修士学位論文

題目

ソースコードの修正作業状況に基づく
メソッド移動リファクタリング候補推薦ツールの提案

指導教員

井上　克郎　教授

報告者

氏原　直哉

平成 29 年 2 月 7 日

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 ソフトウェア工学講座

平成 28 年度 修士学位論文

ソースコードの修正作業状況に基づくメソッド移動リファクタリング候補推薦ツールの提案

氏原　直哉

内容梗概

リファクタリングとは，ソフトウェアの外部から見た振る舞いを変えることなく，ソースコードの理解や修正が容易になるように内部構造を改善する作業である．Java プログラムにおけるメソッド移動リファクタリングは，メソッドが，所属クラスよりも他のクラスの属性を利用している場合，または，所属クラスの他のメソッドよりも他のクラスのメソッドから利用されることが多い場合に実施することで，クラス間の結合度やクラス内の凝集度を向上させることができる．

大規模ソフトウェア開発において適切なメソッド移動リファクタリングの実施が困難であることから，メソッド移動リファクタリングの候補を推薦することでリファクタリング活動を支援する研究が数多く行われている．しかし，既存手法は開発者の作業状況を考慮しないため，開発者にとって優先度が高いと思われるようなメソッド移動リファクタリング候補を推薦する手法はない．そのため，ソフトウェア保守作業を行っている開発者が，追加や修正したコードの設計上の品質を向上させたいとしても，有用でないメソッド移動リファクタリング候補ばかりを推薦してしまう可能性がある．

本研究では，開発者が行っている修正作業の状況に基づいて，メソッド移動リファクタリング候補を推薦することで，修正作業と並行して行われるリファクタリング活動を支援する手法を提案する．具体的には，ソースコードが修正される度に依存関係グラフを作成し，その作業によるソースコードの依存関係への影響をメトリクスとして可視化し，リファクタリング活動の必要性の判断材料を提供することで，開発者を支援する．また，修正作業に関連するコードを特定することで，開発者がその設計を熟知していると思われるコードに対してメソッド移動リファクタリング候補を推薦する．

実際に Java 言語で記述されたオープンソースソフトウェアに対して，提案手法の適用と評価を行った結果，提案手法を用いることで 64.7%，実際に行われたメソッド移動リファクタリングを特定することができていることがわかった．

主な用語

メソッド移動リファクタリング
ソフトウェア品質
ソフトウェア保守

目 次

1 まえがき

リファクタリングとは，ソフトウェアの外部から見た振る舞いを変えることなく，ソースコードの理解や修正が容易になるように内部構造を改善する作業である [6]．大規模ソフトウェアのソースコードは，既存機能の拡張や新機能の追加，欠陥修正などのために継続的に変更され続け，時間の経過とともに設計上の品質は低下していく．そのため，適切なリファクタリングを実施することで，ソースコードの可読性，ソフトウェアの拡張性や保守性等，ソフトウェア品質を改善し，ソフトウェア保守作業のコストを低下させることが重要である．

Murphy-Hill らは，リファクタリングを，実施される際の戦略によって floss リファクタリングと root-canal リファクタリングの 2 つに分類している [14]．Floss リファクタリングは，欠陥修正や機能追加など他の修正作業の途中に，その作業に関連したコードの品質を高く維持するために行われるリファクタリングのことである．Root-canal リファクタリングは，リファクタリング活動に専念する時間を確保して，ソフトウェア全体の設計上の欠陥を取り除くために行われるリファクタリングのことである．Fowler は，リファクタリングのための時間を設けるのではなく，他の保守作業と並行してリファクタリングを実施する，floss リファクタリングを推奨している [6]．また，実際のソフトウェア開発において，実施されたリファクタリングの多くは，開発者が携わっているコードに対して，他の保守作業と並行して行われていると言われている [9][15][16].

リファクタリングのためのソースコード修正の操作は，Fowler [6] によって 72 種類に分類，定義されているが，その中でも欠かすことのできないリファクタリングとして挙げられているのがメソッド移動リファクタリング，すなわち，あるメソッドを実装しているクラスから他のクラスへと移動させる操作である．Java プログラムにおけるメソッド移動リファクタリングは，あるクラス内に定義されているメソッドが，所属クラスの属性よりも他のクラスの属性を利用している場合，または，所属クラスの他のメソッドよりも他のクラスのメソッドから利用されることが多い場合に実施することで，クラス間の結合度やクラス内の凝集度を向上させることができる．また，Fowler が定義した，Feature Envy や Shotgun Surgery といった設計上の欠陥であるコードスメルを取り除くことも可能である．

大規模ソフトウェア開発において適切なメソッド移動リファクタリングを実施することは，ソースコードの設計に関する理解や知識が求められるため，時間がかかり，新たな欠陥を生み出す可能性も多い．そのため，メソッド移動リファクタリングの候補を推薦することで，リファクタリング作業の支援を目的とした研究が数多く行われている．既存手法 [5][7][19] では，ソフトウェア全体からソースコードに含まれている設計上の欠陥を特定し，それを修正するようなメソッド移動リファクタリング候補を推薦する手法が提案されている．そのような手法は，root-canal リファクタリングを行う開発者にとっては非常に役に立つが，一方

で，floss リファクタリングを行う開発者にとっては，保守作業によって，新たに追加したコードや修正したコードの設計上の品質を向上させたいとしても，候補の多くが修正作業に関連しないため，あまり有用ではない．すなわち，保守作業を行っている開発者にとって，関心度や優先度が高いと思われるようなコードを考慮したメソッド移動リファクタリング候補推薦手法はない．

そこで本研究では，開発者の修正作業状況に基づいて，floss リファクタリングを支援するメソッド移動リファクタリング推薦ツール *c-JRefRec* を提案する．具体的な方法としては，開発作業中のある時点のソースコード（通常はバージョン管理システムからチェックアウトした直後）の状態でクラス間の依存関係を計測しておき，開発者が行った修正によって生じた依存関係の増減を構造的メトリクスの増減として表示することで，クラスごとの他クラスとの依存関係の強さと，修正作業による依存関係への影響を可視化し，リファクタリング活動の必要性の判断材料を提供する．また，開発者が熟知していると思われる，修正作業に関連したコードだけをメソッド移動リファクタリング候補の探索対象とすることで，開発者が推薦されたリファクタリングを実際に実施するかどうかの判断を容易に行えるような，実施によって欠陥を作りこむ可能性の低いメソッド移動リファクタリング候補を推薦する．

手法の評価を行うために，Java 言語で記述されたオープンソースのソフトウェアに対して適用実験を行った．まず，実験の準備として，リファクタリング検出ツールである RefactoringMiner[20] を用いて，実験対象ソフトウェアの履歴から実際のソフトウェア保守で実施されたメソッド移動リファクタリングを特定した．検出されたリファクタリング内で移動したメソッドが，移動リファクタリングを実施されるよりも以前に修正されたコミットを特定し，そのコミットの修正作業に対し，本手法を適用することで，そのメソッド移動リファクタリングを候補として推薦できるかを調査する．具体的には，ソースコードをその修正作業が行われる前の状態に戻しておき，ソースコードを修正作業後のソースコードに書き換えることで，実際の修正作業が終了した状態を再現する．修正作業前に本手法のツールを起動し，修正作業後にメソッド移動リファクタリング候補を推薦させ，候補の中に検出したメソッド移動リファクタリングが含まれるかによって評価した．実験の結果，提案手法を用いることで，実際に実施されたメソッド移動リファクタリングのうち，64.7%特定できることがわかった．

以降，2 章では研究の背景について，3 章では開発したツールの設計および機能について述べる．4 章ではツールの利用例を紹介し，5 章では適用実験として，実験の方法や結果を述べる．最後に，6 章でまとめについて述べる．

2　背景

本章では，背景として，本研究に関連する用語，関連研究，既存研究におけるリファクタリング作業を支援する手法とその問題点について述べる．

2.1　ソフトウェア品質

ソフトウェア品質には外部品質と内部品質がある．ソフトウェアの外部品質とは，ソフトウェアの振る舞いである機能性や信頼性，使用性によって評価される品質である．一方で，ソフトウェアの内部品質とは，保守性や効率性，移植性といったソフトウェアの内部構造の状態によって評価される品質である．本研究では，ソフトウェアの保守作業を実施する際に，そのソフトウェアが維持しやすい，管理しやすいソフトウェアであるかどうかを評価する指針である保守性によって評価される品質のことを，単にソフトウェア品質と呼ぶ．

ソフトウェアの保守性とは欠陥の修正や新機能の追加，既存機能の拡張などといった保守作業を開発者が容易に行えるかを示す性質であり，保守性の高いソフトウェアは開発者にとってソースコードの内部構造を理解しやすく，拡張もしやすい．ソースコードの保守性を，その内部構造をによって評価する尺度として，凝集度と結合度が一般的に用いられている．

- 凝集度

  モジュール内の結びつきの強さを測る．この値が高いほど，モジュールが単一の機能の実装だけを含んでおり，保守性が高い．

- 結合度

  モジュール間の結びつきの強さを測る．この値が低いほど，モジュールが単一の機能の実装だけを含んでおり，保守性が高い．

特に，Java 言語のようなオブジェクト指向言語で記述されたソフトウェアは，モジュール内の凝集度を高め，モジュール間の結合度を低くすることで，より再利用がしやすく欠陥を見つけやすいソースコードにすることができる．

しかし，このようなソフトウェア品質を，開発者がソフトウェアから自力で把握することは難しい．そこで，ソフトウェアメトリクスを用いて，ソフトウェア品質を可視化する手法がある [11]．

また，Fowler [6] は，ソースコードに含まれる設計上の欠陥を種類によって分類し，それぞれをコードスメルとして定義した．コードスメルを用いて，ソフトウェアに含まれている設計上の欠陥を特定することで，ソフトウェア品質を評価する手法もある．

2.1.1 ソフトウェアメトリクス

ソフトウェアメトリクスとは，ソースコードの規模，複雑さ，保守性など，ソフトウェアの特定の側面を数値化して定量的に示す指標である．今までに提唱されたソースコードを対象とするソフトウェアメトリクスとしては，Halsted のメトリクス [8]，McCabe のサイクロマチック数 [13]，Chidamber と Kemerer が提案した CK メトリクス [4] などが挙げられる．

ここでは既存研究で提案されているソフトウェアメトリクスをいくつか紹介する．最も定義と計算が簡単であり，長い歴史を持つものの 1 つとしてソースコードの行数を示す LOC が挙げられる．LOC の定義は次のようになる．

- **LOC（ソースコードの行数，lines of code）**

  LOC は計測対象のソースコードの行数である．計測対象のソースコード C の行数を l とする．このとき $LOC(C) = l$ である．

Chidamber と Kemerer が提案した CK メトリクス [4] は，オブジェクト指向ソフトウェアにおけるクラスを対象とするメトリクスとして有名である．CK メトリクスは，WMC，DIT，NOC，CBO，RFC，LCOM の 6 種類のメトリクスから成り，いずれも値が大きいほど複雑であることを意味している．以下にそれぞれのメトリクスの説明を記載する．

- **RFC（クラスへのレスポンス，response for a class）**

  RFC は計測対象クラスのメソッドと，それらのメソッドから呼び出されるメソッドの数の総和である．計測対象クラス C が n 個のメソッド $M_1, M_2, ..., M_n$ を持つとする．また，メソッド $M_1, M_2..., M_n$ は C 以外で定義されているメソッドをそれぞれ $m_1, m_2, ..., m_n$ 個 だけ呼び出すとする．このとき，$RFC(C) = n + \sum_{i=1}^{n} m_i$ である．

- **CBO（クラス間の結合，coupling between object classes）**

  あるクラス C_a が，異なるクラス C_b のメソッドやフィールドを使用するとき，C_a は C_b と結合しているという．CBO は計測対象クラスと結合しているクラスの数である．計測対象クラス C が結合しているクラスの数を c_c とすると，$CBO(C) = c_c$ である．

- **LCOM（メソッドの凝集の欠如，lack of cohesion in methods）**メソッドの凝集度とは，クラスのメソッドがお互いに関連している程度を指す．全てのメソッドが同一のインスタンス変数にアクセスする場合，高い凝集度を持つ．メソッドがアクセスするインスタンス変数の集合が互いに素ならば，凝集度は低くなる．$LCOM$ はメソッドの凝集度の低さを表すメトリクスである．計測対象クラス C が n 個のメソッド $M_1, M_2, ..., M_n$ を持つとする．$I_1, I_2, ..., I_n$ をそれぞれメソッド M_i によって用いられるインスタンス変

数の集合とする．$P = \{(I_i, I_j) \mid I_i \cap I_j = \phi\}$, $Q = \{(I_i, I_j) \mid I_i \cap Ij \neq \phi\}$ と定義する．ただし，$I_1, I_2, ..., I_n$ が 全て ϕ の時は，$P = \phi$ とする．このとき，$LCOM(C) = \mid P \mid - \mid Q \mid$ である．ただし， $LCOM(C) < 0$ となる場合は $LCOM(C) = 0$ とする．

- **NOC（子クラスの数，number of children）** NOC は計測対象クラスを直接継承しているクラスの数である．計測対象クラス C を直接継承している子クラスの数を s_c とする．このとき $NOC(C) = s_c$ である．

- **DIT（継承木における深さ，depth of inheritance tree）** DIT は計測対象クラスの継承木内での深さである．多重継承が許される場合は，DIT は継承木におけるそのクラスを表す節点から，それ以上親クラスが存在しないクラス，すなわち根に至る最長パスの長さとなる．例えば，開発言語が Java であるソフトウェアを対象とする場合，全てのクラスは java.lang.Object を最上位の親クラスとして持つので，java.lang.Object をクラス C_0 と おくと，$DIT(C_0) = 1$ と定義する．C_0 を直接継承したクラスを C_1，C_1 を直接継承 したクラスを C_2 とおくと，それぞれ $DIT(C_1) = 2$，$DIT(C_2) = 3$ となる．

- **WMC（クラスの重み付きメソッド数，weighted methods per class）** WMC は計測対象クラスにおける各メソッドの複雑度の総和である．計測対象クラス C が n 個のメソッド $M_1, M_2, ..., M_n$ を持つとする．これらのメソッ ドの複雑度をそれぞれ $c_1, c_2, ..., c_n$ とする．このとき，$WMC(C) = \sum_{i=1}^{n} c_i$ である．メソッドの複雑度の具体的な算出方法は定められていないが，Halsted のメトリクス [8] や McCabe のサイクロマチック数 [13] などを用いる方法が考えられる．

本手法は，修正作業を行っている開発者に，メソッド移動リファクタリングの必要性の判断材料を提供する目的でメトリクスを表示するため，ソフトウェア品質のうち，凝集度や結合度に関するメトリクスを新たに定義した．

2.1.2 コードスメル

コードスメルとは，ソースコードの設計上の欠陥を示す指標として，Fowler によって提案された．コードスメルは，ソフトウェア開発や保守の作業を困難にする要因の 1 つである．

最初に Fowler が 22 種類のコードスメルを提案した [6]．その後，Emden らや Lanza らなどもコードスメルの種類を提案した．また， コードスメルの種類の分類と，コードスメルの種類間の関係についても研究されている [12][25] ．いくつかのコードスメルを以下に示す．

- **Featuer Envy（属性，操作の横恋慕）**

定義されたクラスのデータよりも他のクラスのデータを多く用いるメソッド．このメソッドは他のクラスで定義されるべきだと考えられる．対応するリファクタリングとしては,，メソッド移動リファクタリングによって，メソッドを適切なクラスに移動することが提案されている．

- **Shotgun Surgey（変更の拡散）**

  変更を行なうと複数のクラスに変更が必要になるメソッド．多くの他のクラスのメソッドを呼び出すまたは呼び出される．対応するリファクタリングとしては，メソッド移動リファクタリングによって，関係するメソッドを一つのクラスに移動してまとめることが提案されている．

- **Blob Operation（長すぎるメソッド）**

  コードが長く，コードの理解に時間がかかるメソッド．複数の機能が実装されていると考えられる．対応するリファクタリングとしては，メソッド抽出リファクタリングによって，複数のメソッドに分割することが提案されている．

- **Duplicate Code(重複したコード)**

  同じようなコードが複数個所にみられるコード．メソッドの抽出リファクタリングによって，各メソッド内で抽出したメソッドを呼び出すことが提案されている．

これらのコードスメルを取り除くために実施すべきとされているのがリファクタリングである．

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から見た振る舞いを変えることなく，ソースコードの理解や修正が容易になるように，内部構造を改善する作業である [6]．すなわち，ソフトウェアの機能は変更せずに，開発者が理解しやすくなるように，機能を追加しやすくなるように，ソースコードを変更する作業である．大規模ソフトウェアのソースコードは，既存機能の拡張や新機能の追加，欠陥修正などのために，継続的に変更され続け，時間の経過とともに設計上の品質は低下していく．そのため，適切なリファクタリングを実施することで，ソースコードの可読性，ソフトウェアの拡張性や保守性等，ソフトウェア品質を改善し，ソフトウェア保守作業のコストを低下させることが重要である．

さらに，Fowler はリファクタリングを行う理由として，以下の 4 つを挙げている．

1. リファクタリングはソフトウェア設計を向上させる

ソフトウェアは，開発者によるコードの変更が短期的な視野にたったものであったり，設計の全体的な理解をせずに行われたりすると，コードは徐々に構造を崩すことになる．さらに，コードの構造が悪化すると累積的に悪影響が及び，コードを読んで設計を把握することも難しくなる．開発者はリファクタリングを定期的に行うことで，コードの構造悪化を未然に防ぎ，良質な状態を保つことができる．

2. リファクタリングはソフトウェアを理解しやすくする

   ソフトウェアは，何かの変更を加えるためにコードを書いた本人とは異なる人がコードを読む可能性がある．開発過程において，少しの時間をリファクタリングに充てるだけで，コードの目的がより伝わるようになる．また，リファクタリングはコードの不明な部分を 理解するにも用いることができる．例えば，コードを理解する際，何をしているのかを突き止めるのに自分の理解をよりよい形で反映するように実際にコードを書き換える．そして，テスト結果が変更前と変わらないことで，自分の理解が間違っていなかったことを確かめることができる．

3. リファクタリングはバグを見つけ出す

   コードが理解できるようになるとバグが見つけやすくなる．プログラム構造を明確 にすることで，コードに対する推測が正しかったとわかり，無理なくバグを発見できる．また，バグレポート自体をリファクタリングが必要な兆候と考えることもできる．コードが不明確なためバグを発見できなかったことを示しているからである．

4. リファクタリングでより速くプログラミングできる

   優れた設計は，ソフトウェア開発のスピードを一定に保つのに役立つ．リファクタリングは，設計を改善しコードを理解しやすくすることで，従来に比べて開発期間の短縮につながる．

以上のように，リファクタリングは，単なるコードのクリーニング作業という意味にとどまらず，欠陥修正や機能追加，コードレビューなど作業全体のあらゆる場面で有用であると言える．以降では，本研究で支援の対象としているメソッド移動リファクタリングについて説明する．

2.2.1 メソッド移動リファクタリング

メソッド移動リファクタリングとは，リファクタリングパターンの 1 つであり，あるメソッドを実装しているクラスから他のクラスへと移動させる操作である．Java プログラムにおけるメソッド移動リファクタリングは，あるクラス内に定義されているメソッドが，所属クラ

スの属性よりも他のクラスの属性を利用している場合，または，所属クラスの他のメソッドよりも他のクラスのメソッドから利用されることが多い場合に実施することが推奨されている．適切に実施することで，クラス間の結合度やクラス内の凝集度を向上させることができる．また，Fowler が定義した，Feature Envy や Shotgun Surgery といった設計上の欠陥であるコードスメルを取り除くことも可能である．

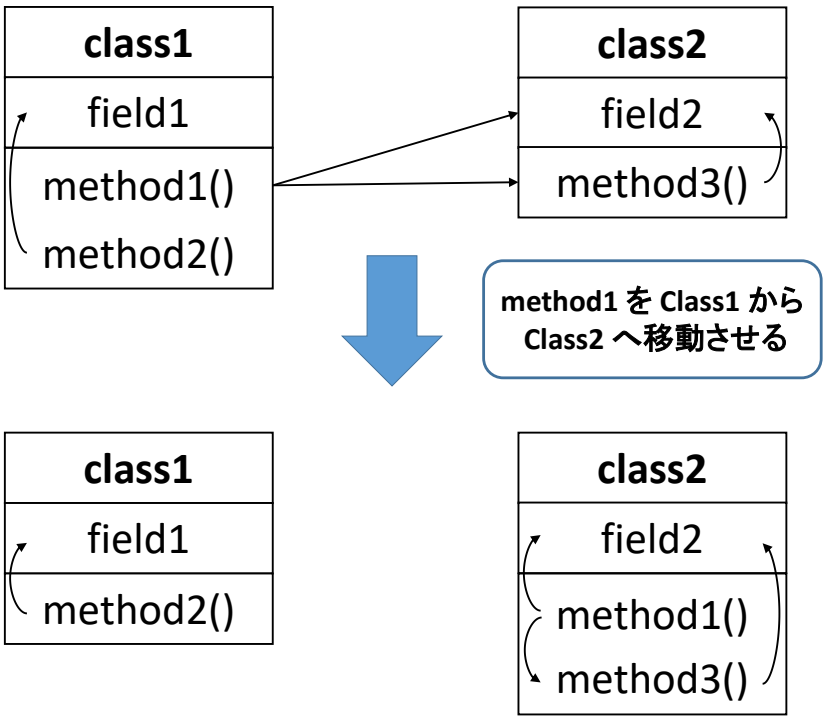

図 1: メソッド移動リファクタリングの例

図 1 を用いて，メソッド移動リファクタリングの例を説明する．class1 クラスに実装されている method1 メソッドは，所属クラスのいずれの属性も使用していないが，class2 クラスの field2 フィールドと method3 メソッドを参照している．また， class1 クラスの method2 メソッドは，所属クラスの field1 フィールドを，class2 クラスの method3 メソッドは，所属クラスの field2 フィールドを，それぞれ参照している．このような状況で， class2 クラスの method3 メソッドを修正するような保守作業を行う必要があるとき，開発者はメソッドの機能を理解するためであったり，修正が周囲に与えるの影響を確認するために，class1 クラスに含まれる method1 メソッドの内容も確認する必要があり，修正作業の範囲が，複数のクラスになってしまうため，コストがかかってしまい，効率的ではない．そこで， method1 メソッドを class1 クラスから class2 クラスへと移動させるメソッド移動リファクタリングを実施すると，各クラスの凝集度が高いソフトウェアへと改善することができる．

2.2.2 リファクタリング活動の調査

リファクタリングは設計上の欠陥を取り除く活動として知られている．たとえば Suryanarayana ら [21] は，技術的負債によって生じるコードスメルとその解決方法としてのリファクタリングを整理している．

しかし，リファクタリングはコードスメルのようなソフトウェアの設計上の欠陥を取り除く目的以外にも実施されていることが報告されている．Bavota らは，リファクタリングとソフトウェア品質に関する調査を行った結果，実際に行われたリファクタリングのほとんどは，コードスメルを取り除いていないと報告した [1]．また，Cedrim らの調査でも同様に，多くの場合，リファクタリングの実施によってコードスメルは取り除かれておらず，むしろコードスメルを生み出しているような場合もあったと報告されている [3]．メソッド移動リファクタリングを対象に行われた Silva らの調査 [20] によると，実際のソフトウェア開発現場において実施されているメソッド移動リファクタリングは，メソッドをより適切なクラスに移動させるため，メソッドを再利用するため，クラス間の参照回数を減少させるためであったと報告している．

開発者は，自分が書いたコードや修正を行っているコードに対して，修正作業と並行したリファクタリングを実施している．Murphy らによるリファクタリング活動に関する調査では，リファクタリング活動を floss と root-canal に分類し，floss リファクタリングの方が root-canal リファクタリングよりも主に用いられている戦略であると結論付けている [15]．また，Hoque らの調査でも，修正作業と同時に行われている floss リファクタリングの方が，他の作業とは区別して行われる root-canal リファクタリングよりも，頻繁に行われていると述べている [9]．Orrú らのリファクタリングとソースコードの所有権に関する調査では，開発者は何度もコミットしているファイルに対して，リファクタリングを実施していることや，ソースコードは，修正を行った回数が多い開発者によって，リファクタリングが行われていることが報告されている [16]．

Rajlich [17] によると，修正作業の一部として実施されるリファクタリング活動は，ソフトウェア変更を容易にするようなソースコードの変更であるプレファクタリングと，変更後のソフトウェアのソースコードの可読性などを向上するポストファクタリングに分類される．本研究の提案手法は，メソッドなどの記述が完了した後に実行するポストファクタリングの支援という位置づけとなる．開発者の作業状況の考慮という観点は Sae-Lim ら [18] も導入しているが，この手法は未解決のバグ修正等に備えて修正すべきコードスメルを探すプレファクタリングの支援であり，本研究とは立ち位置が異なる．

2.2.3 メソッド移動リファクタリング候補推薦手法

メソッド移動リファクタリングを支援する手法として，リファクタリング候補を推薦する様々な手法が提案されている．これらの手法は，不適切なクラスに配置されたメソッドを検出し，それらのメソッドを配置することでソフトウェアの品質を現状よりも改善できるようなクラスを移動先の候補として推薦する．

Tsantalis らは，凝集度と結合度に基づいて，Feature Envy という設計上の欠陥を検出し，それを修正するようなメソッド移動リファクタリング候補を推薦するツール JDeodorant を提案している [5][22]．JDeodorant は，推薦するメソッド移動リファクタリングの候補がソフトウェアの振る舞いを保つかどうか，実施の前提条件の検証を行うことで，問題が起きないリファクタリング候補だけを提示する．しかし，リファクタリング候補推薦の際に，意味的な凝集度に関しては考慮していない．Bavota ら [7] は，メソッド間の構造的関係に加えて，メソッドの内容についての関係トピックモデルを導入した．同じ機能に関係したメソッドの集合を特定することで Feature Envy という設計上の欠陥を検出し，それを修正するようなメソッド移動リファクタリング候補を推薦するツール MethodBook として実現している．

Sales らは，メソッドの静的依存集合の類似度によって，不適切なクラスに実装されたメソッドを特定し，それを修正するようなメソッド移動リファクタリング候補を推薦する JMove というツールを提案している [19].

これらの手法は，ソフトウェアの品質を向上させる有用なリファクタリング候補を推薦するが，これらの手法では，開発者がソフトウェアに設計上の欠陥がありそうであると感じたときにツールを実行してもらう必要がある．また，推薦の対象がソフトウェア全体となるため，推薦されるメソッド移動リファクタリングの候補にはソフトウェア全体の様々なコードに関連したものが含まれることになる．個別のリファクタリングを実施するかどうかの判断には，そのコードに関連する設計の理解が求められるうえ，コードの設計に関する知識や理解がない開発者や経験の浅い開発者がその判断を行うと新たな欠陥を作りこむ可能性もあることから，実際にツールによって推薦されたメソッド移動リファクタリング候補をソフトウェアに対して実施するかどうかの判断を行うことは，開発者にとって非常に大きな負担となる．

開発者がソフトウェアの修正作業の一部としてリファクタリングを実施する場合，本節に記述した既存研究のツールを適用しても，出力結果のほとんどは修正作業や開発者自身とは無関係なコードに対するリファクタリングの推薦となってしまう．また，ある修正作業によって，新たに作られたメソッドや修正されたメソッドが，正しいクラスに実装されているか，依存関係に悪影響を及ぼしていないかを確認するためには，その修正作業ごとにツールを実行し，自分の作業に関係したクラスに関する推薦結果だけを閲覧する必要がある．

そこで，本研究では，既存のメソッド移動リファクタリング候補推薦ツールと，実際のリファクタリング戦略との不一致を解消するために，修正作業状況に基づいたメソッド移動リファクタリング候補の推薦手法を提案する．

2.3 リファクタリング検出ツール

本節では，既存研究で提案されているリファクタリング検出ツールである，UMLDiff[24]，Ref-finder[10]，RefactoringMiner[20][23] について説明する．

UMLDiff は 2 つのバージョンから，設計レベルの差分を抽出し，それらの差分をもとに適用されたリファクタリングを検出するツールである．UMLDiff では，入力として与えられた 2 つのバージョンのソースコードに対して，パッケージやクラス，フィールドな どの設計レベルの情報を抽出し，2 つのバージョン間での設計レベルの差分 (要素の追加，削除，名前変更など) を求める．このような設計レベルの差分から，名前の変更や要素の移動といった簡単なリファクタリングを検出することができる．その他の複雑なリファクタリングについても，設計レベルの差分をいくつか組み合わせることによって検出を行っている．UMLDiff は，32 個のリファクタリングパターンを検出することが可能である．

Ref-Finder は，リファクタリングパターンを論理規則で表現し，それらの規則と 2 つのバージョンから抽出したソースコードレベルの差分を用いて，リファクタリングを検出し ている．Ref-Finder も UMLDiff と同様に，2 バージョン間の差分を抽出して使用するが， Ref-Finder が抽出する情報は，条件分岐やゲッターメソッドとフィールドの関係なども含まれており，UMLDiff より詳細である．また，2 つのメソッドの本体がどの程度類似しているかなどの情報も計算している．そして，リファクタリングパターンを表す論理規則を用いて，その規則を満たすソースコードを推論することによってリ ファクタリングの検出を行っている．Ref-Finder は，63 個のリファクタリングパターンを検出することが可能である．

上記の 2 つのリファクタリング検出ツールは，多くのリファクタリングパターンに対応しており，実験によって高精度でリファクタリングが検出できることが示されている．その一方で，これらのツールは 2 つのバージョン間でのリファクタリング検出を目的としたものであり，リポジトリからリファクタリングが実施されたコミットの検出には適していない．

RefactoringMiner は，オブジェクト指向モデルの差分に基づく，UMLDiff のアルゴリズムを軽量化したものを採用し，Benjamin ら [2] の提案するリファクタリングパターンを表す論理規則を用いて，その規則を満たすソースコードを推論することによってリファクタリングの検出を行っている．RefactoringMiner は，11 個のリファクタリングパターンを検出することが可能である．また，RefactoringMiner は，リポジトリからリファクタリングが実施されたコミットを特定することが可能であるため，本手法ではこのツールを採用した．

3 提案ツール

本研究では，開発者の作業状況を反映してメソッド移動リファクタリング候補を推薦，実行するツールを提案する．このツールは，Eclipse プラグインであり，Change task context based Java REFactoring RECommendation を省略して，*c-JRefRec* と名付けた．図 2 は，*c-JRefRec* の概要を示す図である．*c-JRefRec* は，修正前のソースコードに対し，静的依存解析を行い，依存関係グラフを生成し，それを基に，構造的メトリクスを計算する．修正作業が行われると，修正後のソースコードに対し，同様の操作を行い，修正後の構造的メトリクスを計算する．また，修正前後の依存関係グラフから，修正作業に関連するクラス，メソッドの特定も行う．これらの情報を基に，修正作業に関連するクラスの構造的メトリクスの増減値を表示することで，修正作業によるソフトウェアの構造への影響を可視化する．さらに，*c-JRefRec* は，修正後のソースコードに対し潜在意味解析を行い，クラス，メソッドの特徴ベクトルを生成し，それを基に，メソッドとクラス間の意味的類似度を計算する．構造的メトリクスと意味的メトリクスの 2 つの観点に基づき，修正作業に関連するメソッドに対して，メソッド移動リファクタリング候補を特定し，推薦を行う．このように，*c-JRefRec* は，以下の 2 つの機能を Eclipse に追加する．

1. 修正作業に関連するクラスの構造メトリクス表示機能

   我々が定義するクラスレベルの 4 つの構造的メトリクスが，修正作業によってどれだけ変化したのかを表示することで，リファクタリング活動の必要性の判断材料を開発者に提供する．

2. メソッド移動リファクタリング候補推薦機能

   修正されたコードに関連するメソッド移動リファクタリング候補を推薦，実行することで，保守作業に関連したコードの品質を高く維持することを支援する．

3.1 修正作業に関連するクラスの構造的メトリクス表示機能

この機能は，我々が定義した構造的メトリクスによって，ソフトウェア品質を可視化する機能である．開発者が，ソフトウェア品質を意識しながら保守作業を行い，構造的メトリクスを向上させるようなメソッド移動リファクタリングを実施することで，ソフトウェア保守のコストを低下させることが目的である．

3.1.1 静的依存解析

本手法では，ソースコードの構造的な依存関係を自動的に計算するために，静的依存解析を用いている．具体的には，提案ツールは Eclipse Java Developer Tools の上に構築されて

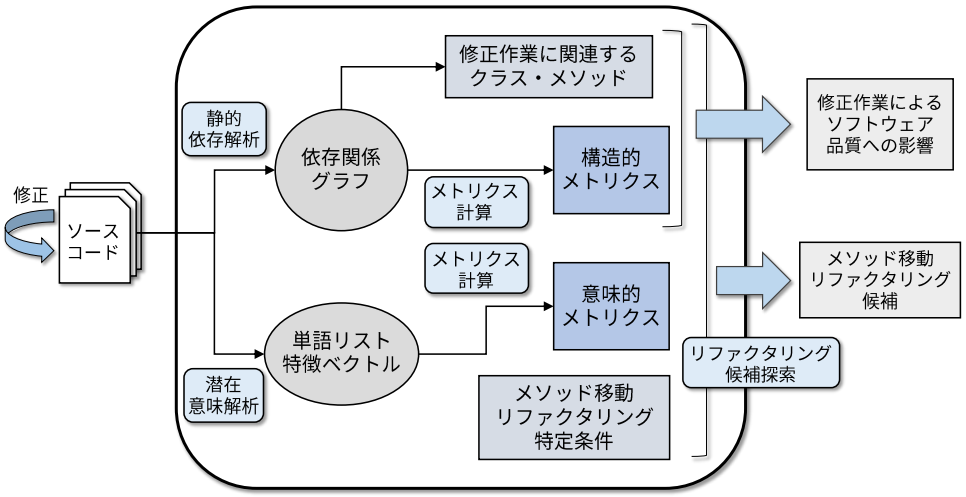

図 2: ツール概要

おり，Eclipse 上でツールが起動されると，JDT が提供する AST Parser を使用して，編集中の Java ソフトウェアのクラスやメソッド間の関係の解析を実行し，依存関係グラフを構築する．本研究では，メソッド呼び出し，もしくは，フィールドアクセスが存在するメソッドやフィールド，クラス間の関係を，依存関係と定義する．依存関係グラフは，ソフトウェアのフィールドとメソッドを頂点とし，依存関係のある頂点間に対して，参照元の頂点から参照先の頂点に対する有向辺を持つグラフである．また，各有向辺は，そのメソッドの呼び出し文やそのフィールドへのアクセス文が記述された文の数だけ重みを持つ．すなわち，依存関係グラフはソフトウェア全体に対するメソッド呼び出しとフィールドアクセスの情報を保持している．ツールは，起動された時点のソフトウェアの状態のグラフと，開発者がソースファイルを修正し，保存したときに，自動的に更新される最新の状態のグラフの 2 つを持ち，この 2 つのグラフの差から，修正作業による依存関係への影響を計算する．

図 3 は，本研究の静的依存解析において，ソースコードから生成される依存関係グラフの例を示す図である．class1 クラスの method1 メソッドは，class2 クラスの field2 フィールドを 1 回参照しており， method2 メソッドを 2 回参照しているので，method1 メソッドから field2 フィールドへ重み 1 の有向辺と，method1 メソッドから method2 メソッドへ重み 2 の

有向辺が生成される．class2 クラスの method2 メソッドと，class3 クラスの method3 メソッドは，class1 クラスの method4 メソッドを，1 回参照しているので，method2 メソッドから method4 メソッドへ重み 1 の有向辺と，method3 メソッドから method4 メソッドへ重み 1 の有向辺が生成される．

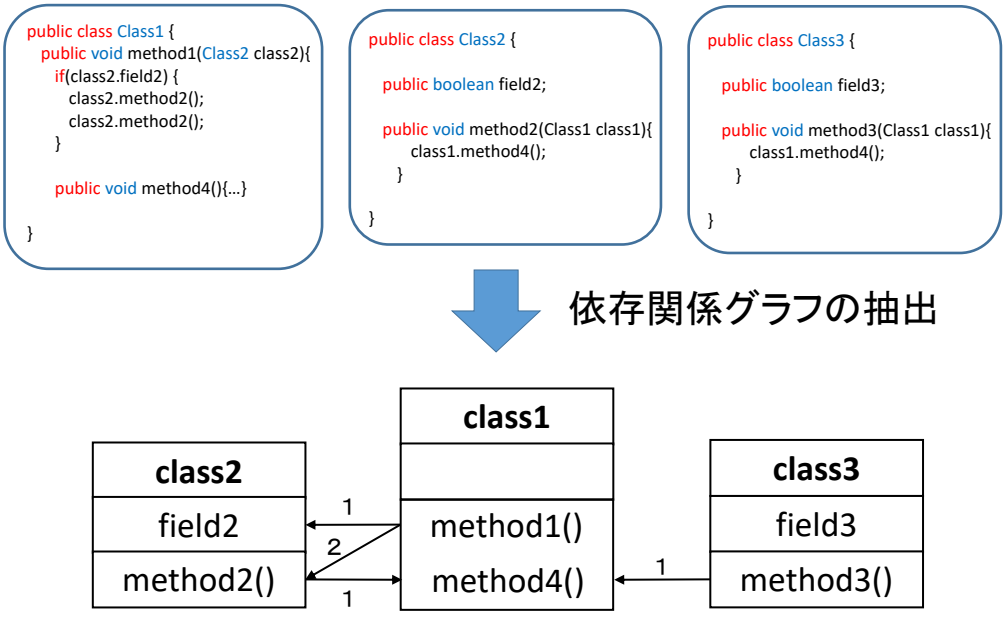

図 3: ソースコードから生成される依存関係グラフの例

3.1.2 定義した構造的メトリクス

2.1.1 節で述べたように，ソフトウェア品質を可視化するメトリクスは，数多く提案されているが，本手法では，メソッド移動リファクタリング候補を実施するかどうかの判断材料としても有用であるような，凝集度や結合度に関するクラスレベルの構造的メトリクスを新たに定義した．

- *methods*(C)

  クラス C に含まれる，いずれかのクラスと依存関係があるメソッドの数を示す．この値が大きいほど、そのクラスの責務は大きいことを意味する．

- *edges*(C)

  クラス C に含まれるメンバが，他のクラスを参照する文の数と，他のクラスに参照される文の数の総和を示す．この値が大きいほど，クラスの凝集度が低いことを意味する．

- *clients*(*C*)

  クラス *C* のいずれかのメンバを参照している，クラス *C* 以外のクラスの数を示す．この値が大きいほど，クラスの凝集度は低いことを意味する．

- *dependents*(*C*)

  クラス *C* に含まれるメソッドによって参照されているメンバを含んでいる，クラス *C* 以外のクラスの数を示す．この値が大きいほど，クラスの凝集度は低いことを意味する．

図 4 に示されている依存関係グラフの例を用いて，これらのメトリクスの計算方法を具体的に説明する．まず，methods(class1) とは，class1 クラスに含まれているメソッドの内，依存関係を持つメソッドの数であるが，条件を満たすメソッドは，modifiedMethod1 メソッドだけであるため，methods(class1) は 1 となる．edges(class1) とは，class1　クラスに含まれるメンバと他のクラスの参照回数の総和であるが，条件を満たすメンバは，modifiedMethod1 メソッドだけである．modifiedMethod1 メソッドは，class4 クラスの method4 メソッドを 3 回，class5 クラスの field5 フィールドを 1 回参照しており，class2 クラスの method2 メソッドに 1 回参照されているため，合計して 5 回の参照回数を持つ．したがって，edges(class1) は，5 となる．clients(class1) とは，class1 クラスに含まれるメンバを参照する，class1 クラス以外のクラスの数であるが，class2 クラスの method2 メソッドだけであるため，clients(class1) は 1 となる．dependents(class1) とは，class1 クラスに含まれるメソッドによって参照されているメンバを含む，class1 クラス以外のクラスの数であるが，class4 クラスと class5 クラスを modifiedMethod1 メソッドが参照しているため，dependents(class1) は，2 となる．参照本数，依存関係のクラスといった情報によって，どのクラスが凝集度の低いクラスであるのか，開発者は理解することができる．

3.1.3　修正作業に関連するクラス

本手法では，開発者にとって，関心度や優先度が高いと思われる，修正作業に関連するクラスのみに対して，メトリクスを表示する．以下の 3 つの条件のいずれかに含まれるクラスを，修正作業に関連するクラスとして定義した．

1. 修正されたメソッドを含むクラス
2. 修正されたメソッドを参照するメンバを含むクラス **(client** クラス**)**
3. 修正されたメソッドに参照されるメンバを含むクラス **(dependent** クラス**)**

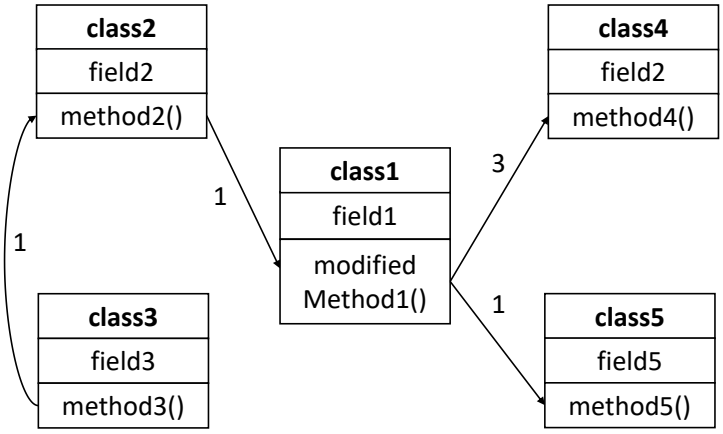

図 4: 依存関係グラフの例

図 4 においては，class1 クラスが修正されたクラス，class2 クラスが client クラス，class4 クラスと class5 クラスが dependent クラスとなる．本手法では，これらのクラスを，修正作業に関連したクラスとみなし，これらのクラスに関して上記のメトリクスを表示する．

3.1.4 クラス状態ビュー

修正作業に関連するクラスの構造的メトリクス表示機能は，クラス状態ビュー (*Class State View*) の形で提供される．クラス状態ビューは，最新のグラフと初期のグラフを比較することで，修正作業によりクラスの依存関係がどれだけ変化しているかを表示する．

図 5 に，ある修正作業を行った時点でのビューの表示状態の例を示す．ビューは表形式で，1 行に 1 つのクラス名を表示しており，各列にはクラスの名前，ツールを起動した時点におけるメトリクスの値と，修正作業によるメトリクスの初期値からの増減値を表示している．この表では，4 つのメトリクスに対し，増加しているものは赤色で，減少しているものは青色で，強調表示されており，増減値の総和が大きい順にソートされている．ソースコードが修正されて保存される度にこの表が自動的に更新されるため，開発者は，自分が行った修正作業による依存関係への影響を知ることができる．

3.2 メソッド移動リファクタリング候補推薦機能

この機能は，本手法のメインである，メソッド移動リファクタリング候補を推薦する機能であり，リファクタリング候補ビュー (*Refactoring Candidates View*) の形で提供される．このビューは，メソッド移動リファクタリング候補として移動するメソッド名，移動先クラス

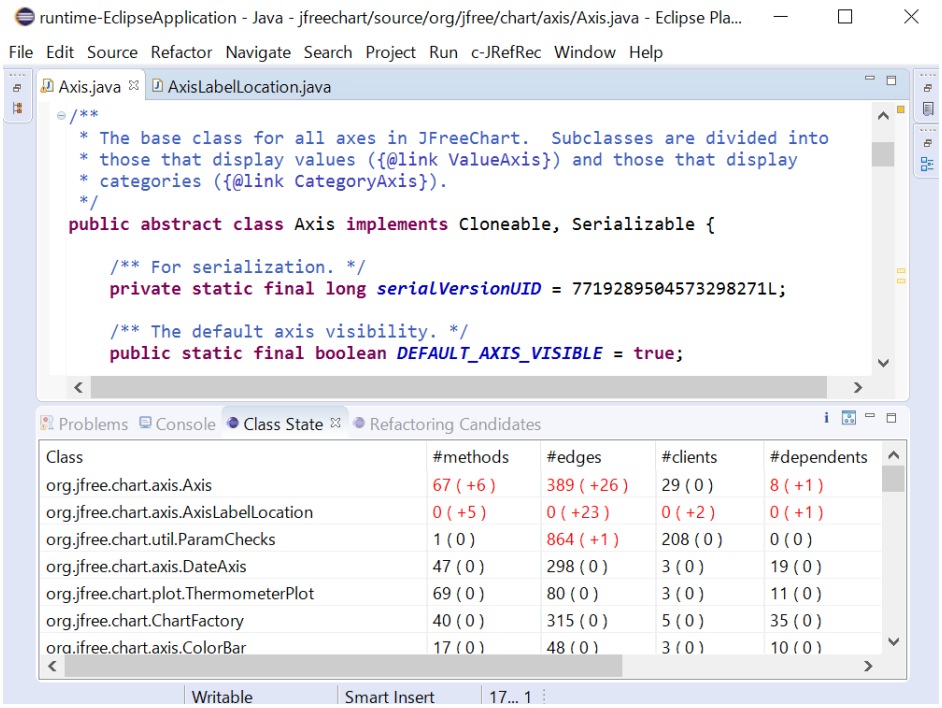

図 5: クラス状態ビュー

名を表示し，また，そのリファクタリングの実施の判断基準となるメトリクスを表示する．そのメトリクスには，3.1.2 節で説明した構造的メトリクスを基に計算されるものの他に，新たにソースコードに対し，潜在意味解析を行って求める意味的メトリクスがある．

3.2.1 潜在意味解析

メソッド移動リファクタリング候補を特定する際に，*c-JRefRec* は，ソフトウェアの構造情報に加えて，クラスやメソッド内に使われている単語を抽出，比較することで得られる意味的類似度も使用する．意味的類似度については，最新のソースコードの状態から計算された値のみを使用する．

まず，修正作業に関連する各クラスと，メソッド移動リファクタリング候補を探索するメソッド内に含まれる単語を抽出し，単語リストをそれぞれ生成する．それらの単語リストから，リスト内における単語の出現頻度と，全リスト内における単語の出現確立を用いて，各単語リストの各単語に対して，重みを計算し，それを各コードの特徴ベクトルとする．メソッドの特徴ベクトルとクラスの特徴ベクトルを用いて，コサイン類似度を計算し，メソッドとクラス間の意味的類似度 (Semantic Simirality) とする．

3.2.2 メソッド移動リファクタリング候補特定条件

まず，3.1.2 節で説明した構造的メトリクスを基に計算される，メソッド移動リファクタリング候補特定条件に用いる新たな式を以下に定義する．

- $\Delta edges(C, R)$

  メソッド移動リファクタリング R を適用することで，追加・減少するクラス C の辺の数．

- $\Delta edges(R)$

  メソッド移動リファクタリング R を適用することで，追加・減少するソフトウェア全体の辺の数．

- $\Delta clients(C, R)$

  メソッド移動リファクタリング R を適用することで，追加・減少するクラス C のクライアントクラスの数．

- $\Delta clients(R)$

  メソッド移動リファクタリング R を適用することで，追加・減少するソフトウェア全体のクライアントクラスの数．

- $\Delta dependents(C, R)$

  メソッド移動リファクタリング R を適用することで，追加・減少するクラス C のディペンデントクラスの数．

- $\Delta dependents(R)$

  メソッド移動リファクタリング R を適用することで，追加・減少するソフトウェア全体のディペンデントクラスの数．

あるメソッド m について，クラス C_{orig} からクラス C_{target} へのメソッド移動リファクタリング R が候補として提示されるのは，依存関係の変化に関するメトリクスと意味的類似度の合計 4 つのメトリクスが以下のいずれか条件を満たした場合である．

$$\Delta edges(R) > 0 \quad \vee$$

$$\Delta clients(R) + \Delta dependents(R) > 0 \quad \vee$$

$$SemanticSimilarity(m, C_{orig}) < SemanticSimilarity(m, C_{target})$$

クラス間における参照回数の減少，依存関係クラスの減少，メソッドとクラス間における意味的類似度の向上の 3 つの条件のいずれかを満たす場合にリファクタリング候補とすることで，現在欠陥となっているメソッドだけでなく，今後欠陥となりそうなメソッドに対しても，リファクタリング候補として推薦する．それぞれの条件の計算方法について，以下で説明する．

まず，$\Delta edges(R)$ とは，メソッド移動リファクタリング R を実施した際の，ソフトウェア全体のクラス間参照回数の増減値である．具体的には，$\Delta edges(C_{orig}, R) + \Delta edges(C_{target}, R)$ である．次に，$\Delta clients(R) + \Delta dependents(R)$ とは，メソッド移動リファクタリング R を実施した際の，ソフトウェア全体の依存関係にあるクラス数の増減値である．具体的には，ソフトウェア全体に $C_1, C_2, ..., C_n$ までの n 個のクラスが存在したとすると，$\sum_{i=1}^{n}\{\Delta clients(C_i, R) + dependents(C_i, R)\}$ である．

3.2.3 修正作業に関連するメソッド探索

本手法では，修正作業に関連するメソッドに対する，メソッド移動リファクタリング候補のみを推薦する．まず，修正作業に関連するメソッドを，以下の 3 つの条件のいずれかを満たすメソッドであると定義した．

1. 修正されたメソッド
2. 修正されたメソッドを参照するメソッド
3. 修正されたメソッドに参照されるメソッド

図 4 においては，modifiedMethod1 メソッドが修正されたメソッド，method2 メソッドが修正されたメソッドを参照するメソッド，method4 メソッドと method5 メソッドが修正されたメソッドに参照されるメソッドとなる．

これらのメソッドに対して，メソッド移動リファクタリング候補特定条件を満たすメソッドがないか探索するが，修正されたメソッドを参照するメソッドと，修正されたメソッドに参照されるメソッドに関しては，その修正されたメソッドにメソッド移動リファクタリング候補がない場合のみ，その修正されたメソッドが実装されているクラスに移動するような，メソッド移動リファクタリング候補が条件を満たすかどうか確認する．図 4 においては，まず modifiedMethod1 メソッドを class2 クラスと class4 クラス，class5 クラスのいずれかに移動させるべきか探索する．もし，メソッド移動リファクタリング候補特定条件を満たすものが存在すれば，method2 メソッドと method4 メソッド，method5 メソッドについては，探索を行わない．もし，modifiedMethod1 メソッドにメソッド移動リファクタリング候補特定条件を満たすものがない場合のみ，method2 メソッドと method4 メソッド，method5 メソッド

を，class1 クラスに移動させるメソッド移動リファクタリング候補があるかを探索する．これは，修正されたメソッドが不適切なクラスに実装されている場合に，関連するメソッドをそのクラスに移動させることは，ソフトウェアの品質をより悪化させる可能性が高いと思われるからである．

3.2.4 メソッド移動リファクタリング候補推薦ビュー

メソッド移動リファクタリング候補推薦機能は，リファクタリング候補ビュー (*Refactoring Candidates View*) という形で提供される．リファクタリング候補ビューは，依存関係グラフによる静的依存解析と潜在意味解析に基づいて，メソッド移動リファクタリング候補推薦機能のために表示する．図 6 に例を示すが，このビューでは，リファクタリング候補として，移動させるメソッドの名前と移動先クラスの名前だけでなく，移動元クラスと移動先クラスの構造的メトリクスにどのような影響があるかも表示する．そのため，開発者は，リファクタリングを適用するかの判断が容易に行えると考えている．

開発者が修正作業に関連したリファクタリング候補を知りたい場合，ビュー右上のボタンをクリックすることで，リファクタリング候補の推薦を要求することができる．また，クラス名をダブルクリックすることで，そのクラスに関するリファクタリング候補だけの推薦を要求することもできる．

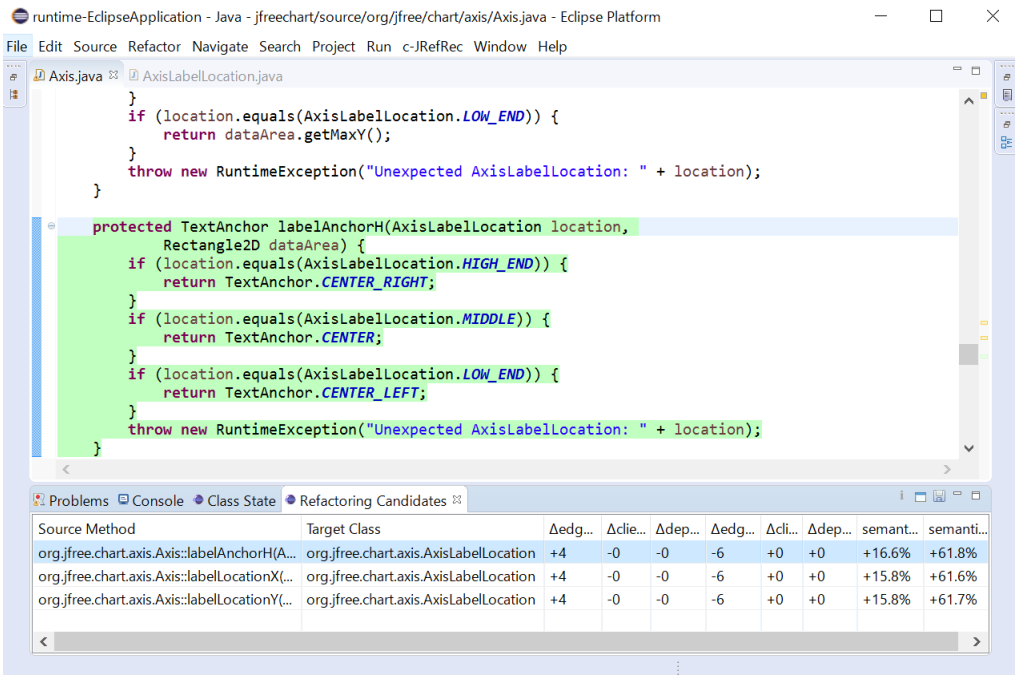

図 6: メソッド移動リファクタリング候補ビュー

4 利用例

有名なオープンソースソフトウェアである *JFreeChart* を題材に *c-JRefRec* の使い方を紹介する．具体的には，github に記録されているコミット ID c7e8c72 内の修正作業に対して，このツールが，どのように機能するかを示す．このコミット内では，*org.jfree.chart.axis.AxisLabelLocation* クラスが作られ，*org.jfree.chart.axis.Axis* クラス内で，フィールドとメソッドの追加と修正が行われている．

4.1 修正作業における *c-JRefRec*

c-JRefRec を起動させると，まず，クラス状態ビューが表示される．クラス状態ビューは，ソースコードが変更されて，保存される度に，自動的に更新される．図 5 は，そのコミットにおける修正がソースコードに実施された後のビューの状態である．この修正作業で新しく作られた *AxisLabelLocation* クラスは，メソッドの数が 0 から 5 へと増加し，辺の数は 0 から 23 へと増加し，クライアントクラスの数は 0 から 2 に増加し，依存クラスの数は 0 から 1 に増加していることがわかる．同様に，*Axis* クラスにも修正が行われた結果，メソッドの数が 67 から 6 だけ増加し，辺の数が 389 から 26 だけ増加し，クライアントクラスの数は 29 から変わっておらず，依存クラスの数は 8 から 1 だけ増加していることがわかる．この修正作業は，*Axis* クラスと *AxisLabelLocation* クラスに対して行われているが，新しく作られた *AxisLabelLocation* クラスに対して，*Axis* クラスは，様々なクラスと依存関係を持つ非常に大きいクラスとなっている．そのため，*Axis* クラスに新しく追加されたメソッドのうち，*AxisLabelLocation* クラスと関連したものを，*AxisLabelLocation* クラスに移動させることで，より凝集度の高いソフトウェアへとすることができると考えられる．このようにして，開発者は，ソースコードの各クラスの結合度や凝集度に関する影響を分析することができる．

4.2 リファクタリング作業における *c-JRefRec*

そこで，開発者が，メソッド移動リファクタリング候補があるかを知りたい場合，図 5 のクラス状態ビュー右上に設置されているボタンをクリックすると，*c-JRefRec* は，その修正作業に関連するメソッド移動リファクタリング候補を推薦する．また，*AxisLabelLocation* クラスに関連するリファクタリング候補だけを知りたい場合，クラス状態ビューにおける *AxisLabelLocation* クラスの行をダブルクリックすることで求めることもできる．結果として，図 6 で示されるようなリファクタリング候補の一覧が自動的に表示される．このビューでは，移動させるべきメソッド名，移動先クラスの名前とリファクタリング後の影響を示すメトリクスの値が表示される．ビューの各列には，$\Delta edges(R, C_{orig})$，$\Delta Clients(R, C_{orig})$，$\Delta Dependents(R, C_{orig})$，

$\Delta edges(R, C_{target})$，$\Delta Clients(R, C_{target})$，$\Delta Dependents(R, C_{target})$，$SemanticSimirality(m, C_{orig})$，$SemanticSimirality(m, C_{target})$ の順に表示されている．

図 6 の一番上のメソッド移動リファクタリング候補について説明すると，*Axis* クラスの *labelAnchorH*() メソッドを，*AxisLabelLocation* クラスへ移動させるリファクタリングを推薦している．*Axis* クラスと *labelAnchorH*() メソッド間で，辺が 4 本増えるけれども，*AxisLabelLocation* クラスと *labelAnchorH*() メソッド間で，辺が 6 本減るため，ソフトウェア全体では，辺が 2 本減ることがわかる．このメソッド移動リファクタリングによって，クライアントクラスの数とディペンデントクラスの数は，変わらない．しかし，*labelAnchorH*() メソッドは，*Axis* クラスと意味的類似度が 16.6% しかないが，*AxisLabelLocation* クラスとは，61.8% もあることがわかる．すなわち，クラス間の参照本数は減少し，*labelAnchorH*() メソッドの実装されているクラスとの意味的類似度は向上することがわかる．このようにして，各リファクタリングを実施することによって，どのようなメリットがあるか，依存関係の観点と，semantic な観点から，開発者は確認することができる．また，これらのメトリクスの値は，3.2.2 節で定義した条件を満たすので，*c-JRefRec* は，*labelAnchoer*() メソッドを *Axis* クラスから *AxisLabelLocation* クラスへ移動させるリファクタリングを候補として，推薦している．

各リファクタリング候補は，その行をダブルクリックすることで，移動させるべきと判断されたメソッドがハイライトされる．また，その候補を選択した状態で，リファクタリング候補ビュー右上に設置されているリファクタリング実施ボタンをクリックすることで，このメソッド移動リファクタリングを実施する場合のソースコードの変更内容がプレビューされる．このプレビュー画面から，実施ボタンをクリックすることで自動的に，メソッド移動リファクタリングが実施される．

図 4.2 は，図 5 の状態から，*labelAnchor*() メソッドを，*Axis* クラスから *AxisLabelLocation* クラスへと移動させた後の，*Class State View* である．*Axis* クラスは，修正作業によりメソッドの数は，+6 に増加していたが，メソッドを 1 つ他のクラスへと移動させたため，+5 へと減少している．辺の数は，修正作業により+26 に増加していたが，リファクタリングによって，+24 へと減少していることがわかる．同様に，*AxisLabelLocation* クラスは，メソッドの数が+5 から+6 へと増加しているが，辺の数は，+23 から+21 へと 2 だけ減少したことがわかる．

このようにして，開発者は開発作業と並行して，修正作業に関連したコードに対するリファクタリング活動を実施することができる．

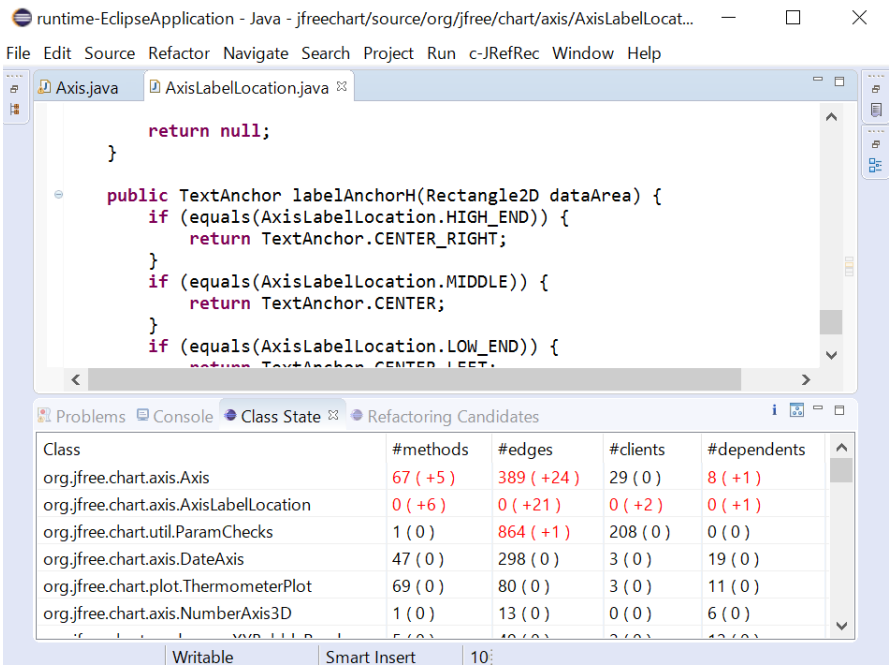

図 7: リファクタリング実施後のクラス状態ビュー

5 適用実験

提案手法の有効性を評価するために，2 つの適用実験を行い，本手法によって推薦されるメソッド移動リファクタリング候補を評価する．本実験では，対象として 3 個の Java 言語で記述されたオープンソースソフトウェアを選択した．実験対象ソフトウェアの一覧を表 1 に示す．JMeter と JFreeChart は 1 つ目の実験のみに使用し，実験 2 では JUnit だけを使用する．また，表 1 中のクラス数は，最新コミットにおける数である．

1 つ目の実験は，ソフトウェアに対して作為的に設計上の欠陥を作り出し，その欠陥を修正するようなメソッド移動リファクタリング候補を推薦できるか調査する．2 つ目の実験は，ソフトウェアの変更履歴で実際に実施されたメソッド移動リファクタリングを，過去の修正作業から推薦できるかを調査する．どちらの実験においても，コミットにおけるソースコードの差分に基づいて実際の修正作業を再現し，その作業状況に基づいて推薦されるメソッド移動リファクタリング候補について評価を行う．Java 言語で記述されたオープンソースソフトウェアに対して，提案手法を適用した．本章では，実験方法と実験結果，妥当性への脅威について述べる．

表 1: 実験対象ソフトウェア

ソフトウェア名	コミット数	クラス数
JUnit	2160	1068
JMeter	13711	1055
JFreeChart	3408	943

5.1 実験手順

まず，1 つ目の適用実験の準備として，各ソフトウェア変更履歴のうち無作為に 10 個のコミットを選択し，それぞれのコミットにおいて修正作業に関連する 2 つのメソッドを無作為に他のクラスへと移動させる．本実験では，これらのメソッドを設計上の欠陥とし，このメソッドを元のクラスに戻すメソッド移動リファクタリングを本手法によって特定できるか，またその精度は既存手法と比較してどうであるかを調査する．本手法は修正作業状況に基づくため，ソースコードをそのコミットの修正作業が行われる前の状態に戻し，ソースコードをその修正作業後のソースコードに書き換えることで，修正作業が行われた状況を再現する．その再現された修正作業に基づき，本手法によるメソッド移動リファクタリング候補を推薦する．その結果得られたメソッド移動リファクタリング候補の中に，移動したメソッドを元のクラスに戻すようなメソッド移動リファクタリングが含まれているかを調べる．また，修正作業終了後のソフトウェアに対して，既存手法である JDeodorant を適用し，得られたメソッド移動リファクタリング候補の中に，移動したメソッドを元のクラスに戻すようなメソッド移動リファクタリングが含まれているか，同じく調べる．この際，移動したメソッドをいずれかのクラスへ移動させるようなメソッド移動リファクタリング候補を推薦できた場合をメソッドの特定とし，実際に移動したクラスへ移動させるメソッド移動リファクタリングを特定できた場合をメソッド移動リファクタリングの特定とする．3 つのソフトウェアに対してそれぞれ 20 個，全体で 60 個の移動させるべきメソッドを作り出している．

次に 2 つ目の実験の準備として，リファクタリング検出ツールである RefactoringMiner を用いて，実験対象ソフトウェアの履歴から実際のソフトウェア保守でメソッド移動リファクタリングが実施された事例の収集を行う．この検出されたメソッド移動リファクタリングを特定できるか調査する．本手法は修正作業状況に基づくため，特定されたメソッド移動リファクタリングの中で移動したメソッドが，メソッド移動リファクタリングを実施される直前に修正されたコミットを特定し，その修正作業が行われた状況を再現する．その再現された修正作業に基づき，本手法によるメソッド移動リファクタリング候補を推薦する．その結果得られたメソッド移動リファクタリング候補の中に，実際に実施されたメソッド移動リ

ファクタリングが含まれているかを調べることによって評価を行う．1 つ目の実験と同様に，メソッドの特定とメソッド移動リファクタリングの特定の 2 つの観点から評価する．

5.2 実験結果

1 つ目の実験の結果には，評価尺度として，precision と recall を用いる．Precision は適合率とも呼ばれ，予測結果のうちどの程度が正解であったかを評価する尺度である．本実験において precision は，ツールによって推薦されたメソッド移動リファクタリング候補のうち移動させたメソッドに対する候補の割合と，ツールによって推薦されたメソッド移動リファクタリング候補のうち移動させたメソッドを元のクラスに戻す候補の割合を用いる．Recall は，再現率とも呼ばれ，データセット中に存在する正解集合のうち，どの程度が正しく予測されたかを評価する尺度である．本実験において recall は，移動させたメソッド移動リファクタリングのうち移動させたメソッドに対する候補の割合と，移動させたメソッド移動リファクタリングのうち移動させたメソッドを元のクラスに戻す候補の割合を用いる．

1 つ目の実験の結果を図 8，図 9，図 10，図 11 に示す．本手法は，メソッド特定に関して precision が 0.48 と recall が 0.73 であり，JDeodorant の precision が 0.38 と recall が 0.25 と比較して，より優れた結果が得られた．また，メソッド移動リファクタリングの特定に関しても，precision が 0.42 と recall が 0.68 であり，JDeodorant の precision が 0.38 と recall が 0.25 と比較して，より優れた結果が得られた．

次に，2 つ目の実験の結果を説明する．まず，RefactoringMiner によって，JUnit の変更履歴の中から特定された実際に実施されたメソッド移動リファクタリング数は 17 個であった．この 17 個のメソッド移動リファクタリングのうち，14 個のメソッドを候補として特定し，いずれかのクラスへのメソッド移動リファクタリング候補を推薦した．そのうち 11 個のメソッドに対して，実際に実施されたメソッド移動リファクタリングを推薦することができた．

表 2: 実際のメソッド移動リファクタリング特定実験結果

ソフトウェア名	検出数	メソッド特定数	メソッド移動リファクタリング特定数
JUnit	17	14	11

5.2.1 妥当性への脅威

本節では，本実験における妥当性への脅威について述べる．

まず，本実験の評価において，候補として推薦すべきとしたメソッド移動リファクタリングについてである．1 つ目の実験では，無作為にメソッドを移動させることによって設計

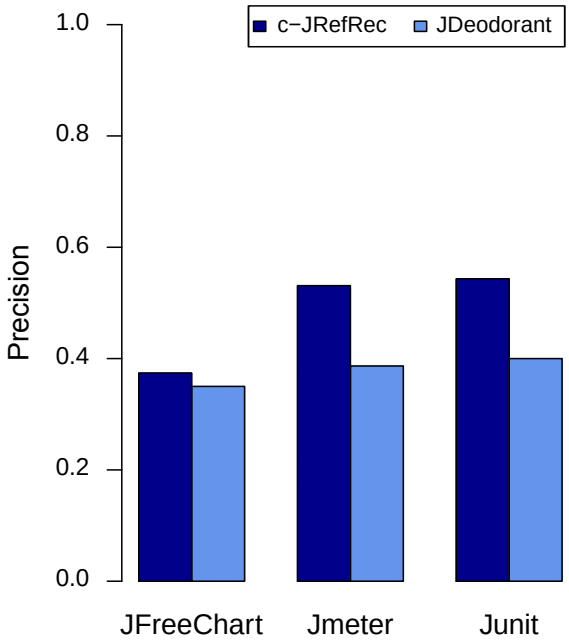

図 8: メソッド特定の precision

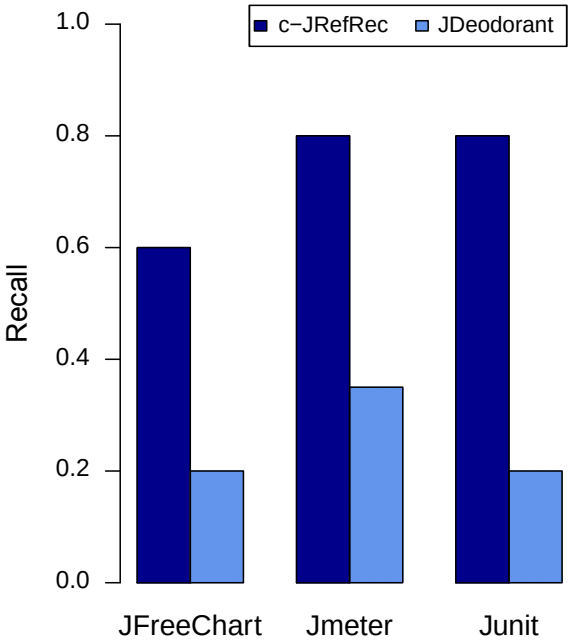

図 9: メソッド特定の recall

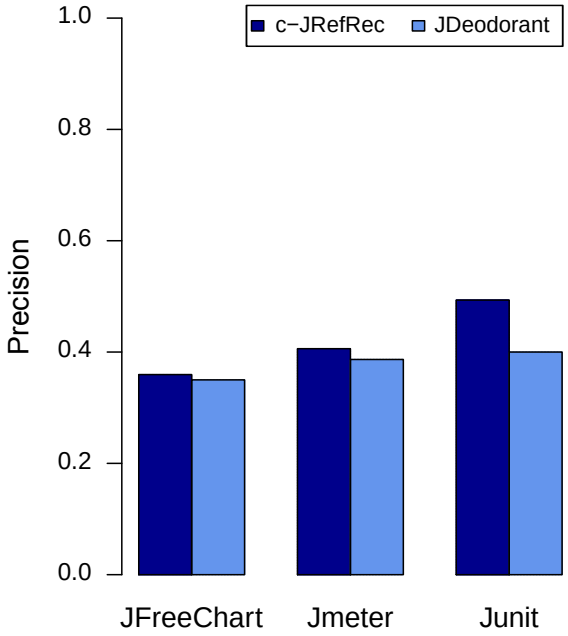

図 10: メソッド移動リファクタリング特定の precision

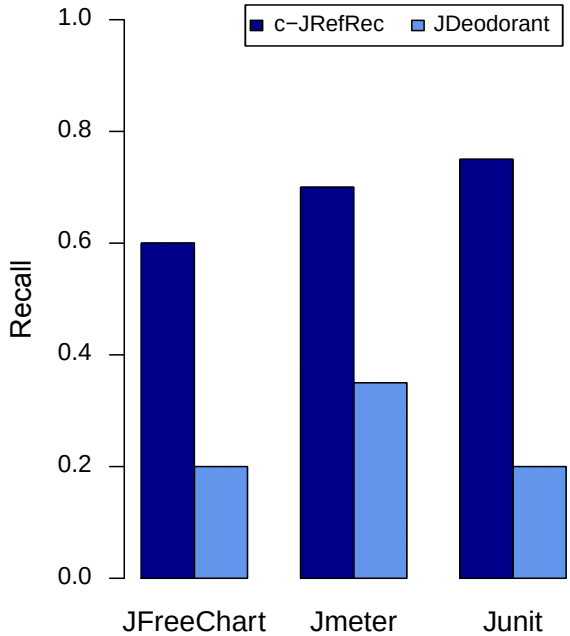

図 11: メソッド移動リファクタリング特定の recall

上の欠陥を作り出し，それを見つけ出すべきメソッド移動リファクタリングとしたが，そのメソッドが設計上の欠陥となっていない可能性がある．また，2 つ目の実験においては，RefactoringMiner というリファクタリング検出ツールを用いて，実際のメソッド移動リファクタリングを検出し，それを見つけるべきメソッド移動リファクタリングとしたが，検出されたもの以外にも実施されたメソッド移動リファクタリングがあり，それを見逃している可能性もある．このような見つけるべきメソッド移動リファクタリング集合の誤差によって，実験の結果に影響があるかもしれない．

次に，実験対象のソフトウェアについてである．1 つ目の実験では 3 つのオープンソースソフトウェアを，2 つ目の実験ではそのうち 1 つを使用したが，今回の実験結果が対象のソフトウェアに限定されたものである可能性がある．今後，さらに他のソフトウェアに対して適用実験を行い，結果がどのように変化するか確認する必要がある．

6 まとめ

本研究では，開発者の修正作業状況に基づいて，修正作業と並行して実施するリファクタリングを支援するためのメソッド移動リファクタリング候補推薦手法を提案し，*c-JRefRec* として実装した．具体的には，修正によって生じた依存関係の増減を構造的メトリクスの増減として表示することで，リファクタリング活動の必要性の判断材料を提供し，修正作業に関連したコードだけをメソッド移動リファクタリング候補の探索対象とすることで，開発者が推薦されたリファクタリングを実際に実施するかどうかの判断を容易に行える，実施によって欠陥を作りこむ可能性の低い候補を提示する．

実験として，3 つのオープンソースソフトウェアに対して，手法を適用し評価を行った．評価の結果，実際に行われたメソッド移動リファクタリングのうち 64.7%を提案手法によって，特定することができた．

今後の課題は，*c-JRefRec* が開発者の作業に与える影響の調査である．開発者に *c-JRefRec* が有効な状態で開発作業を行ったとき，リファクタリングがどれだけ実施されるかを調査し，リファクタリング活動を支援できていることを示せれば，ソフトウェア保守において提案手法がより有効であることを示すことができる．

謝辞

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 教授には，適切なご指導やご助言を頂いたことで，本論文を完成させることができました．井上教授の研究者としての素養や専門知識だけではなく，学生一人ひとりを温かく支援してくださる姿や人間性も尊敬しており，多くの面で学ばさせて頂き，成長することができました．2 年間の研究生活を井上教授のもとで送らせて頂き，本論文を執筆できたことに厚く御礼申し上げます．

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 准教授には，特に発表練習の際に貴重な意見を頂きました．発表および研究の改善点を指摘していただき，それらをご参考にさせて頂くことで本研究を遂行することができました．多くのご指導を頂きました 松下准教授に心から御礼申し上げます．

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 石尾 隆 助教には，直接のご指導を頂きました．本研究の方向性や問題点など，多くのご指導を頂いただけではなく，論文執筆から発表まで，多くのお力添えを頂きました．本論文を執筆できたことは，石尾助教のご指導によるものであると確信しております．心から御礼申し上げます．

I am deep grateful to Associate Professor Ali Ouni belonging to United Arab Emirates University Department of Computer Science and Software Engineering for his valuable comments and helpful criticism which have helped guide and shape the development of this thesis.

最後に，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研究室の皆様に，心から感謝致します．

参考文献

[1] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, Vol. 107, pp. 1 – 14, 2015.

[2] 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 '11, pp. 53–62, New York, NY, USA, 2011. ACM.

[3] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, pp. 73–82. ACM, 2016.

[4] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, Vol. 20, No. 6, pp. 476–493, 1994.

[5] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *IEEE International Conference on Software Maintenance (ICSM)*, pp. 519–520, Oct 2007.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] Bavota Gabriele, Oliveto Rocco, Gethers Malcom, Poshyvanyk Denys, and Lucia Andrea De. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, Vol. 40, No. 7, pp. 671–694, 2014.

[8] Maurice Howard Halstead. *Elements of software science*, Vol. 7. Elsevier New York, 1977.

[9] M. Iftekharul Hoque, V. Nag Ranga, A. Reddy Pedditi, R. Srinath, M. A. Ahsan Rana, M. Eftakhairul Islam, and A. Somani. An Empirical Study on Refactoring Activity. *ArXiv e-prints*, December 2014.

[10] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pp. 371–372, New York, NY, USA, 2010. ACM.

[11] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer Berlin Heidelberg., 2006.

[12] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[13] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, No. 4, pp. 308–320, 1976.

[14] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, Vol. 25, pp. 38–44, 2008.

[15] 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)*, pp. 287–297. IEEE Computer Society, 2009.

[16] Matteo Orrú and Michele Marchesi. A case study on the relationship between code ownership and refactoring activities in a java software system. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, WETSoM '16, pp. 43–49, New York, NY, USA, 2016. ACM.

[17] Vaclav Rajlich. *Software engineering: The current practice*. CRC Press, 2011.

[18] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. Context-based code smells prioritization for prefactoring. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pp. 1–10. IEEE, 2016.

[19] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. Recommending move method refactorings using dependency sets. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 232–241. IEEE, 2013.

[20] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 858–870. ACM, 2016.

[21] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier Science, 2014.

[22] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, Vol. 35, No. 3, pp. 347–367, 2009.

[23] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pp. 132–146, Riverton, NJ, USA, 2013. IBM Corp.

[24] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pp. 263–274, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Aiko Yamashita, Marco Zanoni, Francesca Arcelli Fontana, and Bartosz Walter. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 121–130. IEEE, 2015.