社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

テストケースを利用した Javaプログラムのアクセス修飾子過剰性分析手法

大西　理功† 　小堀　一雄†† 　松下　誠† 　井上　克郎†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1 番 5 号
†† 株式会社 NTT データ 技術開発本部
〒 135-8671 東京都江東区豊洲 3-3-9
E-mail: †{r-ohnisi,matusita,inoue}@ist.osaka-u.ac.jp, ††koborik@nttdata.co.jp

あらまし　Java プログラム内には，実際の被アクセス範囲に比べて過剰に広く設定されているアクセス修飾子 (Accessibility Excessiveness，AE) が多数存在することが知られている．しかし，既存の AE 検出手法では AE の発生理由が考慮されていないため，検出結果には障害となりうる AE 以外に設計者の意図により生まれた AE も含まれてしまっていた．そこで本報告では，意図的なアクセスを自動的に判別して AE から除外して障害となりう る AE の適合率をあげることを，テストケースを用いることで実現する．また，ソースコードのテストカバレッジとその AE 変化の関係を調査した．その結果，テストケースにより意図的に作られた AE を発見することができた．また，調査対象のソフトウェアについては，AE メソッドのテストカバレッジが高いものほど，修正されていることが分かった．
キーワード　アクセス修飾子，Java プログラム，開発履歴，設計情報，テストケース，テストカバレッジ

Analysis of Accessibility Excessiveness in Java Programs Using Test Cases as Design Information

Riku OHNISI†, Kazuo KOBORI††, Makoto MATUSITA†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
Suita, Osaka 565-0871, Japan
†† Research and Development Headquarters, NTT DATA Corporation
Toyosu 3-3-9, Koto-ku, Tokyo 135-8671, Japan
E-mail: †{r-ohnisi,matusita,inoue}@ist.osaka-u.ac.jp, ††koborik@nttdata.co.jp

Abstract　In Java Program, we have found there are many access modifiers declared as wider scope than its real access, and we named Accessibility Excessiveness (AE) to such modifiers. In previous AE detection method, we failed to exclude developer-intentional AE from the result, since the method cannot consider why modifiers are in AE. In this paper, we propose a detection method that identifies developer-intentional AE by using test cases. We also investigated if there are some relations between the test coverage of source code and its AE status. Consequently, we succeeded to find developer-intentional with the proposed method. And we also found that the higher the coverage of the AE methods in subject software, the more AE is fixed during its development.
Key words　Access Modifier, Java Program, Development History, Software Design, Testcase, Test Coverage

1. はじめに

現在のソフトウェア開発においては，大規模化や短納期化などに伴い，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が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場合がある．その場合，例えば Java を用いた開発の場合には，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ってしまう可能性がある．

こういった問題を防ぐために，Java ではアクセス修飾子（Access Modifier）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こと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．

我々はこれまで，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に解析して，過剰に広い範囲に設定された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．さらに ModiChecker の分析対象として，どこからも参照されていないフィールドやメソッドを含めることにより，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 [7]．

一方で，アクセス修飾子過剰性には設計者が意図したものとそうでないものが考えられるが，既存研究ではその区別がなされていなかった．そこで，UML で表される設計情報を用いて設計者が意図したアクセス修飾子過剰性を除去することを試みた．しかし，広く入手可能な、設計情報を含むソースコードは少なかったため，十分な分析を行うことができなかった [8]

本研究では、テストケースをソフトウェアの設計情報と考え，テストケースからアプリケーションへの静的アクセスを分析することにより，アプリケーション単体と比べアクセス修飾子過剰性にどのような変化があるのかを調査した．その結果，アプリケーションのソースコード中のメソッドにおいて，全体として 1 割程度の AE が変更され，その内訳としてアクセスされていない状態を表す NoAccess が半分以上を占めていることがわかった．また，テストケースからのアクセスは NoAccess なメソッドを解消するために大きく寄与していることが分かった．

さらに，ソフトウェアのバージョン間のアクセス修飾子過剰性の変遷とテストカバレッジの関係について調査した．その結果，テストカバレッジ 0%，ついで 100%の AE メソッドが多く，それだけでメソッドの 9 割以上を占めていることがわかった．2 バージョン間で変更された AE メソッドはごく僅かであったが，同程度のテストカバレッジを持つメソッドの中では，テストカバレッジ 100%のメソッドが最も多く変更されていることもわかった．

本論文の構成について説明する．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を行うツール ModiChecker とそれを用いた分析について述べる．4 節ではテストケースを用いたアクセス修飾子過剰性の分析について述べ，5 節では分析結果における考察について述べる．6 節で関連研究について，7 節で本論文のまとめと今後の研究について述べる．

2. アクセス修飾子によって引き起こされる問題

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指示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（Access Modifier）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	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

表 2　AE の種類

* / **	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

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

クラスの外部から直接参照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やメソッドは，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，その利用範囲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必要な範囲以外からのアクセスが可能となるようなアクセス修飾子を設定することがある [5]．

3. Accessibility Excessiveness 検出ツール 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 にて青色の背景色がある位置に示す pub-pro, pub-def, pub-pri, pro-def, def-pri, pub-na, pro-na, def-na, pri-na の 10 種類を AE として定義する．表中の No Access は，対象のフィールドまたはメソッドが実際にはアクセスされていないことを示す．例えば pub-pro 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，pub-pub，pro-pro，def-def，pri-pri 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 × と表示されている箇所は，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．

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，現在宣言されているアクセス修飾子と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をリスト形式にて出力する．これを用い，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．

ModiChecker を用いた既存研究として，ソフトウェアのアクセス修飾子の修正状況における分析を行った [7]．具体的には，ソフトウェア開発の履歴を対象に，AE に対する修正作業の実行頻度について分析した．その結果，AE 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 AE については，分析対象の全てのプロジェクトにおいて修正が行われていることもわかった．また、ソフトウェアの AE を修正すべき AE と修正すべきでない AE に分類し，後者を除去することが可能かどうか分析を行った [8]．ソフトウェアの設計情報を用いて，設計者の判断により生じた修正すべきでない AE を判別する方法を提案し，既存のソフトウェアを対象に適用し，分析を行った結果，一部の AE は除去することができた．しかし，入力に持ちいたソフトウェアに付随する設計情報の内容が不完全だったために，検出できなかった AE も存在した．

4. テストケースを用いたアクセス修飾子過剰性分析

既存ツールである ModiChecker は，AE となっているフィールドやメソッドに対して，実際の被アクセス関係をカバーする最小限のアクセス範囲となるアクセス修飾子を推薦・修正する機能が存在する．しかしこの機能では，静的解析したプログラム内でのアクセス関係のみで判断を行うため，プログラム外からのアクセスを考慮した上で開発者が意図的に設定した AE は考慮されていない．

例えば，MVC(Model View Controller) モデルを実装したプログラムでは，ユーザからの入力は View を通じて Controller を呼び出し，Controller がユーザからの入力に対応した必要な Model を呼び出すという関係にある．また，MVC モデルを Java で実装する場合，Model は開発者が Java で実装するが，Controller はフレームワークによって予め実装されていることが多く，View は Java 以外の方法，具体的には HTML や JSP を用いて実装されることがある．このような場合，ModiChecker に対して開発者が実装した Model に属する Java ソースコードのみを入力として与えても，Controller や View からのアクセスは解析対象外となるため，Controller や View からのアクセスを考慮して意図的に設定されたアクセス修飾子は AE だと誤って判断される．このような誤った判断を防ぐためには，開発者が実装したソースコードだけでなく，Controller や View からのアクセスが記載されると予測される設計情報まで解析対象を広げることにより，ソースコードの解析だけでは判断できないアクセス関係を考慮して AE の解析を行う必要がある．

そこで以前の研究では，設計情報として UML のクラス図を用いた AE の分析を行った [8]．ここではクラス図の各メソッドに対して設定されたアクセス修飾子情報を用いて，意図的な AE を除去することを考えた．その結果，分析対象のソフトウェアではメソッドのアクセス修飾子がクラス図に記載されており，クラス図の情報が劣化なく実装されていたため，クラス図を伴わないソフトウェアと比べて AE の数が少なかったが，クラス図を参照することで，存在していた意図的な AE はすべて除去することが出来た．このことから，クラス図を持ちいることによって AE の抽出精度は高くなる可能性が示された．しかし，UML のクラス図とソースコードがセットになって公開されているオープンソースのソフトウェアがほとんど存在しなかったため，十分な分析結果を得ることができなかった．さらに，実験に用いたソフトウェアのクラス図にはメソッドやクラス間のアクセス関係が記載されていなかったため，メソッドのアクセス関係を取得することができなかった．

そこで本研究では設計情報としてテストケースを用いることとした．テストケースはソフトウェア自身が正しく動作するか確認するために用いられることから，設計書と異なり未完成な部分が少ないと考えられ，したがって設計内容を確実に反映していると考えられる．また，UML のクラス図と異なりテストケースを伴うソフトウェアは数多く公開されているため，多様なソフトウェアに対して分析を行うことが期待できる．本研究では，テストケースのアプリケーション部分への参照状況を利用する．テストケースからのアクセスを考慮することで，アプリケーション部分の被アクセス範囲が広くなり，結果として AE を解消できると考えた．また AE の中には アプリケーション部分での被アクセスのないもの，つまり，フレームワーク部分からのアクセスのみを想定しているメソッドも存在する．このように設計者が意図して作りこんだ AE は NoAccess に多いと考え，テストケースを用いることで除去できると考えている．なお，実行中にアクセス修飾子を変更するリフレクションを用いたソフトウェアは，アクセス範囲を正しく解析できない可能性を含むことから，解析対象とはしない．

また，アプリケーション開発時において AE が修正される時期が分かれば，AE 除去の支援を行ううえで大きく貢献できる．既存研究にてソフトウェアのメソッドのアクセス修飾子の修正状況における分析が行われており，その結果新規作成メソッドの多くは NoAccess であることがわかった．また新規作成メソッドはアクセス範囲が決まっていない場合が多く，利用する側のメソッド数も少ないことがわかっている [7]．このことから，メソッドが利用されるに従って，AE が修正される可能性がある．本研究では，AE が修正される原因を調べるために，メソッドが利用されることの一つの判断として，どの程度テストされているかを表すテストカバレッジと，AE の修正状況の関係性について調べることとした．

以上をふまえて，本研究では以下の二つの RQ を設定した．

RQ1　テストケースを AE 解析対象に含めることで意図的に

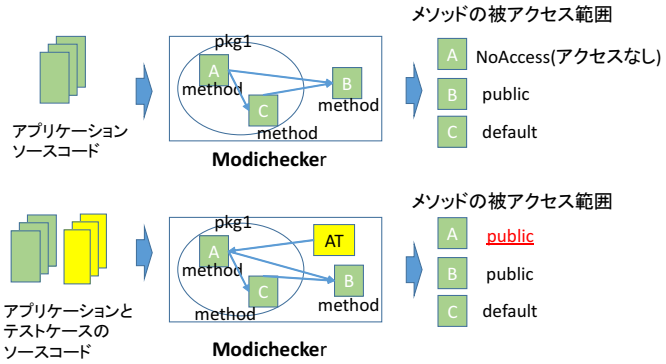

図 1　テストケースからのアクセスを含めた AE 解析

Fig. 1　AE analysis considering access from testcase

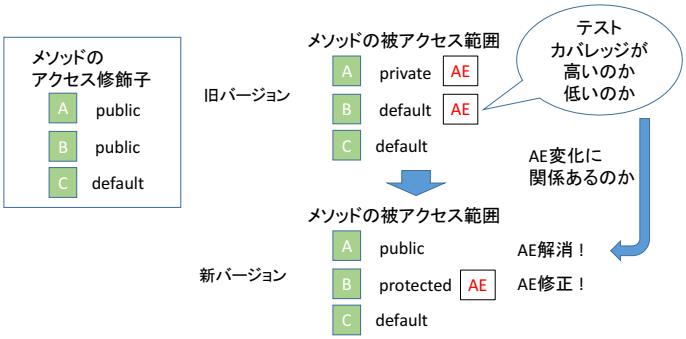

図 2　バージョン間の AE 変化とテストカバレッジ

Fig. 2　AE variation between 2 versions and Coverage

設定された AE を発見できるか

RQ2　バージョン間での AE 変化とテストカバレッジにどのような関係があるのか

RQ1 については，ModiChecker を用いて同じソフトウェアについてテストケースを含まない部分とテストケースを含んだ部分のそれぞれの AE 解析を行い，結果を比較する．テストケースからのアクセスを解析対象に含めた場合と含めない場合の違いを図 1 に示す．テストケースを含んだ部分を解析すると，含まない部分を解析するのに比べて，解析対象としてテストケースからのアクセスが加わる．これにより，メソッドやフィールドの被アクセス範囲が，例えばテストケースが異なるパッケージからアクセスしている場合，図 1 のメソッド A では，NoAccess から public に変化する．このように，テストケースのあるなしで被アクセス範囲が変化するので，被アクセス範囲が広くなる全ての場合において，どれだけ変化したかを求める．また，被アクセスのない NoAccess については，アクセスが存在するようになったか否かについてのみ考える．これは，NoAccess であるメソッドからアクセスされていないものについては，Java プログラム外からのアクセスや将来の機能拡張を考慮して作られた可能性があり，その中でも，テストケースからアクセスされているものについては，現在利用されていることから Java 外部のプログラムからアクセスされていると考えられるからである．つまり，NoAccess であるメソッドがテストから参照されているということは，AE が解消されていることと同義であるため，アクセスの有無についてのみ考えている．

RQ2 については，ソフトウェアの二つのバージョンを用意し，前バージョンのソフトウェアのテストカバレッジを求める，それにより，メソッドをテストカバレッジの高さを用いて分類する．二つのバージョンに対してそれぞれの AE 状況を求め，テストカバレッジによって分類されたメソッドに対して AE が修正されたか否かを判定する．テストカバレッジの値によって，AE が修正される割合を求める．二つのバージョン間で AE が変化する場合の例を図 2 に示す．この例では，古いバージョンで public と宣言され AE であると判断されたメソッド A が，新しいバージョンではアクセス修飾子に変化がないものの AE ではなくなっていることを示している．またメソッド B については，アクセス修飾子が変化したにもかかわらず AE として判定されている．

5. 分 析 結 果

RQ1 および RQ2 の答えを得るために，二つの実験を行う．

5.1 実　験　1

5.1.1 実 験 方 法

RQ1 について調べるために，3 つのオープンソースソフトウェア (Ant, Struts, Javassist) を対象にした実験を行う．ソフトウェアごとに，図 1 のように，入力としてテストケースを含まないソース群 A とテストケースを含む全てのソース群 B の 2 種類を用意する．その際，各ソフトウェアのソースコードのうち，test フォルダに含まれている Java ファイル，または任意のフォルダに含まれる末尾が Test.java のファイルをテストケースと判断した．A，B のそれぞれを ModiChecker の入力として与え，AE 解析を行う．解析結果について，AE の中で被アクセス範囲が広がったものを，どの範囲からどの範囲に広がったか（例えば，被アクセス範囲が private から default に変化という分類のもとでまとめる．ただし，被アクセスの存在しない NoAccess については，アクセスが存在するようになったか否かについてのみ考える．

5.1.2 実 験 結 果

実験 1 の結果を図 3 に示す．これは Ant, Struts, Javassist のそれぞれのメソッド総数，テストケースを含めていないときの AE メソッド数，テストケースを含めたときの AE メソッド数，テストケースを含めることで変化したメソッドの AE の数，そして AE が変化したメソッド中で，AE が解消された NoAccess 以外の AE の数，AE が解消された NoAccess の数，AE が解消されなかったが修正された AE の数を記載している．図 3 から，RQ1 としては，テストケースを用いることで，除去できている AE が存在するため，意図的な AE を発見できるということが言える．特に，意図的な AE の中には NoAccess が多そうと考えていたが，図 3 からもそれを否定しない結果が得られた．また，テストケースを含めることで変化した AE の数は 1 割から 2 割程度であるが，その内訳として 3 つのソフトウェアすべてにおいて，解消された NoAccess の数が最も多く 57.8%から 82.6%の割合を占めていることが明らかとなった．

	Ant 1.9.4	Struts 2.3.16.3	Javassist 3.18.2
総メソッド数	12498	10907	4442
AEメソッド数(テスト除く)	3947	3660	746
AEメソッド数(テスト含む)	3878	3548	642
変化したAE	306	508	147
AE解消(NoAccess除く)	45 (14.7%)	44 (8.7%)	40 (27.2%)
NoAccess解消	177 (57.8%)	420 (82.6%)	91 (61.9%)
AE修正(解消はされない)	84 (27.5%)	44 (8.7%)	16 (10.9%)

図 3　テストケースの有無による AE 変化
Fig. 3　variation of AE within existence of testcase

5.1.3　考　　察

テストケースからのアクセスは，NoAccess であるメソッドの解消，つまり Java 外部からのアクセスを含む意図的な AE を除去するために，大きく寄与している．しかし，全体の AE メソッドに対して，除去される AE の割合は 1 割にも満たない．つまり，テストケースからのアクセスを解析に含めてもごく僅かしか AE は除去されない．テストケースからアクセスされていない NoAccess であるメソッドが多く存在するが，これは本当に利用されていないメソッドか，あるいは重要度が低いためにテストされていないフレームワーク用のメソッドであると考えられる．

5.2　実　験　2

5.2.1　実 験 方 法

RQ2 について調べるために，Apache Ant の 2 バージョン (1.8.2, 1.8.4) を対象に実験を行う．ソフトウェアのソースコードを準備し，2 バージョンのテストケースを含む全てのソースコードに対し AE 解析を行う．解析後に，図 2 のように，Ant 1.8.2 の AE メソッドに対し，Ant 1.8.4 で被アクセス範囲が変化したものを取り出す．一方で，Ant 1.8.2 を対象にテストカバレッジを計測する．テストカバレッジを計測するために JUnit を用いた．解析により求めた AE メソッドを，テストカバレッジが高い (50〜100%) もの，低い (0〜49%) ものに分類し，それぞれの個数を求める．前述した 2 バージョン間で AE が変化したもの（解消された AE，解消された NoAccess，修正された AE メソッド）を，それぞれをテストカバレッジの値によって分類する．同じテストカバレッジの分類に属する，AE が変化したメソッドと AE が変化していないメソッドの割合を求める．これにより，テストカバレッジの高低によって，AE が変化したメソッドの数が変化するかどうか分析し，AE 変化とテストカバレッジの関係を確認する．

5.2.2　実 験 結 果

実験 2 の結果を図 4 に示す．この結果，テストカバレッジが高い AE メソッドは低い AE メソッドより数が少ないにも関わらず，あとのバージョンでより多く修正されていることが分かる．つまり，実験対象のソフトウェアについては，テストカバレッジが高いメソッドほど頻繁に修正されている．

5.2.3　考　　察

実験で用意した 2 バージョンで，テストカバレッジが高いメソッドほど，頻繁に修正されていることが分かった．「テストカバレッジが高いほど，AE が修正されにくい」という帰無仮説に対して，標本数が十分ではなかった．仮説の正しさを確認するためには，より多くの標本に対して実験を行う必要がある．

AEメソッド (個数)	テストカバレッジによる分類	
	0〜49%	50〜100%
Ant 1.8.2 (3879)	2510	1369
Ant 1.8.2->1.8.4 で修正されたAEメソッド (33)	12 (0.47%)	21 (1.53%)

図 4　バージョン間の AE 変化とテストカバレッジ
Fig. 4　AE variation between 2 versions with coverage

また，実験 1 と関連するが，Ant のテストカバレッジが，一般的なソフトウェアシステムと比べてそれほど高くないことが分かった [9]．このため，実験 1 でテストケースからのアクセスを考慮したが，テストカバレッジが低いためにそもそもアクセスされない部分が数多く存在している．このためにテストアクセスを含めても AE メソッドがあまり変化しなかった可能性が考えられる．

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，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 [10]．この研究では，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では，コンパイル時に追加されるフィールドやメソッドが存在するために，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，評価を行った．

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 [11]．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 [12]．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 らによって議論が行われている [13]．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 [14]．この研究では，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 [15] [16]．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 [17]．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を持っていない．

7. まとめと今後の課題

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） テストケースの有無による AE 変化の調査

（2） バージョン間での AE 変化とテストカバレッジの関係性の調査

テストケースの有無による AE 変化の調査については，テストケースを AE 解析の対象に含めることで，NoAccess の解消が最も多く起きることがわかった．これにより，Java 外部からのアクセスをはじめとした意図的な AE の発見には，テストケースが有用であることが分かった．また，テストケースを含めることで変化した AE であるメソッドの数は全体の 1 割から 2 割程度であるが，その中では解消された NoAccess の数が最も多く 57.8%から 82.6%の割合を占めていた．一方で，テストケースからもアクセスされていない NoAccess であるメソッドの数が多く，なぜこのような結果になったのか，その原因を調査する必要がある．

また，AE 変化とテストカバレッジの関係性の調査については，バージョンアップによって修正されたメソッドの中では，テストカバレッジの高い AE メソッドが低い AE メソッドに比べて多く修正されていることがわかった．しかし，この実験で用意した 2 バージョンでは，修正された AE メソッドがかなり少ないため，この結果だけではテストカバレッジと AE メソッドの修正されやすさの関係を，他のソフトウェアに対して適用することは難しい．そのため，より多くのバージョンやソフトウェアに対して分析を行う必要がある．今回のようにテストカバレッジの高いメソッドが最もよく修正されている場合において，バージョン間での修正されるメソッドが多くなれば，AE 変化とテストカバレッジにより一般的な関係性が見られるかもしれない．

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] 石居達也 小堀一雄 松下誠 井上克郎: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，情報処理学会研究報告 Vol.2013-SE-180, No.1, pp.1-8 2013

[8] 大西理功 小堀一雄 松下誠 井上克郎:Java プログラムにおける設計情報を用いた意図的なアクセス修飾子過剰性の抽出手法，情報通信学会研究報告 Vol.2013-SS-79, No.8, pp.43-48 2013

[9] http://www.unisys.co.jp/tec_info/tr93/9306.pdf

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] FindBugs, http://findbugs.sourceforge.net/

[16] Jlint, http://jlint.sourceforge.net/

[17] 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.