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.

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

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より広い範囲が設定された状態である，アクセス修飾子過剰性 (Accessibility Excessiveness, AE) が多数存在することが既存研究により判明している．しかし, 既存研究では, AE が設計者の意図により生まれたかどうか判別できていない．そこで本研究では, Java プログラムの設計情報を用いて, 設計者の意図により生じた AE を判別する手法を提案する．また, 既存のソフトウェアに適用することで，本手法の有効性を考察する．

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

A Detection Method for Intended Accessibility Excessiveness in Java Programs Using 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 that have the status Accessibility Excessiveness(AE) which is declared more extensive area than its real access. In previous research, we couldn't decide whether an AE was made by designer's intention or not. In this paper, we propose a detection method for intended Accessibility Excessiveness in Java programs using design information. We also discuss the validity of the method by applying to an existing example software.

Key words　Access Modifier, Java Program, Development History, Software Design

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

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

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 の分析対象に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，この既存研究ではその区別が考慮されていなかった．本研究では，設計者の意図はソフトウェアの設計情報に現れると考え，その中でもアクセス修飾子の設定に関係のある UML クラス図を分析することで，ModiChecker にアクセス修飾子過剰性を意図的なものとそうでないものに分類する機能を追加した．機能を実装するに当たり，ソフトウェアの設計情報を用いることで設計者の意図したアクセス修飾子の検出を行う手法を提案する．

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するツール ModiChecker について述べる．4 節では意図的な AE を検出する手法について，5 節では，開発ツールによるソフトウェア内の意図的なアクセス修飾子の分析について述べる．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].

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]．この問題を以下に示す 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た後メソッド 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 を用意した．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 に設定した．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，アクセス修飾子を 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，開発途中において開発者間で設計情報が共有されていない場合，想定外の状況下でメソッドが呼ばれてしまい，論理的なバグ発生の原因が作られる．しかし，この状況は Java の構文上問題がないため，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

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

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 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，pub-pub, pro-pro, def-def そして 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．

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，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 (以下, AE リストと呼ぶ) で出力する．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．

4.　意図的なアクセス修飾子の検出手法

既存ツールである ModiChecker には，入力として与えられる Java プログラム内のフィールドやメソッドに対するアクセス関係を静的解析により読み取り，AE となっているフィールドやメソッドの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を持つ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が考慮された AE が出力され，プログラム外からのアクセス関係は考慮されず，結果としてプログラム外からのアクセスを考慮した上での意図的な AE に関する判別は行われない．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 で実装した場合，Controller と Model は Java で実装され，View は HTML や JSP といった別の形式で実装されることがある．この場合，ModiChecker に Controller と Model を入力として与えても，View からのアクセスは解析対象外となるため，View からのアクセスを意図して設定されたアクセス修飾子は AE として判断されることになる．このような場合，ソースコードだけでなく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入手した上で AE の解析を行う必要がある．

そこで，ModiChecker から出力される AE に対して，設計者の意図するアクセス修飾子と意図しないアクセス修飾子を分類することで，本当に修正しなければならない AE（意図的でない AE）を開発者に推薦する際の適合率を上げることを試みた．意図的な AE は，設計者が意図的に広い範囲となるよう設定したアクセス修飾子を指しており，開発者が設計情報やその他の情報を参照することでアクセス修飾子で指定されたアクセス可能範囲が意図されているかどうかを判別可能であるとする．AE リストから全ての意図的な AE を除去することができれば，既存の ModiChecker の修正機能を用いることで，不要な AE を全て除去することができる．

本研究では，意図的でない AE の再現率を落とすことなく適合率を伸ばすことを目的とする．具体的には，既存の ModiChecker が求めた AE リストから意図的な AE をリストから取り除き，修正が必要なものの割合を大きくすることを目指す．また，提案手法により，設計情報から分類できなかった意図的な AE については，設計情報そのものに対するフィードバックとなりうると考える．本来，設計者の意図する AE は，設計情報に記述されているべきであり，意図的な AE を設計情報に反映させることで，設計情報を，ひいてはソフトウェアをより高品質なものにすることができる．

以下，提案手法の具体的な処理について述べる．

4.1　設計情報の利用

AE から意図的なアクセス修飾子を検出するために，設計者の意図が反映される設計情報を利用する．設計情報のモデルとしては，一般的に普及している UML を用いる．

設計情報からフィールドやメソッドのアクセス修飾子の情報を取得するために，クラス図を用いる．また，フレームワークなどとの動的なアクセス関係を取得するために，シーケンス図を用いる．本研究では，これら 2 つのモデル図を利用し，設計者の意図したアクセス修飾子の検出を行う．

UML モデルの設計情報は基本的に図表現により与えられるため，設計情報を文書として処理するために文字列に変換する．ここでは UML モデリングツール astah* [8] を利用し，UML モデル図を XML 出力する．ただし，XML 形式で出力される情報には，UML モデルを管理する astah* のプロジェクト情報が含まれるため，必要な情報のみを抽出する必要がある．後述する ModiChecker の追加機能にて，XML から必要な情報を抽出する処理を実装している．

4.2　ModiChecker の追加機能

ModiChecker の追加機能として，従来の ModiChecker の出力である AE リストと，設計情報を解析するアナライザ機能を実装した．アナライザの機能としては，前述したように，ModiChecker の出力する AE には意図的な AE と意図的でない AE が混在しているため，AE リストから意図的な AE を判

別・除去する役割を担っている．

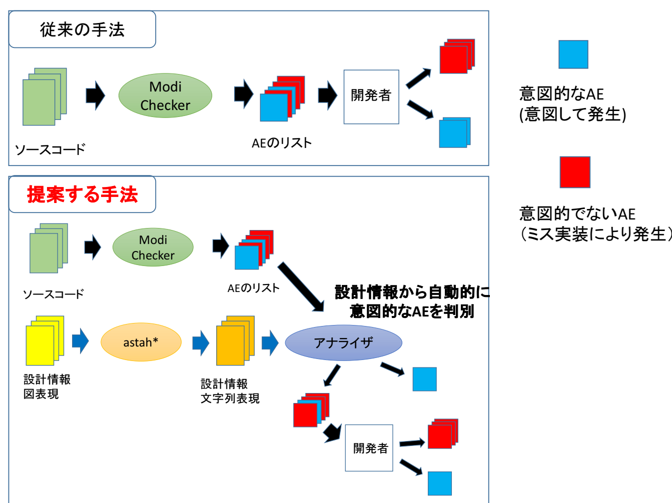

図 1　意図的な AE の検出手法
Fig. 1　detection method of intended AE

図 1 は，設計情報を用いて AE リストから意図的な AE を検出する手法の概略図であり，従来のソースコードのみを用いた場合との対比を示している．従来の場合は，開発者がリストの全ての AE が意図的な AE か否かを判断していたのに対し，提案手法では，AE リストから自動的に意図的な AE の一部が取り除かれる．

設計情報を用いて，AE リストから意図的な AE を判別・削除するアナライザについて説明する．アナライザは，ModiChecker から AE リストを，astah*から設計情報を受け取り，フィールドやメソッドのアクセス関係が乖離している部分，ソースコードと設計情報のアクセス修飾子が異なっている部分を探し出す．これは，AE となっているアクセス修飾子を対象とし，ソースコード上のアクセス関係から推薦されるアクセス修飾子が，設計情報におけるアクセス修飾子すなわち設計者の意図したアクセス修飾子と異なっている場合，当該 AE を修正する必要のないものと判断している．アナライザは，AE リストのうち，設計情報から判別できる意図的な AE を除外したものを，ModiChecker の利用者に提供する．これにより，設計情報を用いて分類できる一部の意図的な AE が取り除かれるので，開発者が確認する AE リストのうち，修正の必要ないものを減らすことが出来る．

4. 2. 1　アナライザ

アナライザの処理は，大きく分けて 2 つある．1 つ目は astah* で記述された設計情報を解析し，フィールドやメソッドのアクセス修飾子および UML 図上での接続関係などを抽出をする処理である．2 つ目は，設計情報と ModiChecker より得られる AE リストのフィールドやメソッドのアクセス修飾子の比較を行い，該当フィールドやメソッドが意図的な AE かを判断し，そうであれば，AE リストから除去する処理である．

1 つ目の処理は，対象 Java のソフトウェアの設計情報からアクセス修飾子などの必要な情報を抽出するという処理である．クラス図のモデルを XML 形式に変換したものを対象に，フィールドやメソッドのアクセス修飾子と所属クラスを取得する．

2 つ目の処理は，ModiChecker から得られる AE となっているフィールドやメソッドに対して、設計情報の該当するフィールドやメソッドを所属クラスにより判別し，ソースコード上で設定されていた該当フィールドやメソッドのアクセス修飾子と設計情報のそれを比較する．これらのアクセス修飾子が一致している場合，該当フィールドやメソッドについて，ソースコードが設計者の意図に従って実装されているとして，意図的な AE と判断する．そして，意図的な AE と判断されたフィールドやメソッドについては，ModiChecker から得られていた AE リストから除去する．

5.　提案手法を用いたアクセス修飾子の意図分析

4 節で説明した意図的なアクセス修飾子の検出手法を用いて，実際のプロジェクトで発生している AE に対し，検出手法により意図されない AE がどの程度削除されることなく維持されるのか (再現率)，および手法を適用した前後で全ての AE に対して意図されない AE の割合はどの程度になるのか (適合率) について分析を行った．今回分析を行うにあたって，2 つの研究課題を設定した．

RQ1　検出手法により意図的でない AE が削除されることなく，すべて検出されるかどうか

RQ2　検出手法の適用前後で，最終的に提示される AE のうち，意図的でない AE はどの程度含まれるか

本研究では，文部科学省の助成事業 enPiT [9] のプログラムの 1 つである，“enPiT Cloud”の演習で用いられた Java のプログラムと UML モデルの設計情報を伴うソフトウェア “EventSpiral” を実験対象とした．以下，RQ1,RQ2 についての分析手順をそれぞれ示す．

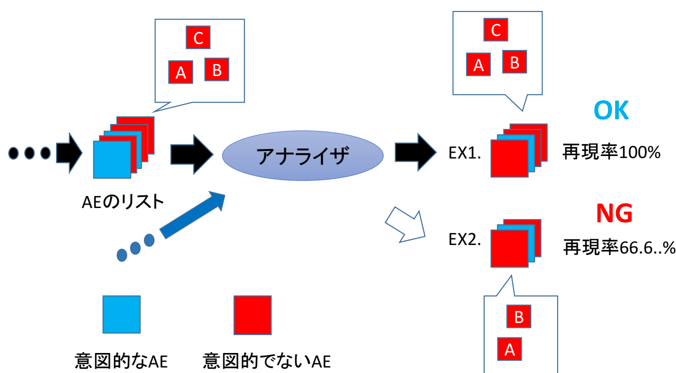

図 2　RQ1
Fig. 2　RQ1

RQ1 に着目した，アナライザの適用前後での AE リストの変化を図 2 に示す．RQ1 はアナライザの適用前後で意図的でない AE が削除されることなく検出されるかを確認する．ソースコードの任意のフィールドやメソッドをランダムに選定して，アクセス修飾子を編集することで故意に意図的でない AE を生成する．生成した AE を含めた AE リストをアナライザに適用

し，意図的でない AE が検出されるかを確認し，RQ1 を分析する．

RQ2 はアナライザ適用前後での AE の中の意図的でない AE の適合率を比較することで，分析を行うことを考えている．しかし，AE が意図的でない AE かどうかの判断ができないため，正確な適合率を求めることができない．なぜなら，現在 AE が意図的な AE かどうかを設計情報のみで判断しているが，設計情報が意図を完全に表現していることを保証できないからである．そこで，検出手法により意図的でない AE が削除されることなく検出されることを前提に，アナライザ適用前後での AE の数の変化を比較することで，適合率の変化を求める．

5.1 実 験 準 備

本実験で，対象とするソフトウェア “EventSpiral”について簡単に説明する．本ソフトウェアは，イベント情報を登録・閲覧し，開催されるイベントの参加チケットなどの購買する機能を提供するウェブサービスを実現する．本ソフトウェアのソースコードは，MVC モデルで実装された Java ソースコードによって成り立っている．また，設計情報として，UML モデルを利用している．基本的な機能は Java プログラムで実装されているが，フレームワーク処理などクライアント側とのインタフェース部分やアカウント情報などデータを管理するデータベース部分は JavaScript や MongoDB など他の言語で実装されている．

本実験では，RQ1，RQ2 の分析を行うにあたり，再現率，適合率の順に実験を行った．

5.2 RQ1 の分析実験

5.2.1 実 験 方 法

RQ1 を分析するにあたり，ランダムに選択した AE でないフィールドやメソッドを故意に意図的でない AE に変更し，変更した AE を含む AE リストを対象に 3 回のテストを行った．テストではアナライザの適用前後で AE の数を確認する．AE でないメソッドやフィールドのうち，AE に変更可能，すなわちアクセス修飾子の設定範囲を広げることができるものが，メソッドに 3 個，フィールドに 63 個確認できた．1 回目は 3 個のメソッドを，2，3 回目のテストはそれぞれ 10 個のフィールドを対象に行った．

5.2.2 実 験 結 果

表 3 再現率についての実験結果

テスト	故意に作成した意図しない AE のメソッドあるいはフィールドの数	アナライザ適用後の故意に作成された AE のメソッドあるいはフィールドの数
1	3	3
2	10	10
3	10	10

表 3 は，メソッドやフィールドの意図的でない AE がアナライザの適用によって，どの程度検出されたかを示している．今回の実験では，再現率は 100 %を示しており，RQ1 について検出手法により意図的でない AE が削除されることなく，すべて検出されることを示す．

5.3 RQ2 の分析実験

5.3.1 実 験 方 法

RQ2 を分析するにあたり，アナライザの適用前後での意図的でない AE の候補数を求めた．

5.3.2 実 験 結 果

表 4 メソッドにおける適合率についての実験結果

状況	全 AE の数	意図的でない AE の候補数
アナライザ適用前	92	92
アナライザ適用後	0	0

表 4 より，メソッドにおいて，AE はすべて意図的な AE という結果が得られた．アナライザにより、意図的でない AE の候補が全て設計情報により意図的な AE と判断できる，つまりソースコード上のアクセス修飾子と設計情報におけるアクセス修飾子が一致していたことを示す．

表 5 フィールドにおける適合率についての実験結果

状況	全 AE の数	意図的でない AE の候補数
アナライザ適用前	28	28
アナライザ適用後	28	28

表 5 より，フィールドの AE は，いずれも意図的な AE とアナライザで判断できないことが分かる．AE のうち，18 個は例外処理クラスの中で宣言されている．設計情報のクラス図において，例外処理クラスが存在しない．また，残り 10 個のフィールドのアクセス修飾子は，設計情報のそれと異なる．

5.4 分析結果の考察

RQ1 について，アナライザにより意図的でない AE が取り除かれないことを高い水準で保証しているように考えられる．ただ，設計情報に記述されていないクラスやメソッド，フィールドを想定し，そういったオブジェクトをどのように分類するかは考える必要がある．ソースコードから設計情報へのリバースエンジニアリングを考えるのであれば，設計情報に漏れている情報を付け足すことで，設計情報の品質を高めることが可能である．また，今回の実験で，対象としたソフトウェアは 1 つのみであるため，アナライザの機能が検証されたとはいい難く，複数のソフトウェアに対して適用する必要がある．

RQ2 について述べる．メソッドの適合率は，すべての AE が意図的な AE という結果が得られた．意図的でない AE の候補数は初期段階では ModiChecker の出力の AE リストそのものである．開発者が確認することで AE が意図的かどうかを判断するのだが，メソッドについては全て設計情報で説明できるため，意図的でない AE は 0 個になる．一方，フィールドの適合率は，設計情報によりいずれの AE についても意図的な AE であるという説明ができなかったため，意図的でない AE の候補数はアナライザ適用前の 28 個から変化しなかった．そこで設計情報により，意図的な AE かどうかの判断ができなかったものについては，設計情報に情報が不足している可能性があるため，設計情報にフィードバックする必要がある．

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） 設計情報を用いて設計者による意図的な AE を検出する手法を提案した

（２） ModiChecker を拡張し，設計情報をもとに AE の中から本当に修正する必要がある AE を検出・分類する機能を開発し，その妥当性を検証した．

検証実験においては，本手法の適用により開発者に対して提示する AE について，意図的でない AE の再現率を落とすことなく，適合率を上げることに成功した．しかし，設計情報に記載のないものなど，意図的な AE の全てを検出するには到らず，結果として意図的でない AE の適合率１００％は達成できていない．検出できていない部分に関しては，網羅的なテストを行うことが証明されているテストコードを分析対象に含めるなど，現手法の拡張を考える必要がある．さらに，ソースコードの設計情報へのリバースエンジニアリングを行うことで，設計情報の品質向上を実現し，ソフトウェアの現場でより高い精度の保守・運用を実現する研究を行うことが考えられる．

今後の課題として 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時としていつが適切かを判断するための支援ツールの構築を行いたい．

謝辞 本研究は，日本学術振興会科研費基盤 (S)(課題番号 25220003) の助成を得た．

その他，記載されている会社名，商品名，又はサービス名は，各社の登録商標又は商標です．

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] astah*, http://astah.change-vision.com/ja/
[9] enPiT, http://www.enpit.jp/
[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.