NAIST-IS-DD1261013

博士論文

リファクタリングがソフトウェア品質に及ぼす影響の実証的評価に関する研究

藤原 賢二

2015年8月11日

奈良先端科学技術大学院大学
情報科学研究科 情報科学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

藤原 賢二

審査委員：

飯田 元 教授	（主指導教員）
松本 健一 教授	（副指導教員）
安本 慶一 教授	（副指導教員）
市川 昊平 准教授	（副指導教員）
吉田 則裕 准教授	（名古屋大学）

リファクタリングがソフトウェア品質に及ぼす影響の実証的評価に関する研究*

藤原 賢二

内容梗概

リファクタリングとは，ソフトウェアが抱える設計上の問題をソフトウェアの外部的な振る舞いを変更することなく取り除くことをいう．高品質なソフトウェアを効率良く開発するためには，適切な時期に適切な箇所に対してリファクタリングを実施することが重要である．本論文では，リファクタリングの実施がソフトウェア品質に与える影響を調査するための分析手法と，その支援手法の提案と評価を行った．

初めに，リファクタリングと欠陥混入に着目し，これらの関係を調査するための分析手法を提案した．提案手法では，ソフトウェアの開発履歴からリファクタリングの実施時期，欠陥の混入時期および修正時期を特定する．そして，これらの時期から各活動の頻度を算出して分析者に提示する．提案手法を Columba プロジェクトに適用した結果，提案手法を用いてリファクタリングと欠陥の関係を定量的に評価することが可能であることを確認した．一方で，数千や数万を超えるコミットから構成される開発履歴を対象に提案手法を適用するためには，リファクタリングの実施時期をより高速に特定する手法が必要であることが分かった．

上記問題を解決するために，開発履歴からリファクタリングの実施履歴を高速に復元するための手法（リファクタリング検出手法）の提案と評価を実施した．従来の手法は，任意の 2 バージョン間から実施されたリファクタリングを検出することを目的としていた．そのため，開発履歴中の隣接する全てのバージョン間からリファクタリングを検出するための工夫がされていなかった．提案手法はリファクタリングの検出に必要な構文情報の解析を差分的に実施することで計算時間の短縮を実現

*奈良先端科学技術大学院大学 情報科学研究科 情報科学専攻 博士論文, NAIST-IS-DD1261013, 2015 年 8 月 11 日.

する．メソッド抽出リファクタリングとメソッドの引き上げリファクタリングを対象に提案手法を実装し，従来手法と計算時間および検出精度の比較を行った．その結果，提案手法は従来手法と比較して，メソッド抽出リファクタリングについては約1.7 倍多く，メソッドの引き上げリファクタリングについては従来手法では検出できなかったリファクタリングの履歴を正確かつ高速に検出することができた．

キーワード

リファクタリング，リファクタリング検出，リポジトリマイニング，版管理システム，定量的評価

Methods for Empirical Analysis and Evaluation of Refactoring Instances*

Kenji Fujiwara

Abstract

Refactoring is a technique for improving design quality of the software. It is the process of changing the structure of the program without changing its behavior. Performing refactorings in the appropriate situation and target is important to develop software efficiently. This dissertation proposes a method for empirical analysis and evaluation of the refactoring instances, and its supporting method.

Firstly, we proposed an analysis method for investigating relations among refactorings and defect introductions. The method locates when refactorings, defect introductions and defect fixes were performed from a software history. Then, it calculates frequencies of these occurrences and provides its transitions to the analysis. We applied proposed method to the Columba project which is an open source software project, and confirmed that the proposed method allows us to evaluate the relation among refactorings and defects quantitatively. However, to apply proposed method to the bigger project that contains more than thousands commits, the faster refactoring detection method is essential.

In order to solve the problem of computation cost of existing refactoring detection methods, we proposed novel faster refactoring detection method and evaluated it. Existing detection methods is not refined to decrease computation cost since these methods focusing to detect refactorings from only two versions of the software history. Our proposed method decreases the computation time by

*Doctoral Dissertation, Department of Information Science, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1261013, August 11, 2015.

analyzing syntactic information incrementally. We implemented our proposed method for Extract Method and Pull Up Method refactorings and compared its performance and preciseness with existing methods. As a result, we confirmed the proposed method can detect about 1.7 times Extract Method refactorings compared with existing methods, and can detect Pull Up Method refactorings that are not detectable by existing methods. We also confirmed that the preciseness and computation time of proposed method is higher and shorter than existing methods.

Keywords:

Refactoring, Refactoring Detection, Mining Software Repositories, Version Control System, Quantitative Evaluation

目次

図目次

表目次

関連発表論文

学術論文誌

1. 藤原賢二, 吉田則裕, 飯田元. 構文情報リポジトリを用いた細粒度リファクタリング検出手法. 情報処理学会論文誌. （採録決定）（第 4 章に関連する）
2. 後藤祥, 吉田則裕, 藤原賢二, 崔恩瀞, 井上克郎. 機械学習を用いたメソッド抽出リファクタリングの推薦手法. 情報処理学会論文誌, volume 56, number 2, pages 627–636, 2015 年 2 月. （第 4 章に関連する）
3. 後藤祥, 吉田則裕, 藤原賢二, 崔恩瀞, 井上克郎. メソッド抽出リファクタリングが行われるメソッドの特徴調査. コンピュータソフトウェア, volume 31, number 2, pages 144–150, 2014 年 8 月. （第 4 章に関連する）

査読付き国際会議・ワークショップ

1. Kenji Fujiwara, Hideaki Hata, Erina Makihara, Yusuke Fujihara, Naoki Nakayama, Hajimu Iida, and Ken-ichi Matsumoto. Kataribe: A Hosting Service of Historage Repositories. In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014), pages 380–383, May 2014. Hyderabad, India. （第 4 章に関連する）
2. Kenji Fujiwara, Kyohei Fushdia, Norihiro Yoshida, and Hajimu Iida. Assessing Refactoring Instances and the Maintainability Benefits of Them from Version Archives. In Proceedings of the 14th International Conference on Product-Focused Software Development and Process Improvement

(PROFES 2013), pages 313–323, June 2013. Paphos, Cypros. (第 3 章に関連する)

査読付き国内研究集会

1. 後藤祥, 吉田則裕, 藤原賢二, 崔恩瀞, 井上克郎. 機械学習を用いたメソッド抽出リファクタリングの推薦手法. ソフトウェアエンジニアリングシンポジウム 2014 論文集, pages 170–175, 2014 年 9 月. (第 4 章に関連する)
2. 藤原賢二, 吉田則裕, 飯田元. ソフトウェアリポジトリを対象とした細粒度リファクタリング検出. 第 20 回ソフトウェア工学の基礎ワークショップ (FOSE 2013), pages 101–106, 2013 年 11 月. (第 4 章に関連する)

査読無し国際会議・ワークショップ

1. Kenji Fujiwara, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida. An Approach to Investigating How a Lack of Software Refactoring Effects Defect Density. In IEICE Technical Report, volume 111, number 107, pages 59–62, June 2011. Seoul, Korea. (第 3 章に関連する)

査読無し国内研究集会

1. 後藤祥, 吉田則裕, 藤原賢二, 崔恩瀞, 井上克郎. メソッド抽出リファクタリングが行われるメソッドの特徴調査. 日本ソフトウェア科学会第 30 回大会講演論文集, FOSE1-3, pages 1–6, 2013 年 9 月. (第 4 章に関連する)
2. 藤原賢二, 吉田則裕, 飯田元. 構文情報を付加したリポジトリによるメソッド抽出リファクタリングの検出. 電子情報通信学会技術報告, volume 113, number 24, pages 19–24, 2013 年 5 月. (第 4 章に関連する)
3. 藤原賢二, 伏田享平, 吉田則裕, 飯田元. オープンソースソフトウェアを対象としたリファクタリングが欠陥混入に与える影響の調査. 日本ソフトウェア科

学会 第 28 回大会 講演論文集, 2011 年 9 月. （第 3 章に関連する）
4. 藤原賢二, 伏田享平, 吉田則裕, 飯田元. ソースコード履歴情報に基づくリファクタリングと欠陥の関係分析. 平成 22 年度 情報処理学会関西支部 支部大会 講演論文集, 2010 年 9 月. （第 3 章に関連する）

その他の発表論文

学術論文誌

1. 崔恩瀞, 藤原賢二, 吉田則裕, 林晋平. 変更履歴解析に基づくリファクタリング検出技術の調査. コンピュータソフトウェア, volume 32, number 1, pages 47–59, 2015 年 2 月.
2. 山田 悠太, 吉田 則裕, 藤原 賢二, 飯田 元. トピック抽出を用いたソフトウェア開発履歴の可視化ツール. コンピュータソフトウェア, volume 31, number 2, pages 144–150, 2014 年 5 月.

査読付き国際会議・ワークショップ

1. Patanamon Thongtanunam, Xin Yang, Norihiro Yoshida, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. Reda: a Web-Based Visualization Tool for Analyzing Modern Code Review Dataset. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), pages 605–608, October 2014. Victoria, Canada.
2. Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, Ana Erika Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. Who Does What During a Code Review? Datasets of OSS Peer Review Repositories. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013), pages 49–52, May 2013. San Francisco, California, USA.

3. Xin Yang, Raula Gaikovina Kula, Ana Camargo Cruz, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, and Hajimu Iida. Understanding Oss Peer Review Roles in Peer Review Social Network (PeRSoN). In Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC 2012), pages 709–712, December 2012. Hong Kong, China.
4. Kenji Fujiwara, Kyohei Fushida, Haruaki Tamada, Hiroshi Igaki, and Norihiro Yoshida. Why Novice Programmers Fall into a Pitfall?: Coding Pattern Analysis in Programming Exercise. In Proceedings of the 4th International Workshop on Empirical Software Engineering in Practice, pages 46–51, October 2012. Osaka, Japan. (Best Student Paper Award)
5. Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, Xin Yang, and Hajimu Iida. Using Profiling Metrics to Categorise Peer Review Types in the Android Project. In Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW 2012), pages 146–151, November 2012. Dallas, Texas, USA

査読付き国内研究集会

1. 槇原絵里奈, 井垣宏, 藤原賢二, 上村恭平, 吉田則裕, 飯田元. 初学者向けプログラミング演習における探索的プログラミングの実態調査と支援手法の提案. 第 21 回ソフトウェア工学の基礎ワークショップ (FOSE 2014), pages 123–128, 2014 年 12 月.
2. Xin Yang, Norihiro Yoshida, Kenji Fujiwara, Yong Jin, and Hajimu Iida. Categorizing Code Review Result with Social Networks Analysis: a Case Study on Three Oss Projects. ソフトウェアエンジニアリングシンポジウム 2014 論文集, pages 200–201, 2014 年 9 月.

査読無し国内研究集会

1. 上村恭平, 藤原賢二, 飯田元. Hardware Description Language におけるコードクローンのパターン分類. 電子情報通信学会技術報告, volume 115, number 20, pages 23–28, 2015 年 5 月.
2. 金勇, 藤原賢二, 飯田元. Linux ディストリビューションにおけるパッチの適用過程の復元に向けて. 電子情報通信学会技術報告, volume 115, number 20, pages 59–62, 2015 年 5 月.
3. 齋藤雄輔, 藤原賢二, 井垣宏, 吉田則裕, 飯田元. Pull Request 駆動型の開発を支援するツールの検討. 電子情報通信学会技術報告, volume 114, number 416, pages 103–108, 2015 年 1 月.
4. 中山直輝, 吉田則裕, 藤原賢二, 飯田元. 開発履歴分析を用いたコードクローン内外における欠陥発生率の調査. 情報処理学会研究報告, volume 2014-SE-186, number 2, pages 1–8, 2014 年 11 月.
5. 槇原絵里奈, 藤原賢二, Putchong Uthayopas, Chantana Chantrapornchai, Jittat Fakcharoenphol, 井垣宏, 吉田則裕, 飯田元. 日本とタイにおけるプログラミング初学者のプログラミング行動の比較. 電子情報通信学会技術研究報告, volume 114, number 260, pages 47–52, 2014 年 10 月.
6. 浦田大地, 吉田則裕, 藤原賢二, 飯田元. ファイル編集履歴に基づいてデザインパターン適用事例を分析する手法の検討. 電子情報通信学会技術研究報告, volume 113, number 277, pages 13–18, 2013 年 11 月.
7. 藤原雄介, 藤原賢二, 吉田則裕, 飯田元. 版管理システムを用いた開発プロセスに適したコードクローン修正支援ツールの検討. 情報処理学会研究報告, volume 2013-SE-182, number 30, pages 1–6, 2013 年 10 月.
8. 山田悠太, 吉田則裕, 藤原賢二, 飯田元. トピック抽出を用いたソフトウェア開発履歴の可視化ツール. 日本ソフトウェア科学会第 30 回大会講演論文集, 2013 年 9 月.
9. 山田悠太, 藤原賢二, 吉田則裕, 飯田元. トピック抽出に基づく開発者の活動に着目したリポジトリ可視化手法. 情報処理学会研究報告 ソフトウェア工学研究会報告, volume 2012-SE-178, number 16, 2012 年 11 月.

10. 濵﨑一樹, 藤原賢二, 吉田則裕, Raula Gaikovina Kula, 伏田享平, 飯田元. Android Open Source Project を対象としたパッチレビュー活動の調査. 情報処理学会研究報告 ソフトウェア工学研究会報告, volume 2012-SE-176, number 12, 2012 年 5 月.
11. 伏田享平, 玉田春昭, 井垣宏, 藤原賢二, 吉田則裕. プログラミング演習における初学者を対象としたコーディング傾向の分析. 電子情報通信学会技術報告, volume 111, number 481, pages 67–72, 2012 年 3 月.

第 1 章

序論

1.1. 研究の背景・目的

近年，情報化社会の成長とともに，ソフトウェアは社会の基盤を支える重要な構成要素となっている．ソフトウェアの信頼性や生産性が社会に与える影響が大きく，高信頼なソフトウェアを効率良く開発することが必須の課題となっている．一般に，設計品質の高いソフトウェアにおける開発ではバグや欠陥などの混入が少ないと言われている．また，設計品質の高いソフトウェアは，機能追加やバグ修正の際に容易に変更が適用できるため，生産性を向上させるとも言われている．ソフトウェア開発の初期段階における設計は変更されやすく，一般的なソフトウェア開発においては機能追加やバグ修正を実施しながら設計の改善を行っていく．

ソフトウェアの設計を改善するための技術として，リファクタリングが注目されている．リファクタリングとはソフトウェアの外部的な振る舞いを変更することなく内部の構造を改善することをいう [1]．Fowler が典型的なリファクタリングパターンをまとめている [2, 3]．これらのリファクタリングは対象となるソフトウェアがオブジェクト指向で設計，実装されていることを前提としている．広く知られているリファクタリングの一例としては「メソッドの引き上げ（Pull up Method）」，「メソッドの抽出（Extract Method）」が挙げられる．図 1.1 に示す「メソッドの引き上げ」リファクタリングでは，共通の基底クラス S を持つクラス A，B が持つメソッド a，b のコードが重複している場合に，基底クラスのメソッド s としてメソッドを集約する．このリファクタリングにより，修正前に重複していたコードに対応する

コードの修正コストの削減が期待される．

一般に，ソフトウェアは開発が進むと，設計品質の低い箇所がソースコード中に増加していく．このような箇所は「コードの不吉な匂い」と呼ばれており，リファクタリングは不吉な匂い除去することを目的として実施される． リファクタリングを実施し，設計品質の低いコードを除去することで，開発時における欠陥の混入を予防できるといわれている．しかし，欠陥の混入を予防するためにどの程度のリファクタリングをどのような頻度で行えば良いのかは明らかになっておらず，次のような課題が存在する．

（課題）リファクタリングがソフトウェア品質に及ぼす影響の実証的評価に関する課題

先に述べたように，設計品質の低いコードを除去することで，開発時における欠陥の混入を予防できるといわれている．しかし，欠陥の混入を予防するためにどの程度のリファクタリングをどのような頻度で行えば良いのかは明らかになっていない．そのため，ソフトウェア開発において，リファクタリングの実施を効果的に行うことは難しい．開発者がより効果的にリファクタリングを実施するためには，欠陥の予防に有効なリファクタリングの種類や実施頻度を明らかにすることが望ましい．

本論文では，この課題に対して以下の二つの観点からアプローチする．

（アプローチ A）リファクタリングの影響の実証的評価手法の提案

これまで，ソースコードの複雑度や CK メトリクスなど，ソフトウェアのプロダクトに対してメトリクスを定義し，ソフトウェアの品質との関係を実証的に評価する

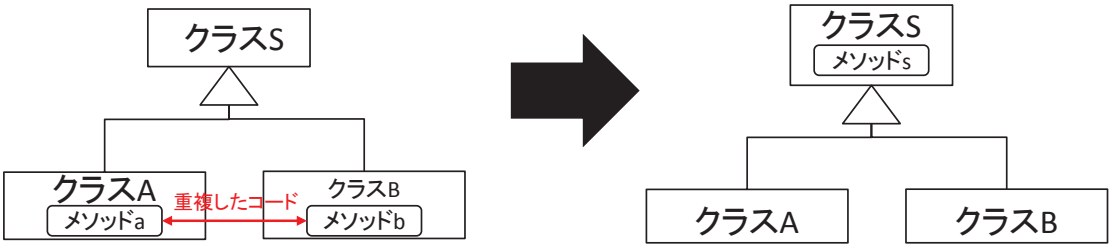

図 1.1　メソッド引き上げリファクタリングの例

手法が多く提案されている．しかし，リファクタリングとソフトウェアの品質の関係を明らかにしようとした研究は少ない．そこで，リファクタリングがソフトウェア品質に与える影響を実証的に評価する手法を提案する．

（アプローチ B）リファクタリングの影響の実証的評価に必要な基盤技術の開発

従来までのメトリクスと同じように，リファクタリングの実施に関する定量的な情報を収集する必要がある．これまで，リファクタリングの実施履歴を収集するための手法は様々なものが提案されているが，実証的評価を実施するのに十分であるとは言えないのが現状である．特に，近年重要視されている細粒度なプロセス分析を実施するために十分なパフォーマンスを備えた手法が存在しない．そこで，リファクタリングの実施履歴を高速に復元する手法を提案する．

結果として，アプローチ A に関して，リファクタリングと欠陥混入の関係に着目し，これらの関係を調査するための分析手法を提案した．提案手法では，開発履歴からリファクタリングの実施頻度，欠陥の混入頻度および欠陥の修正頻度を計測する．提案手法をオープンソースソフトウェアに適用した結果，提案手法を用いてリファクタリングと欠陥の関係を定量的に評価することが可能であることを確認した．第二に，アプローチ B に関して，開発履歴からリファクタリングの実施履歴を高速に復元するための手法を提案し評価した．従来の手法は，任意の 2 バージョン間から実施されたリファクタリングを検出することを目的としていた．そのため，開発履歴中の隣接する全てのバージョン間からリファクタリングを検出するための工夫がされていなかった．提案手法は，リファクタリングの検出に必要な構文情報の解析を差分的に実施することで計算時間の短縮を実現した．

1.2. 本論文の構成

本論文の構成は以下の通りである．第 2 章では，本論文における主要技術である，リファクタリングの定義について述べる．また，第 3 章，第 4 章において共通に取り扱う技術であるリファクタリング検出についてその概要を述べ，以降の章における共通の用語について整理する．第 3 章では，リファクタリングがソフトウェアの欠陥混入に与える影響を調査するための分析手法を提案し，ケーススタディにより

手法の有効性を確認する．第 4 章では，ソフトウェアの開発履歴全体から高速にリファクタリングの実施履歴を復元する手法を提案し，既存手法との結果について述べる．最後に，第 5 章で全体のまとめを述べ，本論文を結ぶ．

第 2 章

関連研究・用語

本論文では，リファクタリングがソフトウェア品質に及ぼす影響をソフトウェア開発履歴を用いて実証的に評価する手法を提案する．本章では，本論文で扱うソフトウェア開発履歴の解説を行う．また，本論文における主要技術であるリファクタリングについて説明し，ソフトウェア開発履歴からリファクタリングの実施に関する情報を得るための技術（リファクタリング検出技術）について説明する．

2.1. 開発履歴を用いたソフトウェア開発

企業やオープンソースソフトウェア（OSS）におけるソフトウェア開発プロジェクトでは多数の開発者が協調して開発を行う．開発者間の協調作業において重要となるのは，情報の共有と変更履歴の管理であり，これらの作業を円滑に進めるための支援環境が多く提案・利用されている．特に，版管理システム，課題管理システムと呼ばれる支援環境は近年のソフトウェア開発プロジェクトで広く利用されている [4, 5].

図 2.1 に版管理システム，課題管理システムを利用したソフトウェア開発の概要を示す．版管理システムは，ソフトウェア開発における成果物（ソースコード，設定ファイルなど）の変更履歴を管理，共有するためのシステムである．開発者は，成果物の変更を適宜，版管理システムに登録し他の開発者は版管理システムを通じて最新版のソースコードを閲覧，取得する．また，開発において手戻りが発生し，ソースコードを以前のバージョンに戻す必要がある場合にも版管理システムは利用可能で

ある．通常，ソフトウェア開発における開発作業は，特定のバグに対する修正作業や，特定の機能追加などの単位で分割することができる．課題管理システムはこのような作業を**課題**として開発者間で共有し，管理することを支援する．課題管理システムを用いたソフトウェア開発では，システムの設計者が追加する機能に対応する課題を登録し，バグの発見者が発見したバグを課題として登録する．開発者らは，課題管理システムに登録されている課題から自身が担当する課題を選択し，課題を解決するための開発作業を実施する．開発作業が完了すると，システムを通じて課題が完了したことを他の開発者と共有する．

版管理システムと課題管理システムを利用した開発形態は，企業および OSS を問わず広く採用されている．近年では，これらのシステムに記録されたソフトウェアの開発履歴を分析することでソフトウェア開発に有用な知見を得るための研究が盛んに行われている [4,6,7]．このような研究はリポジトリマイニング [5,8–10] と呼ばれており，本論文の 3 章にて提案するリファクタリングの定量評価手法もリポジトリマイニング研究の一種である．また，4 章は既存の開発履歴からリファクタリングに関するリポジトリマイニングを実施するための基礎技術を提案している．

以降，本節では版管理システムと課題管理システムについて説明する．

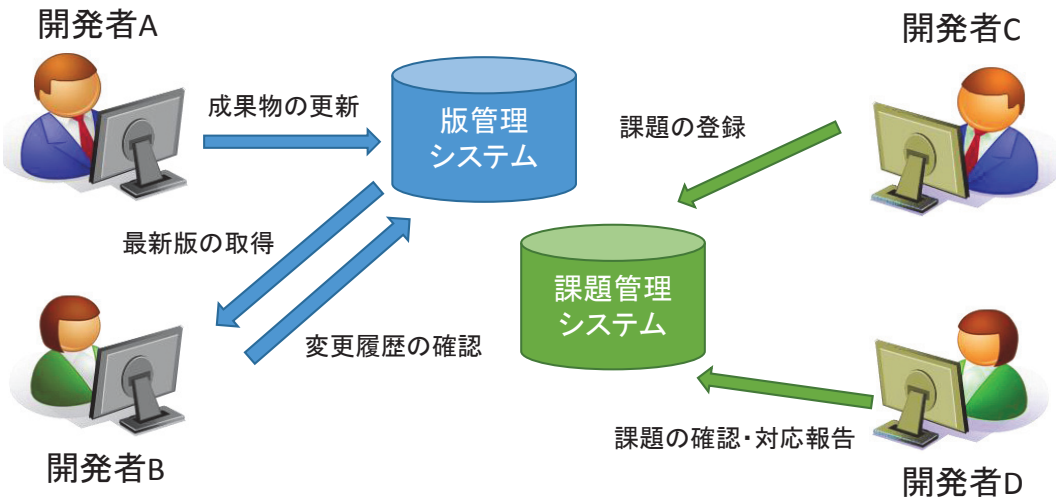

図 2.1　版管理システムと課題管理システムを用いたソフトウェア開発

2.1.1. 版管理システム

版管理システムは，ソフトウェアを構成するソースコードや設定ファイルなどの変更履歴を管理するためのシステムである．版管理システムを用いることで，過去のバージョンの実装（ソースコード）を確認することや，開発者間で最新版のソースコードを共有することが可能である．

版管理システムは，リポジトリと呼ばれるデータベースに変更履歴を格納する．このリポジトリを開発者間で共有する方式によって，版管理システムは大きく 2 つに分類される．**集中型版管理システム**と呼ばれる方式では，プロジェクトに対してリポジトリがただ一つ存在し，開発者らはこのリポジトリに逐次変更を反映する．一方，**分散型版管理システム**と呼ばれる方式では，リポジトリが複数存在することを想定して設計されている．分散型においては，各開発者が個別にプロジェクトのリポジトリを持つ．また，リリース用の履歴を格納するための中央プロジェクトをサーバ上に配置することが一般的である．普段の開発においては各自のリポジトリに変更を反映し，必要に応じて中央リポジトリに変更を統合するのが分散型版管理システムを用いたソフトウェア開発の特徴である．集中型の版管理システムとしては，CVS[*1]や Subversion[*2]，分散型の版管理システムとしては Git[*3]や Mercurial[*4]が代表的である．

本論文で扱う版管理システムの定義について説明する．版管理システムは，ファイルの変更履歴をコミットの集合としてリポジトリに記録する．コミットとは，以下に示す 3 つの要素を保持する履歴の単位である．

$$c = (S_c, A_c, M_c)$$

ここで，S_c はコミット c 時点におけるファイル群のスナップショットを表す．なお，ファイルはパスにより一意に識別可能である．コミットは，一つ以上の親（祖先）を持ち，A_c は親コミットの集合を表す．通常，ある時点でのコミットに対して変更を

[*1] http://www.nongnu.org/cvs/
[*2] http://subversion.apache.org/
[*3] http://git-scm.com/
[*4] http://mercurial.selenic.com/

適用し新しいコミットを作成すると，そのコミットの親は一つになる．しかし，ソフトウェア開発においては複数の時点のスナップショットを統合する形で変更を適用することがある．このような場合には A_c には複数のコミットが格納される．また，各コミットには作成日時や，コミットを作成した開発者，コミットに関するコメントなどのメタデータ（M_c）が記録される．以降，開発者がソフトウェアの変更をリポジトリに登録するためにコミットを作成することを「コミットする」と表記する．

開発者は，版管理システムを通じて任意の 2 コミットを入力として，スナップショット間の差分を得ることができる．変更前のコミット c_{before} と変更後のコミット c_{after} を入力として考える．このとき，版管理システムは c_{before} から c_{after} において新規追加，削除，変更されたファイルをそれぞれ求め開発者へ返す．なお，変更されたファイルに関しては追加と削除および変更された行に関する情報も併せて返す．

先に述べたように，各コミットは複数の親を持つことができる．この場合，リポジトリに格納される開発履歴は，コミットをノードとして見た無閉路有向グラフ（Directed Acyclic Graph）[11] と見なすことができる．本論文では，開発履歴から分岐の無い部分グラフを抽出したときに古いコミットから順に番号を与え，その番号を用いて説明を行うことがある．この時，n 番目のコミットを「第 n コミット」と呼ぶ．

先に紹介した代表的な版管理システムのいくつかは，必ずしも上記定義を満たさない．例えば，CVS は複数のファイルに対する変更を一つのコミットとしてまとめて記録することができない．このような版管理システムのリポジトリについては，必要に応じて事前処理を実施することで本論文で提案する手法が適用可能である [12]．

2.1.2. 課題管理システム

ソフトウェア開発における開発作業は，特定のバグに対する修正作業や，ある機能を追加するための作業といった単位で分割することができる．このような作業を**課題**という単位で同一プロジェクトに属する開発者間で共有し，管理するためのシステ

表 2.1 課題管理システムに記録される情報の例

項目	説明
課題 ID	課題を一意に識別するための ID
タイトル	課題の内容を簡潔に表す文字列
詳細説明	課題の詳細な説明文
起票日	課題票が作成された日時
ステータス	課題の状態を表す．未対応や進行中，完了，却下などがある．
担当者	対応する課題を誰が担当するかを表す．
重要度	課題の緊急性を一定の尺度で表したもの
カテゴリ	課題の分類．課題の検索性を向上させる目的で設定することが多い．

ムとして課題管理システムがある [13]．代表的なものとして，Bugzilla[*5]や Trac[*6]，Redmine[*7]がある．また，近年は版管理システムと統合した形で課題管理システムを提供するシステムやサービス（GitLab[*8]，GitHub[*9]，Bitbucket[*10]など）も増えてきている．機能追加や，バグ修正などの各課題は一意に識別するための ID（課題 ID）が割り当てられ，「課題票」という形で登録される．各課題票に記録される情報はシステムの実装により異なるが，多くの課題管理システムが共通して記録する情報を表 2.1 に示す．

課題管理システムのいくつかは版管理システムと連携して機能するものがある．例えば，ある欠陥の修正のための変更を版管理システムにコミットする際に，コミットログに課題 ID を記述することで，自動的に対応するバグ票の状態を変更することができる．3 章にて扱う SZZ アルゴリズム [6] では，コミットログに記述された課題 ID を手がかりに欠陥の修正を発見する．

[*5] http://www.bugzilla.org/
[*6] http://trac.edgewall.org/
[*7] http://www.redmine.org/
[*8] https://about.gitlab.com/
[*9] https://github.com/
[*10] https://bitbucket.org/

2.2. リファクタリング

リファクタリングとはソフトウェアの外部的な振る舞いを変更することなく内部の構造を改善することをいう [1]．リファクタリングの詳細は，一般にはリファクタリングカタログにまとめられている [2,3]．カタログには，特定の種類のリファクタリング操作に対して，その適用のための事前条件や事後条件，適用の具体的な手続きなどが，リファクタリング操作名（リファクタリング名）を添えた形でまとめられている．ソフトウェアパターンの形で記述されているものもあり，これらはリファクタリングパターンとも呼ばれる．例えば，あるクラスに所属しているメソッドを他のクラスに移動するリファクタリング操作は「メソッドの移動」と呼ばれ，メソッドの移動が適用できるための事前条件などを含めてパターンとしてカタログにまとめられている．リファクタリングカタログは，様々な書籍 [2,3,14] やウェブサイトにまとめられている．

一般に，ソフトウェアは開発が進むと，設計品質の低い箇所がソースコード中に増加していく．このような箇所は**コードの不吉な匂い**と呼ばれており，リファクタリングは不吉な匂い除去することを目的として実施される． リファクタリングを実施し，設計品質の低いコードを除去することで，開発時における欠陥の混入を予防できるといわれている．

2.2.1. リファクタリング検出

実務者・研究者ともに，開発プロジェクトにおいて過去に実施されたリファクタリングを知りたいという要求がある．その要求は，大きく 2 つに分類することができる．

- 実務者が，使用中のライブラリやフレームワーク，API に対して実施されたリファクタリングを理解することで，保守対象のソフトウェアにおいて追随するか否かや追随方法の判断に役立てたい [15,16]．
- 研究者が，開発プロジェクトにおけるリファクタリングの実施事例を収集することで，リファクタリングおよびその効果や支援手法に関する実証的研究

を行いたい [17, 18].

しかし，リファクタリングによる変更とそれ以外の変更が同時に行われた場合は，リファクタリングによる変更が含まれているか否かの判断に時間を要する [19–21]. 大規模開発プロジェクトの中には，数千回の変更履歴を持つものもあり，それらの変更全てに対して目視でリファクタリング実施の有無を検討することは現実的ではない．そこで，リファクタリングの実施を自動的に検出する手法が数多く提案されている．このような手法を，本論文ではリファクタリング検出手法と呼ぶ．本論文の 3 章にて提案する分析手法はリファクタリング検出手法を用いてリファクタリングの実施履歴を得ることを前提としている．また，4 章ではコミット単位でのリファクタリング検出を高速に実施する手法を提案し，評価している．

リファクタリング検出の定義

本論文におけるリファクタリング検出を次のとおりに定義する．あるソフトウェアの変更履歴を記録したリポジトリから抽出したコミットの組 $(c_a, c_b)(0 \leq a < b \leq n)$ が与えられたとき，その間に行われた変更の集合を $M = \{m_0, \ldots, m_{n-1}, m_n\}$ とする．このとき，リファクタリングカタログに記載されたリファクタリング操作が，変更の集合 M に含まれる空でない部分集合に存在するかどうかを推定することをリファクタリング検出と呼ぶ．一般には，リファクタリング検出を行うツールは，例えばコミットの組 (c_1, c_2) が入力されたとすると「コミットの組 (c_1, c_2) 間において，メソッドの引き上げおよびフィールドの移動が行われた」といった情報を出力する．ただし，具体的なリファクタリング名を出力せずにリファクタリングの存在のみを示唆するものもある [22].

リファクタリングは，コミット間で行われた変更の中で必ずしも単独で存在せず，機能追加やバグ修正などの変更と混在することがある [21, 23, 24]．Görg らは，複数のリファクタリングやリファクタリング以外の変更と混在した変更を impure リファクタリングと呼んでいる [25]．これに対し，単一のリファクタリングのみから構成される変更を pure リファクタリングと呼んでいる．pure リファクタリングと異なり，impure リファクタリングに関わるコミットの組では，必ずしも外部的振る舞いが保存されない．また，Murphy-Hill らは，欠陥修正や機能追加の最中にリファクタリングが実施されていることを指摘しており，こういったリファクタリン

グを floss リファクタリングと呼んでいる [21, 26]．リファクタリングとそれ以外の変更のタイミングを明確に分ける root-canal リファクタリングと比べ，floss リファクタリングではリファクタリングと他の変更が混在したコミットの組が生じやすい．Murphy-Hill らは，こういった floss リファクタリングが多数行われていることを報告している [21]．また Herzig らは，多様な種類の変更がこのように「もつれて」存在していることを報告している [27]．

このように，変更の混在は現実にはよく起こるため，リファクタリング検出手法は，コミットの組 (c_a, c_b) 間に行われた変更の集合に，リファクタリングのための変更だけではなく，バグ修正や機能追加のための変更が含まれていた場合であっても，リファクタリングが実施されたことを判定することが望まれる [21, 23]．本論文では，こういった混じりのあるリファクタリングについても検出・分析対象に含める．

リファクタリング検出手法は，多くの差分の解析手法と技術的な背景を共有している．例えば，Systematic change [28] の検出など，変更の集合に対して一般的に認知された名前を付加する研究 [28, 29] が行われている．また，あるコミットにおけるコード片と過去におけるコード片との対応をとる起源解析の多くは，プログラム要素の名前がどのように変更されたかを特定する技術を含んでいる [30, 31]．同様に，ソースコードやソフトウェアモデルの差分をわかりやすく取得する手法もあり，これらも一部のリファクタリング検出と同様の解析を行う [32]．

リファクタリング検出の分類

本節では前節で述べたリファクタリング検出を目的とした手法を大きく 4 つに分類し，本論文で主に取り扱う変更履歴の解析に基づく手法について述べる．

Murphy-Hill らは，リファクタリング検出手法を，2 つの観点を用いて 4 つに分類している [33]．彼らによる分類を表 2.2 に示す．一つの観点として，それぞれの手法がリファクタリングの実施を発見する際に用いる手がかりについて，明示的にリファクタリングの実施が記録されたものを利用するかどうかで分類している（explict または implicit）．別の観点として，得られるリファクタリングの実施が，主観的な判断によるか客観的な事実に基づくかどうかで分類している（context または fidelity）．以降，本節では表中 A1 から A4 の手法についてそれぞれ紹介する．

A1 に分類される手法では，版管理システムに記録されたコミットログを解析する

表 2.2 リファクタリング検出方法の分類

	context	fidelity
explicit	A1：コミットログ	A3：ツールログ
implicit	A2：開発者の観察	A4：変更履歴の解析

ことでリファクタリングの実施を発見する [34–36]．開発者がコミットログにリファクタリングの実施を記録していた場合，そのログを正確に抽出することでリファクタリングの実施を発見することができる．そこで，この手法は「refactor」や「extract」などのリファクタリングに関する単語をコミットログから抽出する．つまり，開発者によるリファクタリング実施の明示的な記録を利用するが，リファクタリングを記録するかどうかの判断や記録される情報は開発者の主観的な判断に基づくという特徴がある．これらの手法は，版管理システムを用いて開発されたソフトウェアの履歴であれば適用可能である．一方で，記録の精度が開発者の性質に依存することと，リファクタリングの具体的な適用箇所について情報を得られないという欠点がある．Murphy-Hill らは，開発者が実際にリファクタリングを実施する頻度に比べて，コミットログにそのことを記録する頻度は少ないと報告している [21]．そのため，開発者のリファクタリング傾向を調査することを目的とした研究にこの手法を利用する場合は，得られるリファクタリングの履歴に偏りがあることに留意する必要がある．

A2 に分類される手法では，研究者が開発者の活動を直接またはツールを使って間接的に観察することでリファクタリングの実施を発見する [37–39]．具体例として，ツールを用いて開発者が利用している開発環境の画面を定期的に記録し，その記録からリファクタリングを検出することが挙げられる．このような手法で利用する記録にはリファクタリングの実施が明示されていない．また，開発者がリファクタリングを実施したことを研究者が判断するため，得られる結果は個人の主観によるものである．これらの手法は適用範囲が限られるが，必要な情報を詳細に収集することが可能である．

A3 に分類される手法では，リファクタリング支援ツールのログを収集することでリファクタリングの実施を発見する [40–42]．ここでのリファクタリング支援ツールとは，統合開発環境が提供するリファクタリング機能に代表されるリファクタリングを自動的に適用するためのものである．開発者がそれらのツールを使用する目的

がリファクタリングの実施であることは自明である．これらの手法では，リファクタリングの実施時期と適用したリファクタリングの種類，適用箇所などを詳細に知ることができる．また，リファクタリング支援ツールは外部的な振る舞いを変更しないことを保証しているため，pure リファクタリングに関する情報を収集できることが特徴である．一方で，対象とするリファクタリング支援ツールを利用して適用したリファクタリングの実施しか得られないという制限がある．

A4 に分類される手法では，ソフトウェア開発における成果物の変更履歴を解析することで，リファクタリングの実施を発見する．これらの手法ではリファクタリングの実施を開発者や分析者の主観によって判断せず，ソースコードを代表とする成果物の変更を基に客観的な判断を行う．一方で，開発者がリファクタリングとして意図した変更を必ず抽出できるという保証はない．

3 章ではリファクタリングと欠陥混入に着目し，これらの関係を調査する分析手法の提案を行う．近年では，企業およびオープンソースソフトウェアのソフトウェア開発において，成果物の変更履歴を記録することは一般的となっている．そのため，これらの手法を適用できる範囲は広く，A1 から A3 に分類される手法よりも多くのリファクタリング実施事例を収集できることが期待される．そのため，A4 に分類される手法は A1 から A3 に分類される手法よりも適用範囲が広く，また，より多くのリファクタリング実施履歴を収集できることが期待される．そこで，提案手法は A4 に分類される手法のひとつである UMLDiff [43] を用いてリファクタリングの実施履歴を復元する．

ソースコードの変更履歴解析に基づくリファクタリング検出に関する研究

Weißgerber と Diehl はソースコードの構文および識別子情報を用いた手法を提案している [44]．彼らの手法は，リファクタリングと思われるコードの変更箇所を，構文情報と識別子情報を用いて抽出する．そして，それらの変更箇所を，コードクローン検出手法 [45] を用いてランク付けし，リファクタリングを検出する．この手法で検出可能なリファクタリングは 10 種類である．

Xing らは任意のバージョン間におけるソースコードの設計情報の差分を求める UMLDiff アルゴリズム [46, 47] を提案している．また，その差分情報を解析することでリファクタリングを検出する手法も提案している [43]．UMLDiff では比較対象

表 2.3 UMLDiff が構築する木構造の要素と親子関係

要素の種類	要素が持つ子要素
Virtual root	パッケージ, 配列型
Package	クラスおよびインタフェース
Class	フィールド，メソッド，コンストラクタ，初期化子，内部クラス，クラスが実装するインタフェース
Interface	フィールド，メソッド，内部クラス，インタフェースが実装するインタフェース
Field	初期化子
Block	ローカルクラス，匿名クラス

となる 2 バージョン間のソースコードを解析し，表 2.3 に示した要素と親子関係で構成される木構造を構築する．次に，構築した 2 つの木構造間で同一の要素を表すものを同定する．同一かどうかの判定は名称が類似しているかどうか（名称の一致）と，対象となる要素周辺の親子関係が類似しているかどうか（構造の一致）を計算することで行う．そして，2 バージョン間における要素の一致度を調べることで，要素の追加，削除，（木構造内での）移動，名称変更を求める．文献 [43] の手法ではこれらの変更情報と各要素の属性についての変更情報を解析することでリファクタリングを検出する．この手法は検出可能なリファクタリングの種類が 33 と他の手法と比べて多いのが特徴である（付録 A）．Xing らは UMLDiff アルゴリズムおよびそれを用いたリファクタリング検出手法を統合開発環境である Eclipse[*11]のプラグインとして実装したツール JDEvAn を公開[*12]している [48]．JDEvAn を用いることで，Java 言語で作成されたソースコード変更履歴からリファクタリングが検出可能である．

Hayashi らは探索手法を利用したリファクタリングの検出手法を提案している [23]．先に紹介したリファクタリング検出手法は，2 バージョン間のプログラム間の差分を解析し，どのようなリファクタリングが行われたのかを推定する．これに対し，Hayashi らの手法では，変更適用前のバージョンのソースコードにどのような

[*11] http://www.eclipse.org/

[*12] http://webdocs.cs.ualberta.ca/~stroulia/Zhenchang_Xing_Old_Home/jdevan.html

リファクタリングを適用すると変更後のバージョンになるかを探索することで実施されたリファクタリングを推定する．この手法は，他の手法と比較して複数種類のリファクタリングが実施された変更に対して高精度な推定が可能となっている．

Prete らは論理プログラミング [49] の概念をリファクタリングの検出に導入した手法を提案している [50]．彼らの手法ではプログラムのソースコードと，リファクタリングを論理プログラミングで利用可能な表現に変換する．まず，2 バージョンのソースコードからそれぞれ Logic Fact を抽出し，データベースに登録する．Logic Fact はクラスやメソッドなどのプログラムの構成要素に関する情報と，それらの関係を基に抽出される．次に，リファクタリングが行われた場合に満たす条件を論理プログラミングにおけるルールとして記述しておき，データベースからルールを満たす解を求めることでリファクタリングを検出する．彼らはこれらの手法を実装したツール Ref-Finder を公開[*13] している [51]．

[*13] https://webspace.utexas.edu/kp9746/www/reffinder/

第 3 章

リファクタリングが欠陥混入に与える影響の分析

3.1. はじめに

高品質なソフトウェアを開発するためには，ソフトウェアの開発中に開発プロセスを評価することが重要である．開発プロセスの評価は，開発中に随時実施することが可能で，早期に問題を検出することができるため近年注目されている．これまで，ソフトウェアの開発履歴を用いた細粒度な開発プロセスの定量評価手法がいくつか提案されている [52,53]

リファクタリングは，ソフトウェアの外部的な振る舞いを変えることなく内部の構造を改善するプロセスである [3]．リファクタリングは，ソフトウェアの開発がある程度進んだ段階で，欠陥の除去や機能追加を行うにあたり，目的の修正を実施しやすくするために実施される．Murphy-Hill らは実施されるリファクタリングの規模と目的の観点からリファクタリングを 2 種類に分類している [21,54]．一つは，リファクタリング以外の作業を実施する際に副次的に実施されるリファクタリング（floss refactoring）と呼ばれ，開発者らはこのようなリファクタリングを日常的に行っていると報告されている．もう一つは，設計の改善そのものを目的としたリファクタリング（root-canal refactoring）と呼ばれ，このリファクタリングは開発者らによって計画的に実施されることが多い．ソフトウェア開発において，より効果的なリファクタリングを実施するためには，これらのリファクタリングプロセスを定量的に評

価し，プロセス改善を実施する必要があると考える．

そこで本章では，欠陥を予防する観点からリファクタリングが欠陥に与える影響を定量的に評価するための手法を提案する．開発者がより効果的にリファクタリングを実施するためには，欠陥を予防するために最適なリファクタリングの種類や実施頻度を明らかにするのが望ましい．そのために，まずリファクタリングが欠陥混入に与える影響を分析するため，リファクタリングの頻度，欠陥が混入された頻度，欠陥が修正された頻度の 3 種類の尺度を定義した．次に，リファクタリングがソフトウェア開発に与える影響を明らかにするために，オープンソースソフトウェアを対象に定義した尺度を測定し，3 種類の値の傾向を分析した．その結果，リファクタリングが頻繁に行われた後は，欠陥が混入する頻度が低下していることを観測した．また，欠陥が混入された頻度，リファクタリングの頻度，欠陥が修正された頻度の順に，値が高くなる傾向にあることを確認した．

3.2. 関連研究

3.2.1. リファクタリングの効果や役割を調査することを目的とした研究

リファクタリングと欠陥の関係を分析している研究として，Ratzinger らはコミットログを用いてリファクタリングを検出し，欠陥との関係を分析している [35]．彼らは，ある期間におけるリファクタリングの回数が多かった場合，後続する期間において欠陥の発生量が減少すると報告している．そして，リファクタリングは欠陥の混入を予防する効果があると述べている．しかし，彼らは時間に対する詳細な変化については分析していない．本章で提案する分析手法は，リファクタリングや欠陥の混入，修正についてそれぞれ実施頻度を求めることで時間に対する変化を分析できるところが新規である．また，Murphy-Hill らはプログラマがどのようにしてリファクタリングを実施しているかを調査している [21, 54]．彼らは 4 種類の異なる方法で収集されたリファクタリングの実施履歴を用いて 9 つの仮説を検証した．その結果として，彼らは版管理システムのログメッセージはソフトウェア開発中に実施されたリファクタリングを検出するには信頼性が低いと結論づけている．そのため，提案手法ではより信頼性の高い変更履歴を用いたリファクタリング検出手法を用いる．

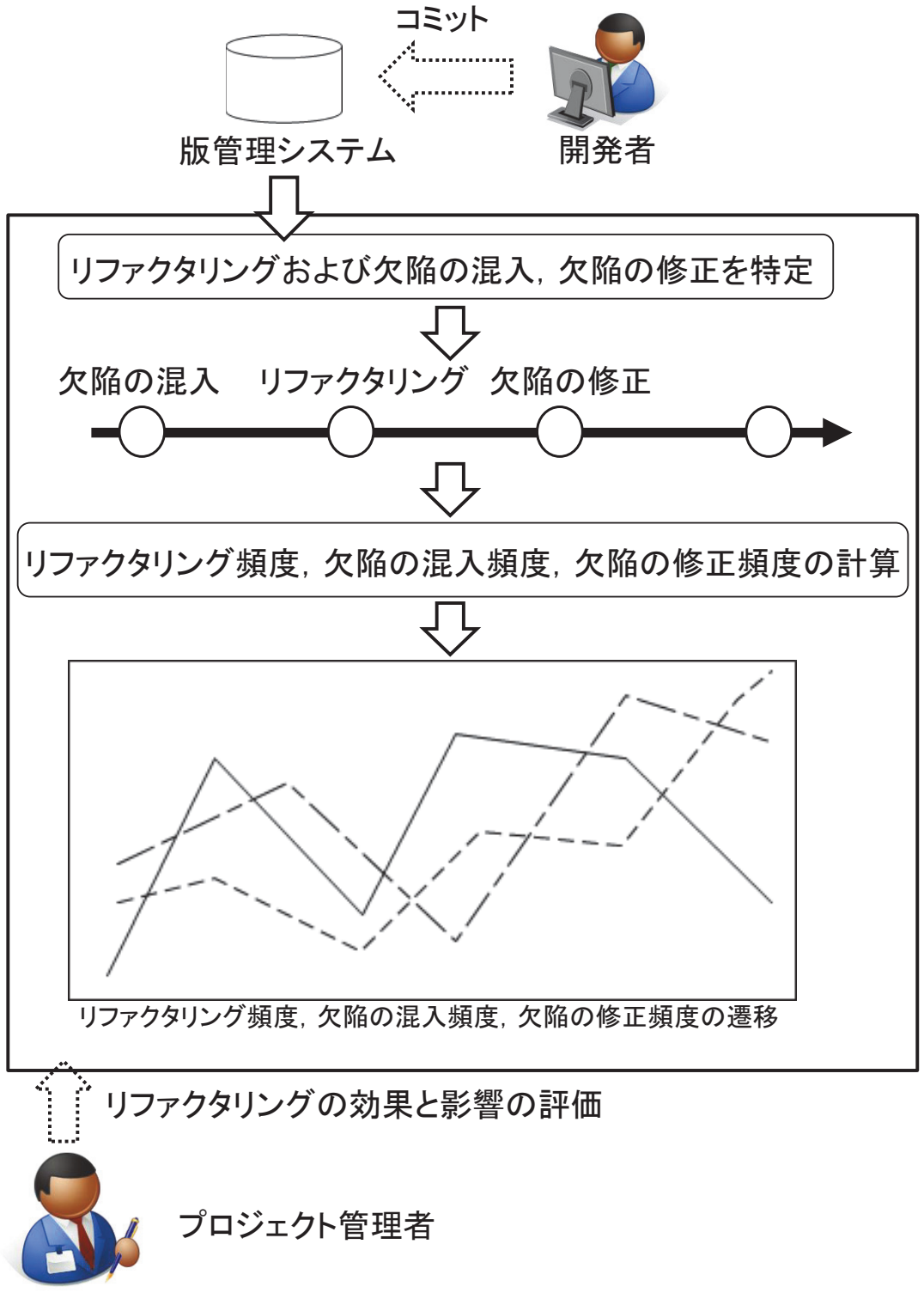
コミット
版管理システム
開発者
リファクタリングおよび欠陥の混入，欠陥の修正を特定
欠陥の混入
リファクタリング
欠陥の修正
リファクタリング頻度，欠陥の混入頻度，欠陥の修正頻度の計算
リファクタリング頻度，欠陥の混入頻度，欠陥の修正頻度の遷移
リファクタリングの効果と影響の評価
プロジェクト管理者

図 3.1　分析手法の概要

Kim らは 3 つのオープンソースプロジェクトを対象に，ソフトウェア開発においてリファクタリングがどのような役割を果たしているのかを調査した [18]．彼女らは，開発履歴からリファクタリングの履歴を復元し，リファクタリングの役割を次の 4 つの観点から評価している．

- リファクタリングの後にバグ修正が行われているか
- リファクタリングは開発者の生産性を向上するか
- リファクタリングはバグ修正を促すか
- ソフトウェアのメジャーリリース前にどの程度リファクタリングが行われているか

彼女らは分析の結果，Eclipse JDT プロジェクトにおいてリファクタリングが行われた 5 リビジョン以内に 26.1% の確率でバグ修正が行われていると報告している．また，リファクタリングによりバグ修正にかかる時間が減少するとも述べている．そして，リファクタリングと共にバグ修正が行われることが多いことと，メジャーリリース直前に多くのリファクタリングが行われていたことを報告している．彼女らは，バグ修正の観点からリファクタリングがソフトウェア開発の品質にどのような影響を与えるかを評価している．本章で提案する手法では，欠陥の混入という観点からリファクタリングが与える影響を分析する手法を提案し，分析を行っている点が新規である．

3.3. 分析手法

3.3.1. 手法の概要

本章では，プロジェクト管理者や開発者（分析者）がリファクタリングプロセスの改善を実施可能にすることを目的として，リファクタリングと欠陥の関係を定量的に評価する手法を提案する．提案手法の概要を図 3.1 に示す．提案手法はリファクタリングの実施時期，欠陥の混入時期，欠陥の修正時期をそれぞれソフトウェアの開発履歴から抽出する．まず，版管理システムに記録されたソースコードの変更履歴に基づいてリファクタリングの実施を特定する．次に，課題管理システムに記録された情報と版管理システムに記録されたコミットの対応付けを行い，欠陥の修正時

期を特定する．そして，ソースコードの変更履歴を利用して，欠陥の混入時期を求める．最後に，これらの時期からリファクタリングの実施頻度，欠陥の混入頻度，欠陥の修正頻度をそれぞれ計算し，これらの遷移を分析者に提示する．

3.3.2. メトリクス

提案手法では，リファクタリングと欠陥の関係を定量的に分析するために，**リファクタリングの実施頻度**，**欠陥の混入頻度**，**欠陥の修正頻度**の 3 つのメトリクスをソフトウェア開発履歴から計測する．これらのメトリクスの計測には，版管理システムおよび課題管理システムに記録されている情報を利用する．提案手法では，版管理システムに記録されているソースコードの変更履歴が，分岐の無いコミットの集合で構成されていることを前提としている．この場合，版管理システムに記録された全コミット C 中に n 個のコミットが含まれるとし，最も古いコミットから順に番号を付与すると $C = [c_1, c_2, \ldots, c_n]$ と表せる．ここで，隣接する 2 コミット間で行われた変更を返す関数 op_i を考える．第 $i-1$ コミットのファイル f_{i-1} から第 i コミットの同一ファイル f_i を作成するために f_i に適用される変更（ソースコードの追加や削除，修正）を Δ_i と表す．このとき，コミット c_{i-1} から次のコミット c_i への変更の集合 op_i を次のように定義する．

$$op_i = \{\Delta_i, \Delta_{i+1}, \ldots\}$$

op_i は，c_{i-1} と c_i 間で追加と削除，修正のいずれかが実施されたファイルの集合を返す．次に，2 コミット間における差分に着目して，3 つのメトリクス対して定義を与える．

まず，op_i においてリファクタリングが実施されたかを返す $r(op_i)$ を定義する．$r(op_i)$ は，op_i においてリファクタリングが実施された場合は 1，そうでなければ 0 を返す．$r(op_i)$ を用いて，第 j コミットから第 k コミットにおけるリファクタリング頻度 $f_r(j,k)$ を次のように定義する．

$$r(op_i) = \begin{cases} 1 & (\text{if } op_i \text{ contains a refactoring}) \\ 0 & (\text{otherwise}) \end{cases}$$

$$f_r(j,k) = \frac{\sum_{i=j}^{k-1} r(op_i)}{k-j} \qquad (j < k, \quad v_j, v_k \in C)$$

同様に，変更 op_i において欠陥が混入されたかを返す $d(op_i)$ と欠陥の混入頻度 $f_d(j,k)$，変更 op_i において欠陥が修正されたかを返す $f(op_i)$ と欠陥の修正頻度 $f_f(j,k)$ を次のように定義する．

$$d(op_i) = \begin{cases} 1(\text{if defects are introduced at } c_{i+1}) \\ 0(\text{otherwise}) \end{cases}$$

$$f_d(j,k) = \frac{\sum_{i=j}^{k-1} d(op_i)}{k-j} \quad (j < k, \quad c_j, c_k \in C)$$

$$f(op_i) = \begin{cases} 1(\text{if defects are fixed at } c_{i+1}) \\ 0(\text{otherwise}) \end{cases}$$

$$f_f(j,k) = \frac{\sum_{i=j}^{k-1} f(op_i)}{k-j} \quad (j < k, \quad c_j, c_k \in C)$$

提案手法では，$f_r(j,k)$，$f_d(j,k)$，$f_f(j,k)$ の 3 種類のメトリクスを用いて，リファクタリングの実施と欠陥混入および欠陥修正の関係を分析する．

3.3.3. リファクタリングの実施時期の特定

前節で定義したリファクタリングの実施頻度を計測するためには，開発履歴中のどのコミットにおいてリファクタリングが実施されたのかを知る必要がある．そのために，2 章で紹介したリファクタリング検出技術を用いる．本章では，リファクタリング検出技術の一つである，Xing らが提案している UMLDiff を用いる [46].

UMLDiff アルゴリズムは任意の 2 コミットを入力として受け取り，コミット間の設計の差分を出力する．詳細な手順としては，まず，クラスやメソッドなどのソースコード中の構文要素を抽出する．次に，コミット間における同一の構文要素の対応付を行う．この際，UMLDiff は識別子名に基づく類似度と，構文要素が持つ構造の類似度を計算して同一の要素を求める．最終的に，リビジョン間でどのようなクラスやメソッドが追加または削除されたのかを求め出力する．Xing らは UMLDiff の適用結果をデータベースに格納し，そこからリファクタリング検出を行うためのクエリを提案している [43].

本章で提案する手法では，以下の手順でリファクタリングの実施時期を特定する．

手順 1-1 全てのリビジョンを n ずつ k 個の組に分ける．

手順 1-2 手順 1-1 で作成した各組に対して，組の始点となるリビジョンと終点となるリビジョンを UMLDiff の入力としてリファクタリング検出を実施する．

手順 1-3 手順 1-2 で検出したリファクタリングに対して，正確な実施時期を求める．

3.3.4. 欠陥の混入時期・修正時期の特定

提案手法では欠陥の混入頻度および欠陥の修正頻度を計測する．提案手法では，Śliwerski らが提案している SZZ アルゴリズムを用いてこれらの時期を特定する [6]．SZZ アルゴリズムは，以下の手順で欠陥の修正時期および混入時期を特定する．

手順 2-1（修正時期の特定） 修正の完了している欠陥に関する課題票について，版管理システムのコミットログと各課題票の情報を基にコミットと課題票の対応付けを行う．

手順 2-2（混入時期の特定） 手順 2-1 で得た，修正を目的としたコミットの集合に対して，修正対象となったソースコードがいつ作成されたのかを版管理システムを用いて特定する．

以下に手順 2-1，手順 2-2 の詳細を示す．

手順 2-1-1 コミットログにバグ ID の候補となるもの（数字）と修正を示すキーワード（例えば “fixes” や “closed” など）が含まれているかを確認する．含まれていない場合には，以降の手順はスキップして次のコミットログを分析する．

手順 2-1-2 先の手順のコミットログに含まれていた番号がバグ ID となっている欠陥が，バグ管理システム上に登録されているかを確認する．次に，その欠陥が既に修正済みかを確認する．対象となる欠陥が実在し，修正も完了している場合，前の手順で得られたコミットログに対応するコミットにおいて，その欠陥が修正されたとする．

手順 2-2-1 手順 2-1-2 で得られるコミットで修正された箇所を，版管理システム上

表 3.1　Columba プロジェクトの概要

種類	開発期間	総コミット数	最終 LOC *2
メールクライアント	2006/7/9 – 2011/7/11	458	192,941

に記録された情報をもとに特定する．

手順 2-2-2　手順 2-1-2 で確認した欠陥が報告された時点よりも前のコミットを版管理システム上で走査し，前の手順で特定した箇所が加えられた（すなわち，欠陥が混入した）リビジョンを特定する．

3.4. 適用実験

オープンソースソフトウェアである Columba[*1]を対象に，前節で述べた手法の適用実験を行った．Columba に関する情報を表 3.1 に示す．Columba では，版管理システムとして Subversion，課題管理システムとして SourceForge.net が標準で提供する課題管理機能を利用している．

3.4.1. 実験手順

実験では，Columba の開発履歴を対象に以下の手順で 3.3.2 章で示した 3 種類のメトリクスを計測し，リファクタリングと欠陥の混入および修正に関する傾向について調査した．

手順 1 UMLDiff を用いてリファクタリングが実施された箇所および時期を特定した．

手順 2 SZZ アルゴリズムを用いて欠陥が修正，及び混入された箇所を特定した．

手順 3 1 および 2 で特定した情報をもとに，25 リビジョン間隔で 3.3.2 章で定義した尺度を算出した．

以下に手順 1 および手順 2 について詳細を述べる．

[*1] http://sourceforge.net/projects/columba/

手順 1: リファクタリング実施時期の特定

3.3.3 項に示した手順に従って，Columba の版管理システムに記録されたソースコードの変更履歴からリファクタリング検出を実施した後，リファクタリングの実施時期を特定した．リファクタリングの検出精度を向上させるために，次の二つのアプローチによりリファクタリングを検出した．

アプローチ A 全リビジョンを分割せずに，全リビジョンにおける最初と最後のリビジョンを UMLDiff の入力する．すなわち，第 1 リビジョンと第 458 リビジョンを入力として，リファクタリング検出を実施する．

アプローチ B $c_1, c_6, c_{11}, \ldots$ と 5 コミット間隔でコミットを抽出する．そして，抽出したコミットについて $(c_1, c_6), (c_6, c_{11}), \ldots$ と 2 コミットずつ UMLDiff の入力としてリファクタリング検出を実施する．

両アプローチにより得られた全リファクタリングに対し，各リファクタリングがどの時点で行われたかを，Subversion の履歴および差分閲覧機能を利用して手作業で確認した．この際，UMLDiff が誤検出したリファクタリングをあわせて除去した．具体例として，メソッドの抽出リファクタリングを行ったと検出されたが，メソッドの名前が似ているのみで，メソッドの定義自体は全く異なるものであった箇所などは誤検出とした．

手順 2: 欠陥修正および混入実施時期の特定

欠陥修正および混入実施時期の特定に関しては，SZZ を実装したツールを利用して欠陥の修正，および混入が発生したコミットを特定した．Columba の開発プロジェクトではコミットログに対するルールが厳格に定められており，バグ修正に対応する変更には **[bug]** や **[fix]** という文字列がタグとして記述されている．そこで本実験では，3.3.4 での手順 2-1 として，上記の文字列が含まれているコミットログのみを分析対象とした．このコミットログに対して，バグ ID が実際にバグ管理システム上に登録されており欠陥修正が対応しているかを確認した．そして確認が取れた場合，このコミットログに対応するコミットが行われた時点を欠陥修正がされた時期とした．さらに，この情報をもとに欠陥が混入した時期についても検出した．

3.4.2. 実験結果

UMLDiff による誤検出を除いた結果を表 3.2 に示す．アプローチ A，アプローチ B 列に記載されている個数が，3.4.1 章で説明した各アプローチにより検出することができたリファクタリングの件数を表す．また，両アプローチにより検出されたリファクタリングについて和集合を作成し，その要素数を合計列に示している．結果として，14 種類のリファクタリングについて計 116 個のリファクタリングを抽出することができた．

欠陥の修正および混入に関しては，修正を目的としたコミットを 322 件抽出した．次に，これらのコミットに関する情報を基に，欠陥を混入したコミットを 243 件抽出した．

以上の情報をもとに，リファクタリング実施頻度，欠陥の混入頻度および欠陥の修正頻度をプロットしたグラフを図 3.2 に示す．図 3.2 において横軸はリビジョン番号，縦軸は提案手法で定義した 3 種類のメトリクス値を表している．図中の頻度はそれぞれ 25 リビジョンを 1 間隔として計算している．例えば．図 3.2 において第 250 リビジョンから第 275 リビジョンの区間における欠陥の修正頻度は 0.2 である．

3.5. 考察

3.5.1. 実験結果からの考察

図 3.2 からは次に示す二つの傾向が見られる．

(傾向 A) リファクタリングが最も実施された第 150 リビジョンから第 175 リビジョンの区間以降は，欠陥の混入頻度が減少傾向にある．

(傾向 B) プロジェクト全体を通して，リファクタリングの実施頻度が高くなった後に，欠陥の修正頻度が高くなる傾向にある．

傾向 A に関して，第 0 リビジョンから第 175 リビジョンの区間における欠陥の混入頻度の平均が 0.23 であるのに対して，第 175 リビジョンから 第 458 リビジョンの区間においては 0.09 となっている．また，リファクタリング頻度が最も高い第

表 3.2 検出されたリファクタリング

リファクタリング	検出数		
	アプローチ A	アプローチ B	合計
Convert top level to inner	1	0	1
Die-hard/legacy classes	1	1	1
Downcast type parameter	3	0	3
Encapsulate field (get)	1	0	1
Extract class	2	1	2
Extract method	7	2	7
Extract subsystem/package	3	2	3
Extract super interface	0	6	6
Generalize type (method)	10	9	10
Generalize type (field)	2	2	2
Generalize type (parameter)	42	40	42
Information hiding	6	0	6
Inline subsystem/package	1	2	2
Move method/field/behavior	0	12	12
Move subsystem/package/class	0	12	12
Pull-up method/field/behaivior	5	4	5
Push-down method/field/behaivior	1	0	1
合計	85	93	116

150 リビジョンから第 175 リビジョンの区間におけるコミットログを確認すると，第 151 リビジョンから第 156 リビジョンにおいて大規模なリファクタリングを実施したと記録されていた．このリファクタリングにより，ソースコードの設計が改善されたため以降の欠陥混入が減少したのではないかと考える．しかし，初期バージョンの開発が完了し，プロジェクトが保守行程に移行したために，頻繁な機能追加が行われず，結果として欠陥の混入が減少したという見方が可能である．

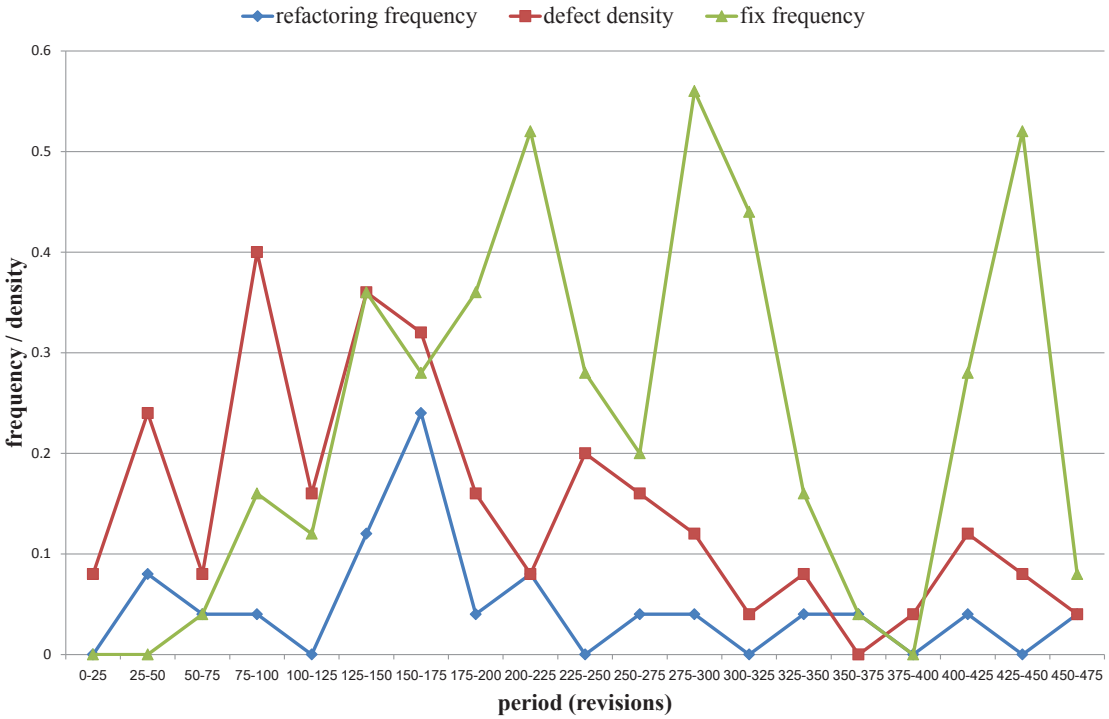

図 3.2　Columba におけるリファクタリング頻度と欠陥の混入・修正頻度

傾向 B に関しては，リファクタリングを実施したためソースコードの可読性が向上し，それに伴い欠陥の発見・修正が容易になったために欠陥の修正頻度も高くなったのではないかと考える．欠陥の修正を行うに当たり，開発者らがソースコードの変更を容易にするために，あらかじめリファクタリングを実施した後に欠陥の修正を行った可能性もある．一方で，リファクタリングを実施したが，失敗し，失敗箇所の修正を行ったため，結果としてリファクタリングの後に欠陥の修正が行われたという場合も考えられる．

3.5.2.　妥当性の検証

本章では提案した手法をオープンソースソフトウェアである Columba を対象に適用した．分析の結果得られた傾向は，あくまで単一プロジェクトから得られた知見である．そのため，別のオープンソースソフトウェアプロジェクトでは別の傾向が

得られる可能性がある．

提案手法で計測するリファクタリング頻度は，手法がリファクタリングの実施時期を特定する際のアプローチに強く依存する．実際，適用実験においてはリファクタリング検出を適用するコミットの粒度を変更したところ，アプローチごとに検出できるリファクタリングの種類と数に差があった．これは，リファクタリング検出の際に始点となるコミットと，終点となるコミットの間で作成されたクラスや，メソッドに関するリファクタリングを検出できないことに起因する．適用実験におけるアプローチ B では，3.3.3 項にて説明した手順 1-1 の n を $n = 5$ としてリファクタリングの検出を行った．n を 5 より小さくしてリファクタリング検出を行うことで，リファクタリングの実施履歴をより正確に復元できると考えられるが，UMLDiff アルゴリズムにより差分を計算するコミットの組み合わせが増えるため，計算時間が増加する．今回の実験では，現実的な時間でリファクタリング検出が終わるように $n = 5$ に設定した．n をより小さくした場合のリファクタリング検出や，n を複数種類設定した検出結果を組み合わせてリファクタリングの実施履歴を特定するためには，UMLDiff よりも高速なリファクタリング検出を用いる必要があると考える．

適用実験の結果は，検出可能なリファクタリングの数は利用するリファクタリング検出手法の精度にも影響を受けていると考えられる．UMLDiff による設計情報の差分抽出については，適合率が 93.7%，再現率が 95.2%[*3] と共に高い値が報告されており，実験結果に与えている影響は小さいと考えられる [46]．UMLDiff アルゴリズムを用いたリファクタリング検出については文献 [43] においてオープンソースソフトウェアプロジェクトに適用した場合の検出数が報告されている．しかし，適合率，再現率の値については言及されていない．適合率に関しては，今回の実験においては誤検出を目視で除去したため影響は小さいと考えられる．一方で，UMLDiff によって抽出されたリファクタリングが網羅していない可能性がある．

対象プロジェクトにおける欠陥の修正，欠陥の混入が行われた時期を特定するにあたり利用した SZZ アルゴリズムが，一部の欠陥の修正，欠陥の混入を抽出できていない可能性がある．欠陥の修正に関しては，今回はコミットログを頼りに特定をしたため，開発者らの記録の厳密さにその再現性は左右されている．いくつかの研究においては欠陥の修正と混入をコミットログを基に特定する際の精度について報

[*3] 再現率は文献 [46] における Table5 中の $\#Correct$ と $\#Reported$ の比率として計算した．

告している [55, 56]．しかしながら，彼らはこれらの手法において全ての修正と混入を特定することはできないと述べている．また，欠陥の混入に関しては，SZZ アルゴリズムが欠陥修正の際に変更されたコードを版管理システムを用いて遡ることで混入時期を特定しているが，コードの削除のみが原因で，欠陥の混入が行われた場合は混入時期を特定することが出来ない．しかし，このような例は稀であると考えられるため，これにより今回得られた傾向が大きく変わるものではないと考える．

提案手法では 3 種類のメトリクスを計測するにあたり，各リビジョンにおいて，リファクタリング，欠陥の混入，欠陥の修正がそれぞれ行われたかどうかに着目している．このため，同一リビジョン内で複数種類のリファクタリングを実施した場合や，複数の欠陥を同時に修正した場合も，単一のリファクタリングや欠陥修正と同様に扱う．これは，提案手法で抽出可能なリファクタリングの粒度が大小様々であり，それら異なるリファクタリングの実施について，同様に 1 回のリファクタリングであると考えるのは不適切であると考えたため，リファクタリングを実施したかどうかのみに着目した．欠陥の混入・修正に関しても同様に，粒度の大小が存在するため欠陥の混入・修正が行われたかどうかのみに着目した．

3.6. 本章のまとめ

本章では，ソフトウェア開発におけるリファクタリングが欠陥混入に与える影響を分析するために，リファクタリングの実施頻度，欠陥の混入頻度，欠陥の修正頻度を定義した．また，それぞれの尺度をソフトウェア開発履歴から収集する手法を提案した．そして，オープンソースソフトウェアを対象にリファクタリングの実施頻度，欠陥の混入・修正頻度を測定し，それらの関係を調査した．Columba からは，リファクタリングが最も行われた期間以降，欠陥の混入頻度が減少傾向にあったことと，プロジェクト全体を通してリファクタリング頻度が高くなった後に，欠陥の修正頻度が高くなる傾向にあることがわかった．

今後の展望として，より多くのオープンソースソフトウェアに対して手法を適用するとともに，今回適用したプロジェクトよりも規模の大きなプロジェクトを対象に調査を行い，Columba に見られた傾向と比較することが考えられる．これにより，ソフトウェア開発において広く適用できる有用な知見を得ることができると期待さ

れる．今回の調査では，リファクタリングや欠陥の混入・修正がどのファイルに実施されたかを考慮せずに，その頻度に着目して分析を行った．リファクタリングが欠陥混入に与える影響をより詳細に調査するには，リファクタリングが実施されたファイルが，リファクタリングによりその後の欠陥混入が無くなったかどうかなど，ファイルに着目して分析を行う必要がある．この際には，より正確なリファクタリングの実施履歴の抽出が求められるため，リファクタリングの実施時期特定におけるリビジョンの分割数を増やし，UMLDiff が抽出するリファクタリングの再現性を高くする必要がある．

第 4 章

構文情報リポジトリを用いた細粒度リファクタリング検出

4.1. はじめに

近年，ソフトウェアの開発履歴からリファクタリングの実施履歴を復元するための技術（リファクタリング検出技術）が数多く研究されている．リファクタリング検出技術はソフトウェア開発者と研究者に次のような利益をもたらす．

利益 1 開発者がライブラリのリファクタリングを検出することで，バージョンアップに追従することができる．

利益 2 リファクタリングの適用事例を収集することで，研究者がリファクタリングの効果を実証的に分析・検証することができる．

利益 3 開発者がどのようにリファクタリングを実施しているのかを分析することで，有効なリファクタリング支援手法の検討に役立つ．

まず，**利益 1** については開発中のソフトウェアで利用している外部ライブラリの開発履歴からリファクタリング検出を行うことで，外部ライブラリのバージョンアップに追従する際の手間を軽減できる．このような目的でリファクタリングを検出している研究は多く，いくつかはライブラリ側で実施されたリファクタリングを基に修正すべき箇所を自動的に推薦する [16, 57]．

利益 2 に関する研究では，リファクタリングがソフトウェア開発に与える影響を実証的に調査するために，ソフトウェアの開発履歴に対してリファクタリング検出手法を適用する．Kim らは 3 つのオープンソースプロジェクトを対象に，リファクタリングが欠陥修正と開発者の生産性に対してどのような影響を与えているかを調査している [18]．また，3 章にて実施した分析や，Ratzinger らはそれぞれリファクタリングが欠陥の混入に与える影響について調査を行っている [35]．

利益 3 について，Choi らはコードクローンとリファクタリングの関係に着目し，開発者らがどのようなコードクローンに対してリファクタリングを実施しているのか分析している [58]．その結果として，メソッドオブジェクトによるメソッドの置き換えリファクタリングを支援するツールが必要であると報告している．

このように，リファクタリング検出技術は開発者や研究者に多くの利益をもたらす．2.2.1 項で述べたように，版管理システムに記録されたソースコードの編集履歴に基づいてリファクタリングを検出するための研究が盛んに行われている．従来のリファクタリング検出手法は，ソフトウェアのリリースバージョン間において実施されたリファクタリングを検出することを暗黙に想定している．リリース間での比較では，リファクタリングの正確な実施時期や，リファクタリングの適用者，リファクタリング適用時点のソースコードなどの情報が失われる．リファクタリングに適切な時期やその効果の分析を行う為には，より細かい時間的粒度での検出が必要である．そのためには，開発履歴へ記録されたコミット単位でのリファクタリング検出を実施することが望ましい．本論文では，開発履歴中の全ての隣接しているコミットの組からリファクタリング検出を行うことを，細粒度リファクタリング検出と呼ぶ．以降，本章では特に断りのない限り隣接する 2 つのリリースバージョン間とコミット間をそれぞれリリースバージョン間，コミット間と表記する．

先に述べたように従来のリファクタリング検出手法は，リリースバージョン間からの検出を目的としている．リリースバージョン間の変更は，差分の大きいコミット間の変更とみなすことができるため，従来手法を用いて細粒度リファクタリング検出を実施することは可能である．しかし，従来手法の評価は高々数十のバージョン間を対象とした検出の精度や速度を評価対象としているため，検出時間を短縮するための工夫が十分になされていない．そのため，数千や数万を超えるコミット間を対象とした細粒度リファクタリングを行うためには多大な時間を要するという問

題がある．ソフトウェアの開発履歴から，リファクタリングの実施に関する情報をコミットの粒度で取得するためには，従来手法よりも高速なリファクタリング検出手法が求められる．そこで本章では，細粒度リファクタリング検出を高速に実施可能なリファクタリング検出手法を提案する．

従来手法の計算時間は，コミット（バージョン）毎に構文解析を行う処理と，あるコミットにおける構文要素が次のコミットにおけるどの構文要素に対応するかを求める処理が大きな割合を占めている．計算時間の観点で，これらの処理について従来手法が抱える問題点は以下の通りである．

問題点 1 コミット間で変更されなかったファイルについても繰り返し構文解析を行っている．

問題点 2 コミット間で変更されなかった構文要素についても繰り返し対応付けを行っている．

問題点 3 本来不要な部分の構文情報を取得し，コミット間での対応付けを行っている．

問題点 3 について，コミット間で構文要素の対応付けを行う処理は，Origin Analysis と呼ばれている [59]．従来手法は，この技術を発展させる形で提案されている．そのため，構文解析においてあらかじめ変数の read/write，メソッドの callee/caller などの情報を取得している．しかし，あるコミットにおける全メソッドの情報が必要な訳ではなく，特定の条件を満たすメソッド内の構文情報があればリファクタリング検出は可能である．また，これらの情報は全てのリファクタリングパターンの検出に必要な訳ではない．

これらの問題点を解決するために，提案手法では構文情報リポジトリ Historage [60] を用いてリファクタリング検出を行う．構文情報リポジトリを用いることでソースコードの構文解析結果をキャッシュすることができ，構文解析にかかる時間を削減できる（問題点 1）．また，構文情報リポジトリは，メソッド本体やクラス階層などの情報からハッシュ値を計算することで，コミット間で差分のある箇所を効率的に知ることができる．そのため，構文情報の対応関係を求める際に重複した計算を避けることができる（問題点 2）．また，構文情報リポジトリに記録されない情報は必要になった段階で取得する（問題点 3）．

リファクタリングには，コードの抽出を行うものと，クラス間でメソッドの移動を行うものが多い．本章では，それらのリファクタリングで代表的な「メソッド抽出」と「メソッドの引き上げ」の 2 種類について提案手法を実装した．そして，オープンソースソフトウェアである jEdit を対象に従来手法である Ref-Finder [50]，UMLDiff [43] との比較を行い，提案手法の有効性を評価した．評価の結果，提案手法は従来手法より高速かつ高精度にリファクタリングを検出できることを確認した．

以降，本章は 4.2 章で背景を，4.3 章で構文情報リポジトリの構築手順と，そのリポジトリを用いたリファクタリング検出について述べる．4.4 章と 4.5 章では従来手法との比較結果とその考察を述べ，4.6 章にて本章をまとめる．

4.2. 背景

本節では，本章におけるリファクタリング検出の定義を説明するとともに，既存のリファクタリング検出手法を紹介する．また，本章で検出の対象とするメソッド抽出及びメソッドの引き上げリファクタリングについて説明し，提案手法が利用する Historage について述べる．

4.2.1. リファクタリング検出

本章で扱うリファクタリング検出は，始点および終点となるコミットにおけるソースコードのスナップショット S_x，S_y を入力として与えると，始点 S_x から終点 S_y において行われた変更を解析し，その間に実施されたリファクタリングの集合を返す．

本章で扱うソースコードの変更解析に基づくリファクタリング検出は次の手順で実施される．

手順 1 ソースコードの構文解析
手順 2 構文要素の対応付け
手順 3 リファクタリングパターンにマッチングする変更をリファクタリングとして検出する

検出の手順としてはまず，S_x と S_y の全てのファイルに対して構文解析を行い，クラスやメソッドなどの構文要素を得る．次に，構文要素の対応付けを行い，S_x から

S_y への変更において構文要素に対して行われた操作（追加，削除，変更など）を求める．最後に，構文要素及びそれらの変更に対して各種リファクタリングパターンとのマッチングを行う．そして，リファクタリングの種類と，適用対象のメソッドやクラスなどの情報を出力する．

以上のような手順でリファクタリング検出を行う既存の手法として，Xing らは UMLDiff アルゴリズムを用いたもの [43] を，Prete らは Ref-Finder [50] を提案している．Xing らの UMLDiff アルゴリズムは構文要素の階層構造と，各要素の名称についてそれぞれ類似度を求め，それらを基に対応付けを行う [46, 47]．このアルゴリズムの適用結果をデータベースに格納し，そこからリファクタリングの検出を行うためのクエリが提案されている [43]．以降，本章では UMLDiff アルゴリズムを用いたリファクタリング検出を UMLDiff と表記する．Prete らの Ref-Finder は，手順 1，手順 2 により得られる情報を論理プログラミングの述語として表現する [50]．そして，手順 3 に対応するパターンを述語の組み合わせとして定義するのが特徴である．彼らは文献 [61] において Fowler のリファクタリングカタログに記載されているリファクタリング 72 件のうち 65 件のパターンを記述している．

4.2.2. メソッド抽出リファクタリング

メソッド抽出リファクタリングとは，あるメソッドから機能的に意味がある単位でコードを選択し，新たなメソッドとして抽出することを言う [3]．このリファクタリングが実施される主な目的として，重複したコードをメソッドとして抽出することでコードの保守性を向上させることが挙げられる．また，長すぎるメソッドを対象にコードの可読性を向上させることを目的とする場合もある．

提案手法では，以下に示す条件を満たすものをメソッド抽出リファクタリングとして定義する．

条件 A-1 抽出対象のメソッドと抽出したメソッドは同一クラス内に定義されている．

条件 A-2 メソッド抽出時に，抽出対象のメソッドに抽出したメソッドの呼び出しが追加されている．

4.2.3. メソッドの引き上げリファクタリング

メソッドの引き上げリファクタリングとは，継承関係にあるクラス間において，子クラスに存在するメソッドを親クラスへ移動することを言う．提案手法では Prete らによる定義 [61] を基に，以下に示す条件を満たすものをメソッドの引き上げリファクタリングと定義する．

条件 B-1 リファクタリング対象となるメソッドが，そのメソッドを保持するクラスの親クラスへ移動されている．

条件 B-2 リファクタリングの対象となるメソッドを保持するクラスは，変更前の時点で移動先のクラスを継承している．

条件 B-3 メソッドの移動後も移動元のクラスが存在している．

Fowler は複数の子クラスから単一の親クラスへメソッドを引き上げる場合をメソッドの引き上げリファクタリングとしている．しかし，本章では UMLDiff や Ref-Finder と同様に単一の子クラスからメソッドを引き上げる場合もメソッドの引き上げリファクタリングとして扱う（条件 B-1）．

4.2.4. Historage

Git や Subversion などに代表される従来の版管理システムは，ソースコードの変更をファイル単位で管理する．そのため，ファイルの追加，削除，変更に関する情報は容易に取得できるが，メソッドの追加や削除に関する情報を得るためには各コミットにおけるソースコードを解析し，その対応を求める必要があった [62]．Hata らはメソッド，コンストラクタ，フィールドの単位でソースコードの変更を管理するための Historage を提案している [60]．Hitorage は各コミットにおけるソースコードを構文解析し，得られた構文木を Git リポジトリ上のディレクトリ階層に対応付けて記録する．具体的には，クラスがメソッドを保持することをディレクトリ上の親子関係で表現し，メソッドの本体をファイルとして表現する．このように構文情報を版管理システムに記録することで，通常のファイル変更を追跡するのと同様の操作でメソッド本体に対する変更を追跡できる．本章で提案する手法は，リファクタリ

ング検出に必要な構文情報とその差分を求めるにあたり，Historage を構文情報リポジトリとして利用する．しかし Historage は，メソッドの引き上げリファクタリングの検出に必要なクラスの継承に関する情報を保存しない．提案手法では Historage の拡張を行うことで検出に必要な情報を取得できるようにする．

4.3. 構文情報リポジトリを用いた細粒度リファクタリング検出

本節では，版管理システムから高速に細粒度リファクタリング検出を行うための手法を提案する．提案手法は，版管理システムに記録された開発履歴（リポジトリ）を入力として，リポジトリ中の全コミット間からリファクタリング検出を実施する．

手順として，まず，分析対象のリポジトリから構文情報リポジトリを構築する．次に，構築した構文情報リポジトリを用いて各コミット間におけるリファクタリングの検出を行う．リファクタリングの検出にあたっては，構文情報リポジトリが提供するファイルの情報を 4.2.1 項にて説明した構文解析の結果として使用する（手順 1 に対応）．また，構文情報リポジトリが提供するファイルの変更情報を用いて構文要素の対応付けを行う（手順 2 に対応）．

以降，本節では構文情報リポジトリを構築する手順と，構文情報リポジトリが提供する情報からメソッド抽出及びメソッドの引き上げリファクタリングを検出する手順について詳しく説明する．

4.3.1. 構文情報リポジトリの構築

提案手法は，Historage を用いてメソッド抽出，メソッドの引き上げリファクタリングの検出に必要な構文情報及びその差分を取得する．それぞれのリファクタリングを検出するのに必要な情報を表 4.1 に示す．メソッド抽出リファクタリングの検出においては，メソッドの変更を行った箇所についてのみ知ることができれば十分である．そこで，提案手法ではメソッド呼び出しの追加を除いた情報を取得できるように構文情報リポジトリを構築し，メソッド呼び出しの追加については必要になった段階で取得する（次節で詳述）．

図 4.1 に変換前のディレクトリ構造を，図 4.2 に変換後のディレクトリ構造を示す．

表 4.1 リファクタリング検出に必要な構文情報

		メソッド抽出	メソッドの引き上げ
構文要素	パッケージ情報	✓	✓
	クラス	✓	✓
	メソッドのシグネチャ	✓	✓
	メソッドの本体	✓	✓
	クラスの継承		✓
差分情報	メソッド呼び出しの追加	✓	
	メソッド本体の変更	✓	
	メソッドの追加	✓	✓
	メソッドの削除		✓

ここで，変換前のディレクトリ構造に存在する `foo/bar/HistorageConverter.java` の構文情報は，変換後の `foo_bar_HistorageConverter.java` ディレクトリ以下に保存されている．この Java ファイルのパッケージ情報は，変換後のディレクトリ直下にある `package` ファイルに保存されている．また，図中の `[CN]` および `[MT]` ディレクトリには，それぞれクラス，メソッドに関する情報が格納されている．例えば，図 4.2 からは，`HistorageConverter` クラスには `convert(Parser)` メソッドが定義されていることが分かる．ここで，`convert(Parser)` ディレクトリ直下の `body` ファイルにはメソッドの本体が，`parameter` ファイルにはメソッドのパラメータ情報が記録されている．

メソッドの引き上げリファクタリングの検出には，あるクラスが継承しているクラスを知る必要がある．しかし，Hata らが提案している Historage ではクラスの継承関係を記録していない．そこで，親クラスの名前をクラスに対応するディレクトリの直下にテキストファイルとして記録し（図 4.2 における `extend`），メソッドの引き上げリファクタリングの検出に利用する．

構文情報リポジトリの構築手順としては，スナップショット S_0 から順番に各ソースコードファイルに対して構文解析を行い，ディレクトリ構造の変換を行い，コミットを実施していく．各コミットを処理する際には，変更されたソースファイルのみ

新規に構文解析を行い，変更されなかった分については直前のコミットで変換されたディレクトリ構造を再利用する．これは，構文木から変換したディレクトリ構造は，基になったソースファイルと一対一に対応するため，同一のソースファイルからは同一のディレクトリ構造が得られるからである．これにより，提案手法は既存の手法と比較して計算時間の大幅な短縮ができる．

4.3.2. メソッド抽出リファクタリングの検出

前節で説明した手順で構築した，構文情報リポジトリのコミット間の差分を分析することで，メソッド抽出リファクタリングの検出を行う．提案手法が検出するメソッド抽出リファクタリングは，メソッド抽出の対象となったメソッド（$targetMethod$）とリファクタリングにより新たに作成されたメソッド（$extractedMethod$），$targetMethod$ から抽出されたコードと $extractedMethod$ の類似度（4.3.4 項参照）に関する情報で構成される．

Algorithm 1. にメソッド抽出リファクタリングの候補を検出する手続きを示す．検出はメソッドの追加及び変更が実施されたクラス毎に実施される（6 行目）．まず，新規に作成されたメソッドを見つけ，$extractedMethod$ の候補とする（7 行目）．次に，行の追加および削除が行われたメソッドを見つけ，$targetMethod$ の候補とする（9 行目）．そして，$targetMethod$ に追加された行に $extractedMethod$ の呼び出しが存在すれば（12 行目），メソッド抽出リファクタリングの候補として，$targetMethod$ と $extractedMethod$ の類似度を計算する（13，14 行目）．なお，前

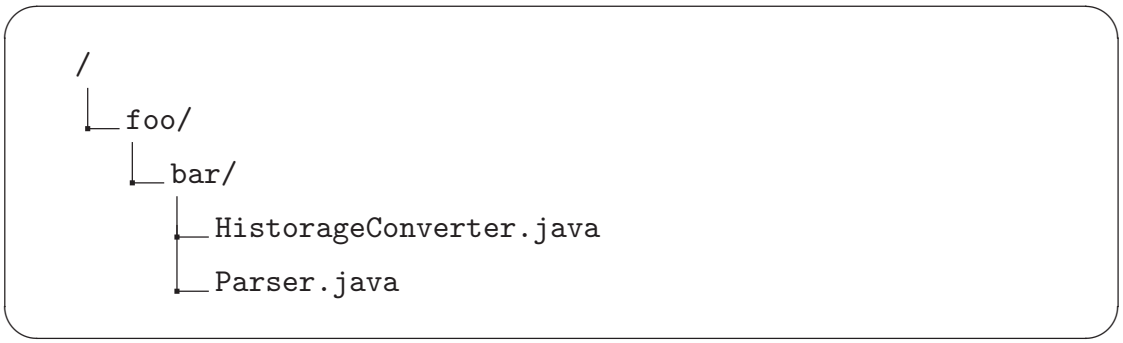

図 4.1 変換前のディレクトリ構造

節で述べたように構文情報リポジトリにはメソッド呼び出しに関する構文情報は記録されていない．そこで，$targetMethod$ に追加された行に対して部分的な構文解析を行うことでメソッド呼び出しに関する情報を取得する．これにより，提案手法はメソッド呼び出しに関する構文解析を行う対象を削減し，計算時間を短縮できる．これらの手続きによりメソッド抽出リファクタリングの候補を検出し，最終的に類似度が閾値以上のものをリファクタリングとして検出する．

4.3.3. メソッドの引き上げリファクタリングの検出

提案手法により検出されるメソッドの引き上げリファクタリングは，変更前に子クラスに存在した引き上げ対象のメソッド（$sourceMethod$），親クラスに作成され

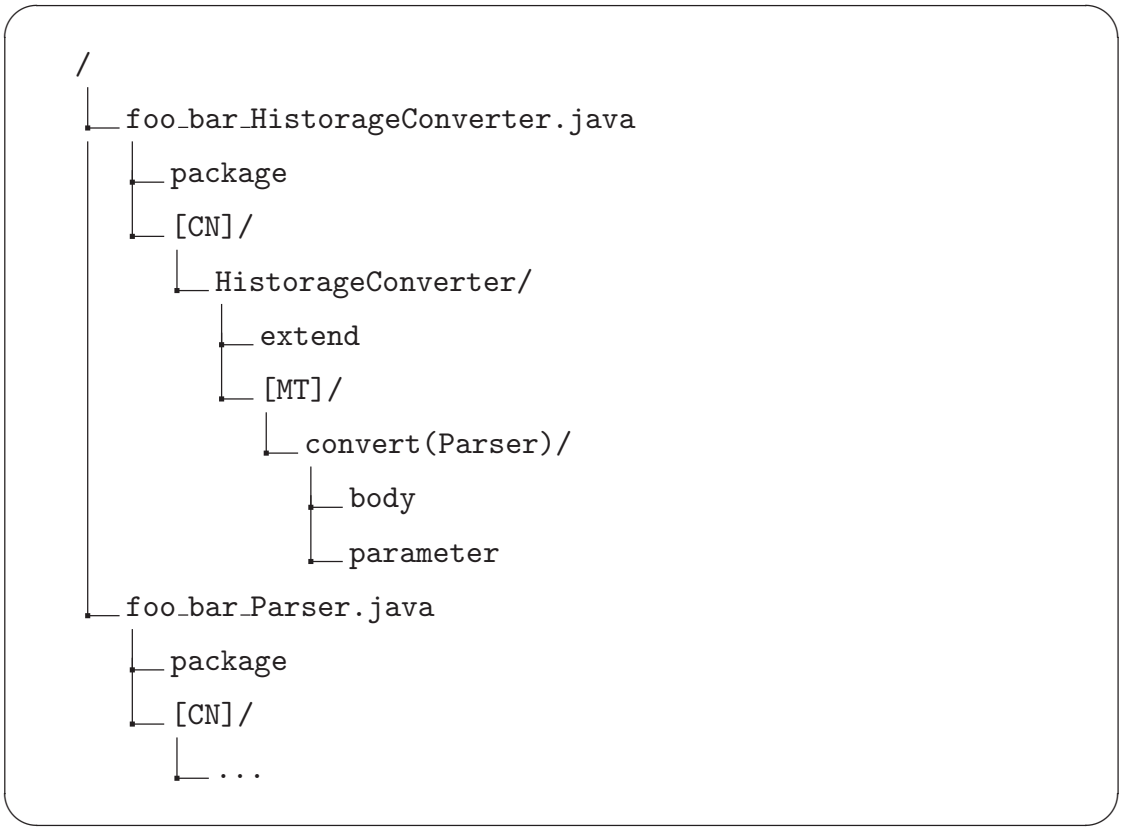

図 4.2 変換後のディレクトリ構造

た引き上げ後のメソッド（*destinationMethod*）とこれらメソッド間の類似度に関する情報で構成される．

Algorithm 2. にメソッドの引き上げリファクタリングの候補を検出する手続きを示す．メソッドの引き上げリファクタリングは，追加されたメソッドと変更されたメソッドの組に対して，追加されたメソッドを保持するクラスが削除されたメソッドを保持するクラスの親クラスである場合に（11 行目）それぞれのメソッドの類似度を計算し，リファクタリングの候補とする（13–15 行目）．ただし，削除されたメソッドを保持するクラスが変更後のコミットにおいて存在しない場合は類似度を計算せず，リファクタリングの候補としない．

```
1  Input  : C, ;                    /* a set of classes modified in r_{i+1} */
2  r_i, ;                                  /* a commit before the change */
3  r_{i+1} ;                                /* a commit after the change */
4  Output: R ;                  /* a set of extract method candidates  */
5  R := ∅;
6  foreach  c ∈ C do
7      foreach  m_a ∈ addedMethod(c, r_i, r_{i+1}) do
8          body_a := getBody(m_a) ;
9          foreach  (m_{c,i}, m_{c,i+1}) ∈ changedMethod(c, r_i, r_{i+1}) do
10             added := getAddedLines(m_{c,i}, m_{c,i+1}) ;
11             deleted := getDeletedLines(m_{c,i}, m_{c,i+1}) ;
12             if  ∃ m_a ∈ getCallee(added) then
13                 sim := calculateSimilarity(body_a, deleted) ;
14                 R.push((m_a, m_{c,i+1}, sim)) ;
15             end
16         end
17     end
18 end
```

Algorithm 1: メソッド抽出リファクタリングの候補の検出

4.3.4. 類似度の計算

本節ではメソッド抽出及び，メソッドの引き上げリファクタリングの検出に使用するメソッド間の類似度を計算する方法を説明する．提案手法は，自然言語処理の分野で文書間の類似度を計算するのに利用される 2 – *shingles* [63] を用いてメソッド間の類似度を計算する．Biegel らはコード間の類似度を 3 種類定義し，リファクタリングの検出性能を比較している [64]．そのうち 2 種類の類似度はトークン列

```
1  Input  : M_a, ;                    /* a set of methods added in r_{i+1} */
2  M_d, ;                           /* a set of methods deleted in r_{i+1} */
3  C_{r_{i+1}} ;                     /* a set of classes exists in r_{i+1} */
4  Output: R ;                  /* a set of pull up method candidates  */
5  R := ∅;
6  foreach  m_a ∈ M_a do
7      c_a := getClass(m_a) ;
8      body_a := getBody(m_a) ;
9      foreach  m_d ∈ M_d do
10         c_d := getClass(m_d) ;
11         if  c_a is super class of c_d then
12             if  c_d ∈ C_{r_{i+1}} then
13                 body_d := getBody(m_d) ;
14                 sim := calculateSimilarity(body_a, body_d) ;
15                 R.push((m_a, m_d, sim)) ;
16             end
17         end
18     end
19 end
```

Algorithm 2: メソッドの引き上げリファクタリングの候補の検出

および抽象構文木を用いたコードクローン検出手法に基づいてそれぞれ定義され，CCFinder [45] と JCCD を用いて計測されている．また，残り 1 種類の類似度の定義中には $2-shingles$ が用いられている．彼らの調査によると，いずれの類似度を用いた場合も精度，再現度ともに大きな差異はない．そこで提案手法では，これらの類似度のうち計算速度が最も速いと報告されている $2-shingles$ を採用した．

$2-shingles$ とは文書 s の先頭から単語を順に 2 個ずつ取り出し，取り出した単語の組を要素とする集合 $SH(s)$ のことを言う[*1]．メソッド間の類似度を計算する際は，コード断片を字句解析した結果得られるトークン列を文書，各トークンを単語として扱う．具体的な例として，次に示すコード断片を考える．

```
int a = i++ + this.getConst();
```

このコード断片は字句解析を行うと以下のトークンに分解される．

$$\{int, a, =, i, ++, +, this, ., getConst, (,), ;\}$$

そして，これらのトークンから次の $2-shingles$ が得られる．

$$\{\{int, a\}, \{a, =\}, \{=, i\}, \{i, ++\}, \{++, +\}, \{+, this\},\\ \{this, .\}, \{., getConst\}, \{getConst, (\}, \{(,)\}, \{), ;\}\}$$

この $2-shingles$ を用いて，文書 s_1，s_2 間の類似度 $similarity(s_1, s_2)$ は 0 から 1 の実数値として次の式で計算される．

$$similarity(s_1, s_2) = \frac{|SH(s_1) \cap SH(s_2)|}{|SH(s_1) \cup SH(s_2)|}$$

提案手法では，メソッド抽出リファクタリングにおいては文書 s_1 を $targetMethod$ から削除されたコード断片，文書 s_2 を $extractedMethod$ に追加されたコード断片として類似度を計算する．従来手法は $targetMethod$ 全体を文書 s_1 としていたが，このように文書 s_1 を設定することで，$extractedMethod$ に対応する部分のみと類似度を計算することができると考えられる．メソッドの引き上げリファクタリングでは，$sourceMethod$ および $destinationMethod$ 本体をそれぞれ s_1，s_2 とし，類似度を計算する．

[*1] 単語を w 個ずつ取り出し，要素とする集合を $w-shingles$ と呼ぶ．$2-shingles$ は $w=2$ における集合である．

4.4. 適用実験

提案手法の有効性を検出精度および，検出時間の観点から評価するために，提案手法をツールとして実装し（以降，提案ツールと表記する）[*2]，従来手法との比較を行った．提案ツールは Eclipse JDT を用いて作成した構文解析器を使用して Java 言語のソースコードを構文解析し，構文情報をディレクトリ構造へ変換する．また，類似度の計算およびメソッド呼び出し情報の取得に必要な字句解析器と構文解析器の作成には Pyrem Torq [65][*3]を利用した．提案手法の比較対象として，以下に示す理由から Ref-Finder および UMLDiff を採用した．

- 実装がツールとして公開されており，入手可能．
- 提案ツールと同じく Java 言語を対象としている．
- リファクタリング検出結果を CSV や XML などのデータ形式に書き出し可能．

適用実験に用いたオープンソースプロジェクトの Columba[*4]と jEdit[*5]の概要を表 4.2 に，実験の概要を図 4.3 に示す．適用実験は以下の手順で行った．

準備: Columba への適用結果を用いた閾値の決定
手順 1: jEdit のリリースバージョンを対象としたリファクタリング検出
手順 2: jEdit の全コミットを対象としたリファクタリング検出

準備として，提案ツールを Columba に適用し，その結果を用いて手順 1，手順 2 にて提案ツールが使用する類似度の閾値を決定した（4.4.1 項にて詳述）．

手順 1 においては jEdit のバージョン 4 から 4.5 までの 6 つのリリースバージョンを対象として，提案ツール，Ref-Finder，UMLDiff を用いてそれぞれリファクタリング検出を行った[*6]．また，各ツールの検出時間を計測した．提案ツールはリポジ

[*2]提案ツールは以下のページから入手可能である：`https://github.com/niyaton/kenja`
[*3]`https://github.com/tos-kamiya/pyrem_torq`
[*4]`http://sourceforge.net/projects/columba/`
[*5]`http://www.jedit.org`
[*6]提案ツールおよび Ref-Finder の実行には Xeon E5650 2.67GHz を 2 基，メモリを 16GB 搭載した OS X 計算機を使用した．また，UMLDiff の実行には Xeon E5540 2.53GHz を 2 基，メモリを

表 4.2 対象プロジェクトの概要

プロジェクト名	Columba	jEdit
種類	メールクライアント	テキストエディタ
開発期間	2006/7/9 – 2012/5/30	2002/4/12 – 2012/1/30
総コミット数	328	4,475
最終 LOC	192,520	177,945

トリを入力として構文情報リポジトリを構築し，リファクタリングを検出する．そのため，対象リリースバージョンに対応するコミットのみを含んだリポジトリを作成し，入力とした．以降，提案ツール，Ref-Finder，UMLDiff の検出結果をそれぞれ REF_{k1}，REF_r，REF_u と表記する．次に，得られた検出結果の統合（和集合の作成）を行った．その際，$targetMethod$ と $extractedMethod$ それぞれについて，メソッドが属するクラスの完全修飾名，メソッド名，引数が一致するものを同一の結果として扱った．Ref-Finder はメソッドの引数に関する情報を出力せず，メソッド名のみを出力する．そのため，Ref-Finder の結果については，手作業で引数情報の対応付けを行った．その際，リリースバージョン間の差分情報と出力結果のクラスの完全修飾名およびメソッド名を基に文献 [50] に記述された論理式を満たすメソッドの組を探索した．なお，対応すると考えられるメソッドのオーバーロードが複数存在した場合は対応付け不可能と判断し，検出結果から除外した．その後，全ての検出結果について手作業で誤検出かどうか確認し，各ツールの精度と再現度を求めた．

手順 2 においては手順 1 で対象とした 6 リリースバージョン間に行われた全ての変更履歴（4475 コミット）から，提案ツールを用いてリファクタリング検出を行った．また，手順 1 と同様にツールの検出時間を計測した．提案ツールの入力には jEdit プロジェクトが公開しているリポジトリを使用し[*7]，検出結果から対象期間に該当する結果のみを抽出した．そして，この検出結果（REF_{k2} と表記する）について手作業で誤検出かどうかを確認し，提案手法の精度を求めた．

12GB 搭載した Linux サーバを使用した．

[*7] `http://sourceforge.net/p/jedit/jEdit.bak/` にて取得可能な Git リポジトリを使用した．

4.4.1.　閾値の決定

既存手法との比較を行うにあたり，Columba の全コミット間に提案ツールを適用し，その結果を用いて類似度の閾値を決定した．適用の結果，メソッド抽出リファクタリングの候補が 56 件，メソッドの引き上げリファクタリングの候補が 114 件得られた．これらの候補に対してリファクタリングであるかどうか手作業で確認を行ったところ，それぞれ 13 件と 6 件のリファクタリングが正しく検出できていることが分かった．

次に，各リファクタリング候補の類似度に着目し，最も多くの正解を検出しつつ，誤検出を少なくすることができる値を閾値とした．その閾値は，メソッド抽出リ

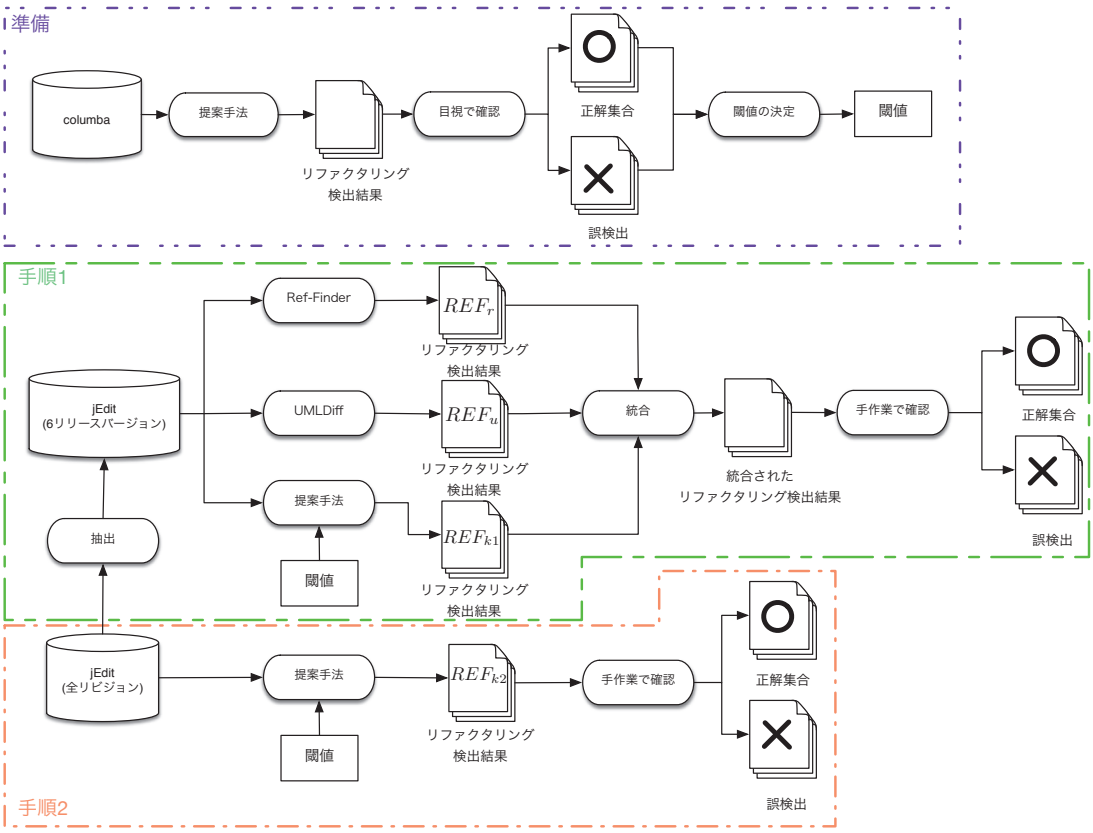

図 4.3　実験概要

ファクタリングについては 0.300，メソッドの引き上げリファクタリングについては 0.895 である．

4.4.2. 精度と再現度

提案手法と，従来手法によるリファクタリング検出の結果を合わせることで，精度と再現度を求める．ここで，精度と再現度の定義について説明する．本章では Bellon らがコードクローン検出ツール同士の性能を比較する際に用いた評価方法と同様の方法で精度と再現度を計算する [66]．説明の為に，先述した REF_{k1} と REF_r，REF_u，を用いてツール間の比較する場合を考える．この場合は初めに，各ツールから得られたリファクタリング検出結果の和集合 $REF_{k1} \cup REF_r \cup REF_u$ を作成する．次に，この和集合の各要素について手作業で誤検出かどうかをそれぞれ確認する．この確認結果を用いて，精度 P と再現度 R はツール毎に以下の式で計算される．

$$P = \frac{TP}{TP + FP}$$
$$R = \frac{TP}{TP + FN}$$

ここで，TP（True Positive）は対象のツールが正しく検出できたリファクタリングの数を，FP（False Positive）は誤ってリファクタリングとして検出した変更の数を表す．また，FN（False Negative）は他のツールは正しく検出したが，対象のツールが検出しなかったリファクタリングの数を表す．

4.4.3. 適用結果

検出時間の比較

手順 1 および手順 2 において各ツールがリファクタリング検出にかかった時間を表 4.3 に示す．表中の提案ツールの検出時間については，構文情報リポジトリの構築にかかった時間，構文情報からリファクタリングを検出するのにかかった時間とそれらの合計を記載している．表 4.3 からは，提案手法はリリースバージョン間からのリファクタリング検出において，既存手法よりも高速にリファクタリング検出が可能なことが分かる．また，コミット数の多い開発履歴からも実用的な時間でリファ

表 4.3　リファクタリング検出にかかった時間

	ツール	検出時間（分）
手順 1	Ref-Finder	35
	UMLDiff	4500
	提案ツール	$3^{\dagger}+4^{\ddagger}=7$
手順 2	提案ツール	$55^{\dagger}+15^{\ddagger}=70$

† 構文情報リポジトリの構築時間
‡ 構文情報リポジトリからのリファクタリング検出時間

表 4.4　jEdit への適用結果（リリースバージョン間でのリファクタリング検出）

	メソッド抽出				メソッドの引き上げ			
	TP	FP	P	R	TP	FP	P	R
提案手法（REF_{k1}）	143	3	0.98	0.44	0	0	0.00	0.00
Ref-Finder（REF_r）	62	70	0.47	0.19	0	1	0.00	0.00
UMLDiff（REF_u）	117	37	0.76	0.36	0	0	0.00	0.00
$REF_{k1} \cup REF_r \cup REF_u$	206	117	–	–	0	1	–	–

クタリング検出を行うことができる．

精度と再現度の比較

■リリースバージョンを対象とした検出　手順 1 により得られたリファクタリング検出の結果と各ツールの精度と再現度を表 4.4 に示す．メソッド抽出リファクタリングの検出においては，提案ツールの精度と再現度は 0.98，0.44 であった．各ツールの結果と比較すると，提案手法はリリースバージョン間のリファクタリング検出を従来手法より正確かつ数多く行えることが分かる．また，メソッドの引き上げリファクタリングについては，いずれのツールにおいてもリリースバージョン間でのリファクタリング検出ができなかった．表 4.4 の結果について，メソッドの引き上げリファクタリングに関しては Ref-Finder から REF_r として 1 件のリファクタリング

表 4.5 jEdit への適用結果（細粒度リファクタリング検出）

	提案手法（REF_{k2}）		
	TP	FP	P
メソッド抽出	356	5	0.99
メソッドの引き上げ	21	0	1.00

が検出されたが誤検出であった．提案ツールからは閾値を超える候補が検出されなかった．また，REF_u は UMLDiff の検出結果から，4.2.3 節で述べた条件を満たしていなかったものを除外した結果 0 件となった．具体的には，UMLDiff がメソッドの引き上げリファクタリングとして検出した結果のうち，106 件が条件 B-2 に反して変更前の時点で移動先のクラスを継承していなかった．また，4 件が条件 B-3 に反して変更後に移動元になったクラスが削除されていた．条件 B-2 に反した例としては，変更後にのみ移動元と移動先のクラスに継承関係が存在した．これは，リリースバージョン間の特定のコミットにおいて継承関係を追加し，後のコミットにおいてメソッドの引き上げリファクタリングを実施した可能性を示唆している．

■細粒度リファクタリング検出（全コミットを対象とした検出） 手順 2 により得られたリファクタリング検出の結果と提案ツールの精度を表 4.5 に示す．精度はメソッド抽出リファクタリングの検出については 0.99，メソッドの引き上げリファクタリングの検出については 1.00 であった．この結果より，提案ツールはメソッド抽出リファクタリングの細粒度リファクタリング検出において，検出精度がリリースバージョン間での検出と変わらないことが分かる．また，305 件とリリースバージョン間の倍以上のリファクタリングを検出できていることが分かる．つまり，提案手法を用いて細粒度リファクタリング検出を行うことでバージョン間でのリファクタリング検出よりも多くの事例を収集可能である．加えて，細粒度リファクタリング検出は，リリースバージョン間で検出できなかったメソッドの引き上げリファクタリングも検出することができている．

4.5. 考察

4.5.1. リファクタリング検出手法

適用実験の結果から，従来手法より提案手法が高速にリファクタリング検出を実施できることが分かる．しかし，適用実験において，従来手法を用いた細粒度リファクタリング検出を実施していないため，提案手法によりどの程度の高速化が実現できているかの定量的な比較はできない．そこで，細粒度リファクタリング検出におけるコミット数に対する計算量の変化について，提案手法と従来手法の両観点から考察を行う．

計算量を議論するにあたり，総コミット数 R[*8]のリポジトリを分析する場合を考える．ここで，構文解析（4.2.1 章における手順 1）にかかる計算量を $O(X)$，構文解析の差分の解析（手順 2）にかかる計算量を $O(Y)$，パターンマッチングによるリファクタリングの検出（手順 3）にかかる計算量を $O(Z)$ とする．これらの手順には順序関係はあるが手続きはそれぞれ独立である．なお，提案手法と既存手法を比較した場合，手順 1 と手順 2 に関しては計算時間の削減するための工夫をしているが手順 3 については従来手法と同様の手順で行っている，そのため，$O(X)$ と $O(Y)$ についてのみ議論する．

まず，構文解析かかる計算量 $O(X)$ を提案手法と従来手法それぞれの場合で求める．リファクタリング検出においては，構文解析はファイルを単位として実施される．そこで，あるコミット r において追加されたファイル数を返す関数 $\delta_a(r)$，変更されたファイル数を返す関数 $\delta_m(r)$ を考える．ここでは，追加されたファイルも変更されたファイルの一部として扱う．コミット r において変更されたファイルのうち，コミット $r-1$ にも存在したファイルの割合を $\eta_m(r)$ とすると，$\delta_a(r)$ は $\delta_m(r)$ と $\eta_m(r)$ を用いて次のように表される．

$$\delta_a(r) = \delta_m(r)(1 - \eta_m(r))$$

提案手法は 4.3 章で述べたように，各コミットにおいて変更されたファイルのみ構文解析を行う．そのため，総コミット数 R のリポジトリに対して，提案手法が構文解

[*8]定数 C との混同を避けるため，リビジョン（revision）の頭文字である R を用いる．

析する必要のあるファイル数 $\widetilde{file}(R)$ はこれらの関数を用いて次の式で表される．

$$\widetilde{file}(R) = \sum_{r=0}^{R} \delta_m(r)$$

一方，従来手法は各コミットにおける全てのファイルを逐一構文解析する．そのため，従来手法が解析する必要のあるファイル数は，コミット r におけるファイル数を返す関数 $file(r)$ を用いて次のように表される．

$$\sum_{r=0}^{R} file(r) = \sum_{r=0}^{R} \sum_{i=0}^{r} \delta_m(i)(1 - \eta_m(i))$$

提案手法および従来手法の計算量は，$\delta_m(r)$ と $\eta_m(r)$ が分析対象の総コミット数 R に対してどのように変化するかによって決まる．そこで，表 4.6 に示す関数で $\delta_m(r)$ と $\eta_m(r)$ をそれぞれモデル化し，モデルの組み合わせによって計算量がどのように変化するかを考察する．

表 4.6 の各式は，ファイルの追加数と変更数が，プロジェクトの進行に従ってどのように変化するかをモデル化している．まず，プロジェクトの進行に関わらず，変更および追加されるファイル数がほぼ一定である場合を考える．この時，コミットあたりの平均変更ファイル数および平均追加ファイル数をそれぞれ α，β として，$\delta_m(r)$ と $\eta_m(r)$ をモデル化した．

次に，プロジェクトの進行に従って，変更されるファイル数が次第に減少する場合を考え，$\delta_m(r)$ を $ke^{-\lambda r}$ としてモデル化した．ここで，k および λ はプロジェクトの規模や成熟速度によってそれぞれ決まる定数である．また，プロジェクトの進行に従い，変更ファイルに占める既存ファイルの割合が増加する場合を考え，$\eta_m(r)$ を $1 - le^{-\gamma r}$ としてモデル化した．ここで，l および γ はプロジェクトの規模や成熟速度によってそれぞれ決まる定数である．このモデルはプロジェクトが成熟すると新規ファイルの作成が減少し，既存ファイルの保守が主になることを表している．

これらのモデルを用いて提案手法と既存手法がそれぞれ構文解析にかかる計算量を求めた結果を表 4.7 に示す．なお，計算量の詳細な導出過程については付録 B を参照されたい．表中の各セルはそれぞれのモデルを採用した場合の提案手法と既存手法の計算量を表している．この表より，いずれのモデルを採用した場合でも提案手

表 4.6 ファイル変更のモデル化

	減少	一定	増加
$\delta_m(r)$	$ke^{-\lambda r}$	α	–
$\eta_m(r)$	–	β	$1 - le^{-\gamma r}$

表 4.7 提案手法および既存手法の計算量

$\eta_m \backslash \delta_m$	減少		一定	
	提案	既存	提案	既存
一定	$O(1)$	$O(R)$	$O(R)$	$O(R^2)$
増加	$O(1)$	$O(R)$	$O(1)$	$O(R)$

法が既存手法よりも少ない計算量で構文解析を実施できることが分かる．特に，分析対象のコミット数が多く，プロジェクトが $\delta_m(r) = \alpha, \eta_m(r) = \beta$ のモデルに従う場合に，提案手法を用いることで従来手法より高速に細粒度リファクタリング検出を実施可能であると言える．

本節で考察したモデルの他に，プロジェクトの進行に従って，修正されるファイル数が増加するモデルと新規作成ファイルが増加するモデルがそれぞれ考えられる．しかし，このようなモデルに合致するソフトウェアプロジェクトは一般的でないと考えられるので本論文では扱わない．

$O(X)$ と同様に，構文要素の対応付けの計算量 $O(Y)$ について考察する．従来手法は，各コミットのスナップショットから得られた全ての構文要素に対して，コミット間での対応付けを行っている．一方，提案手法は変更されなかったファイルから同一の構文要素が得られる特性を利用し，変更の行われたファイルに関する構文要素についてのみコミット間での対応付けを行っている．構文要素の数は構文解析の対象となるファイル数に依存すると考えられるため，$O(Y)$ についても $O(X)$ と同様の高速化がなされていることが考えられる．

4.5.2. メソッド抽出リファクタリングの検出精度の向上

適用実験の結果から，提案手法はメソッド抽出リファクタリングに関して，従来手法よりも正確かつ数多くのリファクタリングを検出できることが分かる．特に，提案手法では誤検出の数が従来手法と比べて圧倒的に少ないことについて議論する．

提案手法はメソッド間の類似度を計算する際に $2-shingles$ を用いて算出している．一方，従来手法である UMLDiff は抽象構文木の構造に基づく類似度を算出する．先に述べたように，Biegel らによると $2-shingles$ を用いた類似度と抽象構文木に基づく類似度のどちらをメソッド間の類似度の算出に利用したとしても，検出の精度に大きな差は生じないと報告している [64]．提案手法は，メソッド抽出の対象となったメソッド（$targetMethod$）から削除されたコード断片と抽出されたメソッド（$extractedMethod$）全体のコード断片の類似度を計算する．一方，UMLDiff は $targetMethod$ 全体のコード断片と $extractedMethod$ 全体のコード断片から類似度を計算する．そのため，従来手法は抽出したコード断片に対応するはずのないコード断片を含んで類似度を計算してしまう可能性がある．例えば，メソッド抽出の対象となったメソッド全体の行数に対して，新たに抽出されたメソッドの行数が少ない場合，提案手法の類似度と比べて従来手法の類似度が小さくなると考えられる．このことが原因で，提案手法に比べて従来手法の精度が落ちているものと考えられる．

4.5.3. 他のリファクタリングパターンへの適用可能性

適用実験において対象とした 2 つのリファクタリングパターン以外への適用可能性について議論する．適用実験において対象としたメソッド抽出は，コードの抽出を行う代表的なリファクタリングパターンである．そのため，コードを手続きとして抽出する手順を含む他のリファクタリングパターン（Template Method の形成等）においても，メソッド抽出に対して示された有効性を期待できると考えられる．また，メソッドの引き上げは，クラス間でメソッドの移動を行う代表的なリファクタリングパターンである．そのため，メソッドの引き上げに対して示された有効性は，クラス間でメソッドの移動を行う他のリファクタリングパターン（スーパークラスの抽出等）に対しても期待できると考えられる．

Prete らは，コミット間においてリファクタリングパターンが適用されているかどうかを判定する条件を論理式で記述している [50]．彼らは，63 個のリファクタリングパターンに対して条件を記述しており，提案手法を拡張しこれら条件の判定を実現することで，メソッド抽出やメソッドの引き上げ以外のリファクタリングパターンに対しても検出を行うことができると考えられる．彼らが記述した条件を表す論理式に用いられている述語のほとんどは表 4.1 に示した構文情報リポジトリの構築において記録する構文情報，およびメソッド呼び出しに関する情報である．そのため，彼らが記述した条件の判定を行い，他のリファクタリングパターンに対応するよう提案手法を拡張することは，多くのリファクタリングパターンにおいて容易であると考えられる．ただし，現状の提案手法では，メソッド内の変数宣言に関する構文情報を取得していない．そのため，一時変数の分離等のメソッド内で行われる変数宣言に関するリファクタリングパターンの検出ついては，構文情報の取得部分に拡張が必要である．

4.6. 本章のまとめ

本章では構文情報を保持したリポジトリを用いて細粒度リファクタリング検出を実施するための手法を提案した．メソッド抽出及びメソッドの引き上げリファクタリングについて手法を実装したツールと既存手法である Ref-Finder および UMLDiff を jEdit に適用し，検出時間及び，精度と再現度について比較実験を行った．その結果，提案手法を用いて細粒度リファクタリング検出を行うことで，より多くのリファクタリングを正確かつ高速に検出できることが確認できた．

本章では，単一のプロジェクトのみを対象として適用実験を行った．jEdit は開発期間が長く，開発規模も大きいため今回の適用実験対象としては十分であると考えられるが，提案手法の有効性評価により一般性を持たせるためには，より多くのプロジェクトに対して提案手法を適用する必要がある．

また，本章では精度と再現度を求めるにあたり，著者が手作業で誤検出の確認を行った．そのため，偏った判断に基づいて精度と再現度を求めている可能性が適用実験に対する妥当性への脅威として考えられる．この問題の解決策として，複数人による手作業での誤検出確認や，分析対象プロジェクトの開発者自身に検出結果が

正しいかどうかを判断して貰う方法が考えられるが，いずれの場合も個人の主観を完全に排除し，客観的な判断を行うことは難しい．

今後の展望として，より多くの種類のリファクタリングを提案手法を用いて検出できるように技術を確立することが挙げられる．また，提案手法により得られる細粒度なリファクタリングの適用事例に基づいて，リファクタリングの効果の定量評価や，より有用なリファクタリングツールの設計が行われることが期待される．

第 5 章

結論

本論文では，リファクタリングの実施がソフトウェアの品質に及ぼす影響を明確にするために，リファクタリングと欠陥の混入，欠陥の修正との関係を定量的に評価する手法を提案した．

提案した評価手法は，リファクタリング検出技術を用いて版管理システムに記録されたソースコードの変更履歴に基づいてリファクタリングの実施を復元する．また，課題管理システムに記録された欠陥に関する情報を基に，欠陥の修正時期を特定する．そして，欠陥の修正時に変更されたソースコードの履歴を遡ることで欠陥の混入時期を特定する．提案手法は，これらの手順により得られた情報からリファクタリングの実施頻度，欠陥の混入頻度，欠陥の修正頻度を計測し，分析者に提示する．オープンソースソフトウェアプロジェクトである Columba を対象に評価手法の適用実験を行った．その結果，リファクタリングが頻繁に行われた後は，欠陥が混入する頻度が低下していることを観測した．また，欠陥が混入された頻度，リファクタリングの頻度，欠陥が修正された頻度の順に，値が高くなる傾向にあることを確認した．これらの結果から，提案手法を用いてリファクタリングと欠陥の関係を定量的に評価することが可能であることを確認した．しかしながら，適用実験に用いた UMLDiff アルゴリズムでは大規模な開発履歴に手法を適用する際に計算時間が膨大になることが判明した．

リファクタリング検出にかかる計算時間を改善するため，構文情報リポジトリを用いたリファクタリング検出手法を提案し，計算時間と精度について評価を行った．

従来の手法は，任意の 2 バージョン間から実施されたリファクタリングを検出することを目的としていた．そのため，開発履歴中の隣接する全てのバージョン間からリファクタリングを検出するための工夫がされていなかった．提案手法は，構文情報リポジトリを用いることで構文情報の解析を差分的に実施する．これにより，リファクタリング検出にかかる計算時間の大幅な短縮を実現した．オープンソースソフトウェアプロジェクトである jEdit を対象に従来手法である Ref-Finder および UMLDiff アルゴリズムとリファクタリングの検出精度を比較したところ，提案手法が検出にかかる時間を短縮できていることを確認できた．また，従来手法と比較してより正確にリファクタリングを検出できることが確認できた．

本論文の成果により，開発者やプロジェクト管理者が自身のソフトウェア開発プロジェクトにおいて実施されたリファクタリングの定量的な評価を行うことができるようになる．そして，提案した分析手法を用いてリファクタリングプロセスの改善を実施することで，開発者らがより効率的なリファクタリングの実施ができるようになることが期待される．また，第 4 章で提案したリファクタリング検出手法を用いることで，研究者や分析者はこれまで得ることのできなかった件数のリファクタリングの適用事例を容易に取得することが可能となった．このことは，リファクタリングの定量評価だけでなく，リファクタリング支援技術の発展にも貢献すると考える．

謝辞

本研究を進めるにあたり，多くの方々に御指導，御協力，御支援を頂きました．ここに謝意を添えて御名前を記させて頂きます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学研究室 飯田 元 教授には，本研究の全過程において熱心な御指導を賜りました．研究方針だけではなく，研究に対する姿勢，論文執筆，発表方法についても多くの御助言を頂きました．心より厚く御礼を申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学研究室 松本 健一 教授には，様々な場面で本研究に対し貴重な御指導，御助言を賜りました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ユビキタスコンピューティングシステム研究室 安本 慶一 教授には，様々な場面で本研究に対し貴重な御指導，御助言を賜りました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 市川 昊平 准教授には，要所要所で本研究に対し貴重な御指導，御助言を賜りました．心より感謝申し上げます．

名古屋大学 大学院情報科学研究科 附属組込みシステム研究センター 吉田 則裕 准教授には，本研究のあらゆる場面において熱心なご指導を賜りました．また，論文執筆，発表方法だけでなく，研究者としての心構えについて，多大なご指導を賜りました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 畑 秀明 助教には，本研究に対し貴重な御助言を賜りました．先生の考案された Historage なくして本研究のリファクタリング検出手法は成り立ちません．また，先生には常日頃より研究に熱心に取り組む姿勢を学ばせて頂きました．心より感謝申し上げます．

株式会社 NTT データ 伏田 享平様には，本研究を進めるにあたり，貴重な御助言を賜りました．また，研究や学生生活についても多数の御助言を頂きました．心より感謝申し上げます．

I would like to thank Assistant Professor Keung Wai Jacky in the City University of Hong Kong for providing invaluable advice to this resaerch. Without his advice, I would not design my refactoring detection tool.

I also would like to thank Dr. Camargo Cruz Ana Erika for providing great feedback.

I wish to thank Assistant Professor Raula Gaikovina Kula in the Osaka University for his helpful advice for this research.

東京工業大学 大学院情報理工学研究科 林 晋平 助教には，本研究を進めるにあたり多くの御助言を賜りました．先生とは学会や共同研究の打ち合わせにおいて大変意義のある時間を共有させて頂きました．心より感謝申し上げます．

大阪大学 大学院国際公共政策研究科 崔 恩瀞 助教には，本研究を進めるにあたり多くの御助言を頂きました．また，博士後期課程に進学した同期として研究生活やプライベートにおいて楽しい時間を共有させて頂きました．心より厚くお礼申し上げます．

新日鉄住金ソリューションズ株式会社 後藤 祥氏には，本研究を進めるにあたり貴重なご意見を頂きました．本研究で提案し，実装したツールは氏のご助力によりデバッグを行うことができました．心より厚くお礼申し上げます．

大阪工業大学 情報科学部 情報システム学科 井垣 宏 准教授，京都産業大学 コンピュータ理工学部 コンピュータサイエンス学科 玉田 春昭 准教授には，博士課程における研究を進めるにあたり，広範囲かつ多大なご協力を頂きました．心より感謝申し上げます．

オムロンソフトウェア株式会社 山田 悠太氏，株式会社インターネットイニシアティブ 濵﨑 一樹氏，株式会社東芝 浦田 大地氏におかれましては，皆様がソフトウェア設計学研究室に在籍している間に楽しく研究させて頂きました．心より厚くお礼申し上げます．

奈良先端科学技術大学院大学情報科学研究科ソフトウェア設計学研究室，ならびにソフトウェア工学研究室，ユビキタスコンピューティングシステム研究室の皆様

には，日頃より多大な御協力と御助言を頂き，公私ともに支えて頂きました．ありがとうございました．

株式会社 日立製作所 藤原 雄介氏，日本電気株式会社 中山 直輝氏，奈良先端科学技術大学院大学 槇原 絵里奈氏，上村 恭平氏，川島 尚己氏，斎藤 雄輔氏には本研究において開発した kenja の開発において多大なご助力を頂きました．心より厚くお礼申し上げます．

奈良先端科学技術大学院大学 川本 理恵氏，呂 悠妃氏，吉川 弥甫氏，小川 暁子氏，荒井 智子氏，森本 恭代氏，高岸 詔子氏，には研究の遂行に必要な事務処理など，多岐にわたり御助力頂きました．心より感謝申し上げます．

最後に，日頃より私を支えてくれた母と祖父，兄姉，そして亡くなった父に心より深く感謝します．

参考文献

[1] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[2] Martin Fowler. Refactoring Home Page. `http://refactoring.com/`.

[3] Martin Fowler. *Refactoring: improving the design of exsiting code.* Addison Wesley, 1999.

[4] Marco D' Ambros, Harald C. Gall, Michele Lanza, and Martin Pinzger. Analysing Software Repositories to Understand Software Evolution. In *Software Evolution*, pp. 37–67. Springer, 2008.

[5] Ahmed E Hassan. The road ahead for Mining Software Repositories. In *Proceedings of the Frontiers of Software Maintenance (FoSM) at the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 48–57. IEEE, 2008.

[6] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR 2005)*, pp. 1–5, 2005.

[7] Raula Gaikovina Kula, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida. Micro process analysis of maintenance effort: an open source software case study using metrics based on program slicing. *Journal of Software: Evolution and Process*, Vol. 25, No. 9, pp. 935–955, 2013.

[8] 門田暁人, 伊原彰紀, 松本健一. 『ソフトウェア工学の実証的アプローチ』シリーズ第 5 回ソフトウェアリポジトリマイニング. コンピュータ ソフトウェア, Vol. 30, No. 2, pp. 52–65, 2013.

[9] Woosung JUNG, Eunjoo LEE, and Chisu WU. A Survey on Mining Software Repositories. *IEICE Transactions on Information and Systems*, Vol. E95-D, No. 5, pp. 1384–1406, May 2012.

[10] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of

software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 19, No. 2, pp. 77–131, March 2007.

[11] Seymour Lipschutz, 成嶋弘 (監訳). マグロウヒル大学演習 離散数学 コンピュータサイエンスの基礎数学. オーム社, 1995.

[12] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, pp. 2–6, 2004.

[13] 小川明彦, 阪井誠. Redmine によるタスクマネジメント実践技法. 翔泳社, 2010.

[14] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

[15] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, pp. 404–428, 2006.

[16] Kunal Taneja, Danny Dig, and Tao Xie. Automated Detection of API Refactorings in Libraries. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pp. 377–380, 2007.

[17] Eunjong Choi, Norihiro Yoshida, and Katsuro Inoue. An Investigation into the Characteristics of Merged Code Clones during Software Evolution. *IEICE Transactions on Information and Systems*, Vol. E97-D, No. 5, pp. 1244–1253, 2014.

[18] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pp. 151–160, 2011.

[19] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. Towards Refactoring-Aware Code Review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2014)*, pp. 99–102, 2014.

[20] Shinpei Hayashi, Sirinut Thangthumachit, and Motoshi Saeki. REdiffs:

Refactoring-Aware Difference Viewer for Java. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*, pp. 487–488, 2013.

[21] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 5–18, January 2012.

[22] Gustavo Soares, Bruno Catao, Catuxe Varjao, Solon Aguiar, Rohit Gheyi, and Tiago Massoni. Analyzing Refactorings on Software Repositories. In *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering (SBES 2011)*, pp. 164–173, 2011.

[23] Shinpei Hayashi, Yasuyuki Tsuda, and Motoshi Saeki. Search-Based Refactoring Detection from Source Code Revisions. *IEICE Transactions on Information and Systems*, Vol. E93-D, No. 4, pp. 754–762, 2010.

[24] Chris Parnin and Carsten Görg. Lightweight Visualizations for Inspecting Code Smells. In *Proceedings of the 2006 ACM Symposium on Software Visualization (SOFTVIS 2006)*, pp. 171–172, 2006.

[25] Carsten Görg and Peter Weißgerber. Detecting and Visualizing Refactorings from Software Archives. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, pp. 205–214, 2005.

[26] Emerson R. Murphy-Hill and Andrew P. Black. Why Don't People Use Refactoring Tools? In *Proceedings of the 1st ACM Workshop on Refactoring Tools (WRT 2007)*, pp. 60–61, 2007.

[27] Kim Herzig and Andreas Zeller. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013)*, pp. 121–130, 2013.

[28] Miryung Kim and David Notkin. Discovering and Representing Systematic Code Changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pp. 309–319, 2009.

[29] Miryung Kim, David Notkin, and Dan Grossman. Automatic Inference of Structural Changes for Matching across Program Versions. In *Proceedings*

of the 29th International Conference on Software Engineering (ICSE 2007), pp. 333–343, 2007.

[30] Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering*, Vol. 31, No. 2, pp. 166–181, 2005.

[31] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, pp. 143–152, 2005.

[32] Beat Fluri, Michael Wursch, Martin PInzger, and Harald C Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, Vol. 33, No. 11, pp. 725–743, 2007.

[33] Emerson Murphy-Hill, Andrew P. Black, Danny Dig, and Chris Parnin. Gathering Refactoring Data: A Comparison of Four Methods. In *Proceedings of the 2nd ACM Workshop on Refactoring Tools (WRT 2008)*, 2008.

[34] 大庭晋. ソフトウェアのバージョン間で実施されたリファクタリング検出手法の改良. Master's thesis, 信州大学大学院工学系研究科, 2013.

[35] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the Relation of Refactorings and Software Defect Prediction. In *Proceedings of the 5th Working Conference on Mining Software Repositories (MSR 2008)*, pp. 35–38, 2008.

[36] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–Does It Improve Software Quality? In *Proceedings of the 5th International Workshop on Software Quality (WOSQ 2007)*, 2007.

[37] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning Development Tools with the Way Programmers Think About Code Changes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2007)*, pp. 567–576, 2007.

[38] Emerson Murphy-Hill and Andrew P. Black. Breaking the Barriers to Suc-

cessful Refactoring: Observations and Tools for Extract Method. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pp. 421–430, 2008.

[39] Markus Pizka. Straightening Spaghetti-Code with Refactoring? In *Proceedings of the 2004 International Conference on Software Engineering Research and Practice (SERP 20014)*, pp. 846–852, 2004.

[40] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pp. 427–436, 2007.

[41] Gail C. Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, Vol. 23, No. 4, pp. 76–83, July 2006.

[42] Romain Robbes and Michele Lanza. SpyWare: A Change-aware Development Toolset. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pp. 847–850, 2008.

[43] Zhenchang Xing and Eleni Stroulia. Refactoring Detection Based on UMLDiff Change-Facts Queries. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pp. 263–274, 2006.

[44] Peter Weißgerber and Stephan Diehl. Identifying Refactorings from Source-Code Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pp. 231–240, 2006.

[45] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, Vol. 28, pp. 654–670, July 2002.

[46] Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pp. 54–65, 2005.

[47] Zhenchang Xing and Eleni Stroulia. Differencing logical UML models. *Automated Software Engineering*, Vol. 14, No. 2, pp. 215–259, 2007.

[48] Zhenchang Xing and Eleni Stroulia. The JDEvAn Tool Suite in Support of Object-Oriented Evolutionary Development. In *Proceedings of Companion of the 30th International Conference on Software Engineering(ICSE Companion 2008)*, pp. 951–952, 2008.

[49] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation–The A-Prolog perspective. *Artificial Intelligence*, Vol. 138, No. 1–2, pp. 3–38, 2002.

[50] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based Reconstruction of Complex Refactorings. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010)*, 2010.

[51] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010)*, pp. 371–372, 2010.

[52] Raula Gaikovina Kula, Kyohei Fushida, Shinji Kawaguchi, and Hajimu Iida. Analysis of Bug Fixing Processes Using Program Slicing Metrics. In *Proceedings of the 11th Product-Focused Software Process Improvement (PROFES 2010)*, pp. 32–46, 2010.

[53] Kimiharu Ohkura, Keita Goto, Noriko Hanakawa, Shinji Kawaguchi, and Hajimu Iida. Project Replayer with email analysis – revealing contexts in software development. In *Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC 2006)*, pp. 453–460, 2006.

[54] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know it. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pp. 287–297, 2009.

[55] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The Missing Links: Bugs and Bug-Fix Commits.

In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010)*, pp. 97–106, 2010.

[56] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, pp. 121–130, 2009.

[57] Barthélémy Dagenais and Martin P. Robillard. Recommending Adaptive Changes for Framework Evolution. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pp. 481–490, 2008.

[58] Eunjgon Choi, Norihiro Yoshida, and Katsuro Inoue. What Kind of and How Clones Are Refactored?: A Case Study of Three OSS Projects. In *Proceedings of the 5th Workshop on Refactoring Tools (WRT 2012)*, pp. 1–7, 2012.

[59] Qiang Tu and Michael W. Godfrey. An Integrated Approach for Studying Architectural Evolution. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, pp. 127–136, 2002.

[60] Hideaki Hata, Osamu Mizuno, and Toru Kikuno. Historage: Fine-grained Version Control System for Java. In *Proc. of the 12th International Workshop on Principles on Software Evolution and 7th ERCIM Workshop on Software Evolution (IWPSE-EVOL 2011)*, pp. 96–100, 2011.

[61] Kyle Prete, Napol Rachatasumrit, and Miryung Kim. A catalogue of template refactoring rules. Technical Report UTAUSTINECE-TR-041610, The University of Texas at Austin, 2010.

[62] Miryung Kim and David Notkin. Program Element Matching for Multi-Version Program Analysis. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pp. 58–64, 2006.

[63] Andrei Z. Broder. On the Resemblance and Containment of Documents. In *SEQUENCES1997,Proc. of the Compression and Complexity of Sequences*

(SEQUENCES 1997), pp. 21–29, 1997.

[64] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR 2011)*, pp. 53–62, 2011.

[65] 神谷年洋. リバースエンジニアリングツールキット Remics の試作. 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, Vol. 109, No. 170, pp. 7–11, 2009.

[66] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

付録

A. UMLDiff により検出可能なリファクタリング一覧

Convert inner type to top-level	Convert top-level to inner
Extract subsystem	Inline subsystem
Extract package	Inline package
Pull-up method/field	Push-down method/field
Pull-up behavior	Push-down behavior
Pull-up constructor body	Extract interface
Extract superclass	Extract subclass
Inline superclass	Inline subclass
Form template method	Replace inheritance with delegation
Replace delegation with inheritance	Extract class
Inline class	Die-hard/legacy classes
Convert anonymous class to nested	Move method/field
Move behavior	Deprecation + delegation
Information hiding	Generalize type
Downcast type	Introduce factory method
Introduce parameter object	Encapsulate field
Preserve whole object	

B. 構文情報を用いたリファクタリング検出手法の計算量

B.1. $\delta_m = ke^{-\lambda x}$ の場合

$$\begin{aligned}
\widetilde{file}(R) &= \sum_{r=0}^{R} \delta_m(r) \\
&= \sum_{r=0}^{R} ke^{-\lambda r} \\
&= k\frac{1-e^{-\lambda(R+1)}}{1-e^{-\lambda}} \\
\therefore O(\widetilde{file}(R)) &= O(1)
\end{aligned}$$

B.2. $\delta_m = \alpha$ の場合

$$\begin{aligned}
\widetilde{file}(R) &= \sum_{r=0}^{R} \delta_m(r) \\
&= \sum_{r=0}^{R} \alpha \\
&= \alpha(R+1) \\
\therefore O(\widetilde{file}(R)) &= O(R)
\end{aligned}$$

C. 従来手法におけるリファクタリング検出の計算量

C.1. $\delta_m = ke^{-\lambda x}, \eta_m = \beta$ の場合

$$
\begin{aligned}
\sum_{r=0}^{R} file(r) &= \sum_{r=0}^{R}\sum_{i=0}^{r} \delta_m(i)(1-\eta_m(i)) \\
&= \sum_{r=0}^{R}\sum_{i=0}^{r} ke^{-\lambda i}(1-\beta) \\
&= k(1-\beta)\sum_{r=0}^{R}\sum_{i=0}^{r} e^{-\lambda i} \\
&= k(1-\beta)\sum_{r=0}^{R}\frac{1-e^{-\lambda(r+1)}}{1-e^{-\lambda}} \\
&= \frac{k(1-\beta)}{1-e^{-\lambda}}\sum_{r=0}^{R}(1-e^{-\lambda(r+1)}) \\
&= \frac{k(1-\beta)}{1-e^{-\lambda}}(R+1+e^{-\lambda}\frac{1-e^{-\lambda(R+1)}}{1-e^{-\lambda}}) \\
\therefore O(\sum_{r=0}^{R} file(r)) &= O(R)
\end{aligned}
$$

C.2. $\delta_m = ke^{-\lambda x}, \eta_m = 1 - le^{-\gamma x}$ の場合

$$\begin{aligned}
\sum_{r=0}^{R} file(r) &= \sum_{r=0}^{R}\sum_{i=0}^{r} \delta_m(i)(1-\eta_m(i)) \\
&= \sum_{r=0}^{R}\sum_{i=0}^{r} ke^{-\lambda i} \times le^{-\gamma i} \\
&= kl\sum_{r=0}^{R}\sum_{i=0}^{r} e^{-(\lambda+\gamma)i} \\
\therefore O(\sum_{r=0}^{R} file(r)) &= O(R)
\end{aligned}$$

C.3. $\delta_m = \alpha, \eta_m = \beta$ の場合

$$\begin{aligned}
\sum_{r=0}^{R} file(r) &= \sum_{r=0}^{R}\sum_{i=0}^{r} \delta_m(i)(1-\eta_m(i)) \\
&= \sum_{r=0}^{R}\sum_{i=0}^{r} \alpha(1-\beta) \\
&= \alpha(1-\beta)\sum_{r=0}^{R}\sum_{i=0}^{r} 1 \\
&= \alpha(1-\beta)\sum_{r=0}^{R}(r+1) \\
&= \alpha(1-\beta)\frac{(R+1)(R+2)}{2} \\
&= \frac{\alpha(1-\beta)}{2}(R^2+2R+2) \\
\therefore O(\sum_{r=0}^{R} file(r)) &= O(R^2)
\end{aligned}$$

C.4. $\delta_m = \alpha, \eta_m = 1 - le^{-\gamma x}$ の場合

$$\begin{aligned}
\sum_{r=0}^{R} file(r) &= \sum_{r=0}^{R}\sum_{i=0}^{r} \delta_m(i)(1-\eta_m(i)) \\
&= \sum_{r=0}^{R}\sum_{i=0}^{r} \alpha l e^{-\gamma i} \\
&= \alpha l \sum_{r=0}^{R}\sum_{i=0}^{r} e^{-\gamma i} \\
\therefore O(\sum_{r=0}^{R} file(r)) &= O(R)
\end{aligned}$$