ソースコードの静的解析による
ソフトウェア保守支援に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2013 年 7 月

小 堀　一 雄

研究業績一覧

主要論文

[1-1] 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, Vol.33, 2013.（学術論文，採録決定）

[1-2] D. Quoc, K. Kobori, N. Yoshida, Y. Higo and K. 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.（学術論文）

[1-3] 小堀 一雄, 山本 哲男, 松下 誠, 井上 克郎: “コードの静的特性を利用した Java ソフトウェア部品類似判定手法”, 電子情報通信学会論文誌 D, Vol.J90-D(4), pp.1158-1160, 2007.（学術論文）

[1-4] Kazuo Kobori, Tetsuo Yamamoto, Makoto Matsushita, Katsuro Inoue: “Classification of Java Programs in SPARS-J”, International Workshop on Community-Driven Evolution of Knowledge Artifact, Session 4-3, Irvine, CA, 2003.（国際会議録）

関連論文

[2-1] 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. (国内会議録)

[2-2] Dotri Quoc, Kazuo Kobori, Norihiro Yoshida, Yoshiki Higo, Katsuro Inoue: “Modi Checker：Accessibility Excessiveness Analysis Tool for Java Program”日本ソフトウェア科学会大会講演, Vol28, 6C-2, pp.1-7, 2011. (国内会議録)

[2-3] 小堀 一雄, 山本 哲男, 松下 誠, 井上 克郎: “メソッド間の依存関係を利用した再利用支援システムの実装”, 電子情報通信学会技術研究報告, SS2004-58, Vol.104, No.722, pp.13-18, 2005.（国内会議録）

[2-4] 小堀 一雄, 山本 哲男, 松下 誠, 井上 克郎: “類似性メトリクスを用いた Java ソースコード間類似性計測ツールの試作”, 電子情報通信学会技術研究報告, SS2003-2, Vol.103, No.102, pp.7-12, 2003. (国内会議録)

書籍

[3-1] 小堀 一雄, 茂呂 範, 佐藤 聖規, 石垣 一, 飯山 教史: “現場で使えるデバッグ & トラブルシュート Java編” 翔泳社, 2010. (共著書籍)
[3-2] 飯山 教史, 町田 欣史, 高橋 和也, 小堀 一雄: “現場で使えるソフトウェアテスト Java編” 翔泳社, 2008. (共著書籍)

内容梗概

社会におけるソフトウェアの重要性が高まってきた現在では，特に社会基盤や企業の基幹業務を担う大規模かつ複雑なソフトウェアの品質を保つことが重要である．特に，ソフトウェア保守はソフトウェアの全ライフサイクルにかかるコストのうち，多くの割合を占めるため，ソフトウェア保守を支援することが重要になっている．

ソフトウェア保守を効率的に進めるためには，保守対象のプログラムの性質や振る舞いを開発者が理解する必要がある．しかし，ソフトウェアの大規模化や複雑化が進むにつれて，人手で十分な理解を行うことが難しくなっている．一方で，コンピュータの計算能力は近年著しく向上しており，コンピュータによるソフトウェア保守の支援を目的としたソースコード静的解析が盛んに研究されている．

本論文では，ソースコード静的解析手法のうち，以下の2つの手法を提案する．

1. アクセス修飾子過剰性に関する解析
2. ソフトウェア部品間における類似性計測

1. については，ソフトウェアを解析し，フィールド及びメソッドの呼び出し関係をグラフ化することで実際の呼び出し元の範囲と,当該フィールド及びメソッドに宣言されているアクセス修飾子の呼び出し範囲の乖離を分析する手法を提案し,この乖離を自動的に解析・修正するツール ModiChecker を開発した．これにより，開発者は意図せずアクセス修飾子を過剰に広く設定してしまったフィールドやメソッドを検知でき，第三者が誤ってアクセスすることを事前に防止できる．また，同じソフトウェアの複数のバージョン間における過剰なアクセス修飾子の数の変化量を比較分析することで,過剰なアクセス修飾子がどのように発生し，どのように修正されていくのかを分析した．

2. については， ソースコード部品における Java 言語の予約語の出現回数に関するメトリクスと複雑性に関するメトリクスを計測し,2 つのソフトウェア部品間でこれらのメトリクス値の比較を行うことで高速に類似部品を検出する手法を提案する．これにより，従来の文字列比較を用いた手法にくらべて解析時に扱う情報量が格段に低減されるため，解析コストを低く抑えることが可能となる．提案した手法は類似性計測ツール Luigi として実現し，従来の文字列比較による類似分析を実装したツール SMMT と，今回開発した Luigi を用いて，同じソフトウェア群に対して類似分析を行い，類似測定に関する精度とコストを比較評価することで，提案手法の優位性を示した．

謝辞

本研究の全般に関し，常日頃より適切なご指導を賜わりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に，心から深く感謝申し上げます．
大阪大学基礎工学部 情報科学科および大阪大学大学院情報科学研究科コンピュータサイエンス専攻在籍中に，適切なご助言とご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 萩原 兼一 教授，楠本 真二 教授に深く感謝申し上げます．
本研究を行うに当たり，直接具体的なご助言とご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授，日本大学工学部情報工学科 山本 哲男 准教授，南山大学 情報理工学部 ソフトウェア工学科 横森 励士 准教授に心より御礼申し上げます．
本研究を行うに当たり，具体的なご助言とご指導を頂きました，奈良先端科学技術大学院大学 情報科学研究科 吉田 則裕 助教に御礼申し上げます．
大阪大学大学院情報科学研究科在学中，様々なご指導，ご協力を頂き，また，様々な相談に乗っていただいた，石居 達也 氏（現 株式会社 日立製作所），Dotri Quoc 氏（現 楽天 株式会社），山本 英之 氏（現 株式会社 NTT データ）に心より御礼申し上げます．
最後に，井上研究室の皆様のご助言，ご協力に御礼申し上げます．

目次

図目次

表目次

第1章　はじめに

1.1. ソフトウェア保守の重要性

ソフトウェアが様々な場所や用途で利用されている現在の社会において，ソフトウェアの担う役割は非常に大きくなっている．特に，社会基盤や大企業の基幹業務を担うような大規模ソフトウェアを高品質に開発，保守することは非常に重要になっており，そのようなソフトウェアが障害を起こした場合，国民や企業の経済的，身体的損失など社会的に大きな影響がでる．また，社会のニーズや制度の変化に対応するため，ソフトウェア開発および保守には高品質だけでなく，高生産性が求められるようになっている．そこで，ソフトウェア工学の分野でも，ソフトウェア開発と保守の品質と生産性を高めるための支援が重要となっている．

ソフトウェア保守は，ソフトウェア開発に比べて長期間実施されることが多く，ソフトウェア保守にかかるコストはソフトウェアの全ライフサイクルにかかるコストの 3 分の 2 を占めるとの報告がなされている[1-1, 1-2]．また，日本情報システム・ユーザー協会（JUAS）によると，開発から 20 年以上保守され続けている基幹系業務システムの存在が報告されている[1-48]．これらのことから，まずはソフトウェアの保守をより効率的に行うことが重要であることが分かる．

次に，ソフトウェア保守の定義と分類について述べる．ソフトウェア保守とは，“ソフトウェアの納入後，ソフトウェアに対して加えられる，欠陥の修正，性能などの改善，変更された環境に適合させるための修正”のことを指す[1-4]．この“修正”はその目的から訂正と改良の 2 つに分類され，さらにソフトウェア保守は，以下の 5 つに分類することができる[1-5，1-6]．修正の分類とソフトウェア保守の分類の関係は図 1.1 に示したとおりである[1-5，1-6]．

- 是正保守(corrective maintenance)
  ソフトウェア製品の引渡し後に発見された問題を訂正するために行う受身の修正（reactive modification）．この修正によって，要求事項を満たすようにソフトウェア製品を修復する．
- 緊急保守　(emergency maintenance)
  是正保守の内，是正保守実施までシステム運用を確保するための，計画外で一時的な修正．
- 予防保守(preventive maintenance)
  引渡し後のソフトウェア製品の潜在的な障害が運用障害になる前に発見し，是正を行うための修正．
- 完全化保守(perfective maintenance)

引渡し後のソフトウェア製品の潜在的な障害が，故障として現れる前に，検出し訂正するための修正．完全化保守は，利用者のための改良，プログラム文書の改善を提供し，ソフトウェアの性能強化，保守性などのソフトウェア属性の改善に向けての記録を提供する．

- 適応保守(adaptive maintenance)
  引渡し後，変化した又は変化している環境において，ソフトウェア製品を使用できるように保ち続けるために実施するソフトウェア製品の修正．適応保守は，必須運用ソフトウェア製品の運用環境変化に順応するために必要な改良を提供する．これらの変更は，環境の変化に歩調を合わせて実施する必要がある．例えば，オペレーティングシステムの更新が必要になったとき，新オペレーティングシステムに適応するためには，幾つかの変更が必要かもしれない．

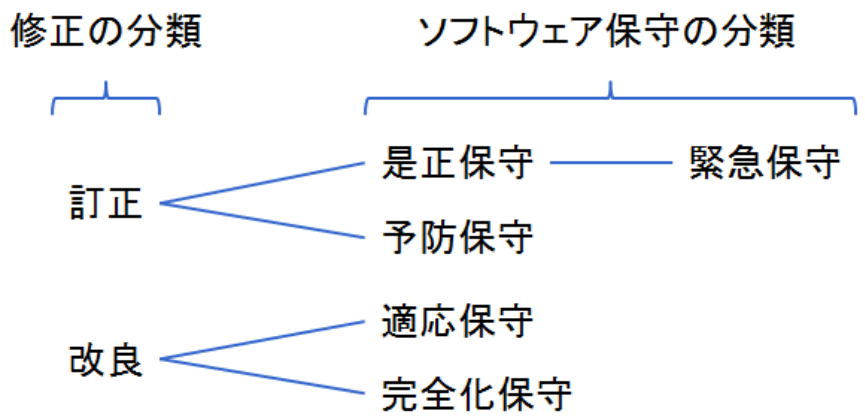


図 1.1　ソフトウェア保守の分類

次に，ソフトウェア保守のコストを増大させている原因について考察する．先に述べたような社会的影響の大きいソフトウェアは大規模化・複雑化が進んでおり，そのようなソフトウェアは大勢の開発担当者によって開発されている．そのようなソフトウェアの保守が必要になった場合，保守対象のソフトウェアの全体像や仕様，修正による影響範囲を把握することは難しくなるため，修正すべきソースコードの箇所を特定するコストや，修正により既存機能にバグが混入しなかったかどうかをテストするコストはソフトウェアの規模や複雑性が大きくなるにつれて増大する．

さらに，ソフトウェアを理解する上で十分な資料が作成されていない場合や，作成されていても，長期間の保守作業の中で最新化が成されず，資料の内容と実際のソフトウェアの動作に乖離がでてしまう場合には，保守コストが増大することが推察される．また，ソフトウェアの開発担当者と保守担当者が異なる場合，開発担当

者から保守担当者に対して，保守対象のソフトウェアに関する資料に書ききれてない仕様や既知の課題に関する引き継ぎが十分なされないこともままあり，その際も同様に保守対象ソフトウェアの理解が難しくなり，保守コストの増大につながる．

いずれの場合も，ソフトウェア保守においては，ソフトウェアの性質，振る舞いを理解することが重要であることがわかる．

ソフトウェアの性質や振る舞いを理解するための単純な方法として考えられるのは，ソフトウェアを実際に動作させてみて振る舞いを見たり，ソースコードを読んだりすることで，ソフトウェアの振る舞いや呼び出し関係を理解する手法がある．これらの手法は，ソフトウェアの規模や複雑さが小さいうちは現実的なコストで実施可能かもしれないが，ソフトウェアの規模や複雑さが大きくなるにつれて非現実的なコストになる．

一方で，コンピュータの計算能力の向上が進む中で，人間ではなくコンピュータにソースコードを分析させ，人間が理解したい内容に合わせてソフトウェアの性質や振る舞いを可視化する手法が盛んに研究されている．これらの手法やツールのうち代表的なものを以下に示す．

- リバースエンジニアリング

  リバースエンジニアリングとは，システムの構成要素(component)および構成要素間の関係を特定し，そのシステムを別の形式，もしくはより高い抽象度で表現することである[1-7]．ソフトウェア開発環境の中には，Imagix 4D[1-8]等のようにソースコードからフローチャートやコールグラフ（関数間の呼び出し関係を表すグラフ）を生成する機能を持つものや，Rational Software Modeler[1-9]等のようにソースコードからクラス階層情報を抽出し，可視化する機能を持つものが存在する．また，リバースエンジニアリングを行う手法の一種として，設計の復元(Design Recovery)[1-10]を行う手法が研究されている．設計の復元とは，設計に関する抽象概念(Design Abstraction)をソースコードおよび他の情報（設計書や開発者の経験，対象とする問題とドメインに関する一般的な知識）から再現することである[1-10]．設計の復元を行う代表的なツールとして，ソースコード中からデザインパターン[1-11]の実装部分を自動的に特定するツール[1-12, 1-13]をいくつか挙げることができる．これらは，ツールの開発者もしくは使用者がデザインパターンに関する一般的な知識（デザインパターンが実装されている部分の構文的特徴など）を予め与えておくと，その知識に基づいてデザインパターンの実装部分を対象ソースコード中から特定する．このように，デザインパターンの実装部分を特定することは，保守作業を行う上で有益であるとされている．例えば，保守対象のソースコード内で実装されているデザ

インパターンを明示すると，保守作業にかかる時間と混入する欠陥の数が減少したという実験結果が報告されている[1-14].

- 回帰テスト

保守作業を困難にする要因の 1 つとして，ソースコードの一部を変更すると，変更部分だけでなく他の部分の振る舞いが変化する可能性があることが指摘されている[1-15, 1-16]．このように，変更が他の部分に影響することは波及効果(Ripple Effect) [1-15] と呼ばれる．

波及効果が発生する可能性があるため，回帰テスト（変更後の振る舞いが要求を満たしているかを確認するためのテスト）では，変更部分のテストだけでなく，他の部分についてもテストを検討する必要が生じる．このことから，必要十分なテストの組み合わせを算出するための手法が数多く提案されている[1-16, 1-17].

保守対象がオブジェクト指向プログラムの場合，Dynamic Dispatch（同じ型の参照型変数であっても，実行時におけるインスタンスの型に依存して呼び出される手続きが変化すること）が原因で，開発者にとって波及効果を理解することが難しくなる[1-16]．よって，一般的な手続き型プログラムと比較して，オブジェクト指向プログラムの方が，回帰テストにおいて実行すべきテストを適切に特定することが難しいと言える．この問題を解決するために，Chianti というツールが開発されている[1-16]．Chianti に，Java 言語で記述された変更前と変更後のソースコードおよび変更前のソースコード用に作られたテストコードの集合を与えると，入力したテストコードの中で，再度動作させるべきもののみを提示する．Chainti は，まず，変更前と変更後のソースコードについて，仮想メソッド（子クラスのメソッドがオーバライドできるメソッド）をオーバライドするメソッドの集合のそれぞれ算出する．そして，それらの差分を求めることで Dynamic Dispatch の変化を特定し，Dynamic Dispatch が変化する可能性があるテストコードを提示する.

- ソースコード解析

ソースコード解析とは，ソースコードの中身や動作を解析する技術の総称である．ソースコード解析の例として、メトリクス計測がある．ソースコードの保守性（保守しやすさ）の評価を行うメトリクスの代表的なものとして，CK メトリクス [1-18] が挙げられる．CK メトリクスは，オブジェクト指向プログラムに含まれるクラスを対象とした5つの複雑度メトリクスから構成されている．CK メトリクスとして，複雑度メトリクスが満たすべき数学的性質 [1-19] を概ね満足していること [1-18]，加えて，他のメトリクスの組み合わせよりも欠陥の発生を予測に有用であること[1-20]が確認されていることが挙げられる.

プログラムスライシングの結果を利用したメトリクスがいくつか提案されている[1-21]．例えば，メトリクス Tightness[1-21]は，C 言語における関数中の文のうち，全てのスライス 1 に共通して含まれる文の割合であり，ほとんど文が返値や大域変数の値に影響与えていると高い値になる．直観的には，単一の目的で作成された関数は Tightness の値が高くなる．このようなプログラムスライシングに基づくメトリクスを用いることで，オープンソースソフトウェアに含まれる関数の凝集性が低下していることを定量化できることが確認されている［1-22］．Kataoka らは，リファクタリングの効果を計測する 3 つのメトリクスを提案している［1-23］．リファクタリング［1-24，1-25］とは，保守性の改善を目的とした変更作業のことである（詳細な定義は 1.2.4 節を参照）．これらメトリクスは，メソッド間の結合に基づいてリファクタリングの効果を計測する．具体的には，1 つ目は返値を介した結合，2 つ目は引数を介した結合，3 つ目は変数の共有に基づく結合を計測する．リファクタリングを行う開発者は，これらメトリクスを用いることで，リファクタリングによりメソッド間に存在する結合がどのように変化したかを調査することができる．

このように、ソフトウェア保守の技術は多数存在する．本論文では，昨今のコンピュータの処理能力の向上を受けて、解析可能な事象が増えてきたことでさかんに研究されているソースコード解析技術に注目し、ソフトウェア保守の支援をおこなう．

1.2. ソースコード解析

ソースコード解析は，静的解析と動的解析に分類することができる．まず，本節ではソースコード静的解析とソースコード動的解析の各々について説明し，その関係について考察する．

- ソースコード静的解析

  ソースコードを実際に動作することなく解析を行うことで，ソースコードからその性質や振る舞いを抽出し，それを開発者に提供する技術である．ソースコードの中身を扱うため，網羅性の高い解析をすることができる．

- ソースコード動的解析

  ソースコードを実際もしくは仮想的な動作環境上で実際にテストケースを与えてプログラムを動作させ，その動作結果や動作中のログなどを解析することで性質や振る舞いに関する情報を開発者に提供する技術である．マルチスレッド処理など，ソースコード静的解析で発見しづらい振る舞いを解析することができるが，網羅的な振る舞いを調べるには大量のテストケースが必要となる．

ソースコード静的解析とソースコード動的解析は解析しやすい性質や振る舞いが異なるため，お互いに補完する技術であるといえる．ただし，ソースコード動的解析はソースコードを実際に動作させる必要があるため，開発途中や修正途中の未完全なソースコードに対して利用することはできない．さらに，ソースコード動的解析を行うためには解析目的に則した十分な量のテストケースを準備する必要があるため，適用に対する初期コストがソースコード静的解析に比べて大きい．

そこで，本研究では実際のソフトウェア保守の現場に適用しやすいソースコード静的解析に注目する．

1.3. ソースコード静的解析

本節では，ソースコード静的解析技術について詳細に説明する．

ソースコード静的解析とは，ソースコード内やソースコード間の関係や性質をグラフ化する，またはメトリクスを計測するなどの処理を施すことでソースコードを抽象化し，抽象化した情報を利用して解析を行う．利用目的に応じてプログラムを様々な視点から抽象化することで，開発者にとって有用な「プログラムの特徴」のみを抽出することが容易となり，ソフトウェア開発を支援することができる．

ソースコード静的解析におけるグラフ化は，解析対象に存在する個々の部品間の関係を抽出し，抽象化し表現することを目的とした場合が多い．グラフ化の代表的な例として，以下を挙げる．

- プログラムのソースコードを木構造で表現した抽象構文木
- 手続き, メソッドなどの呼び出し関係をグラフ化した CFG(Call Flow Graph)
- クラスの継承関係をグラフ化したクラス階層構造
- プログラム間のデータフローや制御構造をグラフ化したプログラム依存グラフ

グラフ化によって個々の部品間の関係が明確になるため，構文木からプログラム依存グラフを作成する場合のように，グラフ化された情報を用いてより高度なプログラム解析が行われることも多い．例としては，エイリアス関係（同一メモリ空間を指す可能性のある式間の同値関係）にある変数の対の情報をもとに，より正確なデータフロー関係の解析を行うなどが挙げられる．一般的に複数の解析を組み合わせた場合，解析コストが上がるが，得られる情報の精度が向上することが知られている．グラフの解析方法の例として代表的なものを以下に挙げる．

- グラフ上の辺の探索による，到達可能な節点の計算
- グラフの比較による，同一部分の検出
- クラスの階層構造の深さ，手続き（メソッド）の数などの数値化
- 利用関係などの関係の行列化

一方で，メトリクス計測を用いたソースコード解析は，解析対象における個々を抽象化し個々の性質を取り出すことを目的としている．

メトリクスを用いた解析の代表的な例として以下を挙げる．

- クラス数，メソッド数，コード行数（LOC） などのメトリクス
- トークンの抽象化を目的とした記号化
- プログラムの品質や再利用性の評価値
- ソフトウェア間の類似性

メトリクスとして計測された情報は個々の性質をある観点から観測したもので，これらの情報を複数組み合わせることで，より多面的な観点から個々の解析対象を観測することができる．そのため，これらの数値を組み合わせることで，部品を評価するための新たな評価基準を生み出すができることも多い．数値化された情報の多くは，統計的手法を用いた評価に利用されることが多い．また，記号化された情報は個々の性質をある観点から観測したものであるが，配列化や行列化を行うことで，解析対象全体の特徴を示すことができる．そのため，統計的手法を用いた評価が行われることもあるが，単に比較するために利用されることも多い．

1.4. ソースコード静的解析によるソフトウェア保守支援

ソースコード静的解析技術を利用してプログラムから抽出された情報をもとに，ソフトウェア保守の支援を目的として様々な解析が行われている．グラフ化した情報を用いたソースコード静的解析の代表的な例を以下に示す．

- 最適化コードの生成:
  コンパイル時に必要のない命令を削除する
- テストデータの自動生成
  テストを行いたい実行経路を通るような入力データを実行履歴から生成する
- プログラムの結合
  似た部分を結合することで，ただ単に結合した場合よりも高速化を計る
- デバッグ支援
  プログラムスライス[1-26] を用いることで，デバッグ対象を限定する
- 影響波及解析
  再テストすべきテストケースを限定することで，テスト工程を効率化する
- モデルチェック
  プログラムの正当性や安全性の検証
- 情報漏洩解析
  プログラムの中で，セキュリティポリシーを満たさない文を検出する

- プログラム理解支援
  解析結果情報を提示することで，保守およびデバッグ作業を支援する

次に,メトリクス計測を用いたソースコード静的解析の代表的な例を以下に示す.

- ソフトウェア部品の評価
  メトリクス値化された部品の性質から再利用性や品質を評価
- コードクローンの把握
  コピーされたソースコードの検出する
- コピー部品の把握
  メトリクス計測された情報を配列化し，解析効率を上げる
- ソフトウェア（部品）のクラスタリング
  メトリクス値等を比較し，同じ傾向にあるソフトウェア(部品)を分類する
- 理解支援
  解析結果情報を選別の基準とし，大量の部品からの選別作業を支援する

なお，ここで挙げる例は一部で，抽出されるプログラム解析情報および利用目的はこれら以外にも多く存在する．さらに，ここで挙げたいくつかのプログラム解析情報を組み合わせることで，新たな解析をする手法も考案されている．

本論文では，上記で示した代表的なソフトウェア保守支援の例をより複雑化させた下記の２つの課題を解決したいと考える．

- 課題１：
  過剰に広いアクセス修飾子をもつフィールド，メソッドに関する理解支援
- 課題２：
  ソフトウェア保守が困難な大規模ソフトウェアにおけるコピー部品の把握

これらを解決するために，課題１に対しては，グラフ化した情報を利用した”アクセス修飾子過剰性の解析手法”を提案し，課題２に対しては，メトリクス計測を利用して高速化を図った” 類似ソフトウェア部品の検出手法”を提案する．

以降，アクセス修飾子過剰性に関する解析，類似ソフトウェア部品の解析に関する課題と既存研究にいてそれぞれ説明し，本論文における詳細な課題の設定とそれを解決する提案手法の内容を述べる．

1.5. アクセス修飾子過剰性に関する解析

1.5.1. 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 のアクセス修飾子には public，protected，private の 3 種類が存在し，何もアクセス修飾子を付けない場合(default) を含めると，フィールドおよびメソッドに対するアクセス範囲について 4 種類の制限を科すことができる（表 1.1）[1-27]．

表 1.1　アクセス修飾子とアクセス可能な範囲の対応

アクセス修飾子	自クラス	同一パッケージ	サブクラス	他クラス
public	○	○	○	○
protected	○	○	○	
default	○	○		
private	○			

public が宣言されたフィールドおよびメソッドには，全てのクラスからアクセスが可能である．protected が宣言されたフィールドおよびメソッドには，自クラス，同一パッケージ，サブクラスからのアクセスが可能である．default なフィールドおよびメソッドには，自クラス，同一パッケージからのアクセスが可能である．private が宣言されたフィールドおよびメソッドには，自クラスのみからのアクセスが可能である．

1.5.2. アクセス修飾子過剰性に関する課題

アクセス修飾子として protected や default，private を設定することで，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[1-28]．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．そういった状況下においては，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[1-3]．

このように，フィールドおよびメソッドに対してアクセス修飾子が過剰に広く設定されている場合，本研究ではこの状態をアクセス修飾子の過剰性と呼ぶ．アクセス修飾子の過剰性が存在すると，開発者が本来意図しなかった，不正な呼び出し操作が可能になり，そのような呼び出し操作が行われることにより不具合や論理的なバグが発生する可能性がある．こういった状況は，開発途中において最終的なアク

セス範囲が不透明な場合のほか，開発者間の設計情報共有が不十分である場合などにも起こりうる．しかし，過剰なアクセス修飾子の宣言は Java の構文上は認められているため，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実際のアクセス範囲に基づいて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．そこで，このようなアクセス修飾子の過剰性を検出し，その情報を開発者に提供し，アクセス修飾子過剰性の是正を支援することが重要と考える．

1.5.3. アクセス修飾子過剰性に関する既存研究の課題

アクセス修飾子の解析に関して，既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[1-29]．しかし，チェックしたアクセス修飾子に対して適切かどうかの分析はなされていない．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[1-30]．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[1-31]．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[1-32]．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[1-33]．しかし，アクセス修飾子の過剰性に関する潜在バグの顕在化を防止するためには，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する必要がある．

1.6. ソフトウェア類似性に関する解析

1.6.1. ソフトウェア類似性

ソフトウェアに対する保守作業の効率を下げている要因の 1 つとして，ソースコード中の類似ソースコードが指摘されている[1-34, 1-35, 1-36, 1-37, 1-38, 1-39, 1-40, 1-41]．類似ソースコードは，以下の理由で作成される[1-35, 1-42]．

- **既存ソースコードのコピーとペーストによる再利用**

  近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，ソースコードの再利用が容易になったために，現実にはコピーとペーストによる場当たり的な既存ソースコードの再利用が多く行われるようになった．

- **定型処理**

  定義上簡単で頻繁に用いられる処理．例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

- **プログラミング言語に適切な機能の欠如**

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

- **パフォーマンス改善**

  リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展などの機能が提供されていない場合に，特定のソースコードを意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

- **ソースコード自動生成ツールが生成するソースコード**

  ソースコード自動生成ツールにおいて，類似した処理を目的としたソースコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたソースコードをベースにして自動的に生成されるため，類似したソースコードが生成される．

特に，Linux や JDK（Java Development Kit）などの大規模ソースコードは大量の類似ソースコードを含むことが報告されている[1-37][1-40]．

1.6.2. ソフトウェア類似性に関する課題

ソフトウェアの保守を行う際に，あるソースコードを修正する必要があった際，そのソースコードの類似ソースコードにも同様の修正を行う必要があることが多い．その場合，全ての類似ソースコードを見つけ出す必要が生じることがある．特に，ソースコード中に欠陥が見つかった場合には，その欠陥を含むソースコードの類似ソースコードを探し，同様の欠陥が無いかを検査する必要がある[1-39][1-40][1-41]．しかし，前節で述べたような方法で作成された全ての類似ソースコードを人手で探すには大きな労力が必要となる．特に，大規模ソフトウェアが対象の場合，全ての類似ソースコードを人手で探すことはより困難となる．そこで，コンピュータを利用して自動的に類似ソースコードを検出し，開発者に提示することで，類似ソフトウェアに対する修正を支援することが重要と考える．

1.6.3. ソフトウェア類似性に関する既存研究の課題

システム間の類似性を求める研究としてさまざまな研究がある．Baxter らは抽象構文木（Abstract Syntax Tree）を利用したクローン検出手法を提案している[1-35]．しかしながら，類似性を求める定義はあるが，その値の有効性については述べられていない．また，実際にシステムに適用した結果はなく，定量的な評価を行っていない．[1-43]，[1-44]，[1-45] は，プログラムの類似性を自動的に計測するツールであるが，大規模システムに適用した結果はない．[1-46] では，提案した類似性を実際のソフトウェアに適用し，用途に応じてどのような類似性が考えられるかについて考察をしている．

1.7. 本論文の概要

本論文では，前節で挙げたソフトウェアのアクセス修飾子過剰性に関する問題，および，類似ソフトウェア部品の検出に関する問題を解決するために，以下の 2 つの手法について提案する．また，提案した手法を実装したツールについて述べ，ツールの評価実験を行った結果について述べる．

1. **Java ソフトウェア部品のアクセス修飾子過剰性分析手法**

   大量のソフトウェア部品の対する保守を行う場合，変数やメソッドに適切なアクセス修飾子が設定されているかどうかを知ることは開発者にとって重要であるが，それを人手で確認することは困難である．そこで，本論文では，ソフトウェアを解析し，変数及びメソッドの呼び出し関係をグラフ化することで実際にどの範囲から呼び出されているかという情報と，実際にその変数及びメソッドに宣言されているアクセス修飾子の呼び出し範囲の乖離を分析する手法を提案し，その乖離を自動的に解析するツール ModiChecker を開発した．さらに，ModiChecker が検出した過剰に広い範囲に設定されているアクセス修飾子を，実際の非参照状況をベースにして自動的にアクセス修飾子の設定を修正する機能を実装した．これにより，開発者は意図せずアクセス修飾子を過剰に広く設定してしまったフィールドやメソッドへ，第三者が誤ってアクセスすることを事前に防止できる．

   次に，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ないため，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手法を提案し，オープンソースのソフトウェアに適用することでその効果を検証する．

   最後に，ソフトウェアバージョンアップの際にアクセス修飾子に対してどのような修正作業がどのような頻度で行われるのか分析を行い，過剰なアクセス修飾子がどのように発生し，どのように修正されていくのかを分析した．

2. **コードの静的特性を利用した Java ソフトウェア部品間の類似判定手法**

   大量のソフトウェア部品に対する類似部品検出を想定する場合，類似部品解析のコストを軽減させることが重要になる．しかし，既存研究で採用されることの多いソースコードの文字列比較を用いた類似部品の検出手法は解析コストが高い．そこで，Java ソースコードにおける Java 予約語の出現回数に関するメトリクスや Java ソースコードの複雑性に注目したメトリクスを計測し，2 つのソフトウェア部品間でこれらのメトリクス値の比較を行うことで高速に類似部

品を検出する手法を提案する．これにより，従来の文字列比較を用いた手法にくらべて解析時に扱う情報量が格段に低減されるため，解析コストを低く抑えることが可能となる．また，提案した手法を類似性計測ツール Luigi として実現した．さらに，従来の文字列比較による類似分析を実装したツール SMMT[1-47]と，今回開発した Luigi を同じソフトウェア群に対して類似分析を行うことで，その類似部品抽出に関する精度とコストを比較評価する適用実験を行うことで，提案手法の有効性を検証する．

第2章　アクセス修飾子過剰性に関する研究

2.1. 導入

ソフトウェア開発において，変数およびメソッドに対するアクセスがソースコード内の全ての場所から可能な状態にあると，変数およびメソッドが開発者の想定していない使われ方をされる可能性がある．そのような状態を放置しておくと，変数へ想定している範囲外の不適切な値が設定されたり，プログラムを正常に実行する上で守るべきメソッド呼び出しの順序が前後したりなど，潜在的な不具合の原因となってしまう．現在広く用いられているオブジェクト指向プログラミング言語である Java において，この問題を解決する手段としては，フィールドおよびメソッドに対するアクセス修飾子の宣言が挙げられる．アクセス修飾子はフィールドおよびメソッドに対して個別に一種類宣言することができ，その種類によって外部からのアクセスを許可する範囲を必要な分へ制限することができる．開発者は，ソフトウェアの設計等に基づく適切なアクセス修飾子をフィールドおよびメソッドに対して宣言することで，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[2-2][2-3].

しかし，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，複数人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．そのような場合においては，コストや期間の制限により，チームに属する開発者全員がソースコード上のフィールドおよびメソッドの利用状況についての情報を共有することが難しくなる．その結果，フィールドおよびメソッドに対して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，そのまま適切なものへと修正されることなく残り続ける場合がある

2.2. 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 のアクセス修飾子には public，protected，private の 3 種類が存在し，何もアクセス修飾子を付けない場合(default) を含めると，フィールドおよびメソッドに対するアクセス範囲について 4 種類の制限を科すことができる（表 2.1）[1-27].

表 2.1　アクセス修飾子とアクセス可能な範囲の対応（再掲）

アクセス修飾子	自クラス	同一パッケージ	サブクラス	他クラス
public	○	○	○	○
protected	○	○	○	
default	○	○		
private	○			

public が宣言されたフィールドおよびメソッドには,全てのクラスからアクセスが可能である.protected が宣言されたフィールドおよびメソッドには,自クラス,同一パッケージ,サブクラスからのアクセスが可能である. default なフィールドおよびメソッドには,自クラス,同一パッケージからのアクセスが可能である. private が宣言されたフィールドおよびメソッドには,自クラスのみからのアクセスが可能である.

2.3. アクセス修飾子過剰性（AE）と No Access

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とメソッドに対し,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)[2-1] を用いる.

AE は表 2.2 の内, (*)印がついているセルに該当する. 表 2.2 において, 行は宣言されているアクセス修飾子を, 列は実際にアクセスされる範囲から導出される必要最小限のアクセス修飾子を表す. 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.

pub-pub, pro-pro, def-def, pri-pri の 4 つの状態は, フィールドおよびメソッドに対して宣言されているアクセス修飾子と, 実際にアクセスされる範囲が一致していることを意味する. 次に, pub-pro, pub-def, pub-pri, pro-def, pro-pri, def-pri の 6 つの状態は, フィールドおよびメソッドに対して宣言されているアクセス修飾子に対し, 実際にアクセスされる範囲が狭いことを意味する. これは, フィールドおよびメソッドに対し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ものとみなし,本論文ではこれらを AE であると定義する. また, pub-na, pro-na, def-na, pri-na の 4 つの状態は, アクセス修飾子は宣言されているが, 実際にはどこからもアクセスされてないフィールドおよびメソッドを意味する. 本論文では, そういった状態を No Access と定義する. 最後に, 表 2.2 において x と表示されている箇所は, フィールドおよびメソッドに対して宣言されているアクセス修飾子に対し, 実際にアクセスされる範囲が広いことを意味

する．しかし，これらはコンパイラによりエラーとして検出されるため，本論文では考慮しない．

表 2.2　AE と NoAccess の種類

	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

2.4. 過剰なアクセス修飾子を設定した場合の問題例

アクセス修飾子として protected や default，private を設定することで，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[1-28]．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．そういった状況下においては，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．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，ソースコード 1 に示すクラス X の例を用いて説明する．

図 2.1 に示すクラス X は，3 行目にある String 型の変数 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．変数 y の文字列長を取得するには 13 行目の methodB を呼び出す必要があるが，変数 y には 3 行目で null が代入されているため，目的を達成するためには，下記の２つの手順を順番に踏む必要がある．

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

この手順を正確に実行するために 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，この例において methodB は外部から直接アクセスされてはならないにもかかわらず，アクセス修飾子に 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 を直接呼び出すことが可能となっている．こうした呼び出され方をした場合，変数 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．

```
1  public class X {
2    // フィールドyの初期値はnull.
3    private String y = null;
4
5    // フィールドyに値を設定する.
6    // クラス外から呼ばれることを想定していない.
7    private void methodA( ) {
8      y = "hello";
9    }
10
11   // フィールドyの文字列長を返す.
12   // クラス外から呼ばれることを想定していない.
13   public int methodB() {
14     return y.length();
15   }
16
17   // 値の設定されたフィールドyの文字列長を返す.
18   // クラス外から呼ばれることを想定している.
19   public int methodC() {
20       this.methodA();
21       return this.methodB();
22   }
23 }
```

図 2.1　アクセス修飾子過剰性による潜在バグ例

このように，フィールドおよびメソッドに対してアクセス修飾子が過度に広く設定されている場合，開発者の意図しない操作が行われることにより不具合や論理的なバグが発生する可能性がある．こういった状況は，開発途中において最終的なアクセス範囲が不透明な場合のほか，開発者間の設計情報共有が不十分である場合などにも起こりうる．しかし，過剰なアクセス修飾子の宣言は Java の構文上は認められているため，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実際のアクセス範囲に基づいて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．

2.5. アクセス修飾子過剰性検出ツール 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，我々はアクセス修飾子過剰性検出ツール ModiChecker を開発した[2-1]．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．

ModiChecker は図 2.2 に示す構成を持つ．最初に，ModiChecker は解析対象のソースコードとそのソースコードのコンパイルに必要なライブラリを入力として取り込む．次に，既存のメトリクス計測プラグインプラットフォームである MASU[2-13]を利用して入力されたソースコードを AST(Abstract Syntax Tree)に

変換する．AST データベースには，各フィールドおよびメソッドを呼び出しているクラスの情報と，各フィールドおよびメソッドにどの種類のアクセス修飾子が設定されているかの情報が格納される．そこで，ModiChecker は各フィールドおよびメソッドに対して，実際に呼び出されている範囲と，宣言されているアクセス修飾子の情報を MASU から入手し，両者の乖離を AE として抽出する．最後に，ModiChecker は各フィールドおよびメソッドの AE に関するレポートを出力し，開発者に提示する．

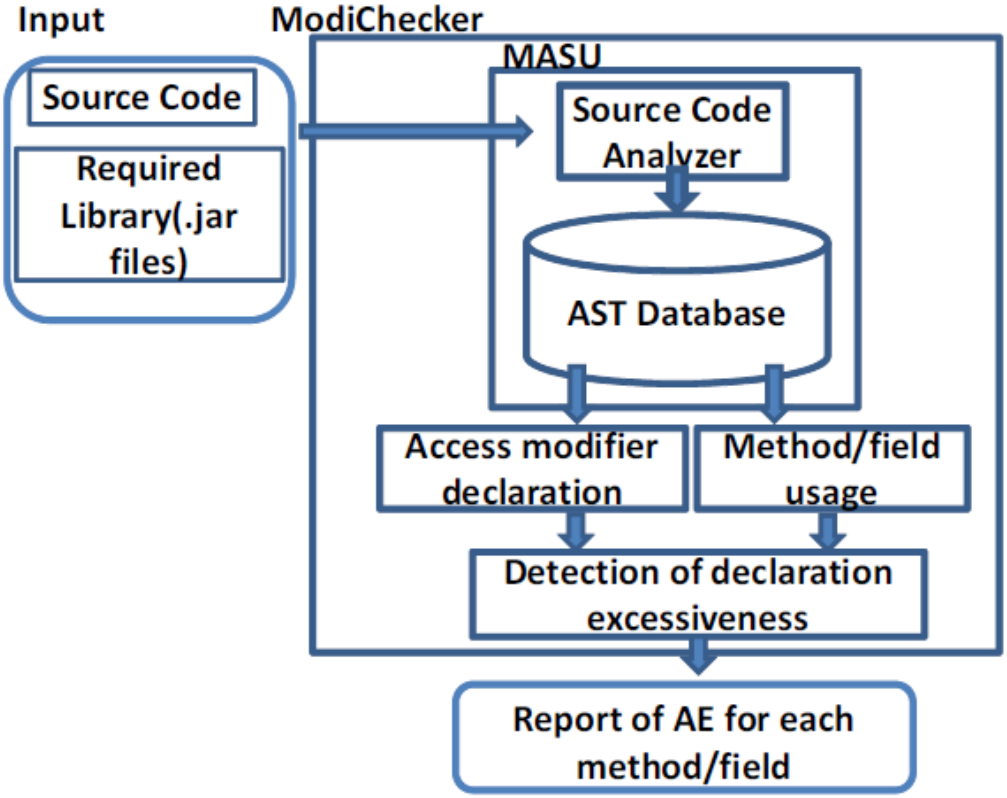


図 2.2　ModiChecker の構成図

2.5.1. ModiChecker の機能

ModiChecker は主に以下に示す 2 つの機能を持つ．

1. **AE であるフィールドおよびメソッドのリスト表示機能**

AE であるフィールドおよびメソッドをリスト表示する．ツール下部のラジオボタンにチェックを入れることで，フィールド/メソッド表示を切り替える．その結果は図 2.3 のように表示される．図 2.3 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．なお，No Access であるフィールドおよびメソッドはツール上には表示されない．

2. 上記リストの CSV ファイル出力機能

AE であるフィールドおよびメソッドのリストを CSV 形式のファイルで出力する(図 2.4).

ModiChecker の開発により，ツール利用者は過剰に広い範囲に設定され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可能となった．

さらにその後の研究[2-14] では，ModiChecker に対して以下の 2 点の機能拡張を行った．

3. AE 修正支援機能

ツール上でアクセス修飾子を変更したいフィールドおよびメソッドの行を選択し，ツール下部の”Change Access Modifier”ボタンを押下することで，ソースコード上のアクセス修飾子を Recommended Modifier に示されているアクセス修飾子へと変更する．

4. No Access であるフィールドおよびメソッドの CSV ファイル出力機能

解析結果として，No Access であるフィールドおよびメソッドを CSV 形式のファイルで出力する(図 2.5). No Access の情報については，3 列目(Access Modifier）が解析時点で宣言されているアクセス修飾子を表す．

この拡張により，ツール利用者は容易にフィールドおよびメソッドのアクセス修飾子を適切なものへと修正することが可能となり，また容易にどこからも参照されていないフィールドおよびメソッドに関する情報を取得することが可能となった．

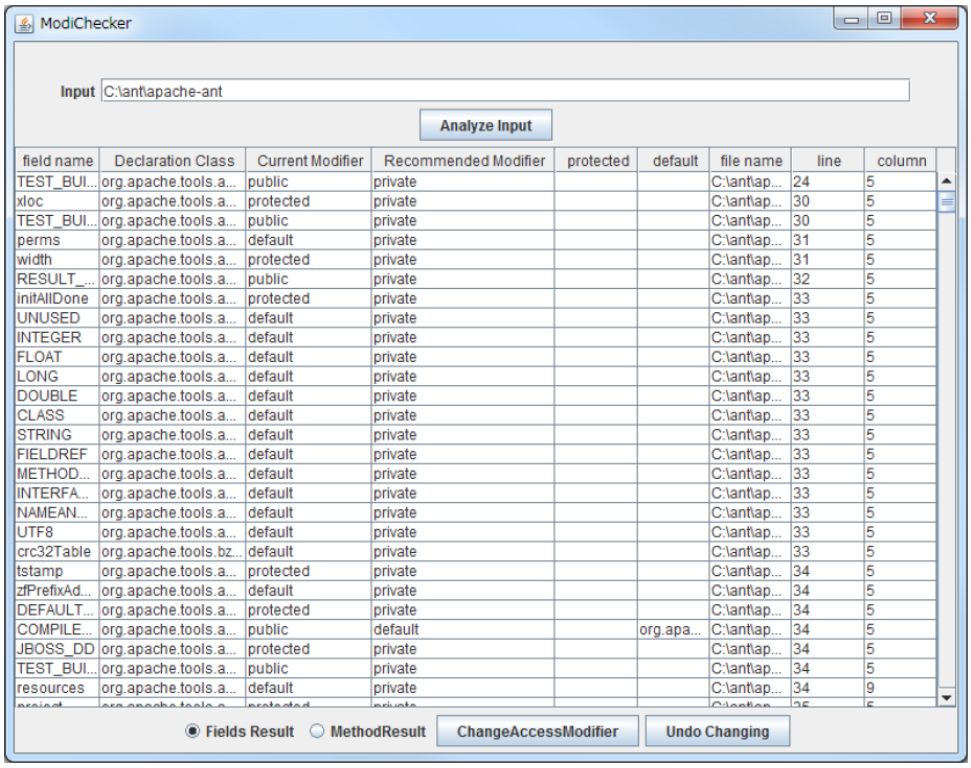


図 2.3　ModiChecker の解析結果画面

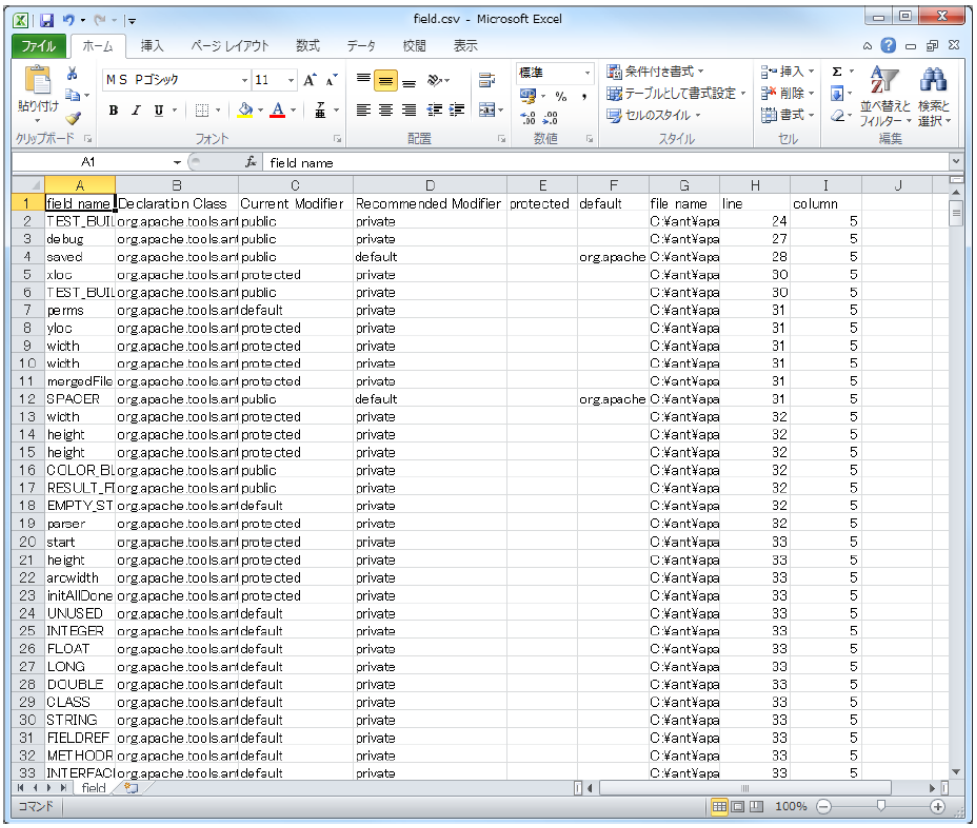
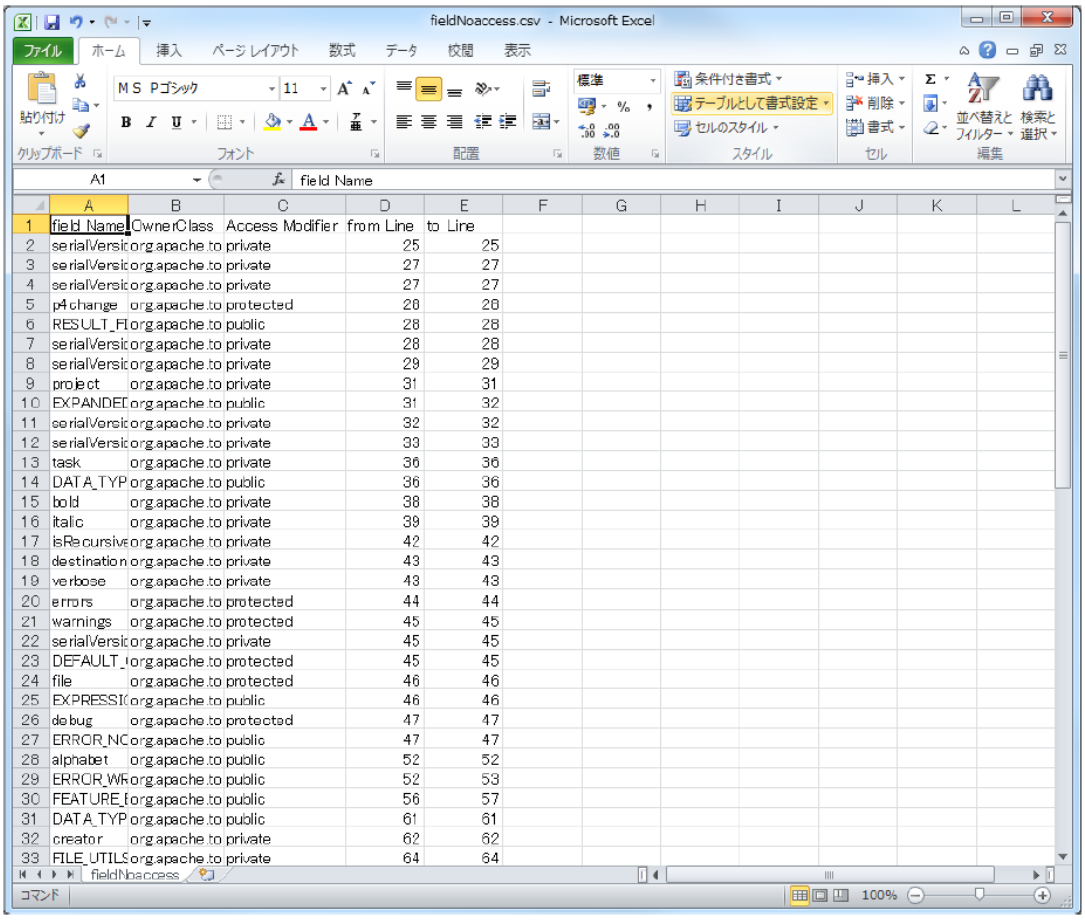


図 2.4　ModiChecker の出力 CSV ファイル



	A	B	C	D	E
1	field Name	OwnerClass	Access Modifier	from Line	to Line
2	serialVersi	org.apache.to	private	25	25
3	serialVersi	org.apache.to	private	27	27
4	serialVersi	org.apache.to	private	27	27
5	p4change	org.apache.to	protected	28	28
6	RESULT_FI	org.apache.to	public	28	28
7	serialVersi	org.apache.to	private	28	28
8	serialVersi	org.apache.to	private	29	29
9	project	org.apache.to	private	31	31
10	EXPANDED	org.apache.to	public	31	32
11	serialVersi	org.apache.to	private	32	32
12	serialVersi	org.apache.to	private	33	33
13	task	org.apache.to	private	36	36
14	DATA_TYP	org.apache.to	public	36	36
15	bold	org.apache.to	private	38	38
16	italic	org.apache.to	private	39	39
17	isRecursive	org.apache.to	private	42	42
18	destination	org.apache.to	private	43	43
19	verbose	org.apache.to	private	43	43
20	errors	org.apache.to	protected	44	44
21	warnings	org.apache.to	protected	45	45
22	serialVersi	org.apache.to	private	45	45
23	DEFAULT_I	org.apache.to	protected	45	45
24	file	org.apache.to	protected	46	46
25	EXPRESSI	org.apache.to	public	46	46
26	debug	org.apache.to	protected	47	47
27	ERROR_NC	org.apache.to	public	47	47
28	alphabet	org.apache.to	public	52	52
29	ERROR_WR	org.apache.to	public	52	53
30	FEATURE_I	org.apache.to	public	56	57
31	DATA_TYP	org.apache.to	public	61	61
32	creator	org.apache.to	private	62	62
33	FILE_UTILS	org.apache.to	private	64	64

図 2.5　ModiChecker の出力 CSV ファイル(NoAccess)

2.6. ソフトウェアのバージョン種別と AE の関連に対する分析

2.6.1. 分析の概要

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[2-1]．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 および NoAccess の値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．

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．

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 および NoAccess の値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 および NoAccess の値の変化量と MinorVU 時の AE および NoAccess の値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 および NoAccess の値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．

2.6.2. 分析の対象

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[2-11]．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．

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，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 または NoAccess 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．

2.6.3. 分析結果と考察

表 2.3，表 2.4，表 2.5 は，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 および NoAccess の値を AE および NoAccess の種類ごとに分類したものである．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 pro-pri 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，No Access 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 2.3　Ant ver1.3 におけるフィールドの AE 等の値

	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

表 2.4　Ant ver1.4 におけるフィールドの AE 等の値

	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 2.5　Ant ver1.4.1 におけるフィールドの AE 等の値

	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 2.6 図 2.6　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に示す．この図では，MajorVU 時（図 2.6 の赤で囲った列）に pro-pri と pub-pri の変化量が多く検出されている．pro-pri と pub-pri 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 および NoAccess の値の差分が大きく，MinorVU 時には小さいことが図 2.6 から推測できる．そこで，MajorVU 時の AE または NoAccess の値の変化量の群と，MinorVU 時の AE または NoAccess の値の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 および NoAccess の値の変化量の集合と MinorVU 時の各フィールドの AE または NoAccess の値の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 2.6 および表 2.7 に示す通りであった.

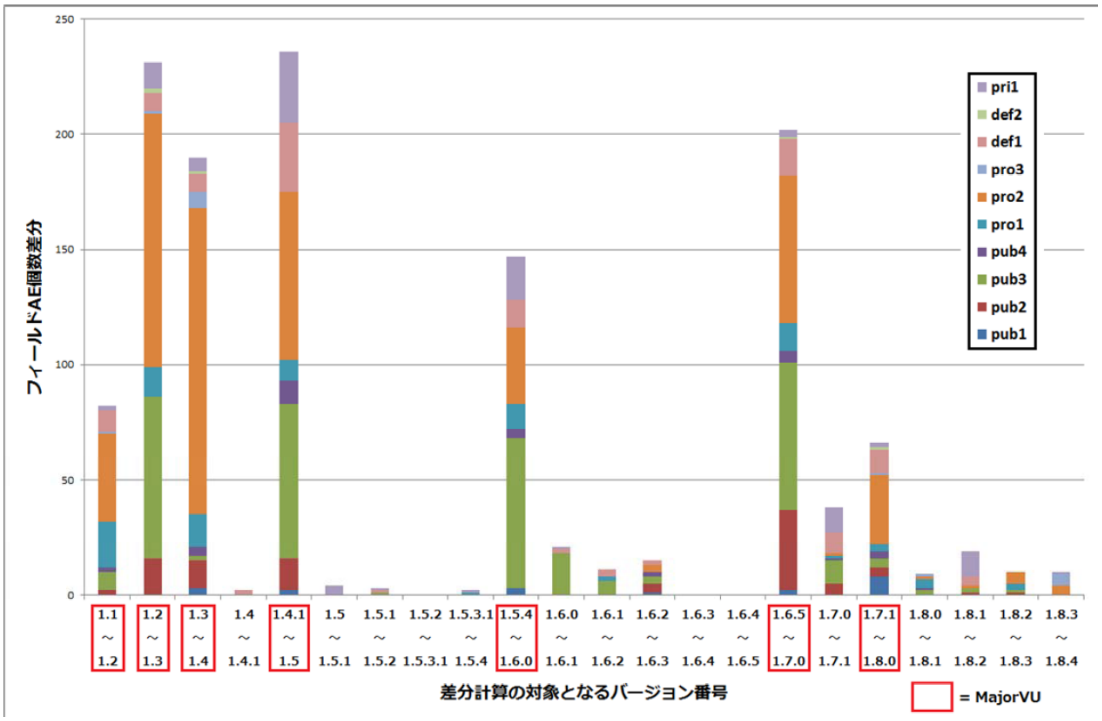


図 2.6　バージョン間におけるフィールドの AE 数の各要素数の差分

表 2.6　MajorVU と MinorVU 間の有意差(フィールド)

AE および No Access	p 値※	有意水準 0.05 における有意差
pub-pro	0. 00080	有り
pub-def	0. 00114	有り
pub-pri	0. 00113	有り
pub-na	0. 00032	有り
pro-def	0. 00002	有り
pro-pri	0. 00001	有り
pro-na	0. 03715	有り
def-pri	0. 00003	有り
def-na	0. 00479	有り

pri-na	0.00192	有り

※数値は小数点以下第 6 位を四捨五入している

表 2.7　MajorVU と MinorVU 間の有意差(メソッド)

AE および No Access	p 値※	有意水準 0.05 における有意差
pub-pro	0.00440	有り
pub-def	0.00562	有り
pub-pri	0.07205	無し
pub-na	0.00122	有り
pro-def	0.00361	有り
pro-pri	0.00225	有り
pro-na	0.02073	有り
def-pri	0.07656	無し
def-na	0.13130	無し
pri-na	0.00919	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 および NoAccess 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，pub-pri，def-pri，def-na 以外の AE および NoAccess 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 および NoAccess の値の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 pub-pri，def-pri および def-na は他の AE や No Access に比べて各バージョン間の値の変化量が小さく，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 または NoAccess の値の差分を分析し，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．

2.7. Java プログラムの開発履歴における AE の遷移に関する分析

2.7.1. 分析の概要

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[2-11] や jEdit[2-12] の 1 バージョンにおいて，過剰なアクセス修飾子が宣言された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 [2-1].

そこで，本節では既存の 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う．今回分析を行うに当たって，以下の 2 点について分析を行った．

分析 1

プロジェクト開発履歴におけるフィールドおよびメソッドに対するアクセス修飾子の遷移状況分析

分析 2

プロジェクト開発履歴における各 AE の修正状況分析

分析 1 は，全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の情報をプロジェクトの全バージョンにわたって追跡調査することで，AE であるアクセス修飾子がどの程度修正されているのかを明確にすることを目的として行った．分析 2 は，プロジェクトの各バージョンにおいて AE の種類ごとに修正状況を調査することで，修正されやすい AE というものが存在するかどうかを明確にすることを目的として行った．

2.7.2. 分析の対象

今回の分析では，SourceForge.jp[2-4] 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 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 2.8 に示す．なお，表 2.8 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を行っている．

表 2.8　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

本研究における分析環境に関する情報は以下の通りである.

- 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.

2.7.3. フィールドおよびメソッドの状態に対する分類

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 4 状態に分類する(表 2.9).

1. 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 2.9 にて*1 記載のあるセル).

2. 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 2.9 にて*2 記載のあるセル).

3. 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 2.9 にて*3 記載のあるセル).

4. なし

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

表 2.9 アクセス修飾子の状態分類

	public	protected	default	private	No Access
public	pub-pub(*1)	pub-pro(*2)	pub-def(*2)	pub-pri(*2)	pub-na(*3)
protected	x	pro-pro(*1)	pro-def(*2)	pro-pri(*2)	pro-na(*3)
default	x	x	def-def(*1)	def-pri(*2)	def-na(*3)
private	x	x	x	pri-pri(*1)	pri-na(*3)

2.7.4. アクセス修飾子の状態遷移に対する分類

前節で定義した４つの状態は，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 2.7 に示す a から r までの 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．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．

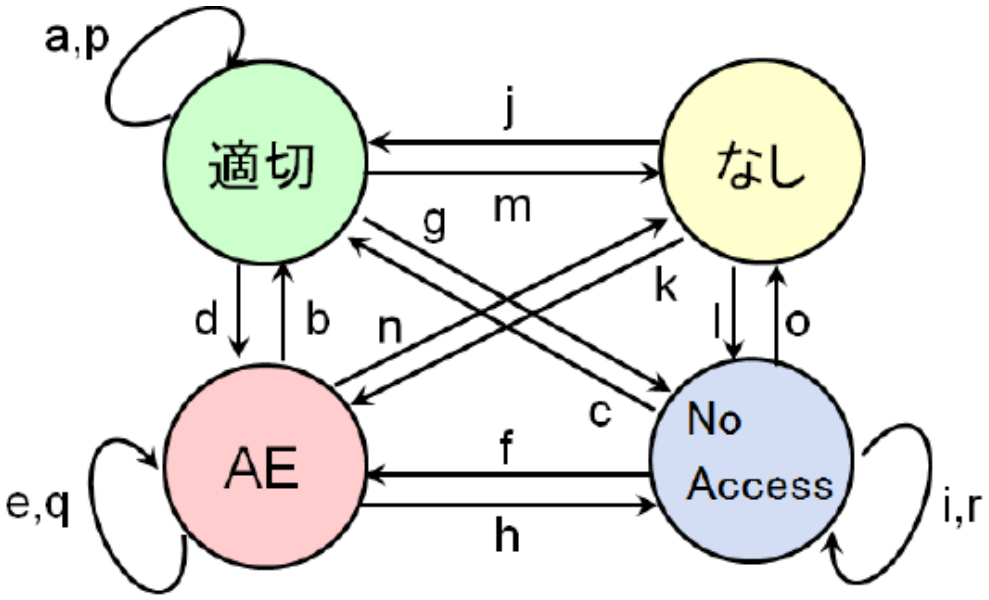


図 2.7　2 つのバージョン間におけるアクセス修飾子の状態遷移図

2.7.5. 分析の手順

今回の分析は以下の手順で実施した．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施している(図 2.8).

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およびパッケージ名，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

2.7.6. 分析結果

各プロジェクトにおいて，2.7.1 節で述べた分析 1 では，2.7.4 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 2.10，表 2.12 は，各種状態変遷総数について，フィールドおよびメソッドそれぞれの値を示している．表 2.11，表 2.13 は，各種状態変遷が状態変遷総数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．

また，分析 2 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 2.14，表 2.15 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．

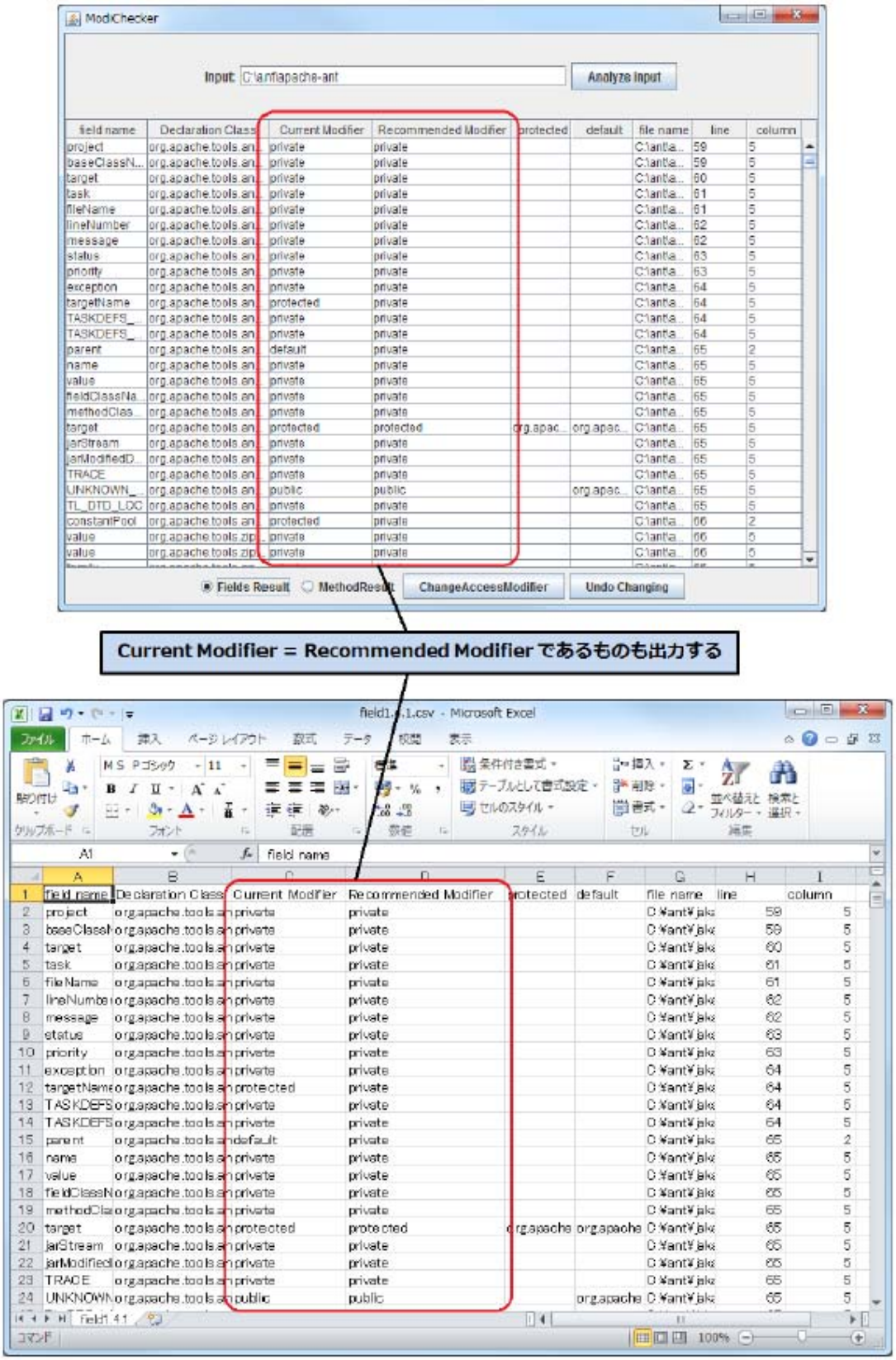


図 2.8　改変後の ModiChecker の出力

2.7.7. 分析 1 の結果考察（フィールドのアクセス修飾子変遷状況）

表 2.10，表 2.11 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，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．

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．

表 2.10　フィールドのバージョン間変遷総数

遷移記号	遷移の意味	Apache Ant	Areca Backup	Argo UML	Free Mind	JDT Core	jEdit	Apache Struts
a	適切→適切	15	7	23	10	33	9	7
b	AE→適切	130	16	358	27	397	74	292
c	NoAccess →適切	20	14	43	6	96	23	14
d	適切→AE	29	50	36	33	299	44	148
e	AE→AE	56	8	33	15	160	12	70
f	NoAccess →AE	1	9	19	1	20	11	7
g	適切 →NoAccess	24	22	160	18	79	33	53
h	AE →NoAccess	21	7	56	2	63	13	27
i	NoAccess →NoAccess	2	7	40	0	3	0	4
j	なし→適切	5190	1905	5818	1960	5480	3117	5806
k	なし→AE	1144	720	1329	619	2401	901	2627
l	なし →NoAccess	168	89	1175	284	328	411	611
m	適切→なし	1642	866	3447	632	1072	1764	2787
n	AE→なし	375	351	1869	243	1046	540	1191
o	NoAccess →なし	104	44	1155	177	127	296	232
p	変化なし (適切)	57796	85917	49936	3110	87322	32232	52464
q	変化なし (AE)	12361	35074	10428	1127	30527	8214	30227
r	変化なし (NoAccess)	1842	6064	9113	412	4921	2932	7651

表 2.11　フィールドのバージョン間変遷割合(%)

遷移記号	遷移の意味	Apache Ant	Areca Back up	Argo UML	Free Mind	JDT Core	jEdit	Apache 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	NoAccess →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	NoAccess →AE	0.00	0.01	0.02	0.01	0.01	0.02	0.01
g	適切 →NoAccess	0.03	0.02	0.19	0.21	0.06	0.07	0.05
h	AE →NoAccess	0.03	0.01	0.07	0.02	0.05	0.03	0.03
i	NoAccess →NoAccess	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 →NoAccess	0.21	0.07	1.38 3	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	NoAccess →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 (NoAccess)	2.28	4.62	10.72	4.75	3.66	5.79	7.34

2.7.8. 分析 1 の結果考察（メソッドのアクセス修飾子変遷状況）

表 2.12，表 2.13 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「NoAccess →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．

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．

表 2.12　メソッドのバージョン間変遷総数

遷移記号	遷移の意味	Apache Ant	Areca Backup	Argo UML	Free Mind	JDT Core	jEdit	Apache Struts
a	適切→適切	60	7	75	35	59	20	7
b	AE→適切	191	66	314	79	527	154	180
c	NoAccess→適切	243	169	645	109	574	127	239
d	適切→AE	85	39	192	38	303	152	107
e	AE→AE	110	11	115	27	278	34	92
f	NoAccess→AE	140	14	64	13	77	34	356
g	適切→NoAccess	132	135	633	102	307	191	129
h	AE→NoAccess	95	12	159	8	113	54	29
i	NoAccess→NoAccess	12	3	52	6	24	1	4
j	なし→適切	4281	2849	9145	3709	5966	3810	5362
k	なし→AE	1992	723	2680	632	1875	1367	4158
l	なし→NoAccess	8230	2835	13815	5232	6534	3287	15728
m	適切→なし	1002	1636	4743	1626	2502	2100	2006
n	AE→なし	522	502	2115	205	1100	854	2951
o	NoAccess→なし	1756	1831	8678	2921	3060	1834	8203
p	変化なし(適切)	49001	114618	71709	6998	93471	38202	52563

q	変化なし (AE)	23863	30743	22479	930	27019	13877	30366
r	変化なし (NoAccess)	93441	102555	114517	7378	96937	32910	151791

表 2.13　メソッドのバージョン間変遷割合(%)

遷移記号	遷移の意味	Apache Ant	Areca Backup	Argo UML	Free Mind	JDT Core	jEdit	Apache 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	NoAccess →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	NoAccess →AE	0.08	0.01	0.03	0.04	0.03	0.03	0.13
g	適切 →NoAccess	0.07	0.05	0.25	0.34	0.13	0.19	0.05
h	AE →NoAccess	0.05	0.00	0.06	0.03	0.05	0.05	0.01
i	NoAccess →NoAccess	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 →NoAccess	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	NoAccess →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	12.89	11.88	8.92	3.10	11.22	14.02	11.07

	(AE)							
r	変化なし (NoAccess)	50.47	39.64	45.42	24.55	40.27	33.24	55.34

2.7.9. 分析 2 の結果考察（フィールドに対する AE 修正状況）

表 2.14 を分析して全体に共通していえることとしては，フィールドに関する傾向と同様に，AE であるメソッドに対するアクセス修飾子修正作業は，ほとんど行われることなく放置されているのが現状である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用いることにより，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．

表 2.14　AE であるフィールドの修正状況(%)

	Apache Ant	Areca Backup	Argo UML	Free Mind	JDT Core	jEdit	Apache 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

2.7.10. 分析 2 の結果考察（メソッドに対する AE 修正状況）

表 2.15 を分析して全体に共通していえることとしては，フィールドに関する傾向と同様に，AE であるメソッドに対するアクセス修飾子修正作業は，ほとんど行われることなく放置されているのが現状である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用いることにより，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．

表 2.15　AE であるメソッドの修正状況(%)

	Apache Ant	Areca Backup	Argo UML	Free Mind	JDT Core	jEdit	Apache 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

2.8. 関連研究

まず，アクセス修飾子の解析に関して，本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[1-29]．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，Müller 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．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[2-5]．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[2-6]．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[2-7]．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．これに対して，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．

また，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[2-8, 2-9]．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[2-10]．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．

2.9. まとめと今後の課題

本研究では，Java のアクセス修飾子の過剰性を AE というメトリクスで測定する手法を提案し，提案手法を実装したツール ModiChecker を開発した．さらに ModiChecker を用いて，下記に示す AE に関する 2 つの分析を行った．

1. ソフトウェアのバージョン種別と AE の関連に対する分析
2. Java プログラムの開発履歴における AE の遷移に関する分析

まず， 分析 1 については，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．ただし，この実験結果は特定の 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．

次に，分析 2 については，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情報について考察を行った．分析を行うに当たり，プロジェクトの各バージョンにおける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，適切，AE，No Access という 3 つの状態に分類した．また，2 つのバージョン間においてフィールドおよびメソッドの状態がどのように遷移するかについて，遷移の性質ごとに 6 つにグループ分けを行った．これらの分類に基づき分析を行った結果，ほとんどのフィールドおよびメソッドに対して宣言された過剰なアクセス修飾子は，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，例えばあるフィールドのアクセス修飾子が pub-pro → pub-pub という遷移を行った場合，アクセス修飾子は変化していないにもかかわらず AE が解消されたことになる．こうした遷移の後に改めて別の適切な状態へと遷移するような例が多数発見されれば，pub-pro → pub-pub という遷移をしたフィールドは，適切となってもまだ修正の余地がある場合が多いということがいえることになる．このように，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することが挙げられる．

第3章　ソフトウェア部品間の類似性計測に関する研究

3.1. 導入

ソフトウェアの大規模化と複雑化に伴い，高品質なソフトウェアを一定期間内に効率良く開発することが重要になってきている．これを実現するために近年のソフトウェア開発において，再利用を用いた開発がよく行われている．再利用とは，既存のソフトウェア部品を同一システム内や他のシステムで利用することを指し，開発期間の短縮や品質向上を期待できるといわれている[3-1,3-2,3-5,3-7]．ソフトウェアの再利用による効果を最大限に引き出すためには，開発者が開発しようとするソフトウェアに必要な部品およびライブラリに関する知識を持つことが重要になってくるが，知識の共有が満足になされていないために，同種のプログラムが別々の場所で，独立して開発されている事も多い．知識の共有が満足になされていないために，同種のプログラムが別々の場所で，独立して開発されている事も多い．

一方でインターネットの普及により,SourceForge[3-10]などのソフトウェアに関する情報を交換するコミュニティが誕生し，大量のプログラムソースコードが簡単に入手できるようになった．これらの公開されている大量の部品の中から，開発者の必要としている機能を持つ部品，その機能の使い方を示している部品のような，再利用に有益な情報を提供する検索システムを実現する事で，知識の共有が実現でき，再利用を促進する事ができると考えられる

そこで我々の研究チームでは利用実績に基づく Java ソフトウェア部品の収集，検索システム SPARS-J(Software Product Archiving, analyzing, and Retrieving System for Java)[3-12]を研究している． SPARS-J においては，継承やメソッド呼び出しによる利用関係だけでなく，ソースコードをコピーして転用することによる利用関係も補足するために，コピーされたと推測される部品を類似部品として判定し，部品群としてグループ化するという特徴がある．これまで SPARS-J では，この類似した部品の検出に，ソースコードの文字列比較を用いた類似性計測ツール SMMT[1-47]を利用していた．しかし大量のソースコードに対して類似性計算を行う場合，既存の文字列比較による類似性計測法では，解析コストが膨大になる問題が発生するため，それを考慮にいれた手法が必要である．そこで，類似性メトリクスを用いた Java プログラム間の類似性計測手法を提案する．

この手法では，ソースコードの静的特性の数値に抽象化したメトリクス値を比較することで類似部品の抽出が可能となるため，解析コストを低く抑えることが可能

となる．また，提案した手法を類似性計測ツール Luigi として実現し，SPARS-J における部品間の類似性計測部として実装した．さらに，Luigi を用いた適用実験を行うことで，その有効性を検証する．

以降，3.2 節では提案手法を説明するための準備として，SPARS-J について述べ，ソースコードの文字列比較を用いた類似性計測手法についての考察を行う．3.3 節では文字列比較を用いた既存の類似性計測ツール SMMT の説明を行う．それを踏まえ，3.4 節では Java ソースコードの部品情報を数値化した類似性メトリクスの比較を用いた類似性計測方法を提案する．さらに，部品間の類似性を測定し，類似部品対を部品群としてまとめる機能を持つ Java プログラム間類似性計測ツール Luigi の説明をする．3.5 節で Luigi の適用実験を行い，提案手法の有効性を検証する．最後に 3.6 節でまとめと今後の課題について述べる．

3.2. ソフトウェア部品の収集，検索システム SPARS-J

3.2.1. SPARS-J とは

我々の研究チームでは，利用実績に基づく Java ソフトウェア部品検索システム SPARS-J(Software Product Archiving, analyzing, and Retrieving System for Java)を研究している．SPARS-J は Java プログラムを対象とした部品検索システムで，収集された部品に対して解析およびインデックス付けを行うことで検索機能を提供する．SPARS-J は現在のところキーワード検索に対応しており，検索はクラス単位で行われる．検索結果を表示する際，検索された部品を評価し順位付けし，選別して表示する仕組みが必要となるが，SPARS-J においては，部品間の利用関係を元に定められた Component Rank[3-4]に基づいて順位付けを行うことで，利用実績の高い汎用的な部品の取得を容易にしている．

SPARS-J は，入力されたソースコードに対してデータベースを構築するデータベース構築部と，データベースから情報を取得することで部品の検索を行う部品検索部から構成されている．SPARS-J を構成する部位の概念図を図 3.1 に示し，次節以降に各部の説明を述べる．

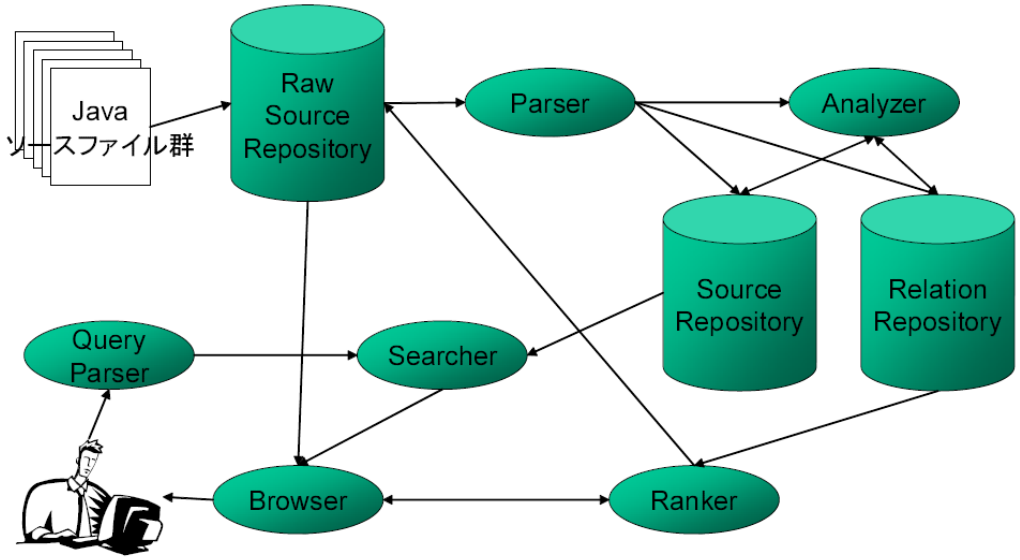


図 3.1　SPARS-J の構成概念図

3.2.2. データベース構築部

データベース構築部では，検索時に必要なデータを提供するために，次の手順で入力された Java ソースコードからデータベースの構築を行う．部品のグループ化，Component Rank の計算については後述する．

1. 入力された Java ソースコードに対して構文解析を行う事で，利用しているクラス名や，LOC，トークン数などのメトリクスを抽出し，部品情報としてデータベースに保存する．
2. 上記の手順１と同時に，ソースコード中で出現する全ての単語に対して，インデックス付けを行い，検索用のデータベースを構築する．その際，どこでその単語が出現したか（コメント中，メソッド宣言中，文中など）を出現した単語と共に保存する．
3. 入力された Java ソースコードに対して意味解析を行い，利用しているクラス名が実際にどのクラスを指すのかを判定し，利用関係情報としてデータベースに保存する．SPARS-J では，クラス継承，メソッド呼び出し，フィールド変数参照，抽象クラスの実装などを利用関係とみなしている．
4. 類似した部品のグループ化を行い，一つの部品として扱う．
5. 部品間の利用関係をもとに，Component Rank を求め，部品の評価値として保存する．

3.2.2.1. 部品のグループ化

一般に，部品の集合には，コピーした部品や，コピーして一部を変更しただけの部品が数多く存在する．異なるシステムをまたいで全く同じ，もしくはほとんど似た部品が現れる場合，それらの部品が再利用されたのではないかと推測する事ができる．しかし，コピーしたという利用関係に関してはコピー元の特定が難しく，部品グラフ上で利用関係として定義するのは難しい．そこで，類似した部品をまとめて一つの部品群としてみなし，グループ化を行う．その際，それぞれの部品への辺が一つの部品群への辺とみなされるため，コピーされた部品への評価を高くすることができる．

部品間の類似性を評価する指標として，当初は 2 つのソースコードファイル間で一致する行の割合[3-12]を求めていたが,現在はソースコードから LOC,トークン数,利用されている変数などのメトリクスを用いている．

3.2.2.2. Component Rank の計算

一般に，ソフトウェア部品の間には互いに利用する，利用されるという利用関係が存在する．本システムにおいては，この利用関係をもとに全ての部品を対象に評価値（Component Rank）を計算する．計算の際には，各部品を頂点，部品間の利用関係を利用する側からされる側への有向辺として，部品間の関係を部品グラフとして表現する．開発者はある部品を参照した後に，グラフ中の辺に沿って利用関係のある部品の一つを見ると仮定することで，この部品グラフを開発者の閲覧行動に関するマルコフ連鎖モデルとみなし，定常状態において各部品が参照されている確率を求め，それを Component Rank としている．この Component Rank により，ただ単に利用数が多い部品だけでなく，利用数が多い部品が利用している部品も重要であると評価する事ができる．

3.2.3. 部品検索部

部品検索部では，構築されたデータベースを用いて，部品の検索を行う．現在のシステムでは，基本検索として全てのソースコードからのキーワード検索を実現しているが，検索条件を指定する事で，コメントのみにキーワードが現れる部品を省く，クラスやメソッド定義にキーワードが出現する部品のみを表示するなどの詳細な検索を行うことができる．検索の手順は以下のようになっている．

1. ユーザーはブラウザを通じてキーワードを入力する事で，部品検索部にクエリーを投げる．
2. 部品検索部はクエリーをキーワードの集合に分解し，データベースに対してそれぞれのキーワードが出現する部品を照会する．キーワードが複数ある場合には，結果を統合する．
3. ユーザーが指定した検索条件に基づいて，与えられたクエリーにマッチした部品を Component Rank の順に並べ，それを検索結果として出力する．
4. ユーザーはブラウザを通じて検索結果を受け取る．ユーザーはさらにキーワードを追加する事でより詳しい検索を行い，検索結果として表示された部品に関して，ソースコードや，その部品のメトリクスなどの部品の詳しい情報を取得する事が出来る．

3.2.4. 本研究との接点

現在，我々のチームで開発している SPARS-J では，コピーアンドペーストを利用した転用先の利用関係も捕捉するために，大量の部品の中から類似した部品をまとめて一つの部品群へのグループ化を行う必要がある．SPARS-J のプロトタイプシステムではこの類似性比較に文字列比較を用いた類似性計測手法を採用していた．しかし，この手法には大量のソースコードを対象にした場合の類似性計測において解析コストの面で問題がある．

そこで本研究では，この文字列比較を用いた類似性計測手法の内容を考察した上で，より解析コストの低い，メトリクスを用いた類似性比較手法の実装を試みた．次節ではこの文字列比較を用いた類似性計測手法の内容と解析コストについて述べる．

3.3. 文字列比較を用いた類似性計測ツール SMMT

本節では，文字列比較を用いた既存の類似性計測システム SMMT の手法とその解析コストについて説明する．

3.3.1. 類似性の定義

ソフトウェアシステムP はそれを構成する要素の集合と考え，$P=\{\mathrm{p1}, \cdots, \mathrm{pm}\}$ と書く．二つのソフトウェアシステム$P=\{p1,\cdots. pm\}$, $Q=\{q1, \cdots,qn\}$ に対し，等価な要素の対応 $Rs \in P \times Q$ が得られるとする．

P とQ のRs に対する類似性 $S(0 \leq S \leq 1)$ を次のように定義する．

$$S(P,Q) \equiv \frac{|\{pi|(pi,qj) \in Rs\} + |\{qj|(pi,qj) \in Rs\}|}{|P| + |Q|}$$

類似性 Sは、対応Rs に含まれるP, Q の要素数をP とQ の総要素数で割ったものとして定義している．Rs に関係しないP, Q の要素が増えることによってS は下がる．$Rs = \phi$ では，$S = 0$ となる．また，P とQ が等価である時，$\forall i(pi, qi) \in Rs$ となり$S = 1$ となる．

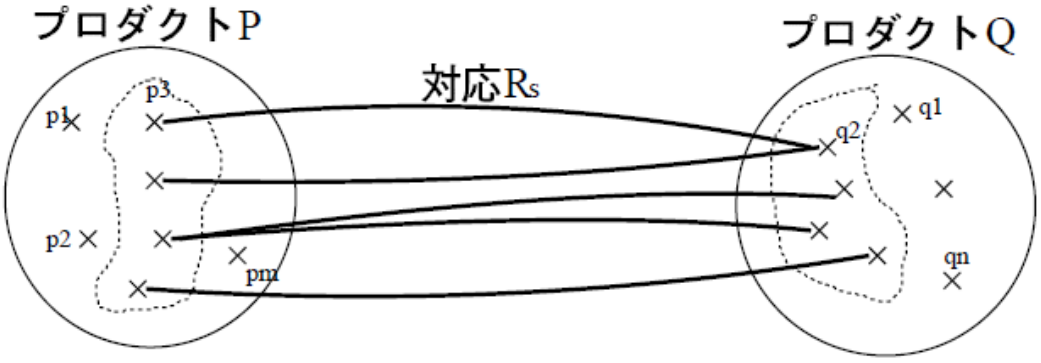


図 3.2　要素の対応 Rs

3.3.2. 類似性のメトリクス

- 等価な行を用いたメトリクスS_{line}

  ソフトウェアシステムP, Qに対し，pi, qi をそれぞれの各ファイルの各行とする．直感的にはP, Qそれぞれ各ファイルを連結した一つのファイルを考え，その各行を構成要素とする．等価な行の対応を調べることによって対応Rs が与えられる．この類似性メトリクスをS_{line} とする．これは，ファイル名やファイルの大きさに影響されず，直感的に近い値が得られることが期待される．その他に，いろいろ類似性のメトリクスを考えることが出来るが，求める手間やその効果を考え，ここではS_{line} について議論する．

3.3.3. 類似性メトリクスの計算手法

二つのソフトウェアシステム P, Qに対し S_{line}を求めるためには, Rsを得ること，すなわち P の各ファイルの各行が Qのどのファイルのどの行に対応するか，又は対応する行がないか，を知る必要がある．

このための簡単な方法としては，Pの各ファイルを連結したファイル p_{all}と Qの各ファイルを連結したファイル q_{all}を作り，p_{all}と q_{all}の間の共通部分を求めて，そこに含まれる各行を Rsとすることが考えられる．共通部分の発見には，通常，最長共通部分列を発見するアルゴリズム LCS[3-8,3-9,3-11]を用いた diff[3-3]等のツール

が便利であるが，ファイル名が変わるなどしてファイルの連結順が変わった場合には，共通部分列として検出が出来なくなる．

そこで，ここではコードの重複（クローンと呼ぶ）を求めるためのツール CCFinder[2-6]と diff とを組み合わせて Rs を求める．

CCFinder は，与えられたプログラムファイルの内に存在する同じプログラム文の系列を効率よく検出し，出力する．ただし，コメントや改行，空白の違い，また，変数や関数の名前（識別子）の違いがあっても同じものとして検出する．

まず CCFinder を用いて p_{all} と q_{all} の間に存在するクローンを検出する．クローン中の各行は Rs の要素とする．CCFinder は，後述のように保守作業にとって無意味なクローンは除去されて出力される．しかし，除去されるものの中には類似性における対応としては有用なものもある．

そこで，検出されたクローンを持つファイル間に対し，共通部分列を diff によって求め，そこに含まれる各行も Rs の要素とする．二種類のツールを組み合わせることで，意味のある行の対応を効率よく求めることができる．

CCFinder では，プリプロセッサ命令（C 言語の#include 行など）は検出対象から除外される．そのため，diff を用いた差分情報を加えることにより，同一行と判断できる行が増加し，より有効な行の対応が求められると考える．後節で述べる適用実験の結果，対応をもつ行は一割程度増加する．

また，diff だけを用いた対応の抽出の場合，ディレクトリ構造を保ったまま二つのソフトウェアシステムを入力とすると，二つのソフトウェアシステムの間で同一の構造を持つ必要があり，ファイル名の変更やディレクトリ構造の変化に追従できない．そのため，ここでは CCFinder と diff を組み合わせて対応を求める方法を採用する．

3.3.4. 類似コードの対応関係の計算方法

CCFinder と diff を用いた対応の求め方の例を述べる．図 3.2 に示すようにソフトウェアシステム P,Q があり，P は 2 つのファイルから構成され，Q は 4 つのファイルから構成されているとする．また，Q は P の後継バージョンとする．Q を構成するファイルの中でファイル A とファイル B は，P のファイル A とファイル B を基にしており，ファイル C はファイル A を基に新しく作成されたファイルである．さらに，ファイル D は新規に作成されたファイルとする．

この二つのソフトウェアシステム P，Q の Rs を求めるために，まず CCFinder を用いて P と Q の間に存在するクローンの検出を行う．検出されたクローンからクローンを含む行の間に対応を結び Rs の要素とする．さらに，クローンが一つでも存在するファイルの組を探す．図 3.3 の場合，矢印で結ばれた P のファイル A と Q の

ファイル A, Pのファイル B と Qのファイル B, Pのファイル A と Qのファイル C の間にクローンが一つ以上検出されたとする.

次に, これらの 3 つの組に対してのみ diff を用いてファイル間の差分を求める. 差分の結果から共通行と判断された各行も Rs の要素とする. 全てのファイルの組み合わせに対して diff を実行しないため, 処理時間は短くなる. また, C 言語の#include 行といった多くのファイルに存在するようなプリプロセッサ命令行の対応もクローンが検出されたファイルの組に対してのみ付けられる. そのため, 単純に共通行であるからといって対応はせず, 意味のある行のみが対応する.

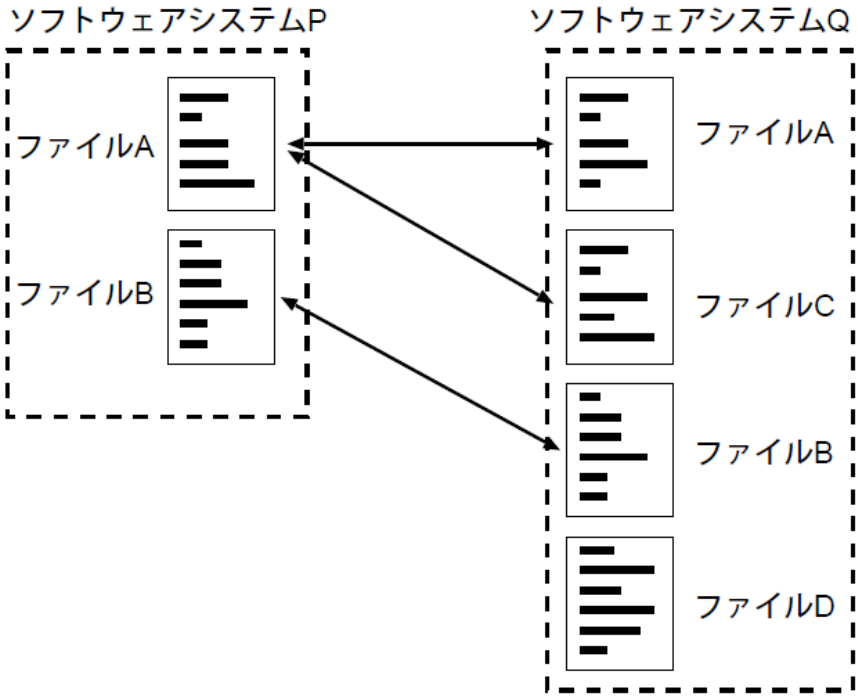


図 3.3 類似コードの対応の求め方

3.3.5. 類似性計測ツール SMMT

S_{line} を求めるためのアプローチに基づき, S_{line} を計測するツールSMMTが作成されている. 以下に, ツールSMMTの入出力と処理の流れを述べる. 図 3.4に処理の流れを表す.

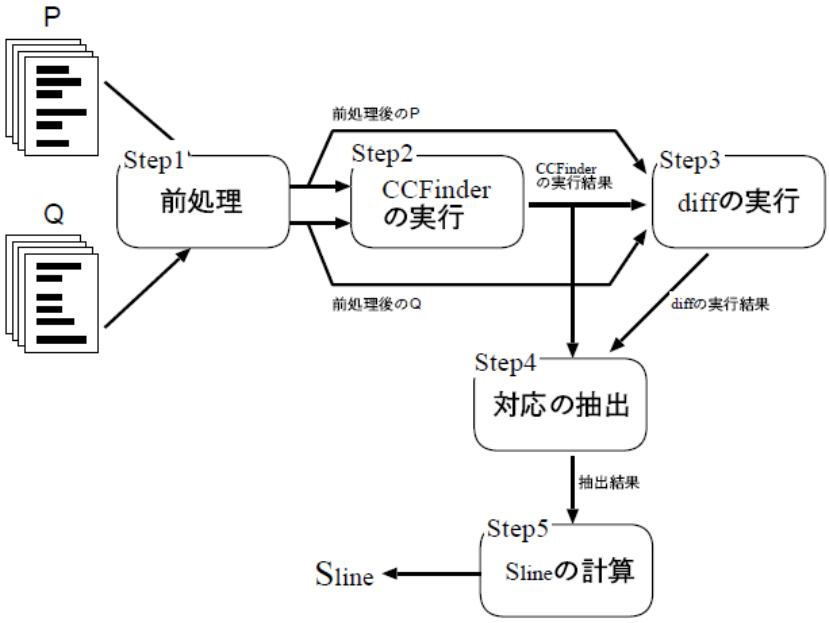


図 3.4　SMMT の処理の流れ

入力：二つのソフトウェアシステムP とQ
出力：P とQ の類似性メトリクス $Sline$ ($0 \leq Sline \leq 1$)

Step1: 前処理

生成されるプログラムの機能に影響を与えない部分を取り除く．この処理は，用いられているプログラミング言語によって異なる．例えば，C 言語で記述されたファイルの場合，コメント部分，空行をすべて取り除く．これにより，diff を実行した時の類似性の精度を向上させる．

Step2: CCFinder の実行

与えられた二つのソフトウェアシステムを入力としてCCFinder を実行させる．CCFinder の実行にあたって，最低一致トークン数を20 とした．最低一致トークン数とは，一致するクローンを含むトークンの長さの最低値を表す．ツールのオプションとして，この値は変更可能である．

Step3: diff の実行

CCFinder の実行の結果，一つでもクローンが発見されたP とQ のファイルの組の全てに対してdiff を実行する．Step4: 対応の抽出

CCFinder で検出されたクローンのトークン列から，一致している行同士を求める．さらに，diff で求まった差分情報から一致している行同士を計算する．CCFinderかdiff のどちらかで一致している判断された行の組をRs に入れる．

Step5: S_{line} の計算

S_{line}の定義より計算する．ただし，二つのソフトウェアシステムの全行数は前処理後の行数を用いる．ツールSMMT はWindows2000 上で稼働し，その対象言語はC，C++，Java，COBOL である．例えば，二つのソフトウェアシステムの総行数が503884 行のS_{line}を計測する場合，Pentium III 1GHz，メモリ2GBytes のシステムでStep1: からStep5: までの処理に掛かる時間は329 秒である（CCFinder とdiff の実行時間を含む）．

ツール SMMT は Windows2000 上で稼働し，その対象言語は C，C++，Java，COBOL である. 例えば, 二つのソフトウェアシステムの総行数が 503884 行の Sline を計測する場合, PentiumⅢ 1GHz, メモリ 2GBytes のシステムで 329 秒を要する．

3.4. 類似性メトリクス比較を用いた類似性計測手法の提案

3.4.1. 本章で解決したい課題

前節で述べた類似性計測手法は，一回あたりの解析コストが高い上に，新しく類似性を比較したいソースコードが現れた場合に，毎回全ソースコードに対して文字列比較するため，非常に時間がかかり，大量の比較を行うのには不向きである．そのため，現在我々の研究チームが開発しているプロジェクト SPARS-J は，類似性を比較する対象が大量のコード群になるので，大量のソースコード間の類似性計測に特化した，高速かつ低コストを実現する類似性検出方法の実装が必要となる．

そこで本研究では，SPARS-J が対象としている Java ソースコードについて，プログラム間の類似性を計測する手法を提案する．本手法では，構文解析時に得られる静的性質メトリクスを数値に抽象化し，それを比較して類似性を求めているため，解析コストを低く抑えることが可能となる．以降，提案手法について説明を行う．

3.4.2. 類似性の定義

本研究では，プログラムの類似性を比較するのに，構文解析時に予め取得しておいた静的性質メトリクスのみを用いる．プログラムの類似性を，以下の 2 つの視点から分析してそれぞれの類似性を算出し，それを掛け合わせたものを最終的な類似

性とする．ただし，本研究では Java の予約語として Java1.3 版の言語仕様[3-6]に従う．

1. 構成トークン類似性

プログラムの表層的特徴の一つとして，そのソースコードに記述されているトークンの種類別使用頻度が考えられる．ソースコードとはトークンの集合であると考えられるので，構成しているトークンの数と種類が良く似ているプログラム同士は類似性が高いといえる．

トークンの構成類似性の分析方法として，分析プログラムの言語仕様に基づく全ての予約語，演算子，に加え，識別子の各数をメトリクスとして用いる．つまり，変数名やメソッド名は全て一つの識別子トークンという範疇に集約される．具体的な名前は変更されてもプログラムの動作に影響しないので考慮しない．

2. 制御構造類似性

プログラムソースコードの構造的特徴を表すために，サイクロマチック数，クラス当たりのメソッド宣言数，ネストの深さなどをメトリクスとして採用し，2 部品間のこれらのメトリクス値の差分が設定した閾値以内であれば類似と判定する．

3.4.3. 類似性のメトリクス

3.4.2 節で定義した 2 つの類似性を算出するため，Java プログラムにおける類似性メトリクスとして，Java ソースコードのトークンの出現回数を用いたメトリクス，Java ソースコードの複雑度メトリクスを定義した．以降，この 2 つのメトリクスについて説明を行う．

1. Java ソースコードのトークンの出現回数を用いたメトリクス

本手法におけるトークンとは，ソースコード上での意味のある単語のこと指す．Java ソースコードのトークンは，4 種類に分けることができ，それぞれのメトリクスを以下のⅠ～Ⅳに定義する．そして，各メトリクスの値の合計を，トークンの出現回数を用いたメトリクス Ttoken として保存する．この時，メトリクス Ttoken は，プログラムのサイズを表すメトリクスとして捉えることができる．

既存のメトリクス測定手法では，サイズのメトリクスに LOC(Line Of Code)が使われるのが一般的であったが，本手法ではコメントや空行，改行位置などのプログラマによる違いは，プログラムの動作には何の影響も及ぼさないノイ

ズであるとみなし，サイズの指標としてそれらのノイズの影響を受けないトークンの総出現回数を採用している．

これにより，性質上小さなサイズのプログラムが多く存在する，オブジェクト指向言語プログラムに対しても，本来のサイズに対するノイズの影響を最小限に抑えることが可能となる．

I. 予約語の出現回数を用いたメトリクス

Java ソースコードに対して，Java の仕様で定められた 49 種類の予約語の各出現回数を，以下の表 3.1 に示すメトリクスとして記録する．なお，Java の予約語の中にはこれ以外に const と goto があるが，この二つには機能が実装されていないため，本手法においてはスペースや改行記号などと同じくノイズとみなし類似性メトリクスには含まない．

II. 記号の出現回数を用いたメトリクス

Java ソースコードに対して，Java の仕様で定められた 9 種類の記号の各出現回数を，以下の表 3.2 に示すメトリクスとして記録する．

III. 演算子の出現回数を用いたメトリクス

Java ソースコードに対して，Java の仕様で定められた 37 種類の演算子の各出現回数を，以下の表 3.3 に示すメトリクスとして保存する

IV. 識別子の出現回数を用いたメトリクス

Java ソースコードに対して，プログラマが任意に定める識別子（変数名やメソッド名など）の延べ出現回数を，メトリクス NOidentifier として保存する

2. Java ソースコードの複雑度メトリクス

Java ソースコードに対して，以下の表 3.4 に示す 6 種類のメトリクスを，ソースコードの複雑さの指標として保存する．さらにそれぞれの部品に対して，類似判定の際の各メトリクス値の差分の許容限界値である閾値を表 3.4 の通りに設定する．

表 3.1　類似性メトリクス（予約語）

通番	小分類	メトリクス名	説明
1	データ型関連予約語	NOvoid	予約語 void の出現回数
2	データ型関連予約語	NOnull	予約語 null の出現回数

3	データ型関連予約語	NOchar	予約語 char の出現回数
4	データ型関連予約語	NObyte	予約語 byte の出現回数
5	データ型関連予約語	NOshort	予約語 short の出現回数
6	データ型関連予約語	NOint	予約語 int の出現回数
7	データ型関連予約語	NOlong	予約語 long の出現回数
8	データ型関連予約語	NOfloat	予約語 float の出現回数
9	データ型関連予約語	NOdouble	予約語 double の出現回数
10	データ型関連予約語	NOboolean	予約語 boolean の出現回数
11	データ型関連予約語	NOtrue	予約語 true の出現回数
12	データ型関連予約語	NOfalse	予約語 false の出現回数
13	クラス関連予約語	NOimport	予約語 import の出現回数
14	クラス関連予約語	NOpackage	予約語 package の出現回数
15	クラス関連予約語	NOclass	予約語 class の出現回数
16	クラス関連予約語	NOinterface	予約語 interface の出現回数
17	クラス関連予約語	NOextends	予約語 extends の出現回数
18	クラス関連予約語	NOimplements	予約語 implements の出現回数
19	クラス関連予約語	NOthis	予約語 this の出現回数
20	クラス関連予約語	NOsuper	予約語 super の出現回数
21	クラス関連予約語	NOnew	予約語 new の出現回数
22	クラス関連予約語	NOinstanceof	予約語 instanceof の出現回数
23	修飾子関連予約語	NOprivate	予約語 private の出現回数
24	修飾子関連予約語	NOpublic	予約語 public の出現回数
25	修飾子関連予約語	NOprotected	予約語 protected の出現回数
26	修飾子関連予約語	NOfinal	予約語 final の出現回数
27	修飾子関連予約語	NOstatic	予約語 static の出現回数
28	修飾子関連予約語	NOabstract	予約語 abstract の出現回数
29	修飾子関連予約語	NOnative	予約語 native の出現回数
30	修飾子関連予約語	NOsynchronized	予約語 synchronized の出現回数
31	修飾子関連予約語	NOvolatile	予約語 volatile の出現回数
32	修飾子関連予約語	NOtransient	予約語 transient の出現回数
33	修飾子関連予約語	NOstrictfp	予約語 strictfp の出現回数
34	制御構造関連予約語	NOfor	予約語 for の出現回数
35	制御構造関連予約語	NOwhile	予約語 while の出現回数
36	制御構造関連予約語	NOdo	予約語 do の出現回数
37	制御構造関連予約語	NOif	予約語 if の出現回数

38	制御構造関連予約語	NOelse	予約語 else の出現回数
39	制御構造関連予約語	NOswitch	予約語 switch の出現回数
40	制御構造関連予約語	NOcase	予約語 case の出現回数
41	制御構造関連予約語	NOdefault	予約語 default の出現回数
42	制御構造関連予約語	NObreak	予約語 break の出現回数
43	制御構造関連予約語	NOcontinue	予約語 continue の出現回数
44	制御構造関連予約語	NOreturn	予約語 return の出現回数
45	例外関連予約語	NOtry	予約語 try の出現回数
46	例外関連予約語	NOcatch	予約語 catch の出現回数
47	例外関連予約語	NOfinally	予約語 finally の出現回数
48	例外関連予約語	NOthrow	予約語 throw の出現回数
49	例外関連予約語	NOthrows	予約語 throws の出現回数

表 3.2　類似性メトリクス（記号）

通番	メトリクス名	説明
1	NOlpar	記号”(“の出現回数
2	NOrpar	記号”)“の出現回数
3	NOlbrace	記号”{“の出現回数
4	NOrbrace	記号”}“の出現回数
5	NOlbrack	記号”[“の出現回数
6	NOrbrack	記号”]“の出現回数
7	NOdot	記号”.“の出現回数
8	NOsemicolon	記号”:“の出現回数
9	NOcommma	記号”,“の出現回数

表 3.3　類似性メトリクス（演算子）

通番	メトリクス名	説明
1	NOassign	演算子”=”の出現回数
2	NOequal	演算子”=”の出現回数
3	NOplus	演算子”+”の出現回数
4	NOplus_assign	演算子”+=”の出現回数
5	NOgt	演算子”>”の出現回数
6	NOlet	演算子”<=”の出現回数
7	NOminus	演算子”-”の出現回数
8	NOminus_assign	演算子”-=”の出現回数

9	NOlt	演算子”<”の出現回数
10	NOget	演算子”>=”の出現回数
11	NOtimes	演算子”*”の出現回数
12	NOtimes_assign	演算子”*=”の出現回数
13	NOcomplement	演算子”!”の出現回数
14	NOnot_equal	演算子”!=”の出現回数
15	NOdivide	演算子”/”の出現回数
16	NOdivide_assign	演算子”/=”の出現回数
17	NObnot	演算子”~”の出現回数
18	NOand	演算子”&&”の出現回数
19	NbandO	演算子”&”の出現回数
20	NOband_assign	演算子”&=”の出現回数
21	NOconditional	演算子”?”の出現回数
22	Noor	演算子”||”の出現回数
23	NObor	演算子”|”の出現回数
24	NObor_assign	演算子”-=”の出現回数
25	NOcolon	演算子”:”の出現回数
26	NOincrement	演算子”++”の出現回数
27	NObxor	演算子”^”の出現回数
28	NObxor_assign	演算子”^=”の出現回数
29	NOdecrement	演算子”--”の出現回数
30	NOmodulo	演算子”%”の出現回数
31	NOmodulo_assign	演算子”%=”の出現回数
32	NOsleft	演算子”<<”の出現回数
33	NOsledt_assign	演算子”<<=”の出現回数
34	NOsright	演算子”>>”の出現回数
35	NOsright_assign	演算子”>>=”の出現回数
36	NOusright	演算子”>>>”の出現回数
37	NOusright_assign	演算子”>>>=”の出現回数

表 3.4　類似性メトリクス（複雑度）

通番	メトリクス名	説明	閾値
1	NOcyclomatic	サイクロマチック数	0
2	NOdeclamethod	メソッド宣言の数	1
3	NOcallmethod	メソッド呼び出しの数	2

4	NOnestingdepth	ネストの深さ	0
5	NOclass	クラス宣言の数	0
6	NOinterface	インタフェース宣言の数	0

3.4.4. 類似性の判定方法

ソフトウェアシステム Pはそれを構成する要素の集合と考え，$P=\{p1,\cdots,pm\}$と表現する．Pをプログラムソースとすると，各 piはソースコードの各トークンの出現数となる．

前節で定義した 96 種のメトリクスを基に，二つのソフトウェアシステム $P=\{p1,\cdots,p96\}$, $Q=\{q1,\cdots q96\}$を例にとって類似の判定方法を説明する．

部品 Pの全トークン数を Ttotal(P)とすると，

$$\mathrm{Ttotal}(P) \equiv \sum_{k=1}^{96} p\mathrm{k}$$

と表せる．

さらに部品 Pと部品 Qの各トークン数の差分の和を diff(P,Q)とすると，

$$\mathrm{diff}(P,Q) \equiv \sum_{k=1}^{96} |pk - qk|$$

と表せる．

diff(P,Q)が 0 なら，全く同じ種類のトークンを，全く同じ回数用いてプログラムソースを構成していることになるため，コピーアンドペーストで作成した類似部品である可能性が非常に高い．

また，diff(P,Q)が同じ 30 トークンだとしても，全トークン 40 トークン中の差分が 30 トークンであるのと，10000 トークン中に差分が 30 トークンあるのとでは，その性質が大きく違うため，非類似性を求める際には全トークン数に対する差分トークン数の割合で求めるべきである．

そこで，部品 Pと部品 Qの非類似度 D(P,Q ）を次のように定義する．

$$\mathrm{D}(P,Q) \equiv \frac{\mathrm{diff}(P,Q)}{\min(\mathrm{Ttotal}(P),\mathrm{Ttotal}(Q))}$$

本手法では，D(P,Q) < 0.03，つまり部品 P,Qの各要素の差分の合計が全体の 3%以下である P,Qの組を類似部品候補とみなす. この 0.03 という値を設定した理由は，SPARS-J が扱う部品群において希望する類似部品判定ができた際の経験則からこ

の値に設定している．類似という概念は直感的なものであるので，プロジェクトごとにこの値は随時変更されるべきものであると考える．

さらに，D(*P,Q*)が 0.03 未満であった類似部品候補 *P,Q* 間の複雑性メトリクスの差分が全てに示す閾値以内であれば，最終的に *P,Q* は類似部品であると判定する．

3.4.5. 主メトリクスを用いた効率化手法

前節では類似性計測の方法を述べたが，新しい判定をする度に，毎回全ソースコードに対して類似性計測を行うのはコストが高くなる．そこで，まず全メトリクスの中から，重要なメトリクスを主メトリクスとして採用する．そして，図 3.5 のように各部品と，その部品が持つ主メトリクス値のハッシュ値とを対応づけたデータベースを構築する．そのハッシュ値が閾値以内である部品だけを対象として，前節の類似性計測を行うようにすれば，判定結果を変えることなく測定の効率のみが上がる．

主メトリクスの候補として，まず閾値をもつメトリクスである複雑度メトリクスが挙げられる．複雑度メトリクスの値の差分が閾値以内であるかどうかは，類似判定の絶対条件であるので，複雑度メトリクスはその閾値とともに主メトリクスとして採用する．

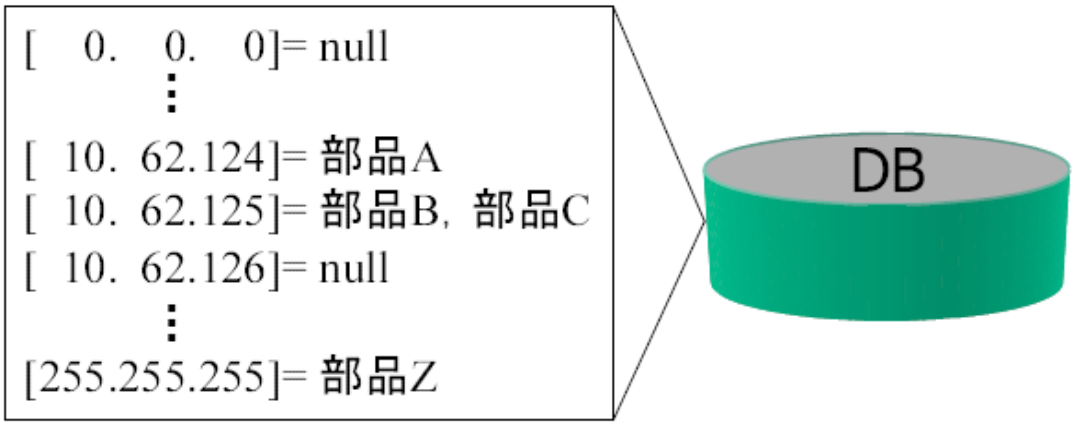


図 3.5　ハッシュ値と部品の対応付けデータベース

しかし，複雑度メトリクスだけを主メトリクスにすると，単純な構造の部品がすべて同じハッシュ値をもってしまい，分類ができずに大きな偏りができる．そこで，部品のサイズを表すメトリクス Ttotal を加工して主メトリクスとして採用することにより，効率的に部品を分類できる．

そこで，まず Ttotal が関係する類似判定の条件について考えてみる．前節の類似性計測条件の一つに相違トークン数の合計が Ttotal の 3%以内であると定義した．この時，Ttotal(*P*)と Ttotal(*Q*)の絶対値の差が 3%を超えるもの(ただし，ここでは Ttotal(*P*) ≥ Ttotal(*Q*)と仮定する)は，たとえ *Q* の全トークンの各値が *P* の各値と同

じか小さかったとしても，Ttotal の差の数は全て差分となるので，最終的に差分は全体の 3%を超え，類似性計測をしても類似でないと判定されるはずである．このことから，主メトリクスを用いてまず Ttotal の差が 3%以内の部品だけを抽出すれば良いと考えられる．

具体的な実現方法として，図 3.6 のように Ttotal を関数 f(n+1)=floor{(f(n)*1.04}(ただし f(0)=0, f(1)=1)を満たす関数 f(n)の値ごとに区切っていき，それぞれの範囲を群に分割する．この時，f(n)と f(n+1)に挟まれた群を群番号 n の群と呼ぶ．

こうすることで，ある群番号 n に入っている Ttotal の最小値 f(n)*0.97 の値を図 3.6 の点 S とすると，この点 S は必ず第 n-1 群の中に入る．また同様に，第 n 群の最大値*1.03 を図 3.6 の点 L とすると，この点 S は必ず第 n+1 群に入る．つまり，Ttotal を群番号に変換し，閾値が 1 である主メトリクスとして使用すれば，変換後の Ttotal が第 n 群に入る部品に対しては，第 n-1 群，第 n 群，第 n+1 群に Ttotal が入る部品のみが最終的に類似部品となる可能性を残していることになり，それ以外の群にある部品を事前に類似比較対象から外すことができる．上記の計算に出てきた 0.97 および 1.03 という係数は，変換前の Ttotal の閾値が 0.03 という設定の場合の数値であり，別の閾値を設定した場合は，その閾値にそった係数を設定する必要がある．

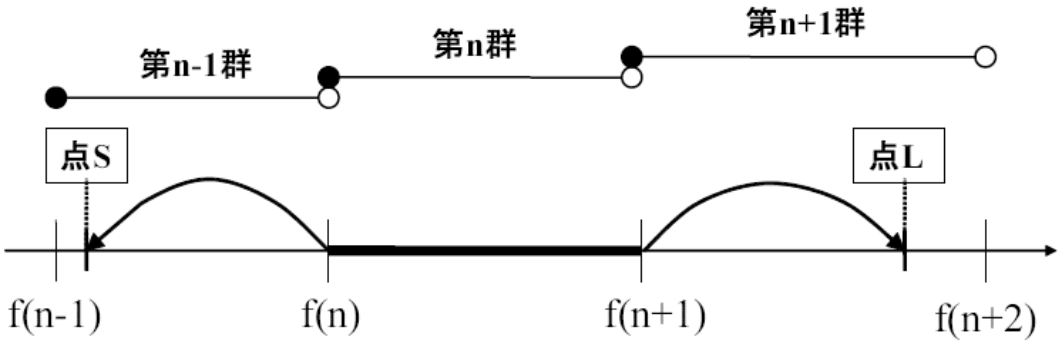


図 3.6　Ttotal の群分割

3.4.6. 類似性計測ツール Luigi

提案手法を基にツールを実装した．ツールは図 3.7 のような構成になっている．

類似性を判定したい Java のソースコードを入力すると，Luigi の構文解析部は入力されたソースコードの類似性メトリクスを計測し，部品の ID とともにデータベースに保存する．同時に，主メトリクスによるハッシュ値も計算し部品 ID と対応させて記録しておく．

メトリクス値の計測が終わると，類似性計測部にてハッシュ値が閾値以内である部品を抽出し，それらのメトリクス値から類似性計測を行って類似部品とされた部

品対を部品群としてまとめる．この際，類似部品が無かったものは単独部品群として扱う．そして，最終的に部品群のデータを出力する．

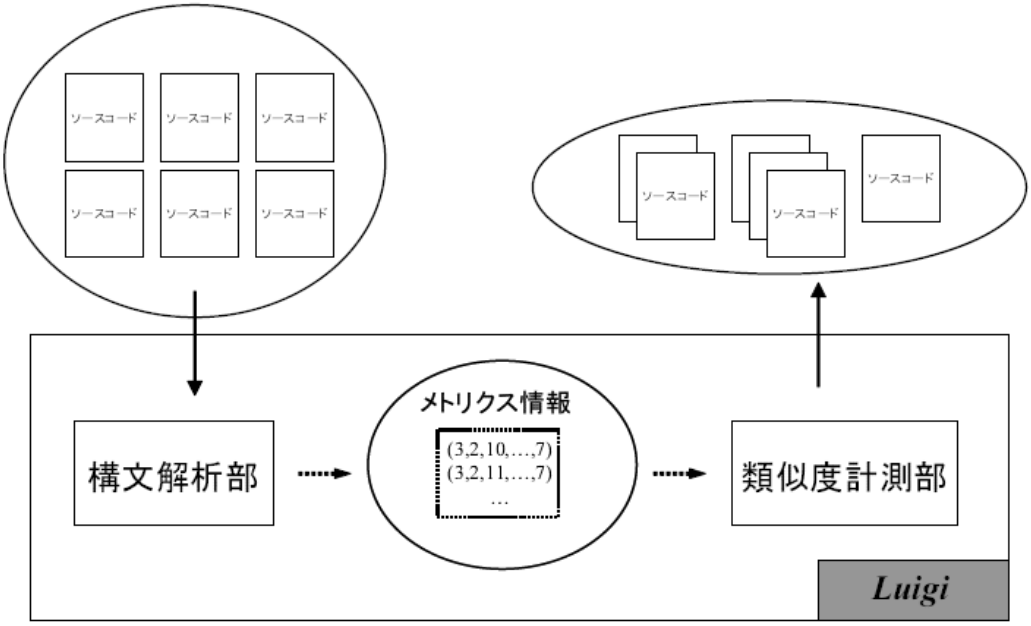


図 3.7　Luigi ツールの内部構成

3.5. 提案手法の検証結果とその分析

3.5.1. 分析の概要

本節では，提案手法を実現した Luigi を用いて適用実験を行い，提案手法の有効性を評価する．実験は，JDK1.3[3-6]内の 431 個のクラスに対して Luigi を適用する．主メトリクスを設定する事で適用対象を絞り込む事ができるが，実験においては，次のような 4 通りの設定を用意し，それぞれの設定下で Luigi を適用した．

1. 主メトリクスを使用しない
   この場合，一つの部品について全てのクラスに対して判定を行う．
2. サイクロマチック数による分類
3. サイクロマチック数およびメソッド数による分類
   この場合，制御構造類似性の一部を満たしている部品のみが抽出される．
4. サイクロマチック数およびメソッド数およびクラスタによる分類
   この場合，制御構造類似性による基準に加えて，構成トークン類似性を満たす可能性がある部品のみが抽出される

3.5.2. 分析の結果

適用実験の結果を表 3.5，表 3.6 に示す．表 3.5 中の主メトリクスＣはサイクロマチック数,Ｍは宣言メソッド数,Ｔはトークン数による分類が行われたことを指し，それぞれ前述した分類方法に対応している．

事前分類クラスタ数とは主メトリクスを用いて分類したときにいくつの部品群に分かれたかを指す．未使用の場合は，主メトリクスにより分類されなかったとし，1としている．Ｃの場合は，431 個のクラスが 21 個のグループに分類され，各グループは平均 21 個の部品から構成されていた．以下，分類条件が細かくなるほど，細かく分類されることがわかる．

最終クラスタ数は，本ツールを用いて解析を行った結果，いくつの類似部品群に分かれたかを示している．主メトリクスにおいて採用された分類条件が制御構造類似性および構成トークン類似性の判定において，類似であるとみなすのに必要な条件を破綻させるものではなかったので，最終クラスタ数は全て同じとなった．さらに，各パターンにおける 278 個の最終クラスタの構成部品を調べたが全て同じであった．これは，主メトリクスにおいて採用された分類条件が制御構造類似性および構成トークン類似性の判定において，類似であるとみなすのに必要な条件を破綻させるものではなかったからであると考えられる．実際には，主メトリクスを細かくしていくことは，主メトリクスを用いた絞込みにおいて，前倒しで判定を行っていることに他ならない．主メトリクスの判定条件を十分考慮する事で最終的な類似判定結果を変えることがないことを確認した．

解析コストは，本ツールの開始から終了までの時間を計ったもので，構文解析におけるメトリクス抽出時間は含んでいない．

表 3.5　Luigi 適用実験結果(解析コスト)

主メトリクス	事前分類クラスタ数	最終クラスタ数	解析コスト(sec)
なし	1	278	5.02
C	21	278	0.56
C,M	85	278	0.29
C,M,T	232	278	0.16

表 3.6　Luigi 適用実験結果（解析精度）

全部品数	Luigi が類似と判定した部品数	SMMT が類似と判定した部品数	Luigi と SMMT 両方が類似と判定した部品数
431	201	199	156

3.5.3. 解析精度に対する考察

まず，定性的な分析として Luigi の分類した類似部品を確認したところ，我々の意図した類似部品が抽出できていることがわかった．

次に，定量的な分析として SMMT の抽出した類似部品を正しい結果と捉えた場合に，Luigi が抽出した類似部品の適合率と再現率を算出することで評価を行った．

まず，適合率について考察する．実験対象の 431 個の部品群に対して類似測定を行った結果，Luigi が判定した類似部品は表 3.6 に示す通り 201 個であった．このうち，SMMT でも類似と判定された部品は 156 個あった．この時，適合率は 156/201=0.776 となる．

次に，再現率について考察する．表 3.6 に示す通り，SMMT が類似と判定した部品 199 個のうち，Luigi も類似部品と判定した類似部品は 156 個であった．よって，再現率は 156/199=0.783 となる．

次に，適合しなかった部品について考察する．適合しなかった部品を調査したところ，コピーしてメソッドを増やしたような部分的に高い類似性をもつものが多かった．この原因として，Luigi では部品全体のメトリクス値のみを扱う方法で高速化を図ったため，このような部分的に高い類似性を評価できないことがわかった．本研究ではクラス単位のコピーアンドペーストを捕捉することを目的としているため，この問題に対応しなかったが，今後さらにメソッド単位のコピーアンドペーストを捕捉するためにはメソッド単位でのメトリクス値比較が有効になると考える．

次に，再現しなかった部品について考察する．再現しなかった部品を調査したところ，非常に規模の小さい部品（総トークン数が 10 数個以内）が多いことが分かった．この様な小さい規模の部品に関しても Luigi は類似判定の対象としていたのに対して，SMMT では類似対象から外していた．そのため，両ツールの類似判定対象の設定が違っていることが原因であった．

以上の分析をまとめると，両ツールの類似判定のアルゴリズムが異なることが原因で生まれた差異が大きいため，類似判定のアルゴリズムが異なるツールに対して，その解析精度を直接比較することは必ずしも適切ではないことがわかった．

以上の評価結果から，コピーアンドペーストを用いた Java ソフトウェア部品の再利用関係を捕捉するという目的に置いては，本手法で提唱したメトリクス値を用いた比較を行うことで定性的には十分精度の高い結果が得られることが分かった．

3.5.4. 解析コストに対する考察

本節では，解析コストについて考察する．まず，アルゴリズムの違いにより，文字列比較を用いた類似性計測ツール SMMT の場合，メトリクス抽出を前もって行う必要は無い．一方で，Luigi の事前にメトリクス抽出を行う必要があるため，このコストをどう扱うかを決める必要がある．SPARS-J においては，類似部品測定ツール

の種類に関わらず利用関係を抽出するために必ず構文および意味解析が必要となる．さらに，構文解析ルーチンにおけるメトリクス抽出の時間は微々たる物であるため，メトリクス抽出のコストを考慮する必要はないと考えられる．つまり，主メトリクスによる分類を行った後に残りの類似性メトリクス比較により類似性を判定する手法は，文字列比較を用いた類似性計測ツールの解析結果と単純比較してよいと考える．

文字列比較を用いた類似測定ツール SMMT を利用して JDK1.3 に属する 431 個のクラスに対して類似測定を行うのに 24.30 秒を要した．一方，全く同じ 431 個のクラスに対して類似度メトリクスを用いた類似部品測定ツール Luigi で類似判定を行った結果，表 3.5 に示すように，主メトリクスを用いた事前分類を行わない場合でも，解析コストは 5.02 秒であり，5.02/24.30＝約 1/5 に低減されている．さらに主メトリクスの組み合わせを調整すると 0.16 秒しか要しない．つまり，SMMT に比べて Luigi の解析コストは 0.16/24.30＝約 1/150 のコスト低減が実現できている．これにより，数値による判定，および主メトリクスを用いた事前分類を用いて，類似判定する対象部品の絞込みを行う事で，従来の文字列比較を利用した手法に比べて非常に効率的に判定を行うことができることを確認した．

さらにこの手法の場合，分類を行うべき部品の数が多くなった場合，主メトリクスによる分類条件を追加する事で，解析対象をより狭くする事ができる．これにより，更なる効率化を図る事ができ，大規模な部品の集合にも対応が可能である．このことから，SPARS-J においては，本提案手法は文字列比較を用いた類似性計測手法よりも大きな威力を発揮すると期待できる．

3.6. まとめと今後の課題

本研究では，Java ソースコードの部品情報を数値化した類似性メトリクスの比較を用いた類似性計測方法を提案した．さらに，ソフトウェア部品の中から類似した部品を検出し，部品群にまとめる機能を実装したツール Luigi の試作を行った．その中で，主メトリクスによる事前分類を導入し，測定回数の効率化も行った．最後に JDK1.3 のソースコードから 431 個のクラスを対象とした適用実験を行い，提案手法が既存の文字列比較を用いた類似性計測手法よりも十分低コストかつ効率的であることを確認した．

今後の課題としては，類似判定の精度に関する詳細な分析や，Java 以外のプログラミング言語への適用などが挙げられる．

第4章　むすび

4.1. まとめ

本論文では，ソフトウェア保守の支援を目的として，ソースコード静的解析手法のうち，以下の2つの手法を提案した．

1. **アクセス修飾子過剰性に関する解析**
2. **ソフトウェア部品間における類似性計測**

1. については，ソフトウェアを解析し，フィールド及びメソッドの呼び出し関係をグラフ化することで実際の呼び出し元の範囲と,当該フィールド及びメソッドに宣言されているアクセス修飾子の呼び出し範囲の乖離の組合せを AE として分析する手法を提案し，AE を自動的に解析・修正するツール ModiChecker を開発した．

また，1.ではさらに，同じソフトウェアの複数のバージョン間における過剰なアクセス修飾子の数の変化量を比較分析することで,過剰なアクセス修飾子がどのように発生し，どのように修正されていくのかを分析した．その結果，著名なオープンソースソフトウェアであっても,一度作りこまれてしまった過剰なアクセス修飾子はほとんど修正されないことがわかった．

これにより，開発者は意図せずアクセス修飾子を過剰に広く設定してしまったフィールドやメソッドが自身の保守対象のプログラムにも多く残っている可能性があることを認識できる．さらに，ModiChecker を利用することで，各フィールドおよびメソッドに宣言されているアクセス修飾子に過剰性があるかどうかを高速に知ることができる．さらには，そのような過剰なアクセス修飾子を設計上呼び出してはいけない範囲から呼び出そうとしていないかどうかを事前に確認したり,ツールが推奨するアクセス修飾子に自動で変更したりすることで,潜在バグの顕在化を未然に防ぐことが可能になる．

2. については， ソースコード部品における Java 言語の予約語の出現回数に関するメトリクスと複雑性に関するメトリクスを計測し,2 つのソフトウェア部品間でこれらのメトリクス値の比較を行うことで高速に類似部品を検出する手法を提案した．これにより，従来の文字列比較を用いた手法にくらべて解析時に扱う情報量が格段に低減されるため，解析コストを低く抑えることが可能となる．提案した手法は類似性計測ツール Luigi として実現し，従来の文字列比較による類似分析を実装したツール SMMT と，今回開発した Luigi を用いて，同じソフトウェア群に対して類似分

析を行い，類似測定に関する精度とコストを比較評価することで，提案手法の優位性を示した．

これにより，大規模かつ複雑なプログラムの保守を行う場合でも，開発者は修正対象のソースコードの類似部品を高速に検索できる．そのため，修正対象のソースコードに対する修正と同じもしくは類似した修正を施す可能性のある類似部品を効率良く探すことができるようになる．

4.2. 今後の研究方針

今後，本論文で述べた研究成果を応用し，ソフトウェア保守作業をより容易なものにしていきたいと考えている．

具体的には，本論文で提案したソースコード静的解析手法およびツールを実際のソフトウェア保守プロジェクトに適用し，開発者からフィードバックを得ることで手法およびツールを改善したい．また，本論文で扱ったアクセス修飾子過剰性が，実際のソフトウェア保守においてどのような障害として顕在化するかに関するデータを集め，統計的に分析することで，障害発生につながりやすい潜在バグを事前に検出できる手法の開発を試みたいと考えている．

参考文献

[1-1] M. Page-Jones,: “The Practical Guide to Structured Systems Design”, New York, Yourdon Press, 1980.

[1-2] A.April, and A.Abran,: "Software Maintenance Management: Evaluation and Continuous Improvement", IEEE Computer Society-John Wiley & Sons, Inc., New Jersey, 2008.

[1-3] Nghi Truong, Paul Roe, and Peter Bancroft,: “Static analysis of students' Java programs”, In Proc. ACE '04, 317-325, 2004.

[1-4] IEEE Std 1219,: "Standard for software maintenance", 1998.

[1-5] ISO/IEC 14764:2006,: "software engineering - software life cycle processes - maintenance", 2006.

[1-6] JIS X 0161:2008,: "ソフトウェア技術-ソフトウェアライフサイクルプロセス-保守 Software Engineering-Software Life Cycle Processes-Maintenance", 2008.

[1-7] E. J. Chikofsky, and J. H. Cross,: "Reverse engineering and design recovery: A taxonomy", IEEE Software, Vol.7, No.1, pp.13-17, 1990.

[1-8] Imagix Corporation,: "Imagix 4D", http://www.imagix.com/products/products.html.

[1-9] IBM,: "Rational software modeler", http://www-01.ibm.com/software/awdtools/modeler/swmodeler/

[1-10] T. J. Biggerstaff,: “Design recovery for maintenance and reuse”, Computer, Vol.22, No.7, pp.36-49, 1989.

[1-11] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides,: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[1-12] N. Shi, and R. A. Olsson,: "Reverse engineering of design patterns from Java source code", In Proc. of ASE 2006, pp.123-134, 2006.

[1-13] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis,: “Design pattern detection using similarity scoring”, IEEE Transactions on Software Engineering, Vol.32, No.11, pp.896-909, 2006.

[1-14]L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. Tichy,: "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance", IEEE Transactions on Software Engineering, Vol.28, No.6, pp.595-606, 2002.

[1-15] K. H. Bennet. Software maintenance: A tutorial. In M. Dorfman, and R. H.Thayer eds,: "Software Engineering", IEEE Computer Society Press, 1997.

[1-16] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley,: "Chianti: a tool for change impact analysis of java programs", In Proc. of OOPSLA 2004, pp.432-448, 2004.

[1-17] G. Rothermel and M. J. Harrold,: "A safe, efficient regression test selection technique", ACM Transactions on Software Engineering and Methodology, Vol.6, No2, pp.173-210, 1997.

[1-18] S. R. Chidamber and C. F. Kemerer,: "A metrics suite for object oriented design", IEEE Transactions on Software Engineering, Vol.20, No.6, pp.476-493, 1994.

[1-19] E. J. Weyuker,: "Evaluating software complexity measures", IEEE Transactions on Software Engineering, Vol.14, No.9, pp.1357-1365, 1988.

[1-20] V. R. Basili, L. C. Briand, and W. L. Melo,: "A validation of object-oriented design metrics as quality indicators", IEEE Transactions on Software Engineering, Vol.22, No.10, pp.751-761, 1996.

[1-21] M. Weiser,: "Program slicing", In Proc. of ICSE'81, pp.439-449, 1981.

[1-22] T. M. Meyers and D. Binkley,: "An empirical study of slice-based cohesion and coupling metrics", ACM Transactions on Software Engineering and Methodology, Vol.17, No.1, pp.1-27, 2007.

[1-23] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya,: "A quantitative evaluation of maintainability enhancement by refactoring", In Proc. of ICSM 2002, pp.576-585, 2002.

[1-24] M. Fowler,: " Refactoring: improving the design of existing code", Addison Wesley, 1999.

[1-25] W. F. Opdyke,: "Refactoring object-oriented frameworks", PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[1-26] M. Weiser,: "Program slicing", Proc. of the 5th International Conference on Software Engineering, pp.439-449, 1981.

[1-27] J.Gosling, B.Joy, G.Steele, G.Bracha, A.Buckley,: "The Java Language Specication, Java SE 7 Edition",

[1-28] K. Khor, Nathaniel L.Chavis, S.M.Lovett and D. C. White,: "Welcome to IBM Smalltalk Tutorial ", 1995

[1-29] 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.

[1-30] 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.

[1-31] D. Evans, and D. Larochells,: “Improving Security Using Extensible Lightweight Static Analysis”, IEEE software, vol.19, No.1, pp. 42-51, 2002.

[1-32] 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.

[1-33] Jlint,: http://jlint.sourceforge.net/

[1-34] B. S. Baker,: “Finding clones with Dup: Analysis of an experiment”, IEEE Trans. Softw. Eng., Vol.33, No.9, pp.608-621, 2007.

[1-35] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier,: "Clone detection using abstract syntax trees", In Proc. of ICSM '98, pp.368-377, 1998.

[1-36] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard,: "Scalable and accurate tree-based detection of code clones", In Proc. of ICSE 2007, pp.96-105, 2007.

[1-37] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multi-linguistic token-based code clone detection system for large scale source code”, IEEE Transactions on Software Engineering, vol.28, no.7, pp.654-670, 2002.

[1-38] R. Komondoor, and S. Horwitz,: "Using slicing to identify duplication in source code", In Proc. of SAS 2001, pp.40-56, 2001.

[1-39] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl,: "Assessing the benefits of incorporating function clone detection in a development process", In Proc. of ICSM '97, pp.314-321, 1997.

[1-40] Z. Li, S. Lu, S. Myagmar, and Y. Zhou,: "CP-Miner: Finding copy-paste and related bugs in large-scale software code", IEEE Trans. Softw. Eng., Vol.32, No.3, pp.176-192, 2006.

[1-41] A. Zeller,: “Why Programs Fail”, Morgan Kaufmann Pub., 2005.

[1-42] M. Kim, L. Bergman, T. Lau, and D. Notkin,: "An ethnographic study of copy and paste programming practices in oopl", In Proc. of ISESE 2004, pp.83-92, 2004.

[1-43] A. Aiken,: "Moss (measure of software similarity) plagiarism detection system", http://www.cs.berkeley.edu/ moss/

[1-44] L. Prechelt, G. Malpohl, and M. Philippsen, “Jplag: Finding plagiarisms among a set of programs”, Technical Report 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, 2000.
[1-45] K. Verco, and M. Wise,:“YAP3 : Improved detection of similarities in computer program and other texts”, Proc. of the 27th SIGCSE Technical Symposium on Computer Science Education, pp.130-134, 1996
[1-46] 長橋賢児,: “類似性に基づくソフトウェア品質の評価,” 情処学研報 2000-SE-126, Vol.2000, No.25, pp.65-72, 2000.
[1-47] 山本 哲男, 松下 誠, 神谷 年洋, 井上 克郎,: “ソフトウェアシステムの類似性とその計測ツール SMMT”, 電子情報通信学会論文誌 D-1, Vol.J85-D-I, No.6, pp.503-511, 2002.
[1-48] 日本情報システム・ユーザー協会,:″非機能要求仕様定義ガイドライン - 検収フェーズのモデル取引・整備報告書 UVC (User Vender Collaboration) 研究プロジェクトⅡ報告書 ″, 2007.
[2-1] 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.
[2-2] G.Booch, R.Maksimchuk, M.Engel, B.Young, J.Conallen, and K.Houston, “Object-Oriented Analysis and Design with Applications”, Addison Wesly, 2007.
[2-3] K. Arnold, J. Gosling, D. Holmes,: ”The Java Programming Language, 4th Edition”,Prentice Hall, 2005,
http://docs.oracle.com/javase/specs/jls/se7/html/index.html
[2-4] SourceForge.jp,: http://sourceforge.jp/
[2-5] 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.
[2-6] D. Evans, and D. Larochells, “Improving Security Using Extensible Lightweight Static Analysis ”, IEEE software, vol.19, No.1, pp. 42-51, 2002.
[2-7] 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.
[2-8] FindBugs,: http://ndbugs.sourceforge.net
[2-9] Jlint,: http://jlint.sourceforge.net/

[2-10] N. Rutar,: C. Almazan, and J. Foster, “A Comparison of Bug Finding Tools for Java”, 15th International Symposium on Software Reliability Engineering (ISSRE04), pp.245-256, 2004.
[2-11] Apache Ant,: http://ant.apache.org/
[2-12] jEdit,: http://www.jedit.org/
[2-13] 三宅達也, 肥後芳樹, 楠本真二, 井上克郎,:″多言語対応メトリクス計測プラグイン開発基盤MASUの開発″, 電子情報通信学会論文誌D, vol. J92-D, no. 9, pp.1518-1531, 2009.
[2-14]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, Vol.33, 2013.
[3-1]V.R.Basili, G.Caldiera, F.McGarry, R.Pajerski, G.Page, and S.Waligora,: “The software engineering - an operational software experience”, in Proceedings of 14th International Conference on Software Engineering(ICSE14), pp.370-381, Melbourne, Australia, 1992.
[3-2] C. Braun,:″Reuse″, in John J. Marciniak, editor, Encyclopedia of Software Engineering, John Wiley & Sons, Vol.2, pp.1055-1069, 1994.
[3-3] Diffutils,: http://www.gnu.org/software/diffutils/diffutils.html
[3-4] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto,: “Component Rank: Relative Significance Rank for Software Component Search”, to be appeared in Proceedings of 25th International Conference on Software Engineering (ICSE 25), pp.14-24, 2003.
[3-5] S.Isoda,: “Experience report on a software reuse project: Its structure, activities, and statistical results”, in Proceedings of 14th International Conference on Software Engineering (ICSE 14), pp.320-326, Melbourne, Australia, 1992.
[3-6] J.Gosling, B.Joy, G.Steele, and G.Bracha,: ”The Java Language Specification, Second Edition”, Prentice Hall, 2000.
[3-7] B.Keepence, and M.Mannion,: “Using patterns to model variability in product families”, IEEE Software, Vol.16, No.4, pp.102-108, 1999.
[3-8] W.Miller, and E.Myers,: “A file comparison program”, Software-Practice and Experience, Vol.15, No.11, pp.1025-1040, 1985.
[3-9] E.Myers,: “An O(N D) difference algorithm and its variations, Algorithmica, Vol.1, pp.251-256, 1986.
[3-10] SourceForge,: http://sourceforge.net/

[3-11] E.Ukkonen: Algorithms for approximate string matching. INFCTRL: Information and Computation (formerly Information and Control), Vol.64, pp.100-118, 1985.

[3-12] 横森 励士, 梅森 文彰, 西 秀雄, 山本 哲男, 松下 誠, 楠本 真二, 井上 克郎,: "Java ソフトウェア部品検索システム SPARS-J", 電子情報通信学会論文誌 D-I, VolJ87-D-I, No.12, pp.1060-1068, 2004.