1.56	7.13	1.79	1.78	2.52
l	**No Existence → No Access**	**0.21**	**0.07**	**1.38**	**3.27**	**0.24**	**0.81**	**0.59**
m	Proper → No existence	2.03	0.66	4.05	7.28	0.80	3.48	2.67
n	**AE → No existence**	**0.46**	**0.27**	**2.20**	**2.80**	**0.78**	**1.07**	**1.14**
o	No Access → No Existence	0.13	0.03	1.36	2.04	0.09	0.58	0.22
p	**Not changed (Proper)**	**71.42**	**65.50**	**58.72**	**35.85**	**64.98**	**63.67**	**50.34**
q	**Not changed (AE)**	**15.28**	**26.74**	**12.26**	**12.99**	**22.72**	**16.22**	**29.00**
r	**Not changed (No Access)**	**2.28**	**4.62**	**10.72**	**4.75**	**3.66**	**5.79**	**7.34**

TABLE IV
STATE TRANSITION RATIO: METHODS (%)

		Ant	Areca	ArgoUML	FreeMind	JDT Core	jEdit	Struts
a	Proper → Proper	0.03	0.00	0.03	0.12	0.02	0.02	0.00
b	**AE → Proper**	**0.10**	**0.03**	**0.12**	**0.26**	**0.22**	**0.16**	**0.07**
c	No Access → Proper	0.13	0.07	0.26	0.36	0.24	0.13	0.09
d	Proper → AE	0.05	0.02	0.08	0.13	0.13	0.15	0.04
e	AE → AE	0.06	0.00	0.05	0.09	0.12	0.03	0.03
f	No Access → AE	0.08	0.01	0.03	0.04	0.03	0.03	0.13
g	Proper → No Access	0.07	0.05	0.25	0.34	0.13	0.19	0.05
h	AE → No Access	0.05	0.00	0.06	0.03	0.05	0.05	0.01
i	No Access → No Access	0.01	0.00	0.02	0.02	0.01	0.00	0.00
j	**No Existence → Proper**	**2.31**	**1.10**	**3.63**	**12.34**	**2.48**	**3.85**	**1.96**
k	No Existence → AE	1.08	0.28	1.06	2.10	0.78	1.38	1.52
l	**No Existence → No Access**	**4.44**	**1.10**	**5.48**	**17.41**	**2.71**	**3.32**	**5.73**
m	Proper → No Existence	0.54	0.63	1.88	5.41	1.04	2.12	0.73
n	**AE → No Existence**	**0.28**	**0.19**	**0.84**	**0.68**	**0.46**	**0.86**	**1.08**
o	No Access → No Existence	0.95	0.71	3.44	9.72	1.27	1.85	2.99
p	**Not changed (Proper)**	**26.46**	**44.30**	**28.44**	**23.29**	**38.83**	**38.58**	**19.16**
q	**Not changed (AE)**	**12.89**	**11.88**	**8.92**	**3.10**	**11.22**	**14.02**	**11.07**
r	**Not changed (No Access)**	**50.47**	**39.64**	**45.42**	**24.55**	**40.27**	**33.24**	**55.34**

We have investigated the repositories of seven OSS projects written in Java obtained from SourceForge as shown in Table II. Those were selected due to their long development histories with sufficient versions.

Transition of a field and method with respect to the AE state changes between two consecutive versions is presented in Figure 4. In this transition there are four states. Proper means that the accessibility of the target field or method is proper. AE indicates that the target is an AE field or method, and No Access shows the target is not accessed by anyone. No Existence means that the target does not exist at the previous or next version. This state is employed to show the transitions

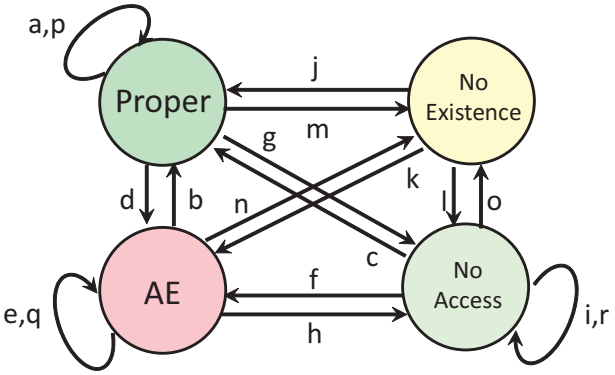

Fig. 4. Transition Diagram of Access Modifiers between Two Versions

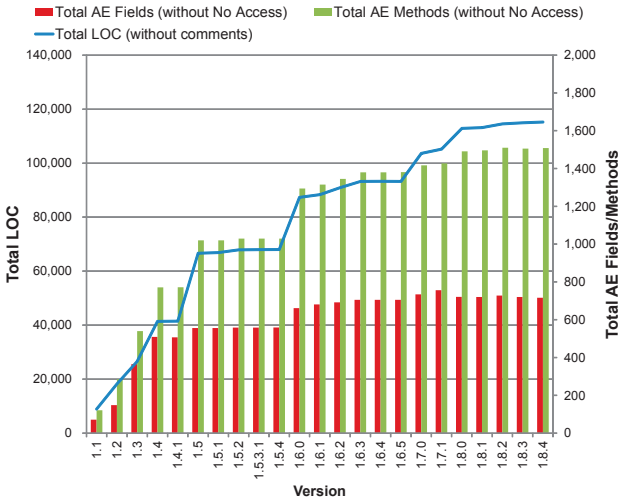

Fig. 5. Grow of AE Fields and Methods over Total LOC

when a field or method is newly added or deleted in the next version.

All possible transitions are labeled from a–r. a and p are both self transitions, but they are different in the sense that at a some change for the access modifier was made, and at p no change was performed. In the same manner, e and i are the cases of some changes, and q and r are those of no change.

Table III and Table IV show the state transition ratios of the fields and methods in each OSS system.

As we can see in these tables, one of the high ratio transition group is self transitions without access modifier change (i.e., p, q, and r). This would mean that the access modifiers are rarely revised in the next version.

Another high ratio is the transition from "No Existence to Proper (j)" and "No Existence to No access (l)." For the field AE transition (Table III), we can see that j is generally higher than l. This would mean that most of newly added fields have proper access modifiers rather than just created. On the other hand, for the method AE transition (Table IV), l is generally higher than j, meaning that many methods are created without preparing their use.

We can observe that the transition from "AE to No Existence (n)" shares higher ration than the transitions from "AE to Proper (b)" in both cases of field and method. This would suggest that the developers tend to remove or completely change the fields and methods, rather than to correct AE access modifiers.

Answer to RQ1: Changing the access modifiers for the fields and methods are generally infrequent. They are fairly stable even in the case of AE.

C. RQ2: Evolution of AE Metric Values

In previous section, we have observed that the AE state are fairly stable once they are created in a version of source code. Here, we have quantitatively analyzed the creation of AE fields and methods over the evolution history.

We have examined 22 versions of Ant [23] from version 1.1 to 1.8.4, including 7 major version releases and 15 minor version releases. We can identify a major version release by the second version number change for a version number composition "1.major.minor". Other changes are minor version releases.

Fig. 5 shows the growth of the total number of LOC of each version without counting comment lines, associated with AE field and method metric values of each version. A red bar represents the total number of AE fields, and a green bar indicates the total number of AE methods, both of which do not include No Access or Proper AE ids.

We see there is a strong correlation between the total LOC and AE fields/methods. The correlation coefficients are 0.93 for LOC and AE fields and 0.98 for LOC and AE methods. These indicate strong correlation between them. Also, as we have discussed in previous subsection, AE methods exist more than AE fields in all versions.

We have investigated the changes of AE metric values of two consecutive versions. Fig. 6 shows the changes of the number of same AE ids in two consecutive versions of Ant. Here, a change means that the the absolute value of the difference of the number of AE fields in two versions. In most cases, the newer version adds new AE fields, so the number of the newer version is larger than that of the older version. There are cases of small decreases in the newer versions, but those are rare ones.

As we can see in this figure, when a major version release happens, many changes on AE fields occur, most of which are addition of new AE fields. In this case, many actual AE ids such as pro-pri are added in the newer major versions. It seems that there is significant distinction of AE id changes between major and minor version release cases.

Each AE id change has been examined by Mann-Whitney U test to assess the significance of the difference between major and minor version release cases. With 5% significance level, the difference was confirmed for each AE id (pub-pro, pub-def, pub-pri, pro-def, pro-pri, and def-pri) and each No Access.

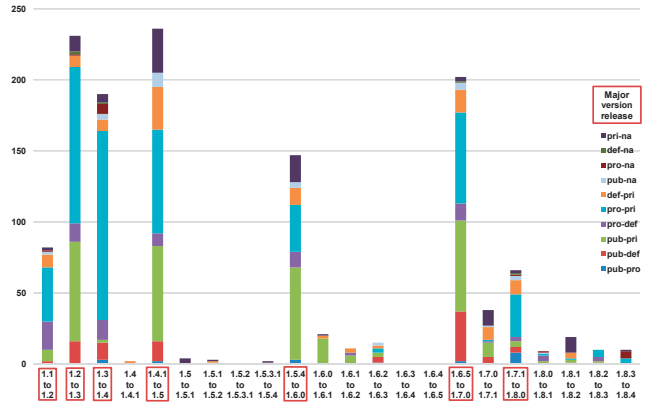

Fig. 6. Change of # of AE Fields in Two Consecutive Versions of Ant

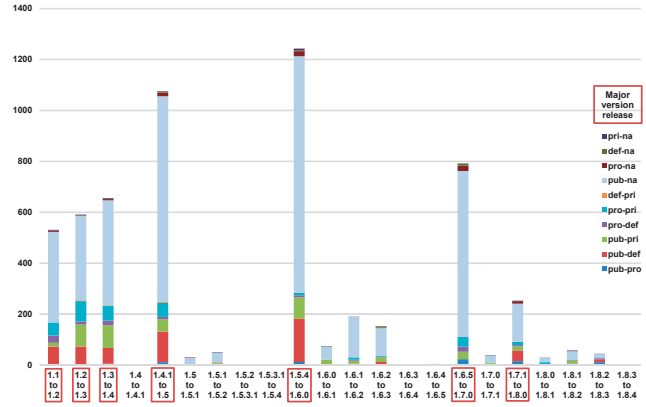

Fig. 7. Change of # of AE Methods in Two Consecutive Versions of Ant

In the same manner, we have analyzed the evolution of AE method metric. Fig. 7 shows the changes of the number of AE fields in two consecutive versions of Ant. Here, the changes are mostly the addition of the new AE methods, with rare cases of very small decreases in the newer versions.

The major version releases contain many No Access AE ids, most of which are newly added pub-na. The major version releases seem to have much more changes than the minor versions. To confirm this, we have also conducted U test for each AE id with 5% significance level. The differences of the major and minor version releases were confirmed for each AE except for pub-pri, def-pri and def-na, due to their small numbers of subjects.

The changes of AE fields and methods might be affected by the total changes of source code in two consecutive versions. The correlation coefficient between the source code changes and the AE field changes was 0.85, and that between the source code changes and the AE method changes was 0.69 (including No Access AE ids). There values are not so high, suggesting that the changes of AE metric values do not simply reflect the source code changes, but they could be used as independent indicators of the characteristic of major and minor version releases. We need further investigation of the AE metric change by tracing each AE field and method, and by analyzing other system evolution to validate this observation.

Answer to RQ2: The AE metric values generally increase along with the growth of the total system size. Also, the numbers of some AE ids for fields and methods are largely changed at the major version releases, compared to the minor version releases.

V. Discussions and Related Work

A. Validity of AE Analysis and ModiChecker Approach

ModiChecker effectively detects and reports issues on the accessibility excessiveness for the input source programs, which cannot be detected by ordinary static analysis tools such as FindBugs [29] or Jlint [30].

A use-case of ModiChecker is that the developer performs the AE analysis by herself to check the problems on her code. The developer would easily recognize the AE fields and methods, which can be accessed from external programs, and also which will be used in the future. So, she can efficiently identify the AE fields and methods caused by the bad design and coding, and she can fix those with the correction feature of ModiChecker.

Another use-case of ModiChecker is to validate program quality at the validation time of system development or maintenance. Program managers and quality managers would run this tool to see the issues of the access modifiers, and they would explore the overall program quality with the AE metric.

As we have seen in our experiments, we would find many AE fields and methods. This might be a bad smell of program quality, but identifying the reasons of AE is not easy for others. We need to device a method of tracing AE fields and methods, and collecting data of future usage of AE fields and methods, so that some mechanism of automatic detection of such AE might be found.

ModiChecker is sufficiently fast to do these use-cases, so that the users can efficiently validate and correct their programs with the interactive correction feature.

We have used a source-code level analysis approach using MASU here. We can think another approach of the bytecode level analysis. This approach would be easily implemented, but tracing back from the bytecode information to the source code level would not straightforward.

B. Empirical Analysis Results of OSS Systems

As shown in previous section, we have found many AE fields and methods in the target systems we have used in our experiments. The AE metric values generally increase along the increase of the system sizes. It seems that the developers add many new AE fields and methods along the version evolution without doing serious refactoring to fix the AE fields and methods.

Life cycle of AE fields and methods in the system evolution would be an interesting research topic. As a partial answer for this, we have investigated the transition of AE states in RQ1. Further interest includes the issues such as what is the life time and who fixed the AE status. To analyze these, we need

to trace each AE field and method over version history. This is an important future work of this research.

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

The first step is checking the source files without test programs and we get the result of ModiChecker for the target program itself. Then we add the test programs and check them together without counting the fields/methods in test programs. The AE 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 programs. The test designer anticipated that those fields/methods should be access from outside the program in the future.

C. Threat to Validity

In our approach, we have used a static analysis method for Java source programs. This approach has a fundamental issue such that reflection code in Java cannot be analyzed correctly. This is not only our own issue but other static analysis approaches, including bytecode level static analysis, contain this issue. However, even with such limitation, the results obtained from ModiChecker and its AE analysis are very useful information to the program developers and maintainers.

We would think of dynamic analysis approach for the AE analysis [5], [18]. The dynamic analysis is performed by collecting execution traces, and by analyzing them for specific objectives. This approach would solve the issue of the reflection easily, but the weakness of the dynamic analysis still remains. In the dynamic analysis, we have to choose appropriate test executions to cover practical use-cases of the target program. Without such consideration, the result would be partial and it cannot be generalized. Another issue is to handle large amount of execution trace data efficiently. To handle the target programs with practical sizes, we need to devise some compaction method to reduce the execution trace sizes [13], [19].

In our experiments, we have used seven OSS systems to know the characteristics of AE and the AE metric. These would be insufficient to get general consequence, especially on the case of evolution pattern of the AE metric, which was obtained only from Ant's 22 versions. We will continue to analyze various OSS systems, as well as proprietary industry systems developed under some organizational development disciplines. It would be considered that such disciplined development would create less AE fields and methods, but we need validation with some data of proprietary industry systems.

D. Related Work

Access Modifier Analysis: There are some previous works related to ours. Müller has proposed bytecode analysis for checking Java access modifiers [11]. They have developed a tool named AMA (Access Modifier Analyzer), which analyzes Java bytecode for the similar objectives of ours. However, the bytecode level analysis for the fields and methods does not always correspond to the source code level analysis, due to the extra fields and methods added at the compilation time. Also, they have reported no empirical results of using their tool. In this paper, we have clearly defined the concept of AE, and shown the empirical results with several target systems.

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 [6]. Among these researches, an issue of access modifier declaration has been discussed by Viega et al. [21], 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.

Vidal et al. have empirically studied similar AE analysis to Java open-source software, and found that at least 20% of defined methods are over-exposed, and 70% of the methods of subject applications are defined as public [20]. They also found that libraries have on average more over-exposed methods than plain applications, less than 10% of the over-exposed methods defined in early versions of the applications become non-over-exposed in future versions.

The idea of using access modifier metrics would be related to our previous work [9]. 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.

Static Code Analyzers: There are a lot of static code analysis tools for Java [29], [30]. Those tools are very popular these days in the development of Java programs. They find possible bugs or bad coding patterns such as deadlock, overloading, array boundary overflow, and so on. Rutar et. al. have compared five of those tools, and reported the difference [14]. However, none of these tool currently available publicly or commercially have the feature for analyzing the AE fields and methods.

Static Code Analysis for Object Oriented Languages: Analyzing actual usage of fields and methods are related to various static analysis techniques of object oriented languages, such as *reference analysis*, *point-to analysis*, *call graph analysis*, and *class analysis*. Reference analysis and point-to analysis are static methods for identifying reference objects of a pointer or variable [1], [10], [17]. It is based on the control flow and data flow analyses, and also context sensitivity or flow sensitivity of method calls is used [22]. In our AE analysis, MASU provides information of possible reference of fields, which would be considered as a simple point-to analysis.

Call graph analysis is a method to establish caller and callee relations for the given program [7]. There are many tradition in ordinary programming languages, and also many methods have been proposed and actually implemented as tools for the object oriented languages [25], [26], [28]. In MASU, the caller-callee relations is constructed by estimating possible candidate callees for each method invocation.

These analyses in object oriented languages need a basis of class analysis, which determines the classes of the objects to which reference variables may point. This is a fundamental analysis so that there are many research works presented [7], [15]. MASU performs this analysis as a part of other analyses mentioned above.

VI. Conclusions

ModiChecker has been used to analyze many OSS systems to investigate our approach with two research questions. The analysis results show the characteristics of AE fields and methods, including their transition and evolution.

We have found that our system is very useful to detect fields and methods with the excessive access modifiers, and that there are many AE fields and methods with various reasons including bad design and coding. According to the result of AE analysis for software evolution, we also found that the number of AEs increases when the major version was released. We understood that it would be better for developers that they would like to check AEs before releasing new major version, to find out potential bugs effectively. We are confident that the AE analysis with ModiChecker is an important method to support quality programming and understanding in Java.

Throughout the experiments, we found that there are two types of AEs, AEs by design or coding errors, and *intentional* AEs for further extensions or accesses from external software. Our tool has a feature to opt-out intentional AEs manually to cope with the situation. As the further work, we are developing a mechanism to detect intentional AEs automatically, by analyzing class diagrams or test codes of target source codes. Investigating abstract classes would also be needed to clarify how many AEs are in existing software. Interviewing software developers to ask why/how they create and resolve AEs in their software would also be an interesting further direction of this research.

Acknowledgment

The authors would like to thank to Quoc Dotri who contributed to the initial development of ModiChecker. Also, we would like to thank to Tatsuya Ishizue and Riku Ohnishi for the support to this work. This work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) (No.25220003) and Osaka University Program for Promoting International Joint Research.

References

[1] L. Andersen, “Program Analysis and Specialization for the C Programming Language”, Ph.D. thesis, DIKU, University of Copenhagen, 1994.
[2] K. Arnold, J. Gosling and D. Holmes, “The Java Programming Language, 4th Edition”, Prentice Hall, 2005.
[3] G. Booch, R.A. Maksimchuk, M.W. Engel, B.J. Young, J. Conallen and K.A. Houston, “Object-Oriented Analysis and Design with Applications”, Addison Wesley, 2007.
[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] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen and R. Koschke, “A Systematic Survey of Program Comprehension through dynamic Analysis”, IEEE Tran. on Software Engineering, Vol. 35, No. 5, pp.684–702, 2009.
[6] D. Evans, and D. Larochells, “Improving Security Using Extensible Lightweight Static Analysis”, IEEE software, vol.19, No.1, pp. 42-51, Jan/Feb 2002.
[7] D. Grove, and C. Chambers, “A Framework for Call Graph Construction Algorithms”, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 23, No. 6,pp. 685–746, 2001.
[8] Y. Higo, A. Saito, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, “A Pluggable Tool for Measuring Software Metrics from Source Code”, The Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM-MENSURA), Nara, Japan, pp.3–12, Nov. 2011.
[9] 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 (in Japanese).
[10] A. Milanova, A. Rountev, and B. Ryder, “Parameterized Object Sensitivity for Points-to Analysis for Java”, ACM Transactions on Software Engineeing and Methodology, Vol. 14, No. 1, pp. 1-41, 2005.
[11] A. Müller, “Bytecode Analysis for Checking Java Access Modifiers”, Work in Progress and Poster Session, 8th Int. Conf. on Principles and Practice of Programming in Java (PPPJ 2010), Vienna, Austria, 2010.
[12] D. Quoc, K. Kobori, N. Yoshida, Y. Higo and K. Inoue, “ModiChecker: Accessibility Excessiveness Analysis Tool for Java Program”, 28th National Convention of Japan Society for Software Science and Technology, Vol. 28, pp. 78-83, Nara, Japan, 2011.
[13] S. Reiss, and M. Reniers, “Encoding Program Executions”, 23rd International Conference on Software Engineering, pp. 221–230, Toronto, Canada, 2001.
[14] N. Rutar, C. Almazan, and J. Foster, “ A Comparison of Bug Finding Tools for Java”, 15th International Symposium on Software Reliability Engineering (ISSRE 04), pp. 245– 256, Saint-Malo, France, 2004.
[15] B. Ryder, “Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages”, 12th International Conference on Compiler Construction, pp. 126–137,Warsaw, Poland, 2003.
[16] A. Saito, G. Yamada , T. Miyake, Y. Higo, S. Kusumoto and K. Inoue, “Development of Plug-in Platform for Metrics Measurement”, International Symposium on Empirical Software Engineering and Measurement, Poster Presentation, Lake Buena Vista, 2009.
[17] B. Steensgaard, “Points-to analysis in almost linear time”, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32–41, St. Petersburg, FL, 1996.
[18] T. Systa, “Understanding the Behavior of Java Programs”, 7th Working Conference on Reverse Engineering, pp.214-223, Brisbane, Australia, 2000.
[19] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto and K. Inoue, “Extracting Sequence Diagram from Execution Trace of Java Program”, 8th International Workshop on Principles of Software Evolution (IWPSE 2005), pp.148-151, Lisbon, Portugal, 2005.
[20] S. A. Vidal, A. Bergel, C. Marcos and J. A. Díaz-Pace, “Understanding and Addressing Exhibitionism in Java Empirical Research about Method Accessibility”, Technical Report of Departamento de Ciencias de la Computación, http://swp.dcc.uchile.cl/TR/2014/TR_DCC-20141204-004.pdf.
[21] 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.
[22] R. Wilson, “Efficient Context-Sensitive Pointer Analysis for C Program”, Ph.D. Thesis, EE Dept., Stanford University, 1997.
[23] Ant, http://ant.apache.org/.
[24] ANTLR, http://www.antlr.org/.
[25] cflow, http://www.gnu.org/software/cflow/.
[26] CodeViz, http://www.csn.ul.ie/ mel/projects/codeviz/.
[27] Controlling Access to Members of a Class, The Java Tutorials, http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html .
[28] Doxygen, http://www.stack.nl/ dimitri/doxygen/.
[29] FindBugs, http://findbugs.sourceforge.net/.
[30] Jlint, http://jlint.sourceforge.net/.
[31] MASU, http://sourceforge.net/projects/masu/.