アクセス修飾子過剰性の変遷に着目したJavaプログラム部品の分析

石居 達也[1,a)]　小堀 一雄[2,b)]　松下 誠[1,c)]　井上 克郎[1,d)]

概要：Java では，フィールドおよびメソッドに対してアクセス修飾子を宣言することで，外部からアクセス可能な範囲を制限することができる．しかし，既存ソフトウェアには実際の利用範囲に対して過剰に広く設定されているアクセス修飾子が多数存在することが知られている．一方で，それらのアクセス修飾子の修正状況については，過去に分析が行われていない．そこで本研究では，ソフトウェア開発の履歴を対象として，過剰なアクセス修飾子に対する修正作業の実行頻度について分析した．分析対象とするデータは，既存のアクセス修飾子過剰性検出ツールを拡張して既存の 7 つの Java プロジェクトから取得した．分析を行うに当たり，宣言されているアクセス修飾子と実際の利用範囲に基づき，フィールドおよびメソッドを 3 状態へ分類した．さらに，バージョン間における状態遷移を，性質ごとに 6 つのグループへと分類した．その結果，過剰なアクセス修飾子の大半は，修正されずそのまま放置されていることを確認した．一方，一部の種類の過剰なアクセス修飾子については，分析対象の全プロジェクトにおいて修正が行われていることを確認した．

キーワード：アクセス修飾子，Java プログラム，開発履歴

An Analysis about Accessibility Excessiveness of Revision Histories in Java Programs

Abstract: Developers can declare access modifiers for fields and methods in Java, and this lets them limit access scopes for fields and methods. However, existing software has a lot of fields and methods that have access modifiers whose access scopes are larger than actual ones. On the other hand, it is not clear how often these access modifiers are modified. In this study, we analyzes how often developers modify excessive access modifiers in software update. We get data of analysis subject from existing seven Java projects by using a existing tool that detects fields and methods with excessive access modifiers, by extension. Moreover, we labeled fields and methods as three states by the combination of declared and actual access scope. Then, we also labeled state transitions of fields and methods between old and new versions as six states by their behaviors. As a result, the tendency was confirmed that most of the excessive access modifiers were leaved and not touched. By contrast, some kind of excessive access modifier was confirmed that they were modified in all of intended Java projects.

Keywords: Access Modifier, Java Program, Revision History

1. はじめに

ソースコード中の変数およびメソッドが，ソースコードのすべての場所から参照可能な状態であると，潜在的な不具合の原因となる可能性がある [1]．Java では，この問題を解決する手段として，フィールドおよびメソッドに対しアクセス修飾子を宣言することができる．開発者は，適切な

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[2] 株式会社 NTT データ
NTT DATA Corporation
[a)] t-isizue@ist.osaka-u.ac.jp
[b)] koborik@nttdata.co.jp
[c)] matusita@ist.osaka-u.ac.jp
[d)] inoue@ist.osaka-u.ac.jp

アクセス修飾子を宣言することで，設計時に意図していない不適切なアクセスを未然に防止することができる [3][4]．しかし，多くの開発者が関わるソフトウェア開発においては，開発者全員がフィールドおよびメソッドの利用状況に関する情報を共有することが難しい．その結果，実際の利用範囲よりも広い範囲のアクセス修飾子を暫定的に宣言しておいたものが，そのまま残り続ける場合がある．

我々の研究グループでは，過去の研究 [2] においてアクセス修飾子過剰性検出ツール ModiChecker を開発している．既存 Java プロジェクトに対し ModiChecker を実行した結果，過剰となっているアクセス修飾子が多数存在していることが確認された．一方，プログラムの開発履歴において，過剰なアクセス修飾子がどのタイミングで修正されるのか，あるいは修正されずに残り続けるのかということについては，過去に分析が行われていない．

そこで本研究では，Java プログラムの開発履歴における，過剰なアクセス修飾子に対する修正作業の実行頻度に関する分析を行った．分析対象としたのは既存の 7 つの Java プロジェクトであり，これらに対して機能拡張された ModiChecker を実行して得られたデータを利用した．分析を行うに当たり，宣言されているアクセス修飾子と実際の利用範囲に基づき，フィールドおよびメソッドを 3 状態に分類した．さらに，バージョン間における状態遷移について，遷移の性質ごとに 6 つのグループへと分類した．

これらの分類に基づき分析を行った結果，過剰なアクセス修飾子の大半は，修正されずそのまま放置されていることを確認した．一方で，一部の種類の過剰なアクセス修飾子については，7 つの Java プロジェクト全てにおいて修正が行われていることを確認した．

以降，2 章では本研究に関連する用語を説明する．3 章では本研究で行った分析の詳細について述べ，4 章では分析結果の提示と結果に対する考察を行う．5 章では関連研究について述べ，6 章でまとめと今後の課題について述べる．

2. 背景

2.1 アクセス修飾子

Java の言語仕様では，フィールドおよびメソッドに対して外部からのアクセス範囲を制限する修飾子を宣言することができる．これを**アクセス修飾子**と呼ぶ．何もアクセス修飾子を付けない場合 (default) を含めると，Java では 4 種類のアクセス制限を科すことができる (表 1)[5]．

アクセス修飾子を適切に設定することで，開発者はフィールドおよびメソッドに対するクラス外部からの想定外の干渉を防ぐことができる．これをカプセル化と呼び，オブジェクト指向プログラミングの主要な性質の 1 つとされている [6]．しかし，実際のソフトウェア開発においては，各フィールドおよびメソッドに対する最終的なアクセス範囲が不透明なままコーディングを開始する場合がある．そういった状況下では，最終的なアクセス範囲よりも広い範囲からのアクセスを許可するアクセス修飾子が設定されることがあり，このことが不具合の原因となる可能性がある．

表 1 アクセス修飾子の種類

アクセス修飾子	アクセスを許容する範囲
public	全ての部品
protected	自身と同じパッケージに所属する部品 および自身のサブクラス
default	自身と同じパッケージに所属する部品
private	自身と同じクラス

想定しているメソッドの用途に対して過剰なアクセス修飾子を設定した場合に起こりうる問題の例として，以下に示すクラス X を用いて説明する．

```
public class X {
   // フィールド y の初期値は null.
   private String y = null;
   // フィールド y に値を設定する.
   // クラス外から呼ばれることを想定していない.
   private void methodA( ) {
      y = "hello";
   }
   // フィールド y の文字列長を返す.
   // クラス外から呼ばれることを想定していない.
   public int methodB() {
      return y.length();
   }
   // 値の設定されたフィールド y の文字列長を返す.
   // クラス外から呼ばれることを想定している.
   public int methodC() {
      this.methodA();
      return this.methodB();
   }
}
```

クラス X は，変数 y の文字列長を取得することを目的としたクラスである．y の文字列長を取得するには methodB を呼び出す必要があるが，y には初期値として null が代入されているため，目的を達成するためには

(1) methodA を呼び出し，y に文字列を代入する．
(2) methodB を呼び出し，length メソッドにより y の文字列長を取得する．

という手順を踏む必要がある．この手順を実行するために methodC が用意されており，開発者は methodC がクラス外から呼ばれることを想定してアクセス修飾子を public としている．しかし，この例では methodB のアクセス修飾子として誤って private ではなく public が設定されてしまっている．これにより，methodA を呼び出す前に methodB を外部から直接呼び出すことができる．こうした呼び出さ

表 2　AE の種類

[a][b]	public	protected	default	private	No Access
public	pub-pub	pub-pro	pub-def	pub-pri	pub-na
protected	x	pro-pro	pro-def	pro-pri	pro-na
default	x	x	def-def	def-pri	def-na
private	x	x	x	pri-pri	pri-na

[a] 列タイトル：宣言されているアクセス修飾子
[b] 行タイトル：実際にアクセスされている範囲

れ方をした場合，y が null の状態で length メソッドを呼び出すことになるため，例外 NullPointerException が発生する．この例のような状況が生じる原因としては，例えば実際のアクセス範囲が不透明なために暫定的に public を宣言しておいたものが，修正されずに残り続けてしまうことが考えられる．

2.2 Accessibility Excessiveness

本研究では，Java のソースコード群に宣言されたフィールドおよびメソッドに対し，宣言されているアクセス修飾子と実際に呼び出されている範囲との差異を表現するために **Accessibility Excessiveness(以下 AE)**[2] を用いる．なお，各 AE の名称については，便宜上 [2] とは異なるものを用いる．

AE は表 2 のように分類される．例えば，あるフィールドに対して宣言されているアクセス修飾子が public であるのに対し，実際にアクセスされる範囲が private 相当である場合，そのフィールドは表 2 の内の pub-pri の状態にあるとみなす．

フィールドおよびメソッドの中には，宣言されてはいるが実際にはどこからもアクセスされないものが存在する．本研究では，そういったどこからもアクセスされない状態 (**No Access**) についても考慮することとする．

本研究においては，pub-pro，pub-def，pub-pri，pro-def，pro-pri，def-pri の 6 つの状態について，開発者の想定しているアクセス範囲よりも広いアクセス修飾子が宣言されている状態とみなし，これらを AE であると定義する．

なお，表 2 において x と表示されている箇所に相当する記述は，通常はコンパイラによりエラーとして検出されるために，本研究では考慮しない．

2.3 ModiChecker

プロジェクト中のフィールドおよびメソッドに対する適切なアクセス範囲の把握を支援するため，我々は過去の研究 [2] においてアクセス修飾子過剰性検出ツール **ModiChecker** を開発した．ModiChecker は，ソースコード群に対して，アクセス修飾子の宣言とフィールドおよびメソッドの被参照状況を静的解析することにより，AE となっている可能性のあるアクセス修飾子を持つフィールドおよびメソッドを抽出する．その結果は図 1 のように表示される．図 1 の 3 列目 (Current Modifier) が解析時点で宣言されているアクセス修飾子を，4 列目 (Recommended Modifier) が静的解析により判明した実際のアクセス範囲に基づく適切なアクセス修飾子を表す．

ModiChecker の開発により，ツール利用者は AE となっている可能性のあるフィールドおよびメソッドの一覧と，それらの実際のアクセス範囲に基づいた適切なアクセス修飾子に関する情報を容易に取得することができる．

3. アクセス修飾子の過剰性分析

フィールドおよびメソッドに対して実際のアクセス範囲に即したアクセス修飾子を宣言することは，開発者の想定していないアクセスによる不具合を未然に防止することにつながる．すなわち，アクセス修飾子を適切に宣言することは，高品質なソフトウェアを構築するための重要な手段の一つであるといえる．しかし，現在のソフトウェア開発現場においては，要件の複雑化などに伴い，開発者が全てのフィールドおよびメソッドに関する適切なアクセス範囲を把握することは困難であるのが実情である．実際，Ant や jEdit の 1 バージョンにおいて，AE であるようなフィールドおよびメソッドが多数存在していることが確認されている [2]．一方で，AE であるアクセス修飾子の修正状況については，過去に分析が行われていない．

そこで，本研究では既存の Java プロジェクトを対象として，現在のソフトウェア開発においてアクセス修飾子の修正作業がどの程度行われているのかについての分析を行った．今回分析を行うに当たって，以下の研究課題 (research question) を設定した．

RQ1　アクセス修飾子の修正作業はどれほどの頻度で行われているのか

RQ2　AE の種類ごとに修正頻度の差は存在するのか

RQ1 についてはアクセス修飾子の遷移状況について，RQ2 については AE の種類ごとの修正状況について，それぞれ分析対象とした Java プロジェクトの全バージョンにおける全てのフィールドおよびメソッドを対象として追跡調査を行った．

3.1 分析対象とした Java プロジェクト

今回の分析では，SourceForge.jp[7] からダウンロード可能な Java プロジェクトの中から比較的バージョン数が多く，開発期間が長いものを 7 つを分析対象とした．分析の対象としたプロジェクトの一覧を表 3 に示す．なお，表 3 中の開発期間については，分析対象のバージョンのリリース日を基に記述している．

3.2 フィールドおよびメソッドの状態の分類

分析を円滑に行うため，本研究ではまず，プロジェクト

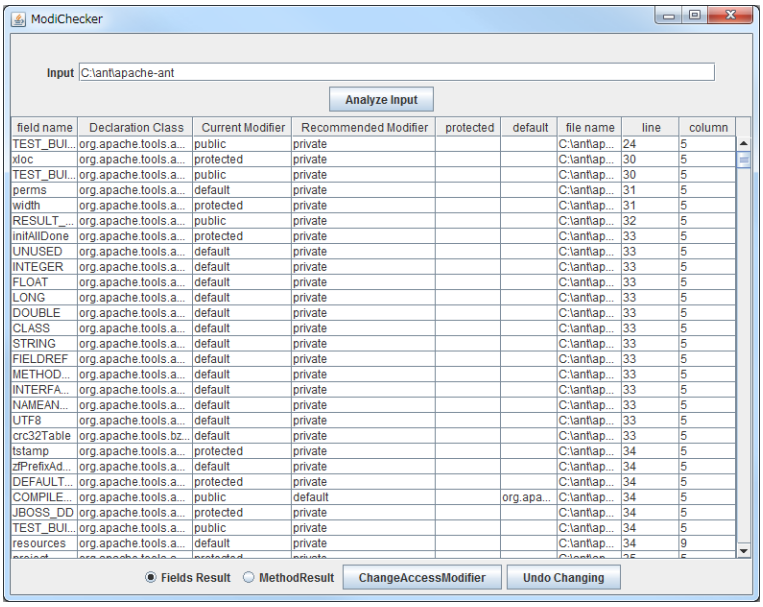

図 1 ModiChecker の解析結果表示画面

表 3 分析対象としたプロジェクト一覧

プロジェクト名	バージョン番号	バージョン数	フィールドアクセス修飾子変遷総数	メソッドアクセス修飾子変遷総数	開発期間 (年)
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

表 4 アクセス修飾子の組み合わせによる状態の分類

	public	protected	default	private	No Access
public	pub-pub	pub-pro	pub-def	pub-pri	pub-na
protected	x	pro-pro	pro-def	pro-pri	pro-na
default	x	x	def-def	def-pri	def-na
private	x	x	x	pri-pri	pri-na

の各バージョンにおけるフィールドおよびメソッドの状態について分類を行った．

ここでは，ソースコード上のフィールドおよびメソッドについて，宣言されているアクセス修飾子と実際のアクセス範囲との組み合わせにより以下の 3 状態に分類する (表 4)．

適切 実際のアクセス範囲に即したアクセス修飾子が宣言されている状態 (表 4：白色セル)．

AE 実際のアクセス範囲に比べて過剰なアクセス範囲が宣言されている状態 (表 4：薄灰色セル)．

No Access フィールドおよびメソッドが宣言されてはいるが，プロジェクト内のどこからもアクセスがなされていない状態 (表 4：濃灰色セル)．

3.3 バージョン間における状態遷移の分類

前節で定義した 3 状態は，プロジェクトがバージョンアップされる際に，リファクタリングなどの操作によって別の状態へと遷移したり，同じ状態の中の別のパターンへと遷移したりする．フィールドおよびメソッドの生成・削除や，2 バージョン間で状態の変遷が生じなかった場合を考慮に含めると，アクセス修飾子の 2 バージョン間における状態遷移には，図 2 に示す 18 種類が存在する．これら 18 種類の遷移は，その性質ごとに 6 つにグループ分けできる．なお，図 2 中の「なし」は，対象となるフィールドおよびメソッドがあるバージョンにおいては存在していないことを表す状態である．また，「なし」から「なし」への遷移については，状態遷移そのものが発生していないものとし，本研究においては考慮しない．

AE 修正 「適切」に向かって伸びる矢印 a,b,c の 3 つが該当する．アクセス修飾子の修正，あるいはアクセス範囲の調整により，アクセス修飾子が適切なものへと変化するような遷移を指す．なお，a についてはバージョン間でアクセス修飾子が修正されたもののみを対象とする．以降で解説する e と i についても同様と

する．

AE 発生　「AE」に向かって伸びる矢印 d,e,f の 3 つが該当する．アクセス修飾子，あるいはアクセス範囲の変化により，アクセス修飾子が AE となるような遷移を指す．

アクセス消失　「No Access」に向かって伸びる矢印 g,h,i の 3 つが該当する．アクセス修飾子，あるいはアクセス範囲の変化により，フィールドおよびメソッドへのアクセスが消失するような遷移を指す．

フィールド/メソッド作成　「なし」から伸びる矢印 j,k,l の 3 つが該当する．旧バージョンに存在しなかったフィールドおよびメソッドが，新バージョンにおいて新たに作成されるような遷移を指す．

フィールド/メソッド削除　「なし」に向かって伸びる矢印 m,n,o の 3 つが該当する．旧バージョンに存在したフィールドおよびメソッドが，新バージョンにおいて削除されるような遷移を指す．

変化なし　「適切」,「AE」,「No Access」から自身へとループする矢印 p,q,r の 3 つが該当する．2 バージョン間でアクセス修飾子およびアクセス範囲の双方共に変化がないような遷移を指す．

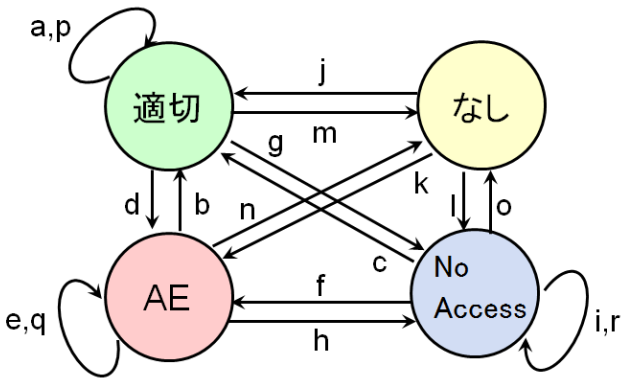

図 2　プロジェクト 2 バージョン間におけるアクセス修飾子の状態遷移図

3.4　分析手順

今回の分析は以下の手順で実施した．

(1) 分析対象のプロジェクトの各バージョンに対し ModiChecker を実行し，各バージョンごとの全フィールドおよびメソッドのアクセス修飾子宣言状況に関するデータの記載された csv ファイルを生成する

(2) 全バージョン中の重複しないフィールドおよびメソッドの一覧を取得する．重複の判定には，フィールドの場合はフィールド名とパッケージ名，メソッドの場合はメソッド名，パッケージ名およびシグネチャの組を用いる

(3) 2. で取得したフィールドおよびメソッドの各バージョンにおけるアクセス修飾子宣言情報を perl スクリプトにより一つの csv ファイルに統合する

(4) 3. で取得したデータを基に各種分析を行う

なお，初期状態の ModiChecker は AE であるフィールドおよびメソッドのみを出力とするが，本研究においてはアクセス修飾子が適切であるフィールドおよびメソッドを考察対象に含めるため，事前準備として ModiChecker が適切なフィールドおよびメソッドも同時に出力するように改変を行った．

3.5　分析環境

本研究における分析環境に関する情報は以下の通り．

- OS : Microsoft Windows 7 Enterprise Service Pack 1 (64bit)
- CPU : Intel(R) Xeon(R) CPU E5507 @ 2.27GHz 2.26GHz (2 プロセッサ)
- RAM : 24.0GB
- Eclipse classic 3.7.2
- JDK 1.7.0_07
- perl v5.14.2

また，分析にかかる時間はソフトウェアの規模に比例し，一つのソフトウェアバージョンにつき最小で約 60 秒，最大で約 360 秒程度である．

4.　分析結果と考察

各プロジェクトにおいて，RQ1 では，3.3 節で定義を行ったアクセス修飾子の各種変遷がそれぞれどの程度発生したのかについて集計を行った．表 5，表 6 は，各種変遷が変遷全体に占める割合について，フィールドおよびメソッドそれぞれの値を示している．RQ2 では，各 AE がそれぞれどの程度修正されているのかについて集計を行った．表 7，表 8 は，適切なアクセス修飾子への修正作業がフィールドおよびメソッドそれぞれについてどの程度行われたかの割合を示している．また，表中の「N/A」は全バージョン間において対象 AE が一度も出現しなかったことを表す．

4.1　RQ1：アクセス修飾子変遷分析 (フィールド)

RQ1：グループごとに見られる傾向

表 5 を基に，フィールドのアクセス修飾子変遷について，6 つのグループそれぞれに見られる傾向の分析を行う．

AE 修正　全体に対する割合としては，全プロジェクトにおいて 1%に満たない．グループ内でみると，b の「AE →適切」について，Areca を除く 6 プロジェクトにおいて a,c に比べて約 2.6〜28 倍の頻度で修正が行われていることがわかる．

AE 発生，アクセス消失　全体に対する割合としては，2 グループ共に全プロジェクトにおいて 1%に満たない．グループ内でみると，f,i の No Access からの遷移は，

表 5 フィールドのバージョン間変遷割合 (%)

		Ant	Areca	ArgoUML	FreeMind	JDT Core	jEdit	Struts
a	適切→適切	0.02	0.01	0.03	0.12	0.02	0.02	0.01
b	AE →適切	0.16	0.01	0.42	0.31	0.30	0.15	0.28
c	No Access →適切	0.02	0.01	0.05	0.07	0.07	0.05	0.01
d	適切→ 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	適切→ 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	なし→適切	6.41	1.45	6.84	22.59	4.08	6.16	5.57
k	なし→ AE	1.41	0.55	1.56	7.13	1.79	1.78	2.52
l	なし→ No Access	0.21	0.07	1.38	3.27	0.24	0.81	0.59
m	適切→なし	2.03	0.66	4.05	7.28	0.80	3.48	2.67
n	AE →なし	0.46	0.27	2.20	2.80	0.78	1.07	1.14
o	No Access →なし	0.13	0.03	1.36	2.04	0.09	0.58	0.22
p	変化なし (適切)	71.42	65.50	58.72	35.85	64.98	63.67	50.34
q	変化なし (AE)	15.28	26.74	12.26	12.99	22.72	16.22	29.00
r	変化なし (No Access)	2.28	4.62	10.72	4.75	3.66	5.79	7.34

表 6 メソッドのバージョン間変遷割合 (%)

		Ant	Areca	ArgoUML	FreeMind	JDT Core	jEdit	Struts
a	適切→適切	0.03	0.00	0.03	0.12	0.02	0.02	0.00
b	AE →適切	0.10	0.03	0.12	0.26	0.22	0.16	0.07
c	No Access →適切	0.13	0.07	0.26	0.36	0.24	0.13	0.09
d	適切→ 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	適切→ 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	なし→適切	2.31	1.10	3.63	12.34	2.48	3.85	1.96
k	なし→ AE	1.08	0.28	1.06	2.10	0.78	1.38	1.52
l	なし→ No Access	4.44	1.10	5.48	17.41	2.71	3.32	5.73
m	適切→なし	0.54	0.63	1.88	5.41	1.04	2.12	0.73
n	AE →なし	0.28	0.19	0.84	0.68	0.46	0.86	1.08
o	No Access →なし	0.95	0.71	3.44	9.72	1.27	1.85	2.99
p	変化なし (適切)	26.46	44.30	28.44	23.29	38.83	38.58	19.16
q	変化なし (AE)	12.89	11.88	8.92	3.10	11.22	14.02	11.07
r	変化なし (No Access)	50.47	39.64	45.42	24.55	40.27	33.24	55.34

他 2 状態からの遷移に比べて出現頻度が少ないことがわかる．

フィールド/メソッド作成 全体に対する割合としては，約 2〜33%を占める．グループ内でみると，全プロジェクトにおいて遷移先が適切 (j)，AE(k)，No Access(l) の順に出現頻度が高い．

フィールド/メソッド削除 全体に対する割合としては，約 1〜12%を占める．グループ内では，全プロジェクトにおいて遷移前が適切 (m)，AE(n)，No Access(o) の順に出現頻度が高い．

変化なし 全体に対する割合としては，6 グループの中で最も大きい約 53〜97%を占める．全プロジェクトにおいて適切 (p)，AE(q)，No Access(r) の順に出現頻度が高い．

RQ1：**考察**

フィールドにおけるアクセス修飾子の変遷について，最も多く見られたのは「変化なし」に属する p の「変化なし (適切)」であった．また，「フィールド/メソッド作成」中でも j の「なし→適切」は比較的多い傾向が見られる．

これらのことから，フィールドは最初から用途を明確にして作成されることが多く，一度適切なアクセス修飾子が宣言されると，その後長期にわたって利用される場合が多

表 7　AE であるフィールドの修正状況 (%)

	Ant	Areca	ArgoUML	FreeMind	JDT Core	jEdit	Struts
pub-pro	0.0055	0.0000	0.0013	0.0000	0.0150	0.0000	0.0038
pub-def	0.0038	0.0017	0.0195	0.1607	0.0076	0.0078	0.0036
pub-pri	0.0024	0.0005	0.0379	0.0073	0.0269	0.0070	0.0013
pro-def	0.0254	0.0023	0.0149	0.0000	0.0068	0.0326	0.0267
pro-pri	0.0112	0.0002	0.0177	0.0000	0.0085	0.0045	0.0118
def-pri	0.0157	0.0000	0.0365	0.0063	0.0060	0.0077	0.0021

表 8　AE であるメソッドの修正状況 (%)

	Ant	Areca	ArgoUML	FreeMind	JDT Core	jEdit	Struts
pub-pro	0.0141	0.0000	0.0119	0.0398	0.0074	0.0083	0.0077
pub-def	0.0091	0.0026	0.0132	0.0606	0.0279	0.0121	0.0056
pub-pri	0.0038	0.0028	0.0112	0.0169	0.0098	0.0111	0.0007
pro-def	0.0153	0.0000	0.0160	0.0000	0.0054	0.0000	0.0142
pro-pri	0.0046	0.0000	0.0113	0.1325	0.0038	0.0037	0.0100
def-pri	0.0000	N/A	0.0066	0.0000	0.0141	0.0015	0.0015

いといえる．一方，その他のグループの状態遷移について考察を行った場合，「AE 修正」「AE 発生」「アクセス消失」のようなアクセス範囲の変化に伴う状態遷移の数と比べると，「フィールド/メソッド削除」のようなフィールドそのものが消滅する場合の状態遷移の数が多くなる傾向にある．このことは，フィールドの利用方法が変更されるような場合には，アクセス修飾子の修正ではなくフィールドそのものが変更されることが多いことを示している．

4.2　RQ1：アクセス修飾子変遷分析 (メソッド)

RQ1：グループごとに見られる傾向

表 6 を基に，メソッドのアクセス修飾子変遷について，6 つのグループそれぞれに見られる傾向の分析を行う．

AE 修正　全体に対する割合としては，全プロジェクトにおいて 1%に満たない．グループ内でみると，c の「No Access →適切」が，b の「AE →適切」と同等もしくはやや高い頻度で出現していることが分かる．

AE 発生，アクセス消失　全体に対する割合としては，2 グループ共に全プロジェクトにおいて 1%に満たない．グループ内でみると，d,g の適切からの遷移が他 2 状態からの遷移に比べてやや出現頻度が少ないことがわかる．

フィールド/メソッド作成　全体に対する割合としては，約 2〜32%を占める．グループ内でみると，jEdit を除く 6 プロジェクトにおいて，遷移先が No Access(l)，適切 (j)，AE(k) の順に出現頻度が高い．

フィールド/メソッド削除　全体に対する割合としては，約 1〜16%を占める．グループ内では，jEdit を除く 6 プロジェクトにおいて遷移前が No Access(o)，適切 (m)，AE(n) の順に出現頻度が高い．

変化なし　全体に対する割合としては，6 グループの中で最も大きい約 51〜96%を占める．グループ内では，Areca と jEdit では適切 (p)，No Access(r) の順である以外は，No Access(r)，適切 (p)，AE(q) の順に出現頻度が高い．

RQ1：考察

メソッドにおけるアクセス修飾子の変遷については，最も多いのが「変化なし」に属する r の「変化なし (No Access)」であり，次いで p の「変化なし (適切)」であった．「フィールド/メソッド作成」においても同様の傾向が見られ，jEdit で逆転が見られる以外では，o の「なし→ No Access」が m の「なし→適切」を上回った．

これらのことから，メソッドについては，作成時点から利用されているものよりも，作成時点では用途が定まっていないか，利用する側のメソッドがまだ作成されていないもののほうが多いことがわかる．また，その他のグループの状態遷移について考察を行った場合，フィールドと同様に「AE 修正」「AE 発生」「アクセス消失」に比べて「フィールド/メソッド削除」の遷移が多い傾向にある．このことは，フィールドと同様に，メソッドの利用方法が変更されるような場合には，メソッドそのものが変更されることが多いことを示している．

4.3　RQ2：AE 修正状況の分析と考察 (フィールド)

フィールドに対する AE の修正作業が行われる割合は最大でも 0.16%であり，大半は修正がなされていない，ということが挙げられる．各 AE ごとの傾向を見ていくと，pub-def，pub-pri は全プロジェクトで，pro-def，pro-pri，def-pri は 1 プロジェクトを除く 6 プロジェクトで修正作業が行われていることがわかる．

よって，これら 5 つの AE については，ModiChecker のようなツールを用いてアクセス修飾子の修正を行うことにより，後に開発者が費やすことになるアクセス修飾子修正作業へのコストを軽減することが可能であると考えられる．

4.4 RQ2：AE 修正状況の分析と考察 (メソッド)

フィールドと同様に，AE であるメソッドに対するアクセス修飾子修正作業が行われることはほとんどない，ということが挙げられる．また，各 AE ごとの傾向としては，pub-def，pub-pri については全プロジェクトで，pub-pro，pro-pri については 1 プロジェクトを除く 6 プロジェクトで修正作業が行われる傾向にある．

よって，これら 4 つの AE については，アクセス修飾子過剰性検出ツールを利用することによるアクセス修飾子修正作業のコスト削減が期待できる．

5. 関連研究

アクセス修飾子の解析に関して，我々の研究以前にいくつかの研究がなされている．

Muller は Java のアクセス修飾子をチェックするためのバイトコード解析手法を提案している [8]．しかし，バイトコードに対する解析は，コンパイル時に追加されるフィールドやメソッドの影響で，本研究で行ったようなソースコードに対する解析とは必ずしも同じ結果にはならない．また，Muller の研究ではチェックしたアクセス修飾子に対する分析はなされていない．一方，本研究では既存の複数のソフトウェアに対して取得したデータを用いて，複数の側面からの分析を行った．

Tal Cohen は複数のサンプルメソッドにおける各アクセス修飾子の数の分布を調査した [9]．また，Evans らは静的解析によるセキュリティ脆弱性の解析を研究した [10]．これらの研究で課題となっているアクセス修飾子の宣言に関しては Viega らによって議論されている [11]．Viega らは，private にすべきだがそのように宣言されていないメソッドやフィールドについて警告を出すツール Jslint を開発している．一方，本研究では，private だけでなく全ての過剰なアクセス修飾子を分析対象としている．アクセス修飾子の数を分析対象としている点については，小堀らの研究とも関連がある [12]．この過去の研究では，Java のソースコードの類似性を計算するための分析手段の一つとして，アクセス修飾子の宣言数が用いられている．

6. まとめと今後の課題

本研究では，既存の 7 つの Java プロジェクトの全バージョンに対して ModiChecker を実行し，取得できたフィールドおよびメソッドに関するアクセス修飾子について分析を行った．その結果，大半の AE となっているアクセス修飾子は，変更されることはなくそのまま放置される傾向が見られた．また，一部の種類の AE であるフィールドおよびメソッドについては，7 つの Java プロジェクト全てにおいて修正が行われていることを確認した．

今後の課題としては，アクセス修飾子の修正がソフトウェアの品質向上にどの程度寄与するのかを調査することが挙げられる．また，今回行った分析ではバージョン間の状態遷移数および AE の修正数に関する割合の大小についてしか調査を行っていないため，統計的に検定を行い，結果に有意差が存在するかどうかを調査するということが挙げられる．さらに，各遷移が行われた後の遷移状況を詳しく調べることで，アクセス修飾子の修正が推奨されるフィールドおよびメソッドに特徴的な遷移というものが存在するかどうかを調査したい．

参考文献

[1] Bertrand Meyer, “Object-Oriented Software Construction SECOND EDITION”, Prentice Hall, 2000.
[2] Dotri Quoc, Kazuo Kobori, Norihiro Yoshida, Yoshiki Higo, Katsuro Inoue, ModiChecker: Accessibility Excessiveness Analysis Tool for Java Program, コンピュータソフトウェア, Vol.29, No.3, pp.212-218(2012).
[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 Wesly, 2007.
[4] K. Arnold, J. Gosling, D. Holmes, “The Java Programming Language, 4th Edition”, Prentice Hall, 2005.
[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, The Java Language Specification, Java SE 7 Edition, http://docs.oracle.com/javase/specs/jls/se7/html/index.html
[6] K. Khor, Nathaniel L.Chavis, S.M.Lovett and D. C. White. “Welcome to IBM Smalltalk Tutorial”, 1995
[7] SourceForge.jp, http://sourceforge.jp/
[8] A. M ¨ uller, “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.
[9] 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.
[10] D. Evans, and D. Larochells, “Improving Security Using Extensible Lightweight Static Analysis”, IEEE software, vol.19, No.1, pp. 42-51,Jan/Feb 2002.
[11] 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.
[12] 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.
[13] FindBugs, http://findbugs.sourceforge.net/
[14] Jlint, http://jlint.sourceforge.net/
[15] 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.
[16] Apache Ant, http://ant.apache.org/
[17] jEdit, http://www.jedit.org/
[18] 小堀一雄，石居達也，松下誠，井上克郎，Java プログラムのアクセス修飾子過剰性分析ツール ModiChecker の機能拡張とその応用例，SEC Journal，(採録決定・採録号等未定)