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.

動的解析を用いたコード移動リファクタリングの支援

木村 秀平† 肥後 芳樹† 井垣 宏† 楠本 真二†

† 大阪大学大学院情報科学研究科
E-mail: †{s-kimura,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

あらまし あるモジュールの凝集度を高め結合度を下げるために，Move Method リファクタリングや Extract Method リファクタリングといった，コード片を適切な位置に移動させるリファクタリングが行われる．このようなリファクタリングの候補を検出するために，従来の手法では，静的解析を用いてソースコードから導き出したクラスやメソッドの関係を利用していた．しかし，静的解析を用いる手法は，繰り返し回数，動的束縛や実際に実行される経路などの実行時に定まる情報を反映することができないため，検出できないリファクタリング候補があると考えられる．本論文では，動的解析を用いてリファクタリング候補の検出を支援する手法を提案する．実験の結果，提案手法を用いて実際のソフトウェアから複数の有効なリファクタリング候補を検出することができた．
キーワード リファクタリング，動的解析，メソッド移動リファクタリング，メソッド抽出リファクタリング，ソフトウェア保守

Supporting Code Moving Refactoring Based on Dynamic Analysis

Shuhei KIMURA†, Yoshiki HIGO†, Hiroshi IGAKI†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
E-mail: †{s-kimura,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

Abstract Software refactoring is the process of reorganizing the internal structure of code while preserving the external behavior. It is performed to reduce coupling and increase cohesion of modules. Move Method and Extract Method refactorings are a kind of refactoring techniques that move a piece of code to another place. Previous approaches detect refactoring candidates based on static analysis. However, these approaches cannot detect some refactoring opportunities, because static analysis cannot obtain runtime information. This paper proposes a technique to detect refactoring candidates based on dynamic analysis. Our technique detects refactoring candidates by using method traces. The evaluation of our technique showed that our technique can detect appropriate refactoring candidates from actual software.
Key words refactoring, dynamic analysis, Move Method refactoring, Extract Method refactoring, software maintenance

1. はじめに

大規模ソフトウェアの開発プロジェクトでは，複数のプログラマがさまざまな要求を満たすようにソフトウェアを開発，修正していくため，ソフトウェアの設計品質を高く保つことは難しい [1]．このような問題に対処するために，ソフトウェアの設計品質を向上させる技術の 1 つである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振る舞いを保ちながら，内部構造を改善する技術のことである [2]．これまでに多くのリファクタリング候補を検出する手法がツールとして実装され，リファクタリングの支援が行われてきたが，それらは主に静的解析を用いてリファクタリング候補を検出していた [3]〜[5]．

一方，動的解析とは，プログラムを実行し得られた履歴を解析する手法である．プログラムを実行することで，ソースコードの解析では得られない，実際に起こった動作を取得することができる．例えば，それぞれのメソッドが呼び出された回数，動的束縛の結果や，プログラム実行中に呼び出されたメソッドの系列などを簡単に知ることができる．

本論文では，メソッド呼び出し履歴を解析することによって，リファクタリング候補の検出を支援する手法を提案する．この手法を用いることにより，静的解析では検出することが困難だったリファクタリング候補を検出することができると考えられる．本手法の特徴は，メソッド呼び出し履歴を用いることに

より，メソッド同士がプログラムの実行中にどのように関連しあっているか，ということに着目している点である．

本手法では，メソッド呼び出し履歴を用いてリファクタリング候補の検出を支援するために，フェイズ分割の情報を用いる．フェイズとは，ある機能を実現するためのメソッド呼び出し列であり，プログラムの実行は複数のフェイズから成り立っている [6]．フェイズ分割の情報は，メソッド呼び出し履歴を適切な位置で区切ることによって得られる．この分割は，繰り返し回数や動的束縛などの実行時の状態を反映している．あるメソッドが他のメソッドと異なるフェイズに属している場合，このメソッドは他とは異なった機能を実現するために呼び出されたと考えられる．そのため，このようなメソッドをリファクタリング候補として検出すればよい．ただし，メソッドの呼び出し情報は文字や数値で算出されるため，リファクタリング担当者がこの情報のみを利用して候補を検出するのは困難である．そのため，提案手法では，フェイズ単位でのメソッド呼び出し情報を色の濃淡に変換し可視化することにより，リファクタリング候補の検出を支援している．

本手法の有効性を確認するために，提案手法をツールとして実装し，実際のソフトウェアに対して実験を行った．実験対象には規模の異なる 2 つのソフトウェアを用意した．実験の結果，提案手法を用いることにより，それぞれのソフトウェアからリファクタリング候補を検出することができた．また，それらをリファクタリングすることにより，ソースコードの品質が向上することを開発者への聞き取りを行い確かめた．

以降，2 章では本研究の背景として既存研究と本研究の動機となった例を示す．3 章では，提案手法で用いる語とツールの説明を行い．4 章で提案手法について述べる．提案手法の評価と考察を 5 章にて行い，最後に 6 章で本論文のまとめと今後の課題について述べる．

2. 背　　景

2.1 既存研究

Tsantalis らは，ソースコードから Feature Envy となるコード片を検出し，メソッド移動リファクタリングの候補とする手法を提案している [5]．この手法では，Jaccard distance と呼ばれるフィールドやメソッド間の距離を定義し，リファクタリング候補となるクラスを抽出している．このように，静的解析では主にクラス，フィールドやメソッドなどの関連を求め，その情報を用いてリファクタリング候補を検出している．

しかし，静的解析では，プログラム実行中にどの要素が互いに影響しあっているかを考慮することができず，リファクタリングすべきコードを見落としてしまう可能性がある．

2.2 Motivating Example

図 1 は，FRISC [7] というコードクローン検出ツールのコールグラフである．なお，同じクラスに属するメソッドは点線で囲っている．このようなグラフは静的解析を用いることで容易に取得でき，静的解析を用いたリファクタリング候補の検出ではよく用いられる．

しかし，静的解析では実行時の情報を得られない．例えば，

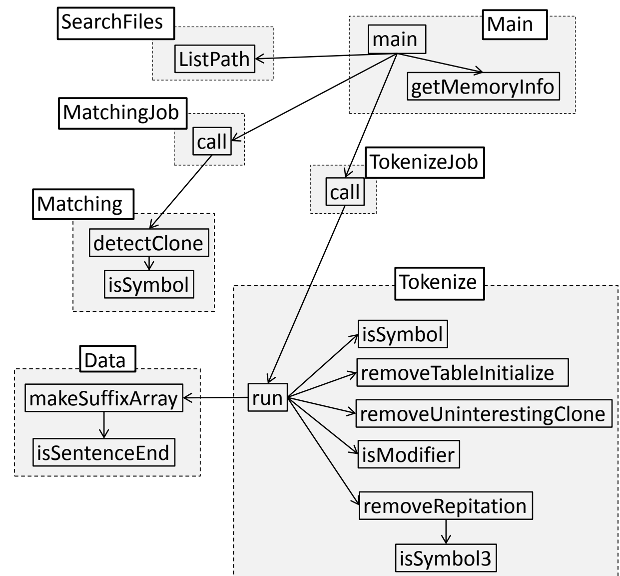

図 1: FRISC のコールグラフ

“Tokenize.isSymbol” と “Tokenize.isModifier” という 2 つのメソッドは，プログラムの実行中に組として繰り返し呼び出されており，強い関連を持っている．しかし，このような情報は静的解析では取得できず，メソッド呼び出しの関連を表したコールグラフにも現れない．

さらに，静的解析では，動的束縛の結果呼び出されるメソッドの情報を取得することができない．図 2 は，FRISC のソースコードの中で動的束縛を用いている箇所である．このコードの中で，変数 v は，FileSort クラスのインスタンスが代入されるのか，ArraySort クラスのインスタンスが代入されるのかは判明しない．そのため，静的解析では v から FileSort クラスのメソッドが呼び出されるのか，ArraySort クラスのメソッドが呼び出されるのかが確定できない．結果として，このような情報が得られず，既存手法ではリファクタリング候補を見逃す可能性がある．

このようなリファクタリング候補を検出するため，本研究では動的解析を用いている．この手法を用いることにより，静的解析では検出することが難しいリファクタリング候補を検出することができる．

本研究では以下の問題を調査する．

Research Question 動的解析を用いて有効なリファクタリング候補を検出することができるか？

3. 準　　備

3.1 コード移動リファクタリング

複数のモジュールに分かれているコードがある 1 つの機能を実現している場合，または 1 つのモジュール内に複数の機能を実現するためのコードが存在する場合，コード片を移動させ機能ごとにまとめるリファクタリングを行うことにより，モジュールの凝集度を高め結合度を下げることができる．このようなリファクタリングの代表的な例として，メソッド移動リファクタリングが挙げられる．あるメソッドが不適切なクラスで宣言さ

```
class SearchFiles{
    static class FileSort implements Comparator<File>{
        public int compare(File src, File target){
            int diff = src.getName().
                        compareTo(target.getName());
            return diff;
        }
    }
 ...
    private static void ListPath(File dir) {
        // dynamic dispatch
        if( target.isFile() )
            s = new FileSort()
        else
            s = new ArraySort()
 ...
        Arrays.sort(fs, s);
    }
}
```

図 2: 動的束縛を用いているコードの例

れている場合，メソッド移動リファクタリングを用いて適切なクラスに移動することにより，クラスが持つ責任を明確にすることができる．また，異なった処理を行っているコードを切り出し，別のメソッドやクラスを作成することで，機能ごとにまとめるリファクタリングも存在する．例として，メソッド抽出リファクタリングやクラス抽出リファクタリングがある．

本手法では，メソッド移動リファクタリングやメソッド抽出リファクタリングなどと同様に，コード片を移動させるリファクタリングを支援する．コード片の単位は，メソッドやクラスである．本論文では，このようなリファクタリングをコード移動リファクタリングと呼ぶ．

3.2 Amida

動的解析とは，プログラムを実行しその実行履歴を解析する手法である．本研究では，実行履歴の一種であるメソッド呼び出し履歴を利用する．これを取得するためのツールとしてAmida を用いた [8], [9].

Amida は，Java プログラムの実行中にメソッド呼び出しを記録し，UML のシーケンス図として可視化するツールである．さらに，メソッド呼び出し履歴を複数のフェイズに分割する機能を備えている．フェイズとは，ある機能を実現するためのメソッド呼び出しの列である．異なるフェイズに属しているメソッドを検出することにより，異なる機能を実現するために用いられたメソッドを容易に特定することができる．なお，フェイズの分割数は Amida に与えるパラメータによって変動する．

4. 提案手法

4.1 手順

この章では，提案手法を用いてリファクタリング候補を検出する手順を説明する．図 3 は，提案手法の手順を表した図である．提案手法は実行可能 Jar ファイルとテストケースを入力として受け取り，コード移動リファクタリングの候補を出力として返す．

本手法では，動的解析を用いるために，事前にテストケースを用意する必要がある．多くのテストケースを用意するほど，複数の異なる出力を得ることができるが，テストケース数の増加に伴い解析に掛かる時間も増える．

以降，図 3 に示した各処理について詳細に述べる．

(Step 1) メソッド呼び出し履歴とフェイズ分割情報の取得

Amida を用いて，対象ソフトウェアを与えられたテストケースで実行し，メソッド呼び出し履歴を得る．また，得られたメソッド呼び出し履歴に対しフェイズ分割を行う．フェイズ分割の結果は，フェイズ分割情報として出力される．

なお，Amida の実行時にパラメータを与え，分割数を調整することができる．ソフトウェアの規模や機能の数に応じてパラメータを設定することが望ましいが，今回の研究では，フェイズ数が 5-10 程度となるように設定した．これは，分割数が少なすぎるとメソッドごとの属するフェイズに差がつかず，分割数が多すぎると可視化した際にメソッドごとの比較が行いにくくなってしまうためである．

(Step 2) フェイズ内で，メソッドごとの呼び出し回数のカウント

メソッド呼び出し履歴には，同じメソッドの呼び出しが繰り返し現れる．このステップでは，Step 1 で得られたメソッド呼び出し履歴とフェイズ分割情報を用いて，各フェイズにおける各メソッドの呼び出し回数を数える．この処理によって得られる呼び出し回数のデータを，以降”フェイズ単位でのメソッド呼び出し情報”と呼ぶ．

(Step 3) フェイズ単位でのメソッド呼び出し情報の可視化

このステップでは，Step 2 で得られたフェイズ単位でのメソッド呼び出し情報を可視化する．可視化の方法は以下のとおりである．

あるクラスを c とする．c に含まれるメソッドを $m_0, m_1, ..., m_n$ とする．また，フェイズ分割数を N ，フェイズ a における m_i の呼び出し回数を p_{i_a} とし，それらの合計を P_i とする．

$$P_i = \sum_{k=0}^{a} p_{i_k} \quad (1)$$

次に，メソッド呼び出しの回数を色の濃淡を表す値に変換する．この値は，0 から 1 の範囲で，数値が大きくなるほど濃い色とする．すなわち，値が 1 であれば完全な黒，0 であれば完全な白であり，0.5 は灰色である．フェイズ a での m_i の呼び出し回数を p_{i_a} とする時，p_{i_a} に対する色の濃淡 (p'_{i_a}) は以下の式で与えられる．

$$p'_{i_a} = \frac{p_{i_a}}{P_i} \quad (2)$$

すなわち， m_i の全呼び出しのうち，フェイズ a で呼び出された回数の割合を色の濃淡として表している．

最後に，$p'_{i_0}, p'_{i_1}, ..., p'_{i_N}$ のそれぞれの色で塗りつぶしたセルを，フェイズの番号順に一行に描画する．セルの色が濃いフェイズほど，他のフェイズに比べてそのフェイズ内で多く呼び出されたことを表す．この手順を，i を 0 から n へ 1 ずつ増やして繰り返すことにより，c に含まれる全てのメソッドを可視化する．

上記の手順により，図 4 のような，各行にメソッドの名前と

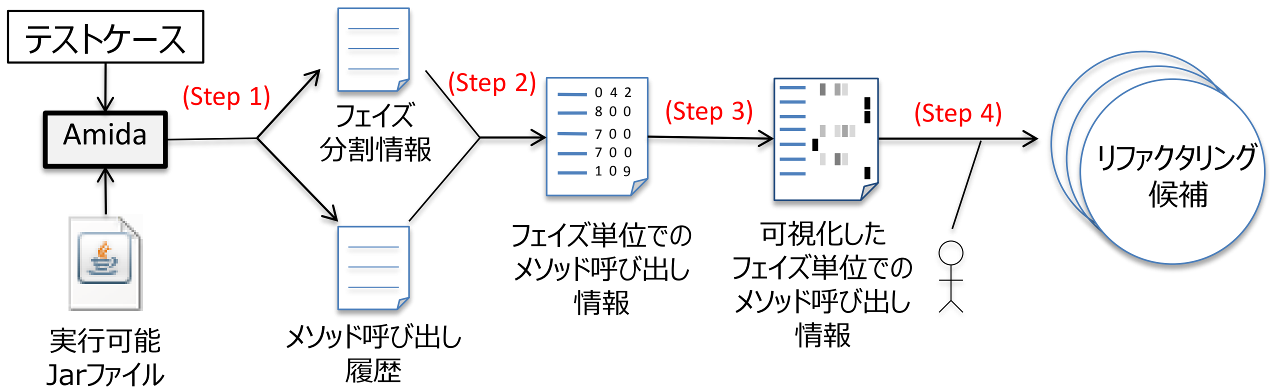

図 3: 提案手法の概要

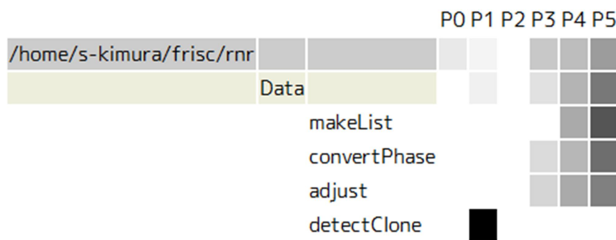

図 4: Step 3 の出力例

フェイズごとの色付けされたセルが存在する表示が得られる．この表示を，以降"可視化したフェイズ単位でのメソッド呼び出し情報"と呼ぶ．なお，クラス単位，パッケージ単位で同様の処理を行うことにより，より大きな単位での可視化が行える．図 4 では，メソッド，クラス，パッケージのそれぞれに対する可視化を行っており，図中の一番上の行は rnr というパッケージのフェイズ情報，2 行目は rnr.Data というクラスのフェイズ情報，3 行目以降は rnr.Data クラスに含まれるメソッドのフェイズ情報を表している．

(Step 4) リファクタリング候補の検出

このステップでは，Step 3 で得られた可視化したフェイズ単位でのメソッド呼び出し情報を用いて，リファクタリング担当者がリファクタリング候補の検出を行う．あるクラスに含まれるメソッドの，可視化したフェイズ単位でのメソッド呼び出し情報の中で，他のメソッドと異なった表示がされているメソッドをリファクタリング候補として検出する．

例えば，図 4 中の “detectClone” というメソッドは，P1 のセルの色のみが黒く，他のメソッドと異なった表示をしている．このようなメソッドをリファクタリング候補とする．

なお，プログラムの開始時に一度だけ呼び出される main メソッドはリファクタリング候補から除外している．なぜなら，メソッド呼び出し履歴の中で必ず 1 度しか現れず，かつ先頭付近のフェイズで呼び出されるため，多くの場合他のメソッドと異なる表示となるためである．

5. 実験と考察

5.1 実験の準備

提案手法の Step 2 と Step 3 の手順を自動化したツールを実装し，提案手法を FRISC [7] と MASU [10] という 2 種類のソフトウェアに対して適用した．FRISC は 2.2 でも例示した Java 言語で記述されたコードクローン検出ツールである．このソフトウェアは比較的小規模であり，8 つのファイル，846 行のソースコードから構成されている．一方，MASU は Java 言語で記述されたメトリクス計測ツールであり，最新バージョンは 1,153 ファイルと 133,045 行のソースコードからなる，比較的大規模なソフトウェアである．なお，実験にあたり，それぞれのツールの基本的な機能のみを実行するテストケースを用意した．

また，検出されたリファクタリング候補は，それぞれのソフトウェアの開発者に提示し適切かどうかの回答を得た．

5.2 実験結果

図 5 は FRISC に対する提案手法の適用結果である．図中の色を付けた 18 のメソッドをリファクタリング候補として検出した．また，図 6 は MASU に対する提案手法の適用結果の一部である．図中の色を付けたメソッドを含む，58 のメソッドをリファクタリング候補として検出した．

以下に実験対象ごとの結果に対する考察を述べる．

5.2.1 FRISC

図 5 に示したように，4 つのリファクタリング候補のグループが存在する．P0 のみで使用されているクラス (A)，P1 のみで使用されているクラス (B)，P2-P6 の中でまんべんなく利用されているクラス (C)，P6 でのみ使用されているメソッド (D) の 4 つである．異なるグループに含まれるクラスおよびメソッドは，それぞれ異なる機能を実現するために用いられたと考えられる．これらのクラスやメソッドを，グループごとにそれぞれ異なるパッケージへ移動させるリファクタリングを適用すると，各パッケージがそれぞれの機能を実現するためのクラスおよびメソッドのみを含むこととなる．結果として，モジュールの凝集度が向上し，結合度が減少することとなる．

リファクタリングの適用例としては，例えば，図中に (B) で表した “Data”，“TokenizeJob”，“Tokenize” を，新しく作成したサブパッケージの中に移動させる，という方法が挙げられる．また，図中に (D) で表した “Main.getMemoryInfo” のようにメソッドがリファクタリング候補となっている場合は，他のクラスにこれと似ている表示を持つものがあれば，そのクラ

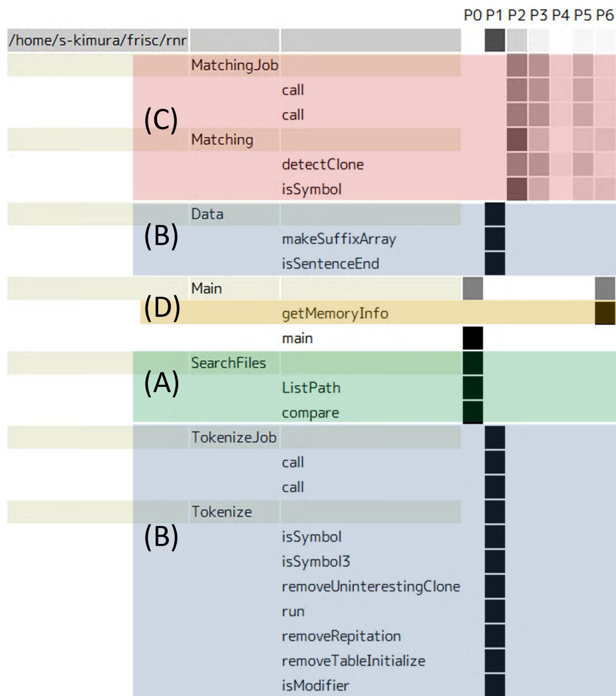

図 5: FRISC に対する提案手法の適用結果

スへ移動させる (メソッド移動リファクタリング)．今回は似ている表示を持つクラスが存在しないため，サブパッケージ，クラスを作成しそこへ移動させる．

これらのグループ分けを開発者に提示し，それぞれを移動させることを提案したところ，全ての要素に対して同意を得られた．そのため，これらのリファクタリング候補は適切であるといえる．

5.2.2 MASU

MASU は規模が大きいソフトウェアのため，FRISC と比較しリファクタリング候補が多く検出された．図 6 は，main.data.target.unresolved パッケージに含まれるメソッドに対する表示の一部である．このパッケージは 99 個のクラスを含み，58 個のクラスが実行履歴中に含まれていた．以下に，このパッケージで検出されたリファクタリング候補の中から 3 種類取り上げ，それらについての考察を述べる．

サブパッケージへの移動

図 6 中に (A) で示した，“JavaUnresolvedExternalFieldInfo”，“JavaUnresolvedExternalClassInfo”，および図には現れていないが同様の表示を持つ “JavaUnresolvedExternalMethodInfo” の 3 つは，クラスに含まれる全てのメソッドが P0 でのみ呼び出されている．このパッケージは，他の大半のメソッドが P11 で呼び出されている，または P0-P11 の間でまんべんなく呼び出されている．そのため，これらの 3 つのクラスは出力結果から容易にリファクタリング候補として検出することができる．これらの候補は，FRISC に対する考察で示した例と同様に，サブパッケージへの移動を行うリファクタリングが適用できる．

なお，名前に着目すると，これらの 3 つの候補には，クラス名に “JavaUnresolvedExternal” という接頭語が付いている．このパッケージで実行履歴に現れた 58 個のクラスのうち，“JavaUnresolvedExternal” という接頭語が付いているものは候補として検出された 3 つのみで，54 個のクラスは “Unresolved” という接頭語が付き，残りの 1 つは “NameResolver” という名前だった．そのため，名前からもこれらのクラスがリファクタリング候補であると考えられる．

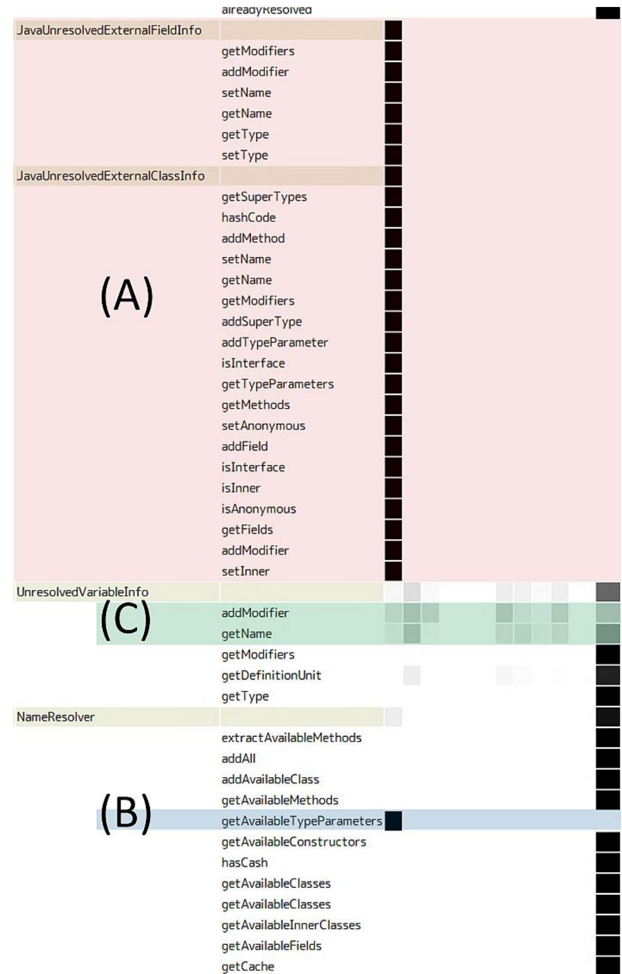

図 6: MASU に対する提案手法の適用結果の一部

開発者に聞き取りを行ったところ，検出されたリファクタリング候補は適切で，これらの 3 つのクラスはパッケージ中の他のクラスと異なった機能を実装しているため，サブパッケージを作成し，その下に配置するのが適切である，との評価を得た．そのため，これらのリファクタリング候補は適切であるといえる．

他クラスへの移動

図 6 中に (B) で示したものは，“NameResolver” というクラスに含まれる “getAvailableTypeParameters” というメソッドに対する表示である．このメソッドは P0 でのみ呼び出され，同じクラスに属する他のメソッドは全て P11 のみで呼び出されている．

開発者に聞き取りを行ったところ，検出されたリファクタリング候補は適切で，このメソッドは他のクラスに移動すべき，との評価を得た．ただし，このクラスはユーティリティとして用いられているクラスであり，含まれているメソッドはそれぞ

れ異なる機能を実現するものである．そのため，このクラスに含まれる他のメソッドも同様に他のクラスに移動すべきである，との評価も得た．同じクラスに属する他のメソッドがリファクタリング候補として検出できなかったのは，同一フェイズ内でのメソッドの呼び出しが発生してしまい，表示に差が生まれなかったためである．そのため，このリファクタリング候補は適切であるといえるが，このクラスに含まれる他のメソッドは，本手法ではリファクタリング候補として検出できないといえる．

setter/adder，getter に対する検出

図 6 中に (C) で示したものは，“addModifier” と “getModifiers” という，“UnresolvedVariableInfo” クラスのフィールドにある配列変数に対する adder，getter である．一般に，setter/adder はプログラム実行の前半で用いられ，getter は後半で用いられることが多い．そのため，このようなメソッドは異なるフェイズに属することとなる．しかし，これらのメソッドはそのクラス内のデータに対するアクセスのみを行うため，1 つのクラスに存在する方が凝集度が高いといえる．よって，提案手法ではリファクタリング候補として検出されるが，これは誤検出である．

5.3 Research Question への回答

2.2 で述べた，Research Question に対して得られた考察を述べる．Research Question は，動的解析を用いて有効なリファクタリング候補を検出することができるか，という課題である．

提案手法では，メソッド呼び出し履歴に対して動的解析を適用し，リファクタリング候補の検出を支援している．また，実験を行い，FRISC に対して 18 のメソッドをリファクタリング候補として検出し，全ての候補に対して開発者の同意が得られた．MASU に対しては 58 のメソッドをリファクタリング候補として検出し，そのうち 39 のメソッドに対して開発者の同意が得られた．

このことから，動的解析を用いて有効なリファクタリング候補を検出することは可能であるといえる．

5.4 結果の妥当性

本研究における結果の妥当性に関して，以下で挙げる点に留意する必要がある．

今回の実験では，それぞれのソフトウェアに対し 1 つのテストケースしか用意していない．しかし，メソッド呼び出し履歴はテストケースに大きく依存する．そのため，異なるテストケースを用いた場合，今回の実験結果とは大きく異なる結果となる可能性がある．

また，フェイズ分割数によって，可視化したフェイズ単位でのメソッド呼び出し情報は大きく変動し，リファクタリング候補かどうかの判定に影響を与える．そのため，適切でないフェイズ分割数を設定してしまうと，検出すべきリファクタリング候補を検出できない，または誤検出をする可能性がある．

6. おわりに

本研究では，メソッド呼び出し履歴を動的解析することにより，リファクタリング候補の検出を支援した．2 つの実際のソフトウェアに対する実験の結果，提案手法を用いて有効なリファクタリング候補を検出できることを確かめた．

本研究の今後の課題は以下のとおりである．

- 既存研究との定量的な比較
- 複数のテストケースに対する出力の統合
- メソッド呼び出し履歴以外の実行履歴を用いる動的解析手法との組み合わせ

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究 (A)(課題番号 : 21240002)，萌芽研究 (課題番号 : 23650014，24650011)，若手研究 (A)(課題番号 : 24680002) の助成を得た．

文 献

[1] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, “Does code decay? assessing the evidence from change management data,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 27, pp. 1–12, Jan/Feb 2001.
[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
[3] T. Mens and T. Tourwe, “A survey of software refactoring,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 30, pp. 126–139, Feb 2004.
[4] F. Simon, F. Steinbruckner, and C. Lewerentz, “Metrics based refactoring,” in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pp. 30–38, Mar 2001.
[5] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 35, pp. 347–367, May/June 2009.
[6] Y. Watanabe, T. Ishio, Y. Ito, and K. Inoue, “Visualizing an execution trace as a compact sequence diagram using dominance algorithms,” in *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis*, pp. 1–5, Oct 2008.
[7] 村上寛明, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二, “ソースコード中の繰り返し部分に着目したコードクローン検出手法の提案,” 電子情報通信学会技術研究報告, Mar 2012.
[8] “Amida.” http://sel.ist.osaka-u.ac.jp/ ishio/amida/.
[9] T. Ishio, Y. Watanabe, and K. Inoue, “Amida: a sequence diagram extraction toolkit supporting automatic phase detection,” in *Proceedings of the 30th International Conference on Software Engineering*, pp. 969–970, May 2008.
[10] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, “A pluggable tool for measuring software metrics from source code,” in *Proceedings of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pp. 3–12, Nov 2011.