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

小堀 一雄 †
石居 達也 †
松下 誠 †
井上 克郎 †

我々は，Java プログラムの各フィールドやメソッドのアクセス修飾子の宣言とその利用状況との差をアクセス過剰性（AE:Accessibility Excessiveness）と呼び，AE を静的解析するツール ModiChecker を開発してきた．本研究では，検出した AE を容易に解消できるようにするために，AE に対して，開発者の指示に基づいて容易に修正できる機能を ModiChecker へ実装した．さらに応用例として，OSS である Ant の複数のバージョン間における AE の数の変化を調べ，AE の変化量とバージョンアップの種類との関連について調査した．

Correction Support and Version Analysis of Accessibility Excessiveness for Java Programs

Kazuo Kobori†,‡
Tatsuya Ishizue‡
Makoto Matsushita‡
Katsuro Inoue‡

Abstract:
We have developed a tool named “ModiChecker”, which statically analyzes the declaration and usage of the fields and methods in Java programs, and reports excessiveness of the declaration as “Accesibility Excessiveness (AE)”. In this paper, we will add a feature to ModiChecker to support developers resolving AE declarations. Also, we will explore the relation between AE and program evolution, by analyzing multiple versions of an OSS program, Ant.

† 大阪大学大学院情報科学研究科

1. はじめに

現在のソフトウェア開発においては，要件の複雑化などに伴い，複数の開発者がチームを組んで設計，プログラミング，テストを実施することが多い．チームに所属する開発者は全員，ソースコードの仕様を完全に把握していることが望ましいが，コストや期間の関係上難しいこともある．その場合，ソースコード上のフィールドやメソッドに対して設計時には意図していない不適切なアクセスの仕方をプログラミング時に行なってしまう可能性がある．

こういった問題を防ぐために，アクセス修飾子を適切に設定することで，意図しないフィールドやメソッドへのアクセスを防止することができる[1][2]．しかし，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のはコストがかかるため，何らかの支援が必要であると考えた．

我々は，アクセス修飾子過剰性検出ツール ModiChecker を開発した[6]．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とメソッドの被参照状況を静的解析することにより，過剰に広い範囲に設定されている可能性のあるアクセス修飾子を抽出する．これにより，開発者は意図しないフィールドやメソッドへのアクセスを事前に防止できる．

しかしこの既存研究では，どこからも参照されていないフィールドやメソッドを分析対象としない点や，抽出した後の修正支援をしないなどの課題があった．そのため，本研究では，本ツールの分析対象に無参照のものも含めた．またユーザがアクセス修飾子の修正を効率的に行えるような支援機能を追加した．

さらに，本ツールの応用例としてある特定の時点のソフトウェアにおけるアクセス修飾子の過剰性だけではなく，ソフトウェアのバージョンが上がるとともにアクセス修飾子の過剰性がどのような変遷をするかという視点で，OSS に対する調査を行った．ここでは，Ant を調査対象とし，その 22 個のバージョンに対して ModiChecker を適用して，結果を比較した．その結果，ソフトウェアの大きな変更に伴いアクセス修飾子の過剰性も大きく変化することが分かった．

本論文の構成について説明する．まず 2 節では，研究の背景となる Java アクセス修飾子の仕様および過剰なアクセス修飾子の宣言によって引き起こされる問題について述べる．3 節では，アクセス修飾子の過剰性の定義や分析するツール ModiChecker について述べる．4 節では ModiChecker への機能拡張について述べ，5 節では，ModiChecker の応用例としてソフトウェアのバージョン間におけるアクセス修飾子の過剰性を分析した結果について説明する．最後に 6 節で本論文のまとめと今後の研究について述べる．

2. アクセス修飾子とそれによって引き起こされる問題点

Java の言語仕様では，フィールドやメソッドに対して外部からのアクセス範囲を制限でき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．Java のアクセス修飾子は表1に示す 4 種類があり，上にいくほど広い範囲からのアクセスを許容する[3]．

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（指定なし）	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

特に，クラスの外部から直接変更されるとプログラムの動作に異常をきたすようなフィールドは，クラス外部からの直接アクセスを許可しないアクセス修飾子”private”を宣言しておくことで，フィールドの利用方法をクラス設計者の想定内に収めることができる．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[4]．ところが実際には，ソフトウェアを開発する際，各部品の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ことがあり，そのような状況では最終的に必要なアクセス範囲以外からのアクセスを許可するアクセス修飾子を設定することがある[5]．この問題を以下のような 3 つのメソッドを持つクラス X を例に説明する．

```
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 methodA( ) {
      y = new String(“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 methodB() {
      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 methodC() {
      this.methodA();
      this.methodB();
   }
}
```

上記のソースコードでは，まずメソッド methodA を呼び出して y にオブジェクトを代入してからメソッド methodB を呼び出す必要がある．そこで，そのようなメソッド呼び出し順序を実装したメソッド methodC を用意している．このメソッド methodC は外部から呼び出されることを期待して作られているため，アクセス修飾子を public に設定している．一方，メソッド methodB は外部から直接呼び出されてはならないにもかかわらず，アクセス修飾子も public に設定してしまっている．これにより，methodA を呼ばずに methodB を呼ぶことが可能となってしまう．このような呼び出され方をした場合，フィールド y が初期値 null の状態でメソッド length が呼ばれるため，例外 NullPointerException が発生する．

このように，アクセス修飾子が過度に広く設定されている場合，意図しないメソッド呼び出しが不具合を産む．また，開発途中における開発者間の設計情報共有不足などにより，想定外の状況下でメソッドが呼ばれることで，論理的なバグ発生の原因が作られる．しかし，Java の構文上は問題がないため，このような状況をコンパイラ等を用いて機械的に検出することは難しい．また，全てのアクセス修飾子が適切に設定されているかどうかを，レビューによって確認するには高いコストが必要である．

3. ModiChecker

我々は，指定された Java のソースコード群に宣言されたメソッドとフィールドに対して，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 Accessibility Excessiveness (以下 AE)を定義し，AE をソースコード中から検出するためのツール ModiChecker を開発した[6]．本節では，AE と ModiChecker について説明する．

宣言されているアクセス修飾子と，実際にアクセスされている範囲の組み合わせにより，表2の背景色がある位置に示す pub1，pub2，pub3，pro1，pro2，def1 の 6 種類を AE として定義する．例えば，pub1 とは， アクセス修飾子として public が宣言されているフィールドまたはメソッドで，かつ，実際にアクセスされている範囲は protected と同じである状態を意味する．

一方，pub0，pro0，def0 そして pri0 は，メソッドやフィールドのアクセス修飾子の宣言と実際にアクセスされている範囲が等しい状態，つまり適正な宣言が行われている状態を意味する．また，表 2 で x と表示されている箇所の記述は通常コンパイラによりエラーとして検出される状態を意味する．

表 2 AE の種類

†＼‡	public	protected	default	private
public	pub0	pub1	pub2	pub3
protected	x	pro0	pro1	pro2
default	x	x	def0	def1
private	x	x	x	pri0

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

ModiChecker は，与えられたソースコード中の AE であるフィールドやメソッドを探し出し，その結果を図 1 のように表示する．図 1 で表示される主な分析結果として，3 列目(Current Modifier)に現在宣言されているアクセス修飾子，4列目(Recommended)に静的解析によって判明した実際にアクセスされている範囲が表示される．

このように，ModiChecker を用いることで，開発者はソースコード中で AE であるフィールドおよびメソッドの状態を知ることができる．

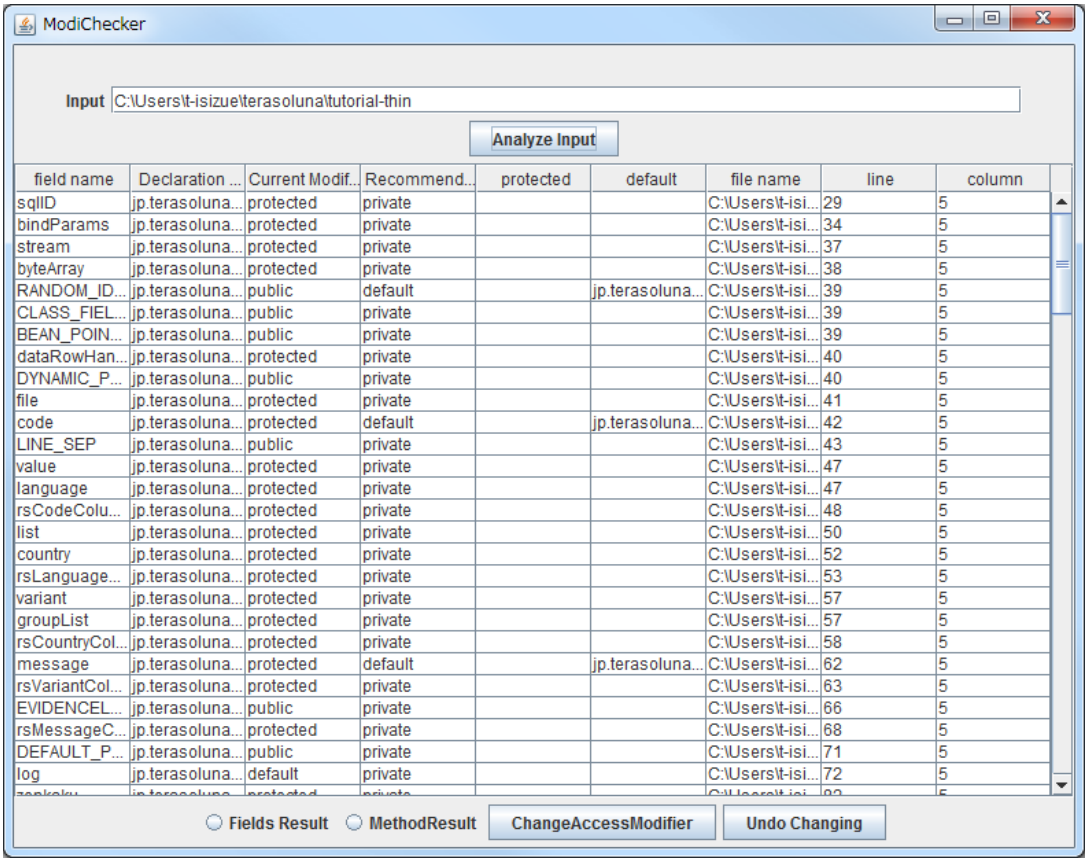

図 1 ModiChecker の AE 表示画面

4. ModiChecker の機能拡張

4.1. アクセスが無いフィールドやメソッドの分析

既存のModiCheckerでは，宣言されてはいるものの，実際にはどこからもアクセスされていないフィールドやメソッドは分析の対象としていなかった．しかし，実際のプログラムを分析してみると，そのようなどこからもアクセスされないフィールドやメソッドが多数あることに気づいた．

そこで，これらのどこからもアクセスされていないことも AE の一種と考え，表 3 のように，pub4，pro3，def2，pri1 の4種類の AE を新たに定義した．例えば，pub4 はフィールドもしくはメソッドのアクセス修飾子として public が宣言されているが，実際にはどこからもアクセスされていない状態を意味している．

また，ModiChecker を拡張し，これらの新しい AE についても図 1 に示す AE 表示画面で扱えるようにした．

表 3 拡張した AE の種類

†＼‡	public	protected	default	private	No Access
public	pub0	pub1	pub2	pub3	pub4
protected	x	pro0	pro1	pro2	pro3
default	x	x	def0	def1	def2
private	x	x	x	pri0	pri1

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

4.2. AE 修正支援機能

ModiChecker を用いて分析を行い，AE が多数存在すると分かった場合，そのようなアクセス修飾子を ModiChecker が推奨するアクセス修飾子へ修整するためには，別途，エディタ等を用いて，それぞれのアクセス修飾子を編集しなければならない．AE が大量に検出された場合，この作業は大きな手間が必要であった．

そこで本研究では，AE となっているフィールドおよびメソッドに対して，推奨するアクセス修飾子へと自動的に修正する機能を ModiChechker に実装した．本機能は，元のソースコードからアクセス修飾子の宣言部のみを変換して，推奨するアクセス修飾子に変更した新しいソースコードを自動的に作成する．

ただし，一般的なソフトウェア開発においては，理由があって特定のフィールドやメソッドに過剰なアクセス修飾子を設定している場合も存在する．例えば，以下のような例が考えられる．

- ライブラリや AP フレームワークを開発している際に，特定のフィールドやメソッドが開発対象外からの呼び出しを前提として実装されている場合
- ソースコードが開発途中である場合，将来的に内部および外部からアクセスされる可能性を残しているため，アクセス修飾子は現状必要な範囲より広いものを選択している場合

上記のような理由を持つフィールドおよびメソッドを開発者自身が選択し，自動修正の対象から外すことのできるユーザインタフェースを実装した．具体的には，図 1 に示す AE 表示画面において，下記の手順による選択的なアクセス修飾子の自動修正に対応した．

1. アクセス修飾子変更したいフィールドやメソドの行をマウス操作によって選択する．
2. 「ChangeAccessModifier」ボタンを押下する．
3. 選択した部分に対して推奨するアクセス修飾子に変更されたソースコードを自動生成する．

図 1 に示す出力結果を例に説明する．3 行目でフィールド「stream」のアクセス修飾子が現在は protected になっている．しかし，実際の呼び出し関係を ModiChecker が分析した結果，アクセス修飾子として private が推奨されている．ここで，前述に示すような特別な理由がないと開発者が判断した場合は，上記に示した手順によりフィールド「stream」のアクセス修飾子は private に修正される．

この機能を用いることで，開発者は必要なフィールドおよびメソッドだけに対して AE を効率良く修正することができる．

4.3. AE 修正支援機能の検証

前述の AE 修正支援機能は，対象プログラム群に対するリファクタリングの一種であるため，修正前と修正後のプログラムの動作が変わってはいけない．そこで，プログラムの動作確認を行うテストスイートを持つ Ant のバージョン 1.8.2 を対象として，そのテストスイートを用いて，修正の実施前後でテストを行い，動作が変わっていないことを確認する[7].

実験の手順は以下の通りである．

1. 実験対象の Ant に対して，動作を確認するテストスイートを実行する．
2. テストスイートが正常に終了したことを確認する．
3. 実験対象のソースコードをModiCheckerに入力し，AE の検出を行う．
4. 実験対象のソースコード群の外部からのアクセスを前提としているなど，開発者が意図的に設定している AE を修正対象から外す．
5. それら以外の全ての AE に対して修正機能を実行する．
6. AE の修正が行われたソースコードが自動生成されたことを確認する．
7. 自動生成されたソースコードに対して手順1で使用したテストスイートを実行する．
8. テストスイートが正常に終了したことを確認する．

実験対象のプログラムに上記手順を実施したところ，

798 個のフィールドおよび 6076 個のメソッドが AE であると判定された. これらの内, 5695 個のメソッドに関しては, ビルド時に Ant外部からアクセスされる必要があるため, 開発者が意図的に AE となるようアクセス修飾子を設定したと考えられるため AE 修正対象から外した. 残りの 798 個のフィールドおよび 381 個のメソッドに対して AE の自動修正を実施したところ, 修正後のプログラムに対してもテストスイートは正常に終了し, ModiChecker による自動修正によって機能性を損ねていないことが確認できた.

これにより, プログラムの動作を変えることなく, ModiChecker を用いてアクセス修飾子を必要最低限の範囲へ変更できることがわかった.

なお, 上記手順4で実施した修正対象の選別においては, 下記の手順をビルドエラーが出なくなるまで 6 回繰り返し, 合計で約 63 分を要した.

1. ビルドエラーのログを確認する
2. ビルドエラーの原因となったメソッドのアクセス修飾子を ModiChecker の AE 修正対象から外す
3. AE 修正前の Ant ソースコードに対して, 再度 ModiChecker の AE 修正機能を利用した AE 修正を実行し, AE 修正済みの Ant のソースコードを作成する
4. AE 修正済みの Ant のソースコードに対して再度ビルドを実行する

このうち, ModiChecker の実行時間は約 33 分で, 1 回あたり平均 333 秒であった. これは実用に耐えうる性能であると判断した.

5. 機能拡張の応用例としてのバージョン間分析

AE に関する既存研究では, ある時点でのソースコード群を対象とした AE 数について考察を行っていた[6]. しかし, ある時点での AE 数の情報だけでは, 不適切なアクセス修飾子もしくは将来的な拡張性を考慮したアクセス修飾子を持つ, 未熟な機能の多寡については判断できていなかった. そこで本節では, 同じソフトウェアの複数のバージョン間における AE 数の変化量を比較分析することで, どのようなバージョンアップ時に過剰性を残したアクセス修飾子が追加されるのか分析する.

まず, ソフトウェアのバージョンアップを, 機能拡張など比較的大きな変化を伴うと予想される「メジャーバージョンアップ(以降, MajorVU と呼ぶ)」と, 機能のバグ修正など, 比較的小さな変化を伴うと予想される「マイナーバージョンアップ(以降, MinorVU と呼ぶ)」の 2 種類に分類する. 機能が新しく追加される際には, 将来的な拡張性を考慮して, アクセス修飾子には過剰性を残した設定が行われて AE 数の変化量が増加するが, 機能のバグ修正が行われる際には, アクセス修飾子が関連しない限り AE 数は変化しないことが予想される.

上記の予想を検証するために, 本論文ではソフトウェアの MajorVU 時の AE 数の変化量と MinorVU 時の AE 数の変化量に有意差があるかどうか実験により調べる. 両者に有意差があることが分かれば, バージョンアップ時の AE 数の変化量を調べることで, 開発者がソフトウェアの機能が成熟していると判断したかどうかを推測できると考えられる. なお, MajorVU 時および MinorVU 時に実際にどのような修正が行われたかという分析は, 本論文の対象外とする.

本節では上記の有意差を確認する実験の対象ソフトウェアとして OSS の Ant を用いる[7]. Ant は Apache ソフトウェア財団が開発を行っているビルドツールであり, 多くのバージョンのソースコードが入手可能である. 本実験では Ant のバージョン 1.1 からバージョン 1.8.4 までの 22 バージョンを対象とした.

この実験における Ant の MajorVU と MinorVU の定義を以下に示す.

- MajorVU:左から 2 つ目以前のバージョン数が変化するタイミング. 例えば 1.2→1.3 や, 1.6.5→1.7.0 など.
- MinorVU:左から 3 つ目以降のバージョン数が変化するタイミング. 例えば 1.4→1.4.1 や, 1.6.4→1.6.5 など.

なお, 今回の実験対象のバージョンアップ 22 回の内訳は, MajorVU が 7 回, MinorVU が 15 回であった.

また, 4.3 節でも述べたように Ant 自身のビルドに必要なパッケージに属するフィールドやメソッドは開発者が意図的に AE であるように設定していることが予想されるため実験の対象から除外した.

表 4, 表 5, 表 6 に, それぞれ Ant のバージョン 1.3, 1.4, 1.4.1 におけるフィールドの AE 数を AE の種類ごとに分類して示す. バージョン 1.3 から 1.4 へは MajorVU であり, バージョン 1.3 から 1.4.1 へは MinorVU である.

これらのデータから, 例えば pro2 については, MajorVU で 181 個から 314 個と急増し, MinorVU では, 1 個の変化もなく, MajorVU と MinorVU で AE 数変化量に大きな差があることがわかる. また, 全てのバージョンにおいて AE の約 80%は実際のアクセス範囲が private であり, 多くのフィールドが実際にはカプセル化可能であることがわかる. また, NoAccess に関する AE は全体の 2～2.5%しかなくデッドコードが少ないことが推測できる.

表 4 バージョン 1.3 のフィールドの AE 数

†＼‡	public	Protected	default	private	No Access
public	39	0	20	84	2
protected	x	15	37	181	3
default	x	x	1	43	2
private	x	x	X	952	21

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

表 5 バージョン 1.4 のフィールドの AE 数

†＼‡	public	Protected	default	private	No Access
public	49	3	8	82	6
protected	x	16	51	314	10
default	x	x	2	51	3
private	x	x	X	1214	27

表 6 バージョン 1.4.1 のフィールドの AE 数

†＼‡	public	Protected	default	private	No Access
public	49	3	8	82	6
protected	x	17	51	314	10
default	x	x	2	49	3
private	x	x	X	1217	27

各フィールドに対する AE の各要素の数についてバージョンアップ前後の差分を示した棒グラフを図 2 に示す．この図では，MajorVU 時（図 2 の赤で囲った列）に pro2 と pub3 の変化量が多く検出されている．pro2 と pub3 は共に実際のアクセス範囲が private である AE であるため，多くのフィールドが実際にはカプセル化可能であることがわかる．さらに，多くの場合，MajorVU 時には AE 数の差分が大きく，MinorVU 時には小さいことが図 2 から推測できる．

そこで，MajorVU 時の AE 数変化量の群と，MinorVU 時の AE 数変化量の群との間に有意水準 5%における統計的な有意差があるかどうかを検定する実験を行った．

検定の対象としてバージョン 1.1 から 1.8.4 までの MajorVU 時の各フィールドの AE 数変化量の集合と MinorVU 時の各フィールドの AE 数変化量の集合を用いるが，両群のデータが正規分布に従っているかどうか不明であるため，正規分布であることを前提条件としないマン・ホイットニーの U 検定を用いたところ，検定における危険率 p 値および有意水準5%における有意差の有無は表 7 および表 8 に示す通りであった．

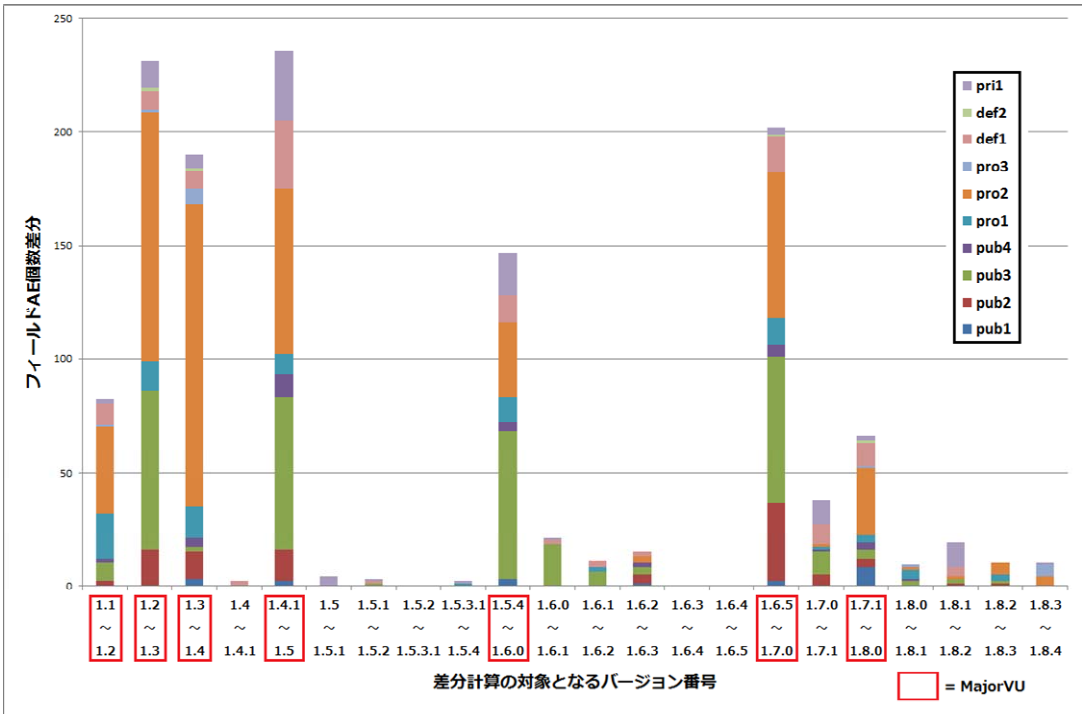

図 2 Ant のバージョン間における，フィールドの AE 数の各要素数の差分

表 7 Ant のフィールドに対する MajorVU と MinorVU の間の有意差の有無

AE	p 値※	有意水準 0.05 における有意差
pub1	0.00080	有り
pub2	0.00114	有り
pub3	0.00113	有り
pub4	0.00032	有り
pro1	0.00002	有り
pro2	0.00001	有り
pro3	0.03715	有り
def1	0.00003	有り
def2	0.00479	有り
pri1	0.00192	有り
NoAccess	0.01162	有り

※数値は小数点以下第 6 位を四捨五入している

表 8 Ant のメソッドに対する MajorVU と MinorVU の間の優位差の有無

AE	p 値※	有意水準 0.05 における有意差
pub1	0.00440	有り
pub2	0.00562	有り
pub3	0.07205	無し
pub4	0.00122	有り
pro1	0.00361	有り
pro2	0.00225	有り
pro3	0.02073	有り
def1	0.07656	無し
def2	0.13130	無し
pri1	0.00919	有り
NoAccess	0.00012	有り

※数値は小数点以下第 6 位を四捨五入している

以上の実験結果から，今回の実験対象である Ant のフィールドに関しては全ての種類の AE 数に関するバージョンアップ時の差分について，ソフトウェアの MajorVU 時と MinorVU 時の間で有意差がみられることがわかった． また，メソッドに関しては，pub3,def1,def2 以外の AE についてはバージョンアップ時の AE 数の差分について，ソフトウェアの MajorVU 時と MinorVU 時の間で有意差がみられ，フィールドとメソッドの両者とも MajorVU 時の方が MinorVU 時よりも AE 差分量が大きくなることがわかった．有意水準 0.05 における有意差が見られなかった pub3，def1 および def2 は他の AE に比べて各バージョン間の AE 変化量の値が小さく，順位がタイとなる値も多かったためマン・ホイットニーの U 検定では誤差が出やすい状況下にあった．

本実験の結果を応用すると，ある保守対象のソフトウェアのフィールドおよびメソッドに対してバージョン間の AE 数の差分を分析し，差分が大きく変化したバージョンアップでは本実験の MajorVU に相当するような変化が発生しており，アクセス修飾子に過剰性を残した機能が追加された可能性があることを発見できる．

6. 関連研究

6.1. アクセス修飾子の解析に関する関連研究

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．Müller 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]．Müller の研究では，我々と似た目的のために Java のバイトコードを解析する AMA(Access Modifier Analyzer)というツールを開発している．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，必ずしもソースコードに対する解析と同じ結果にはならない．さらに，Müller はツールを用いた実践的な実験結果を報告していない．一方，我々の研究では実際に既存のソフトウェアに対して複数の側面から実験したデータを明示し，評価も行った．

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]．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，我々の開発したツール ModiChecker では，private だけでなく全てのアクセス修飾子に対する警告を出すことができる．

アクセス修飾子の数をメトリクスとしている点については，我々の過去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メトリクスの一部として，アクセス修飾子の宣言数が用いられている．

6.2. ソースコードの静的解析に関する関連研究

Java に関するソースコード静的解析ツールは多数存在する[13][14]．これらのツールはデッドロックやオーバーロード，配列のオーバーフロー等のコーディング上の悪いパターンや，潜在的なバグを検出するため，今日

の Java プログラム開発では重要なツールである.

Rutar らは, このような機能を持つ5つのツールを比較分析した[15]. しかし, これらのツールは, 本論文のようにアクセス修飾子の冗長性を解析する機能は持っていない.

7. まとめと今後の研究

本研究では, Java のアクセス修飾子を分析するツール ModiChecker を利用して, 以下のことを行った.

1. AE の概念を拡張し, アクセスが無いフィールドやメソッドの分析も可能にした.
2. 過剰なアクセス修飾子を持つフィールドおよびメソッドの中から, 開発者が選択したものに対してアクセス修飾子を自動修正する機能を開発し, その妥当性を検証した.
3. 複数のバージョンの AE の分析を通じて, 全ての AE に関して, AE 数の差分とソフトウェアのバージョンアップ種別(MajorVU／MinorVU)に相関があることを確認した.

本結果は, 特定の OSS の分析に基づいている. 今後, 多様なソフトウェアの分析を通じて, より一般的な関係を導き出す必要があろう. さらに, 発展的なテーマとしてソフトウェアの機能が成熟しているかどうかを区別するための AE の閾値の考え方を整理することで, ソフトウェアの現場が判断しやすくするための研究を行うことが考えられる.

また, AE とプログラムの品質との関係の調査が挙げられる. 具体的には, 過度に広い範囲のアクセスを許可しているアクセス修飾子を設定している部品と, 最適なアクセス修飾子の設定になっている部品の間で, バグの検出率に有意差がでるかどうかを調べ, 有意差が出るような AE の閾値を見極めることができた後には, AE を用いたプログラムの品質判定方法を提案することで, プログラムのリリース判定を支援することを目指すことが考えられる.

謝辞

本研究を進めるにあたり, 検定手法の選択等において, 株式会社 NTT データの山本英之氏に多くの知識や示唆を頂きましたことを感謝致します.

参考文献

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

[2] K. Arnold, J. Goslin and D. Holmes,: “The Java Programming Language, 4th Edition”, Prentice Hall, 2005.

[3] J. Gosling, B. Joy, G. Steele, G. Bracha and A. Buckley,: The Java Language Specification, Java SE 7 Edition, http://docs.oracle.com/javase/specs/jls/se7/html/index.html

[4] K. Khor, N. Chavis, S. Lovett and D. White,: “Welcome to IBM Smalltalk Tutorial” , 1995

[5] Nghi Truong, Paul Roe, and Peter Bancroft,: “Static analysis of students’ Java programs”, In Proc. ACE ’04, 317-325, 2004.

[6] D. Quoc, K. Kobori, N. Yoshida, Y. Higo and K. Inoue,: ModiChecker: Accessibility Excessiveness Analysis Tool for Java Program,日本ソフトウエア科学会大会講演論文集 vol.29, pp.212-218, 2012, コンピュータ・ソフトウェア.

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

[8] 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.

[9] T. 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, 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, 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.