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.

処理の委譲の有無に基づくリファクタリング支援手法

高　良多朗† 　堀田　圭佑† 　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
〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †{r-kou,k-hotta,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

あらまし　近年，ソフトウェア保守作業の効率を高める手段としてリファクタリングが注目されるようになり，様々な視点からリファクタリングを支援する手法が提案されている．オブジェクト指向言語においては，メソッドの長さや複雑度などのメトリクス値に着目してリファクタリング候補を特定する手法が多数存在しているが，実際にリファクタリングを適用すべきかの判断が難しい場合がある．著者らは，上述のメトリクス値によって特定のコードだけを評価するのでなく，類似した処理を行う複数のコードでの比較を行うことでリファクタリングの検討が容易になるのではないかと考えた．そこで本研究では，メソッドが行う処理の委譲の有無に着目し，これが一貫していないコードをリファクタリング候補として特定する手法を提案する．また，提案手法を開発者が異なる複数のソフトウェアに対して実験を行った結果，類似した処理を実装しており，かつ，処理の委譲の有無が異なる箇所が存在することが確認できた．

キーワード　ソフトウェア保守, ソースコード解析, ソースコード理解, リファクタリング

Supporting Refactoring Based on Presence of Delegation

Ryotaro KOU†, Keisuke HOTTA†, Yoshiki HIGO†, Hiroshi IGAKI†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita–Shi, Osaka, 565–0871 Japan
E-mail: †{r-kou,k-hotta,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

Abstract　Refactoring has attracted attention for software maintenance. Therefore, many approaches supporting refactoring have been proposed from a variety of viewpoints. In object-oriented languages, most of the approaches suggest refactoring candidates by using code metrics such as length or complexity for each of methods. However, it may be difficult to judge whether those candidates should be refactored. The authors conjectured that comparing some source code makes judging candidates easier than checking source code individually. Therefore, this paper proposes an approach identifying refactoring candidates by using presence of delegation. We experimented the proposed approach on different developers' software. As a result, the proposed approach was able to suggest several code portions that performs similar procedures and varies in presence of delegation functionality as refactoring candidates.

Key words　Software Maintenance, Source Code Analysis, Source Code Understanding, Refactoring

1. まえがき

近年，ソフトウェア開発は大規模化・複雑化する傾向にあり，それに伴い保守作業に要するコストも増加している．保守作業の中でも，ソースコードを読んで理解する作業は特に大きなコストがかかるため，理解性の高いソースコードの実装による保守作業の効率化が求められている．ソースコードの理解性を高めるための経験則として，“1 つのメソッドは 1 つの処理のみを行うべき” というものがある [1]．この経験則に従うことで，処理の流れや入出力などについての理解性が向上する．しかし，理解性の高いソースコードが実装されていたとしても，追記・修正を重ねることでソースコードはにしだいに劣化し，理解性が低下していくと報告されている [2]．

劣化したソースコードを改善する技術としてリファクタリングが存在する [3], [4]．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，内部の構造を改善する作業” と定義されている．劣化したソースコードに対して適切なリファクタリングを適用し，上述の経験則を満たすように変更するこ

```
01:  public void continuous() {
02:      ...
03:      for(int num = 1;num < 5;num++) {
04:          String str = "continuous" + num;
05:          RepeatString rep = new RepeatString(str,num);
06:          String str2 = rep.toString();
07:          System.out.println(str2);
08:      }
09:  }
```

```
11:  public void separate() {
12:      int num = 1;
13:      while(num<5) {
14:          String str = "separate" + num;
15:          RepeatString rep = new RepeatString(str,num);
16:          writeRepeat(rep);
17:          num++;
18:      }
19:  }
20:  private void writeRepeat(RepeatString target) {
21:      String output = target.toString();
22:      System.out.println(output);
23:  }
```


(a) 1 つのメソッドで実装した例（委譲なし）　(b) 2 つのメソッドで実装した例（委譲あり）

図 1　処理の委譲の有無が異なるコードの例

とで低下した理解性を改善できる．オブジェクト指向言語におけるメソッドを対象としたリファクタリングパターンの一例としては，メソッドが行う処理の一部を別のメソッドに委譲する “メソッドの抽出” や，メソッドが処理の委譲を行わないようにする “メソッドのインライン化” がある．しかし，大規模なソースコードからリファクタリングすべき箇所を手動で特定することは困難であるため，リファクタリングを支援するための手法が多数提案されている [5], [6].

既存研究で提案されているメソッドの抽出やメソッドのインライン化を支援する手法の多くは，メソッドの長さや複雑度などのメトリクス値 [7] が一定のしきい値を超えるものをリファクタリングの候補として提示する．しかし，提示された候補が実際にリファクタリングを適用すべきであるかを判断することの支援は不十分である．例えば，あるメソッドがメソッドの抽出の候補として提示され，実際にメソッドの抽出を適用すべきかを判断する基準を “複数の処理を行っているか” とした場合，提示された候補を見るだけでは適切な判断ができるとは限らない．ここで，類似した処理が 2 箇所で実装されており，かつ，それらの間で処理の委譲の有無が異なる場合，すなわち，ある処理が一方では 1 つのメソッドのみを用いて実現されており，もう一方では複数のメソッドを用いて実現されている場合を考える．このような場合，それらを比較することによって，メソッドの抽出やメソッドのインライン化を適用すべきか否かを検討することができる．したがって，それらを同時に開発者に提示することによって，それらに対してメソッドの抽出やメソッドのインライン化を適用すべきか否かの判断を支援することができる．

そこで，本研究では処理の委譲の有無に着目してメソッドの抽出やメソッドのインライン化の候補を特定する手法を提案する．提案手法では，類似した処理を実装しており，かつ，処理の委譲の有無が異なる箇所を特定し，それらを同時に開発者に提示することによって，リファクタリングを適用すべきか否かの判断を支援する．

提案手法をツールとして実装し，これを用いた被験者実験を行うことで提案手法の有効性を評価した．実験では，7 つのソフトウェアに対して本ツールを適用して検出された処理の委譲の有無が異なるコードを用意しておき，これをソフトウェアの開発者自身に提示し，開発者には提示された処理の委譲の有無が異なるコードがリファクタリング候補として妥当かを判断してもらった．実験の結果，提案手法で検出された箇所の中に，類似した処理を実装しており，かつ，処理の委譲の有無が異なる箇所が存在することが確認できた．

2. 研 究 動 機

2.1　処理の委譲の有無が異なるコード

本研究では，類似した処理を行っていて，かつ，処理の委譲の有無が異なるような箇所を “処理の委譲の有無が異なるコード” と呼び，これに着目する．

図 1 に類似した処理を行っているが，処理の委譲の有無が異なるコードの例を示す．図 1(a) では処理を 1 つのメソッドで実現しているのに対し，図 1(b) では処理を 2 つのメソッドで実現している．すなわち，図 1(a) は処理の委譲を行っていないのに対し，図 1(b) は処理の委譲を行っている．

図 1 のように処理の委譲の有無が異なるコードがある場合，どちらかのコードは “1 つのメソッドは 1 つの処理のみを行うべき” という経験則に反している可能性が高い．このような処理の委譲の有無が異なるコードを特定して開発者に提示することによって，開発者はどのコードに対して，どのリファクタリングを行うべきかを判断しやすくなる．

2.2　関 連 研 究

ソースコード中から類似した処理を特定する手法として，コードクローン検出が存在する [8], [9]．既存のコードクローン検出手法の多くはソースコード上で同一あるいは類似している箇所を検出する．しかし，処理の委譲の有無が異なるコードは一部がメソッドに抽出され，ソースコード上で連続していない．したがって，上述の手法をそのまま用いても処理の委譲の有無が異なるコードの検出は困難だと考えられる．

本研究に関連する研究として，類似した処理を行っているが表現方法が異なる，ソースコード上で不連続なコード片をコードクローンとして検出することができる手法がいくつか提案されている．

神谷は，モジュールの実行，すなわちメソッド呼び出しを単位とした実行パスをモデル化し，モデル上で一致する実行列をコードクローンとして検出する手法を提案した [10]．この手法はモジュール内の実行パスを展開することで，モジュール内部で閉じた実行パスのみでなくプログラム全体での実行パスを調

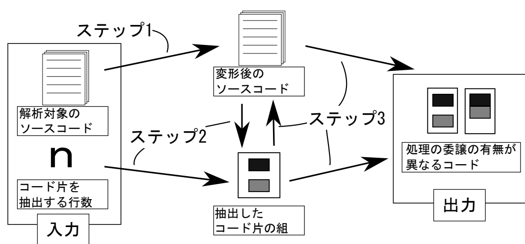


図 2　提案手法全体の流れ

```
01:  public void adapter(){
02:    int num1 = 15;
03:    int num2 = 25;
04:    int num3 = num1 * num2;
05:    Divisor divi = new Divisor(num3);
06:    adaptee(divi.getArray());
07:  }
08:
```

２つのメソッド呼び出しが存在

(a) 変形前

```
01:  public void adapter(){
02:    int num1 = 15;
03:    int num2 = 25;
04:    int num3 = num1 * num2;
05:    Divisor divi = new Divisor(num3);
06:    int[] var0 = divi.getArray();
07:    adaptee(var0);
08:  }
```

メソッド呼び出しを切り離す

(b) 変形後

図 3　1 つの文中に複数のメソッド呼び出しが存在するコードの例

```
01:  public void adapter(){
02:    int num1 = 15;
03:    int num2 = 25;
04:    int num3 = num1 * num2;
05:    Divisor divi = new Divisor(num3);
06:    int[] var0 = divi.getArray();
07:    adaptee(var0);
08:  }
09:
```

(a) 正規化前

```
01:  public void adapter(){
02:    int $V = $N;
03:    int $V = $N;
04:    int $V = $V * $V;
05:    Divisor $V = new Divisor($V);
06:    int[] $V = $V.getArray();
07:    adaptee($V);
08:  }
```

(b) 正規化後

図 4　変数名とリテラルについての正規化を行う例

べることが可能であり，複数のモジュールにまたがった実行パスをコードクローンとして検出することができる．その反面として計算量が膨大になる課題点を抱えていたため，神谷は実行パスの共通部分をまとめた状態でモデル化する手法を新たに提案している [11].

吉田らは，異なるコードクローンに含まれるコード片間の依存関係に着目し，強い依存関係を持つコードクローンの集合をまとめて検出する手法を提案した [12]．この手法ではコードクローンの依存関係を調べるためのメトリクス値を新たに定義し，これを用いることで強い依存関係を持つコードクローンの分類，検出を行っている．実験の結果より，この手法を用いて検出されたコードクローンは既存手法を用いて検出されたコードクローンと比べ，リファクタリングを施しやすいものが多いと報告している．

3. 提案手法

3.1 概　要

本研究で提案する手法はソースコードから処理の委譲の有無が異なるコードを特定する．提案手法はソースコード中のすべてのメソッド呼び出しについて，メソッド呼び出しの直前のコード片と呼び出されるメソッドの先頭のコード片を抽出し，抽出した 2 つのコード片が 1 つのメソッド中に連続して出現する箇所を特定する．2 つのコード片を抽出した箇所および，それらのコード片が 1 つのメソッド中に連続して出現する箇所の組が処理の委譲の有無が異なるコードとなる．提案手法は検出対象となるソースコードに加え，メソッド呼び出しの直前ならびに呼び出されるメソッドの先頭から何行を抽出するかを表す整数値を入力とする．

提案手法の全体の流れを図 2 に示す．本手法は以下に示す 3 つのステップによって行われる．

ステップ 1　1 つの文中に出現するメソッド呼び出しが高々1 つになるようにソースコードを変形する．

ステップ 2　ソースコード中の各メソッド呼び出しについて，メソッド呼び出しの直前のコード片と，呼び出されるメソッドの先頭のコード片を抽出する．

ステップ 3　抽出した 2 つのコード片をクエリとして，1 つのメソッド中に連続して出現する箇所を各メソッドから検出する．

以下，提案手法の各ステップについて，その処理内容を詳細に述べる．

3.2 ステップ 1

1 つの文に複数のメソッド呼び出しが含まれる場合，そのままでは同じ文に含まれるメソッド呼び出しから抽出される直前のコード片は同じものとなる．しかし，それらのメソッドの呼び出しは同時ではないため，メソッド呼び出しの順序を保つようにコード片を抽出するべきである．この問題を解決するために，本ステップでは入力として受け取ったソースコードの変形を行い，1 つの文中に出現するメソッド呼び出しの数が高々1 つになるようにする．

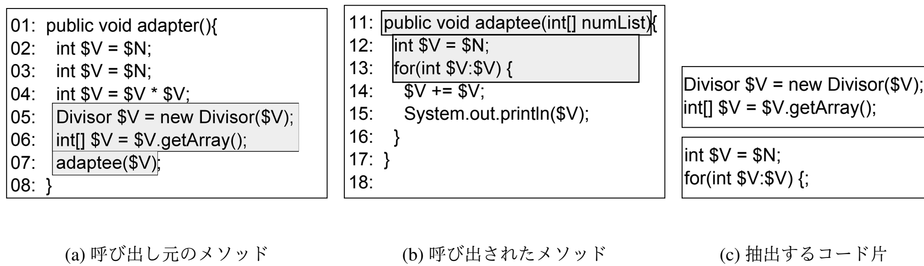
図 3 に示す例では，図 3(a) の状態では 6 行目の文中に getArray メソッドと adaptee メソッドの 2 つのメソッド呼び出しが存在しているため，図 3(b) のように getArray メソッドと adaptee メソッドを別の文に切り離し，1 つの文中に出現するメソッド呼び出しの数が高々1 つになるように変形している．

また，ユーザー定義名の違いを吸収するために，図 4 に示す例のように，ソースコード中に現れる変数名とリテラルを対応する字句に置き換える正規化を行う．各メソッド呼び出しから呼び出されるメソッドを区別するためメソッド名と型名の正規化は行わない．

3.3 ステップ 2

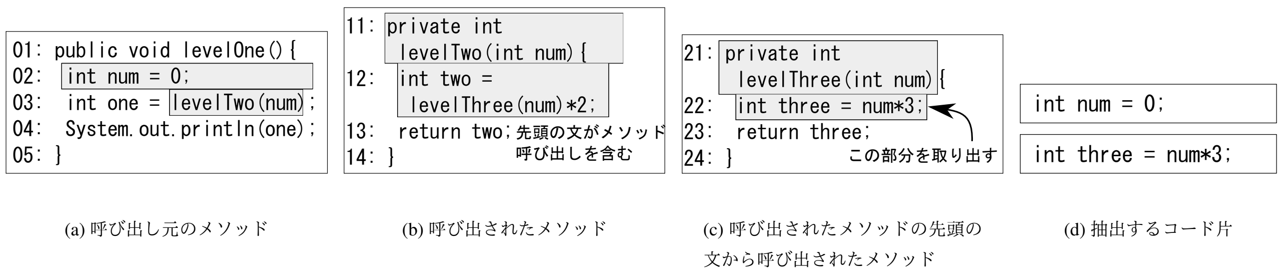
このステップでは，各メソッド呼び出しについて，メソッド呼び出しの直前のコード片と，呼び出されるメソッドの先頭のコード片を抽出する．抽出されるコード片の行数は入力で指定した整数値となる．どちらかのコード片について抽出できる行数が入力値に満たない場合，そのメソッド呼び出しについては処理を行わない．

図 5 に示す例ではコード片を 2 行ずつ抽出するように設定



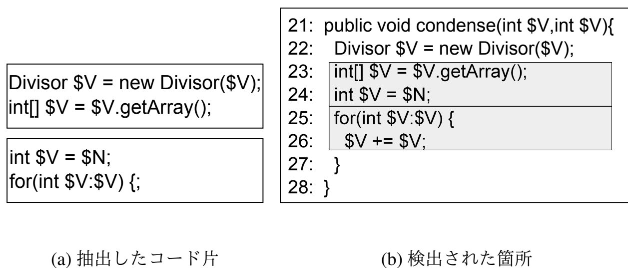
(a) 呼び出し元のメソッド　(b) 呼び出されたメソッド　(c) 抽出するコード片

図 5　メソッド呼び出しについてコード片を抽出する例



(a) 呼び出し元のメソッド　(b) 呼び出されたメソッド　(c) 呼び出されたメソッドの先頭の文から呼び出されたメソッド　(d) 抽出するコード片

図 6　複数回のメソッド呼び出しに対応してコード片を抽出する例



(a) 抽出したコード片　(b) 検出された箇所

図 7　抽出したコード片から一致する箇所を検出する例

し，adaptee メソッドについて抽出している．抽出されるコード片の組は図 5(c) に示すものになる．本ステップにおいて，呼び出されるメソッドの先頭の文がまた別のメソッド呼び出しを含む場合，さらに呼び出されるメソッドを参照して先頭のコード片を抽出する．図 6 に示す例では，複数回のメソッド呼び出しに対応してコード片を抽出している．図 6(a) の levelOne メソッドは図 6(b) の levelTwo メソッドを呼び出しているが，levelTwo メソッドは先頭の文で図 6(c) の levelThree メソッドを呼び出している．このため，levelOne メソッドと levelThree メソッドからコード片を抽出し，図 6(d) に示すものとなる．

3.4 ステップ 3

このステップでは，ステップ 2 で抽出した 2 つのコード片が 1 つのメソッド中に連続して現れる箇所を検出する．

図 7 に検出の例を示す．図 7(a) に示すコード片は，ステップ 2 の例で使用した図 5(c) のものである．このコード片は図 7(b) のソースコード中に連続して現れている．

4. 評価実験

提案手法がリファクタリング対象として妥当なコードを提示できるかを調べるため，複数の開発者による被験者実験を行った．

4.1 実験概要

本実験では，複数の開発者を被験者として募集し，各被験者が記述したソースコードに対して提案手法を適用，検出されたコードがリファクタリング対象として妥当であるかを調べた．実験に使用したソースコードの規模を表 1 に示す．

リファクタリングを行うべきか否かの判断基準は，開発者によって異なると考えられるため，本実験では開発者による判断基準の違いを吸収するために，提示されたコードがリファクタリング対象として妥当かを判断する基準を “処理の委譲の有無が意図的なものであるか” とし，処理の委譲の有無が意図的であるコードはリファクタリング対象として妥当でないものとする．

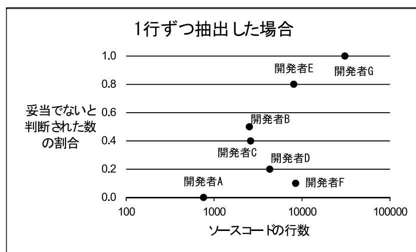
4.2 実験方法

あらかじめコード片を抽出する行数を 1 から 3 までに設定した上で各ソースコードに対してツールを実行した結果を用意する．実験のコストを削減するため，検出された処理の委譲の有無が異なるコードが 10 箇所を超える場合はその中から 10 箇所をランダムに抽出して使用する．

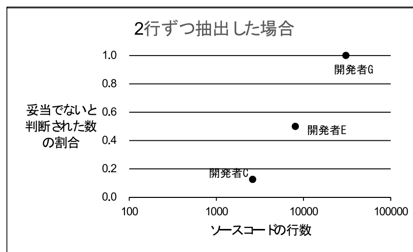
開発者には自身のソースコードに対するツールの実行結果を確認してもらい，それぞれがリファクタリング対象として妥当かを判断してもらう．上述の通り，妥当かを判断する基準は “処理の委譲の有無が意図的なものであるか” とする．最後に判断

表 1　被験者実験に使用したソースコード

開発者	A	B	C	D	E	F	G
総行数	759	2,525	2,604	4,307	8,052	8,475	30,691
ファイルの数	10	15	20	73	83	127	214



(a) 1 行ずつ抽出した場合



(b) 2 行ずつ抽出した場合

図 8　ソースコードの行数と妥当でないと判断された数の割合の関係

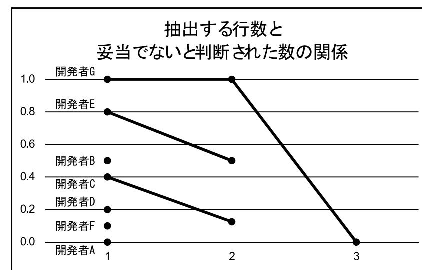


図 9　コード片を抽出する行数と妥当でないと判断された数の割合の関係

してもらった結果を集計して開発者ごと，およびコード片を抽出する行数ごとに妥当でないと判断された数の割合を算出する．

4.3　実験結果

実験の結果を表 2 に示す．本ツールを適用する際にコード片を 2 行ずつ，あるいは 3 行ずつ抽出するように設定した場合，実験対象によってはコードが検出されなかったため，該当箇所にはハイフンを表示している．また，4 行ずつ抽出した場合はどのソースコードについても処理の委譲の有無が異なるコードが検出されなかった．

また，実験対象となるソースコードの行数と妥当でないと判断された数の割合の関係を表すグラフを図 8 に，コード片を抽出する行数と妥当でないと判断された数の割合の関係を表すグラフを図 9 に示す．図 8，9 のグラフより，コード片を抽出する行数を増やすほど妥当でない検出の割合が低くなり，妥当な検出の割合が高くなることが分かる．

検出されたコードの例を図 10，図 11 に示す．図 10 は開発者 F のソースコードに対して，コード片を 1 ずつ抽出した場合の検出結果の 1 つである．この例では検出箇所は同クラスのオブジェクトに対して類似した処理を行っており，メソッドに抽出したほうが良い，つまり妥当な検出結果であると判断された．

表 2　被験者実験の結果

	開発者	A	B	C	D	E	F	G
1 行ずつ抽出	検出された数	2	4	187	16	145	139	7,261
	使用した数	2	4	10	10	10	10	10
	妥当でないと判断された数	0	2	4	2	8	1	10
	妥当でないと判断された数の割合	0.0	0.5	0.4	0.2	0.8	0.1	1.0
2 行ずつ抽出	検出された数	-	-	8	-	2	-	10
	使用した数	-	-	8	-	2	-	10
	妥当でないと判断された数	-	-	1	-	1	-	10
	妥当でないと判断された数の割合	-	-	0.13	-	0.5	-	1.0
3 行ずつ抽出	検出された数	-	-	-	-	-	-	2
	使用した数	-	-	-	-	-	-	2
	妥当でないと判断された数	-	-	-	-	-	-	0
	妥当でないと判断された数の割合	-	-	-	-	-	-	0.0

図 11 は開発者 G のソースコードに対して，コード片を 1 ずつ抽出した場合の検出結果の 1 つである．この例では検出箇所の前後で異なる処理を行っており，検出に使用したコード片が偶然一致したと判断された．

5.　考　　察

実験の結果から，提案手法はどのソースコードに対しても，コード片を抽出する行数を増やすほど，リファクタリング対象として妥当なコードを多く検出できるといえる．したがって，提案手法を使うときはコード片を抽出する行数を増やし，抽出する行数を増やした場合の検出結果から順番に調べることが望ましいと考えられる．

また，図 11 に示した例のように，異なる処理を実装している箇所が検出されることがあるが，その理由として，抽出したコード片の中身が整数値の加算や null の確認など，どの処理においても頻繁に用いられる文であることが考えられる．特に，呼び出されたメソッドの先頭の文は null の確認やローカル変数の宣言であることが多いため，妥当でない検出結果が増える原因の 1 つとして考えられる．したがって，本研究の課題点はこのような処理の委譲の有無が異なるコードの間に高い頻度で生じる違いを取り除くことができるようにすることである．

6.　妥当性の脅威

本研究での手法および実験の妥当性について注意点がある．

6.1　手法の妥当性

本研究の提案手法は 3.2 節で述べたように，1 つの文中に出現するメソッド呼び出しの数が高々1 つになるようにソースコードを変形している．提案手法とは異なる方法を用いて変形を行った場合，得られるソースコードは提案手法での変形によって得られるものと異なる可能性があるため，検出される処理の委譲の有無が異なるコードが変化すると考えられる．

6.2　実験の妥当性

本研究の実験では，個人によって開発されたソースコードのみを使用している．複数人によって開発されたソースコードは開発者よる実装方針の違いの影響により，委譲の有無が異なるコードが発生する可能性が高くなると考えられる．このような

```
257: int index=storeStatement(Variable55,
       Variable56,StatementInfo.SYNCHRONIZED);
258: Block Variable57=node.getBody();
259: accept(Variable57);
260: Block Variable58=node.getBody();
261: int Variable59=Variable58.getStartPosition();
```

(a) 呼び出し元のメソッド

```
353: private void accept(ASTNode node){
354:   if(node!=null) {
355:     node.accept(this);
356:   }
357: }
```

(b) 呼び出されたメソッド

```
086:   map.put(binding,info);
087: }
088: Block Variable9=node.getBody();
089: if(Variable9!=null) {
090:   CallVisitor callVisitor= new CallVisitor
         (map,info,binding,id,manager);
091:   Block Variable10=node.getBody();
092:   Variable10.accept(callVisitor);
093: }
```

(c) 検出された箇所

図 10　妥当だと判断された検出結果の例

```
957:    int Variable98=readInt(u+4);
958:    values[i]=labels[offset+Variable98];
959:    u+=8;
960:  }          ← 括弧閉じは無視
961:  mv.visitLookupSwitchInsn(
        labels[label],keys,values);
962:  break;
963: case ClassWriter.VAR_INSN:
```

(a) 呼び出し元のメソッド

```
298: public void visitLookupSwitchInsn(final Label
       dflt, final int[] keys, final Label[] labels){
299:   if(mv!=null) {
300:     mv.visitLookupSwitchInsn(dflt,keys,labels);
301:   }
302:   execute(Opcodes.LOOKUPSWITCH,0,null);
303:   this.locals=null;
```

(b) 呼び出されたメソッド

```
024: public void visitInsn(final int opcode){
025:   minSize+=1;
026:   maxSize+=1;
027:   if(mv!=null) {
028:     mv.visitInsn(opcode);
029:   }
030: }
```

(c) 検出された箇所

図 11　妥当でないと判断された検出結果の例

ソースコードに対して実験を行った場合，本研究とは異なる実験結果が得られる可能性がある．

また，同じく実験において，本ツールによって検出されたコードが 10 箇所を超える場合はその中から 10 箇所をランダムに抽出して使用した．抽出されるコードはツールを実行する度に変化するため，同じソースコードを対象に実験を行っても，常に同じ実験結果が得られるとは限らない．

7. あ と が き

本研究では，メソッドの抽出やメソッドのインライン化の適用候補の特定を支援するために，類似した処理を実装しており，かつ，処理の委譲の有無が異なる箇所をリファクタリング候補として特定する手法を提案した．提案手法はメソッド呼び出しに着目し，類似した処理を実装しており，かつ，処理の委譲の有無が異なる箇所を処理の委譲の有無が異なるコードとして検出する．

また，提案手法を実装したツールを用いて，手法の有効性を調べるために複数の開発者による被験者実験を行った．実験の結果，検出された処理の委譲の有無が異なるコードのうち，一致する行数が多いものから順番に調べることが望ましいことを確認した．また，検出結果を確認したところ，リファクタリング対象として妥当な検出が可能であることも確認した．

今後の課題として，引数が null であるかの確認やメンバ変数の宣言など，メソッドの先頭に頻繁に実装される記述を抽出するコード片に含めないよう選択できるようにすることを検討している．これによって処理の委譲の有無が異なるコードの間に高い頻度で生じる違いが取り除かれ，よりリファクタリング候補の特定に有用な手法になると考えられる．

謝辞　本論文は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号：25220003)，挑戦的萌芽研究 (課題番号：24650011)，及び文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の支援を受けて行われた．

文　　献

[1] R.C. Martin, Clean Code A Handbook of Agile Software Craftsmanship, Prentice Hall, Aug. 2008.

[2] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, “Does code decay? assessing the evidence from change management data,” IEEE Trans. on Software Engineering, vol.27, no.1, pp.1–12, Jan. 2001.

[3] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley, June 1999.

[4] T.mens and T.Tourwe, “A survey of software refactoring,” IEEE Trans. on Software Engineering, vol.30, no.2, pp.126–139, Feb. 2004.

[5] E. Murphy-Hill and A.P. Black, “Breaking the barriers to successful refactoring: Observations and tools for extract method,” Proc. of the 30th international conference on Software engineering, pp.421–430, May 2008.

[6] E. Murphy-Hill, C. Parnin, and A.P. Black, “How we refactor, and how we know it,” IEEE 31th international conference on Software engineering, pp.287–297, May 2009.

[7] N.E. Fonton and S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, Course Technology Ptr (Sd), Feb. 1998.

[8] 肥後芳樹，楠本真二，井上克郎，“コードクローン検出とその関連技術,” 電子情報通信学会論文誌，vol.J91-D，no.6，pp.1465–1481，June 2008.

[9] 井上克郎，神谷年洋，楠本真二，“コードクローン検出法,” コンピュータソフトウェア，vol.18，no.5，pp.47–54，Sep. 2001.

[10] 神谷年洋，“任意粒度機能モデルに基づくバイトコードからのコードクローン検出手法,” 電子情報通信学会技術研究報告，vol.113，no.24，pp.43–48，May 2013.

[11] 神谷年洋，“任意粒度機能モデルに基づくコードクローン検出手法の大規模プログラムへの適用に向けた改善,” 電子情報通信学会技術研究報告，vol.113，no.159，pp.133–137，July 2013.

[12] 吉田則裕，肥後芳樹，神谷年洋，楠本真二，井上克郎，“コードクローン間の依存関係に基づくリファクタリング支援,” 情報処理学会論文誌，vol.48，no.3，pp.1431–1442，Mar. 2007.