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

信学技報
TECHNICAL REPORT OF IEICE.

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推薦に向けて

氏原　直哉† 　アリ　ウニ† 　石尾　　隆† 　井上　克郎†

† 大阪大学情報科学研究科　〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †{naoya-u,ali,ishio,inoue}@ist.osaka-u.ac.jp

あらまし　メソッド移動リファクタリングはソフトウェアの品質を向上させるために役立つ技術である．本研究では，Java プログラムの修正作業によって発生したクラス間の依存関係の変化を可視化し，不必要な依存関係を取り除く可能性の高いメソッド移動リファクタリングを推薦，実行するツールを開発した．
キーワード　メソッド移動リファクタリング，　*floss* リファクタリング，　ソフトウェア品質，　ソフトウェア保守

Towards Move Method Refactoring Recommendation using Change Task Context

Naoya UJIHARA†, Ali OUNI†, Takashi ISHIO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita-shi, 565–0871 Japan
E-mail: †{naoya-u,ali,ishio,inoue}@ist.osaka-u.ac.jp

Abstract　Move Method Refactoring is an effective technique to improve software quality. In this research, we have developed a refactoring tool that visualizes changes of dependence relation among classes caused by a maintenance task and makes a recommendation of move method refactroings to remove unnecessary dependence relations.
Key words　move method refactoring, floss refactoring, software quality, software maintenance

1. ま え が き

リファクタリングとは，ソフトウェアの外部的振る舞いを保ったままで，内部の構造を改善していく作業を指す [5]．大規模システムのソースコードは，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 [6]．Floss リファクタリングは，欠陥修正や機能追加など他の変更作業の途中に，高品質なコードを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，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 らは，リファクタリングのための時間を設けるのではなく，floss リファクタリングを推奨している [5]．

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 ら [5] 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，また，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と，クラス間の結合度を減少させることができ，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にもつながる．

ソフトウェア開発において，特に大規模ソフトウェアの開発では，移動すべきメソッドを開発者が自力で特定することは難しい．そのため，メソッド移動リファクタリング候補を自動的に特定するための手法が多数提案されている．しかし，それらの多くは，ソフトウェア全体から設計上の欠陥を特定し，その欠陥を修正するようなるメソッド移動リファクタリングを推薦

する手法である．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，推薦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．

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，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．

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シナリオを紹介し，5. 章でまとめと今後の課題について述べる．

2. 背　　景

リファクタリングに関する研究としては，リファクタリング作業を支援する手法の研究と，開発者が実施したリファクタリングの検出および分析に関する研究が行われている．

2.1 メソッド移動リファクタリング候補推薦

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 [4], [15]．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 [2] 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 [13]．

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．

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 [14] 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 [11] 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 [7]．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 [8]．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 [9]．

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，2.1 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．

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．Rajlich [10] 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 [12] 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．

3. ツール概要

このツールは，Change task context based Java REFactoring RECommendation を省略して，*c-JRefRec* と名付けた．*c-JRefRec* が持つ機能は，修正作業に関するクラスの状態表示機能と，メソッド移動リファクタリング候補推薦機能である．1 つ目は，私たちが定義した，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提供する機能である．2 つ目は，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することで，開発者の floss リファクタリングを支援する機能である．

3.1 ツール設計

c-JRefRec は，Eclipse plugin として実装した．Eclipse 上で修正を行うソースコードを開いた状態で，このツールを起動すると，クラスやメソッド間の関係を解析するために，Eclipse Java Developer Tool の AST Parser を用いて，有向依存グラフを生成する．このグラフは，メソッド呼び出しとフィールドアクセスを辺の情報とし，メソッドとクラスが頂点である．すなわち，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持っている．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クラス間の関係を，依存関係と定義する．また，開発者がソースファイルを修正し，保存したときに，このグラフは自動的に更新される．このツールは，ツール起動時のソフトウェアの状態のグラフと，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．

さらに，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だけでなく，クラスやメソッド内に使われている単語を抽出することで，クラスやメソッド間の意味的類似度を求める semantic 解析も行う．

c-JRefRec は，*Class State View* と *Refactoring Candidates View* という，2 つのビューを持つ．*Class State View* 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し，1 つ目の修正作業に関するクラスの状態表示機能のために表示する．*Refactoring Candidates View* は，有向依存グラフによる依存関係の解析と semantic 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．それぞれのビューについて，以下で説明する．

3.2 クラス状態ビュー

このビューは，クラスの結合度や凝集度を示すための以下の 4 つのメトリクスを表示する．

- $methods(C)$ は，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．
- $edges(C)$ は，クラス C に含まれるメンバーが持つ，他のクラスとの辺の数を示す．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．
- $clients(C)$ は，クラス C のメンバを使うクラスの数を示す．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．
- $dependents(C)$ は，C に含まれるメソッドによって使われるメンバを含んでいるクラスの数を示す．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．

修正作業の間に，修正されたメソッドを含んでいるクラス，そのメソッドのクライアントクラス，ディペンデントクラスのことを修正作業に関連したクラスとし，それらのクラスに関して，これらのメトリクスを表示する．図 1 で示されているように，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と，修正作業によるメトリクスの初期値からの増減値が表示される．このリストは，4 つのメトリクスの増減値の総和が大きい順にソートされている．このビューは，ソースコードが修正されて，保存される度に，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．

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．

3.3 リファクタリング候補ビュー

このビューは，メソッド移動リファクタリング候補と，4 つのメトリクスに関する値を表示する．

メトリクスは，以下のように定義した．

- $\Delta edges(R, C)$ は，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 の辺の数である．
- $\Delta clients(R, C)$ は，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

ラスの数である．

- $\Delta dependents(R, C)$ は，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である．

グラフ全体に含まれる辺の数が減少するか，依存関係にあるクラス数が減少する場合，そのメソッド移動リファクタリング R を実施すべきであると仮定する．

また，semantic なメトリクスとして，メソッド m と，クラス C の意味的類似度を，$SS(m, C) = cosine(tf - idf(m), tf - idf(C))$ として，tf-idf ベクトルを用いて，計算する．C_1 クラスに含まれる m が，C_1 クラスよりも C_2 クラスとの意味的類似度が大きい場合，メソッド m は C_2 に移動させるべきであると仮定する．

これらのメトリクスを用いて，メソッド移動リファクタリング候補と判断するための条件は，以下の通りである．$\Delta edges(R, C_{original}) + \Delta edges(R, C_{target}) > 0$ OR $\Delta clients(R, C_{original}) + \Delta clients(R, C_{target})$ $+ \Delta dependents(R, C_{original}) + \Delta dependents(R, C_{target}) > 0$ OR $SS(m, C_{original}) < SS(m, C_{target})$

参照本数の減少，依存関係クラスの減少，意味的類似度の向上，いずれかの条件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．

図 2 のように，このビューは，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，このリファクタリングによって，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．

4. ユーザシナリオ

c-JRefRec の使い方を紹介するために，リファクタリング対象のソフトウェアとして，有名なオープンソースソフトウェアである *JFreeChart* を選んだ．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．

c-JRefRec を起動させると，まず，*Class State View* が表示される．*Class State View* 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 1 は，そのコミットにおける修正作業が行われた後の *Class State View* の状態である．このコミット内で，*AxisLabelLocation* クラスは，新しく作られたクラスであるため，メソッドの数は 0 から 5 だけ増加し，辺の数は 0 から 23 だけ増加し，クライアントクラスの数は 0 から 2 だけ増加し，依存クラスの数は 0 から 1 だけ増加していることがわかる．同様に，*Axis* クラスも修正されているため，メソッドの数は，67 から 6 だけ増加し，辺の数は，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．

そこで，開発者が，*AxisLabelLocation* クラスにリファクタリング候補があるかを知りたり場合，その行をダブルクリックすることで求めることができる．結果として，図 2 で示されるような，*Refactoring Candidates View* が自動的に表示される．このビューは，移動させるべきメソッド名，移動先クラスの名前とメトリクスの値が表示される．このメトリクスは，$\Delta edges(R, C_{original})$，$\Delta Clients(R, C_{original})$，$\Delta Dependents(R, C_{original}$，$\Delta edges(R, C_{target})$，$\Delta Clients(R, C_{target})$，$\Delta Dependents(R, C_{target})$ の順に表示されている．図 2 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，**??**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．

図 3 は，図 1 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

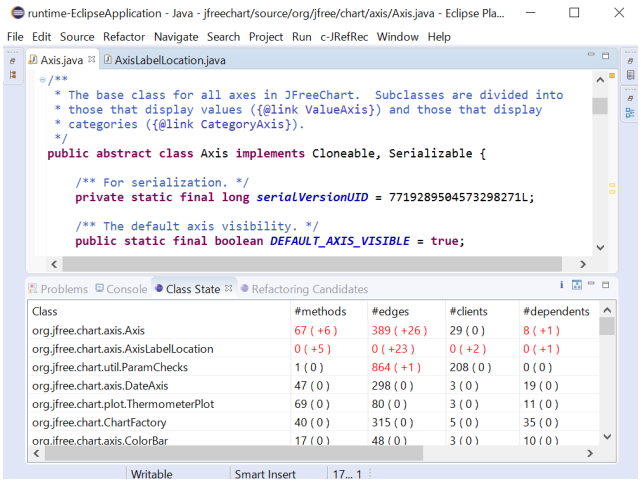

図 1　Class State View (リファクタリング前)

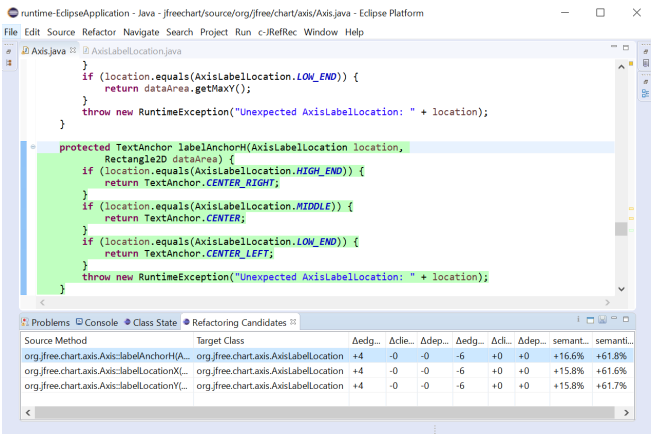

図 2　Refactoring Candidates View

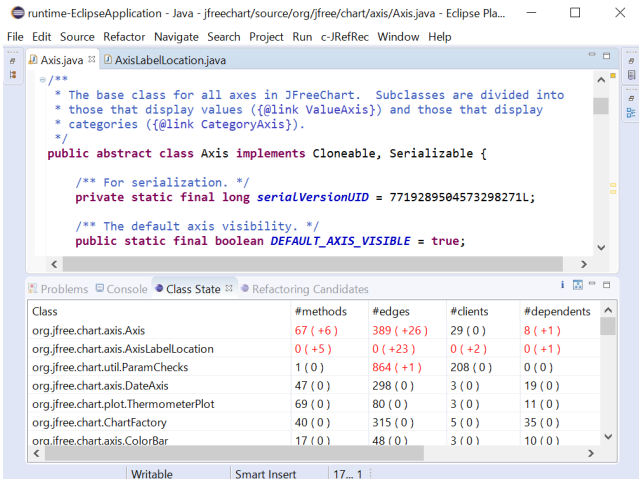

図 3　Class State View (リファクタリング後)

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．

5. まとめと今後の課題

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．

今後の課題は，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の評価と，それが開発者の作業に与える影響の調査である．リファクタリング候補の評価としては，実際のソフトウェア開発で実施された floss リファクタリングにおけるメソッド移動リファクタリングを，提案ツールを用いてリファクタリング実施前の修正作業から推薦できるかを調査する予定である．また，人為的にメソッドを不適切な場所に移動し，それらを元の場所に戻せるかによっても評価を行うことを考えている．開発者の作業に与える影響については，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する予定である．

謝辞　本研究は JSPS 科研費 JP25220003, JP26280021, JP15H02683 の助成を受けたものです．

文　献

[1] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba. An Experimental Investigation on the Innate Relationship between Quality and Refactoring. Journal of Systems and Software, vol.107, pp.1–14, 2015.

[2] G. Bavota, R. Oliveto, M. Gathers, D. Poshyvanyk, and A. D. Lucia. Methodbook: Recommending Move Method Refactoring via Relational Topic Models. IEEE Transactions on Software Engineering, vol.40, issue.7, pp.671–694, 2014.

[3] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi. Does refactoring improve software structural quality? A longitudinal study of 25 project. In Proc. of SBES, pp.73–82, 2016.

[4] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells." In Proc. of ICSM, pp.519–520, 2007.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts.

Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co, Inc, 1999.

[6] E. Murphy-Hill and A. P. Black. Refactoring Tools: Fitness for Purpose. IEEE Software, vol.25, issue.5, pp.38–44, Sept.-Oct. 2008.

[7] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. IEEE Transactions on Software Engineering, vol.38, issue.1, pp.5–18, 2012.

[8] M. I. Hoque, V. N. Ranga, A. R. Pedditi, R. Srinath, M. A. A. Rana, M. E. Islam, and A. Somani. An Empirical Study on Refactoring Activity. ACM Computing Research Repository abs/1412.6359, 2014.

[9] M. Orrú and M. Marchesi. A case study on the relationship between code ownership and refactoring activities in a Java software system. In Proc. of WETSom, pp.43–49, 2016.

[10] V. Rajlich. Software Engineering: The Current Practice. CRC Press, 2011.

[11] D. Silva, N. Tsantalis, and M. T. Valente. Why We Refactor? Confessions of GitHub Contributors. In Proc. of FSE, pp.858–870, 2016.

[12] N. Sae-Lim, S. Hayashi, and M. Saeki. Context-Based Code Smells Prioritization for Prefactoring. In Proc. of ICPC, pp.1–10, 2016.

[13] V. Sales, R.Terra, L.F. Miranda, and M. T. Valente. Recommending Move Method Refactorings using Dependency Sets. In Proc. of WCRE, pp.232–241, 2013.

[14] G. Suryanarayana, G. Samarthyam, and T. Sharma. Refactoring for Software Design Smells. Morgan Kaufmann, 2015.

[15] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering, vol.35, issue. 3, pp.347–367, 2009.