特別研究報告

題目

処理の委譲の有無に基づく一貫性が欠如したコードの特定

指導教員
楠本 真二 教授

報告者
高　良多朗

平成 26 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 25 年度 特別研究報告

処理の委譲の有無に基づく一貫性が欠如したコードの特定

高　良多朗

内容梗概

ソフトウェア保守を行うにあたり，ソースコードの理解性を向上させる技術としてリファクタリングが存在する．この技術は保守作業の効率化に有効だが，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難しい．このため，リファクタリングの候補を自動で特定するなど，リファクタリングを支援する手法が多数提案されている．

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，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．これらのリファクタリングパターンを支援する既存手法の多くはメソッドの長さや複雑度などを調べることでリファクタリングの候補を提示するが，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もまた難しいといえる．

そこで本研究では，処理の委譲の有無に着目してリファクタリングの候補を特定する手法を提案する．類似した処理を行い，かつ，処理の委譲の有無が異なる箇所を同時に開発者に提示することで，特定のしきい値によるコード単体での判断でなく，複数のコードを見比べることによる判断が可能となり，リファクタリングを適用すべきであるかを判断するための支援ができると考える．また，提案手法をツールとして実装し，異なる開発者による複数のソフトウェアと，オープンソースソフトウェアに対して実験を行った．実験の結果，提案手法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リファクタリングの候補として提示できることを確認した．

主な用語

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

目 次

1 まえがき **1**

2 準備 **3**

2.1 リファクタリング 3

2.1.1 メソッドの抽出 3

2.1.2 メソッドのインライン化 4

2.2 処理の委譲の有無が異なるコード 4

2.2.1 定義 4

2.2.2 発生の原因 4

2.3 関連研究 5

2.3.1 リファクタリング 5

2.3.2 類似した処理の特定：コードクローン検出 6

3 提案手法 **7**

3.1 概要 7

3.2 ステップ 1 8

3.3 ステップ 2 10

3.4 ステップ 3 12

4 実装 **13**

4.1 概要 13

4.2 検出結果の表示 13

5 評価実験 **15**

5.1 実験 1:複数の開発者による被験者実験 15

5.1.1 実験対象 15

5.1.2 実験方法 15

5.1.3 実験結果 16

5.1.4 検出結果の例 17

5.2 実験 2:オープンソースソフトウェアに対する実験 19

5.2.1 実験対象 19

5.2.2 実験結果 19

6 考察 **20**

7　妥当性について **21**

8　あとがき **22**

謝辞 **23**

参考文献 **24**

図 目 次

1 メソッドの抽出の適用例 3
2 メソッドのインライン化の適用例 4
3 処理の委譲の有無が異なるコードの例 5
4 提案手法全体の流れ 7
5 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 8
6 抽象構文木によるソースコードの変形 9
7 変数名とリテラルについての正規化を行う例 10
8 メソッド呼び出しについてコード片を抽出する例 11
9 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 11
10 抽出したコード片から一致する箇所を検出する例 12
11 出力の例 14
12 被験者実験の結果：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 16
13 被験者実験の結果：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 16
14 妥当だと判断された検出結果の例 17
15 妥当でないと判断された検出結果の例 18

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る．それに伴い保守作業に要するコストも増加するため，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に寄与する経験則として，“1 つのメソッドは 1 つの処理のみを行うべき” とされている [1]．この経験則に従うことで，処理の流れや処理の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，度重なる追記・修正によって劣化し，しだいに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で，内部の構造を改善していく作業” と定義されている．すなわち，劣化したソースコードに対してリファクタリングを適用することで，ソフトウェアの振る舞いを変えずに低下した理解性を改善できる．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，“メソッドの抽出” や “メソッドのインライン化” がある．メソッドの抽出とは，メソッドが行う処理の一部を別のメソッドに委譲し，同じ処理を複数のメソッドを用いて行うようにソースコードを変更することである．また，メソッドのインライン化とは，処理の一部を別のメソッドに委譲しているメソッドについて，処理の委譲を行わないようにソースコードを変更することである．すなわち，委譲先で行っている処理をすべて委譲元で行わせることをいう．1 つのメソッドが複数の処理を行っているような場合はメソッドの抽出を適用し，それぞれの処理を 1 つのメソッドで行うように抽出することで上述の経験則を満たすことができる．反対に，複数のメソッドで 1 つの処理が実現されている場合はメソッドのインライン化を適用し，処理を 1 つのメソッドで行うようにすることで上述の経験則を満たすことができる．このように，ソースコードに対して適切なリファクタリングを行うことによってソースコードの理解性を向上させることができる．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．このような背景から，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 “複数の処理を行うか” 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，これを用いて提案手法の有効性を評価した．実験では被験者実験として，7 つのソフトウェアに対してツールを適用し，ソフトウェアの開発者自身に提示された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ほか，約 20 万行のオープンソースソフトウェアに対してツールを適用して検出されたコードの数を調べた．実験の結果，規模が大きめのソフトウェアに対して提案手法が使用可能であることと，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．

以降，2 章でリファクタリングの説明，処理の委譲の有無が異なるコードの例とその発生原因，関連研究について述べる．3 章で提案手法について説明し，4 章で提案手法の実装について説明する．5 章で提案手法の有効性を調べるための評価実験について説明する．6 章で実験結果の考察を行う．7 章で妥当性について述べ，最後に 8 章で本研究のまとめと今後の課題について述べる．

```
01:  void printPoint(Point point) {
02:    printFormat();
03:    System.out.println("X:" + point.getX());
04:    System.out.println("Y:" + point.getY());
05:  }
```

(a) 適用前

```
01:  void printPoint(Point point) {
02:    printFormat();
03:    printLocation(point);
04:  }
05:  ...
06:  void printLocation(Point p) {
07:    System.out.println("X:" + p.getX());
08:    System.out.println("Y:" + p.getY());
09:  }
```

(b) 適用後

図 1: メソッドの抽出の適用例

2　準備

2.1　リファクタリング

リファクタリングとは，Fowler らによって “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” と定義されている [3]．リファクタリングの方法は同文献 [3] の中で，メソッドの抽出やメソッドのインライン化などのリファクタリングパターンとしてまとめられている．また，Fowler らはソースコード中に存在する，将来的に問題を引き起こす可能性のある箇所を “不吉なにおい（Bad Smell）” と呼び，これをリファクタリングを検討すべき箇所と述べている．

以下，本小節では本研究で着目するメソッドの抽出とメソッドのインライン化について説明する．

2.1.1　メソッドの抽出

“メソッドの抽出（ExtractMethod）” とは，長すぎるメソッドや複雑すぎるメソッドが行う処理の一部を，新たなメソッドとして抽出するリファクタリングパターンである．すなわち，メソッドが行う処理の一部を，他のメソッドに委譲するようにソースコードを変更することを指す．

メソッドの抽出の適用例を図 1 に示す．図 1(a) では，標準出力を printPoint メソッドが行っている．図 1(a) の printPoint メソッドに対してメソッドの抽出を適用し，標準出力への出力処理を新たに作成した printLocation メソッドに抽出すると図 1(b) のようになる．

```
01:  int getAbsolute(int index) {
02:    return (isMinus(index)) ?  index * -1 : index ;
03:  }
04:  ...
05:  boolean isMinus(int num) {
06:    return num < 0;
07:  }
```

(a) 適用前

```
01:  int getAbsolute(int index) {
02:    return (index < 0) ?  index * -1 : index ;
03:  }
```

(b) 適用後

図 2: メソッドのインライン化の適用例

2.1.2　メソッドのインライン化

“メソッドのインライン化（InlineMethod）” とは，短すぎるメソッドや中身が単純すぎるメソッドが行う処理をメソッドの呼び出し元に展開するリファクタリングパターンである．

メソッドのインライン化の適用例を図 2 に示す．図 2(a) では，getAbsolute メソッド中の三項演算子の条件式の計算を isMinus メソッドに委譲している．図 2(a) の getAbsolute メソッドに対してメソッドのインライン化を適用し，条件式を委譲しないようにすると図 2(b) のようになる．

2.2　処理の委譲の有無が異なるコード

2.2.1　定義

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 3 に処理の委譲の有無が異なるコードの例を示す．図 3(a) と図 3(b) は類似した処理を行っているが，図 3(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 3(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 3(a) は処理の委譲を行っていないのに対し，図 3(b) は処理の委譲を行っている．

図 3 のように処理の委譲の有無が異なる場合，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．

2.2.2　発生の原因

処理の委譲の有無が異なるコードが発生する原因としては，以下のものが挙げられる．

```
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:  }
```

(a) 1 つのメソッドで実装した例（委譲なし）

```
11:  public void separate() {
12:      ...
13:      int num = 1;
14:      while(num<5) {
15:          String str = "separate" + num;
16:          RepeatString rep = new RepeatString(str,num);
17:          writeRepeat(rep);
18:          num++;
19:      }
20:  }
21:  ...
22:  private void writeRepeat(RepeatString target) {
23:      String output = target.toString();
24:      System.out.println(output);
25:  }
```

メソッドの呼び出し

(b) 2 つのメソッドで実装した例（委譲あり）

図 3: 処理の委譲の有無が異なるコードの例

部分的なリファクタリング 元々は処理の委譲の有無が同じであったコード群の内，一部のコードに対してメソッドの抽出やメソッドのインライン化といったリファクタリングを行った場合，リファクタリングを行ったコードと行っていないコードの間では処理の委譲の有無が異なる [8].

開発者による実装方針の違い 複数人でソフトウェアの開発を行う際，実装方針が統一されていない場合，類似した処理が異なる方法で実装されることがある.

差分を含む処理 類似した処理を行うコードの間に文の追加や変更などの差分が含まれる場合，差分による影響を取り除いた上で同様のリファクタリングを適用すると，データ結合が複雑になり，かえって理解性が低下することがある [9]. このように，類似した処理を行うコード間で委譲の有無を揃えない方が理解性が高くなる場合がある.

2.3 関連研究

2.3.1 リファクタリング

丸山は，オブジェクトが持つ変数とメソッド（属性と操作）の関係を正しく理解していない場合はリファクタリングの適用が困難であると述べた上で，プログラムスライシング [10] という手法を拡張し，基本ブロックスライシングによるメソッドの抽出の候補を特定する手法を提案した [11]. この手法では，開発者によって指定された，ある変数に依存関係を持つコード片をメソッドの抽出の候補とすることで，リファクタリングの前後での変数の依存関

係の確認を行う手間を削減できると報告している．

また丸山らは，リファクタリングがソースコードの改善に有効かを判断することは実際に適用した結果を確認してみなければ難しいと述べ，リファクタリングを適用した履歴を監視することで，過去に適用された不適切なリファクタリングを取り消す手法を提案した [12]．

若林らは，クラス構成やデータの依存関係が適切でない場合に理解性が下がると述べ，クラス構成の問題点が発生する原因について議論した上で，発生したクラス構成の問題点を自動的に発見する手法を提案した [13]．

2.3.2 類似した処理の特定：コードクローン検出

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 [14, 15]．コードクローンとは，ソースコード中に存在する同一または類似したコード片のことである．既存のコードクローン検出手法の多くはソースコードを一連の文字列とみなし，一定のしきい値以上連続して一致する部分文字列をコードクローンとして検出する．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てソースコード上で連続していないため，上述の手法をそのまま用いた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，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手法が提案されている．神谷は，モジュールの実行パスを単位とした上でソフトウェアプロダクトをモデル化することで，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 [16]．この手法はモジュールを構成する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考慮するため，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．しかし，この手法は全てのモジュールを構成する実行パスを展開するため，時間計算量および空間計算量についての課題点を抱えていた．そこで神谷は，実行列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することで，この課題点を解決している [17]．

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 [18]．吉田らの手法では，コードクローン間の距離を表すメトリクス値 DCH[19] を用いることで，メソッド間のクラス階層上における距離を表す新たなメトリクス値を定義している．また，定義したメトリクス値を用いることで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．

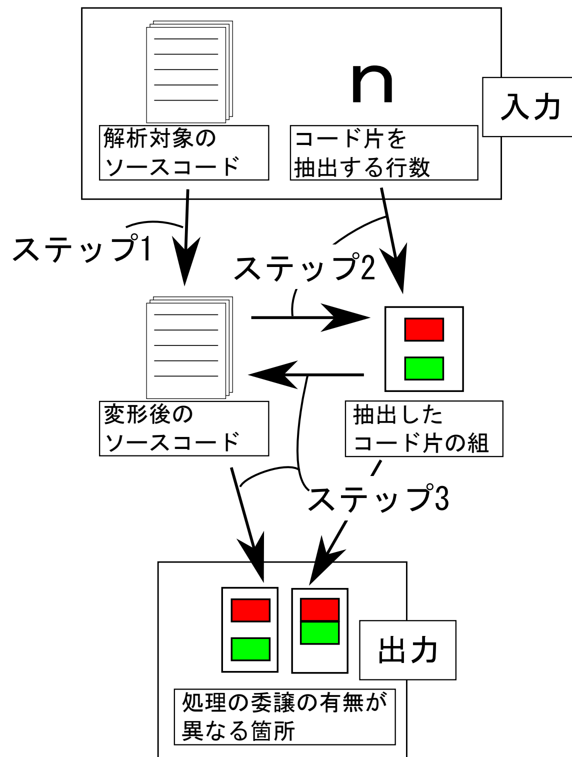


図 4: 提案手法全体の流れ

3　提案手法

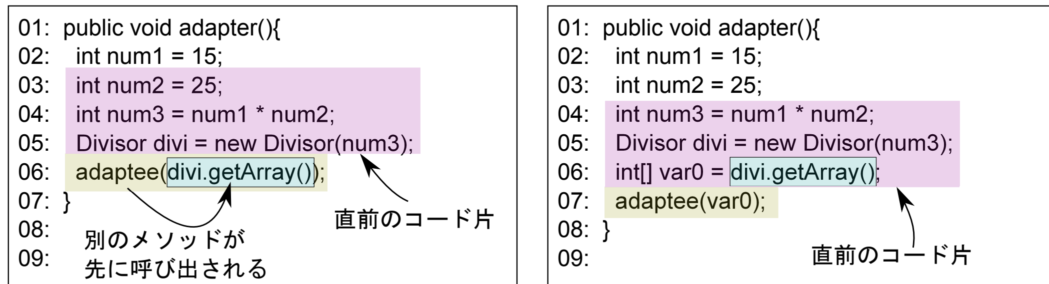
3.1　概要

本研究で提案する手法はソースコード中から処理の委譲の有無が異なるコードを特定する．提案手法では，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する．そして，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．これにより，処理の委譲の有無が異なるコードを検出することができる．

提案手法は以下の 2 つを入力とする．

- 検出対象となるソースコード
- 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

提案手法の全体の流れを図 4 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．



(a) 変形前

(b) 変形後

図 5: 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

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．

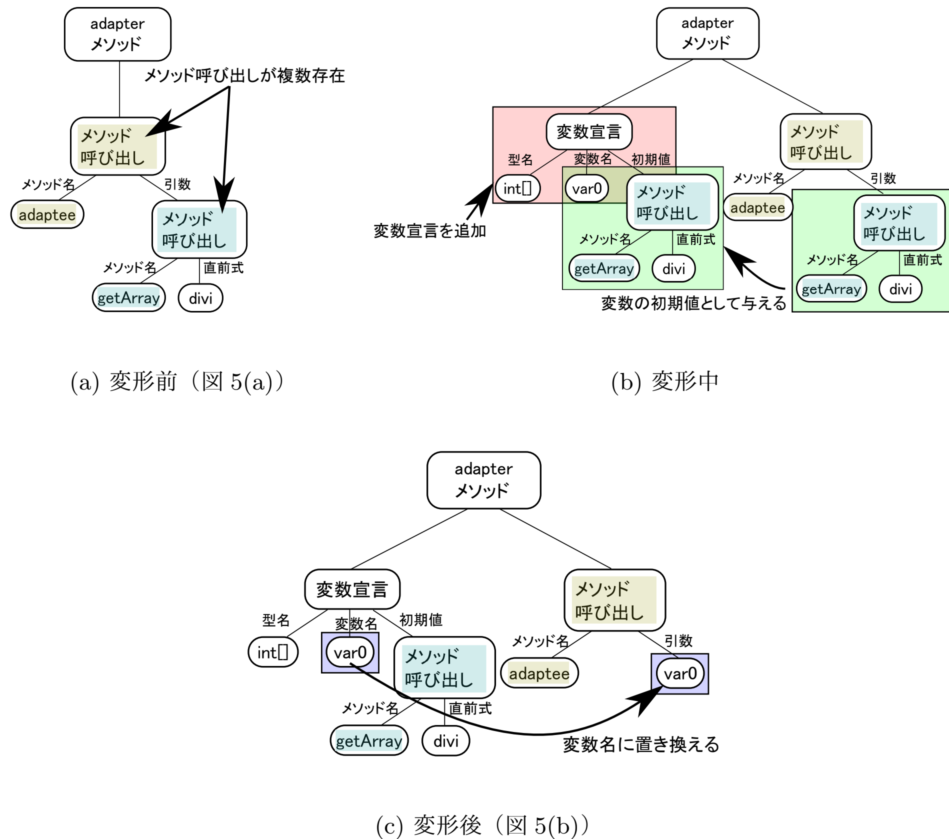
ステップ 3 抽出した 2 つのコード片をクエリとして，1 つのメソッド中に連続して一致する箇所を各メソッドから検出する．

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

このステップでは，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 5 にこのステップで変形の対象となるソースコードの例を示す．この例ではコード片を 3 行分抽出するものとしている．変形前の図 5(a) 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．図 5(a) の状態ではどちらのメソッド呼び出しについても，4 行目から 6 行目までを直前のコード片として抽出する．しかし，2 つのメソッドの実行は同時ではなく，getArray メソッドの呼び出しの後で adaptee メソッドが呼び出される．すなわち，adaptee メソッドの直前のコード片を抽出するとき，抽出するコード片に getArray メソッドの呼び出しを含ませる必要がある．この問題を解決するために，複数のメソッド呼び出しを含む文を変形して各メソッド呼び出しを別の文に切り離す．



(a) 変形前（図 5(a)）

(b) 変形中

(c) 変形後（図 5(b)）

図 6: 抽象構文木によるソースコードの変形

変形後の図 5(b) では getArray メソッドと adaptee メソッドがそれぞれ別の文に切り離され，1 つの文中に出現するメソッド呼び出しの数は高々1 つになっている．図 5(b) の状態では adaptee メソッドの呼び出しについて，4 行目から 6 行目までを直前のコード片として抽出し，これは getArray メソッドの呼び出しを含む．

ソースコードの変形の流れを図 6 に示す抽象構文木を用いて説明する．2 行目から 5 行目まではソースコードの変形には影響しないため省略している．図 6 に示す抽象構文木は図 6(a) が図 5(a) に，図 6(c) が図 5(b) に対応している．変形の手順は以下のようになる．

(a) 複数のメソッド呼び出しを含む文を変形の対象として特定する．

(b)1 変形を行う文が含むメソッド呼び出しを抽出する．

(b)2 抽出したメソッド呼び出しの返り値を初期値とする，新たな変数の宣言文を変形を行う文の直前に追加する．

(c) 抽出したメソッド呼び出しを追加した変数名で置き換えて文を変形する．

```
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:  }
09:
```

(b) 正規化後

図 7: 変数名とリテラルについての正規化を行う例

また，ユーザー定義名の違いを吸収するために，ソースコード中に現れる変数名とリテラルの正規化を行う．メソッド名と型名については正規化を行わない．

図 7 にこのステップにおいて正規化を行う例を示す．変数名は全て同一の字句に置き換えられ，リテラルについても型に対応する字句に置き換えられている．

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 8 にコード片を抽出する例を示す．この例では adaptee メソッドについて，コード片を 2 行分抽出するものとしている．図 8(a) で呼び出す adaptee メソッドを図 8(b) に示す．この adaptee メソッドの呼び出しについて，直前のコード片と，呼び出される adaptee メソッドの先頭のコード片を抽出する．抽出したコード片の組は図 8(c) に示すものになる．

本ステップの処理において，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ようにしている．図 9 に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を示す．この例ではコード片を 1 行分抽出するものとしている．図 9(a) の levelOne メソッドは図 9(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 9(c) の levelThree メソッドを呼び出している．このため，抽出するコード片は図 9(d) に示すように，levelOne メソッドと，levelThree メソッドのものとなる．

```
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:  }
09:
```

(a) 呼び出し元のメソッド

```
11:  public void adaptee(int[] numList){
12:    int $V = $N;
13:    for(int $V:$V) {
14:      $V += $V;
15:      System.out.println($V);
16:    }
17:  }
18:
```

(b) 呼び出されたメソッド

```
Divisor $V = new Divisor($V);
int[] $V = $V.getArray();
```

```
int $V = $N;
for(int $V:$V) {
```

(c) 抽出するコード片

図 8: メソッド呼び出しについてコード片を抽出する例

```
01:  public void levelOne(){
02:      int num = 0;
03:      int one = levelTwo(num);
04:      System.out.println(one);
05:  }
```

(a) 呼び出し元のメソッド

```
11:  private int levelTwo(int num){
12:      int two = levelThree(num)*2;
13:      return two;
14:  }
```

先頭の文がメソッド呼び出しを含む

(b) 呼び出されたメソッド

```
21:  private int levelThree(int num){
22:      int three = num*3;
23:      return three;
24:  }
```

この部分を取り出す

(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

```
int num = 0;
```

```
int three = num*3;
```

(d) 抽出するコード片

図 9: 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

```
21:  public void condense(int $V,int $V){
22:    int $V = $V * $V;
23:    Divisor $V = new Divisor($V);
24:    int[] $V = $V.getArray();
25:    int $V = $N;
26:    for(int $V:$V) {
27:      $V += $V;
28:    }
29:    System.out.println($V);
30:  }
```

(a) 検出された箇所

```
Divisor $V = new Divisor($V);
int[] $V = $V.getArray();
```

```
int $V = $N;
for(int $V:$V) {
```

(b) 抽出したコード片

図 10: 抽出したコード片から一致する箇所を検出する例

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 10 に検出の例を示す．図 10(b) に示すコード片は，ステップ 2 の例で使用した図 8(c) のものである．このコード片は図 10(a) の中に連続して現れている．

4　実装

4.1　概要

ツールは Java 1.7 を用いて実装した．本ツールでは Java のソースコードのみを検出の対象とする．また，本ツールは “Java Development Tools”(JDT) というライブラリ [20] を用いて生成された抽象構文木を使用することでメソッド呼び出しや識別子などを判別している．各メソッド呼び出しについて，呼び出されるメソッドの特定は同ライブラリの Binding という機能を使用することで実現している．

4.2　検出結果の表示

ツールの実行結果は図 11 に示すような GUI で確認できる．GUI の左側にはステップ 2 で抽出したコード片が黄色でハイライトされて表示され，右側にはステップ 3 で同じコード片から検出されたコード片が黄色で表示される．また，青色でハイライトされた部分がコード片の抽出に使用したメソッド呼び出しと，呼び出されるメソッドの宣言部を示す．左右のコードが処理の委譲の有無が異なるコードとなる．下のボタンによって表示するコードを切り替えることができる．

ShowPair

org.apache.tools.ant.filters.StripLineBreaks.read()

```
}
public StripLineBreaks(final Reader in){
    super(in);
}
public int read() throws IOException{
    if(!getInitialized()) {
        initialize();
        setInitialized(true);
    }
    int ch=in.read();
    while(ch!=-1) {
        int Variable0=lineBreaks.indexOf(ch);
```

org.apache.tools.ant.filters.ClassConstants.read()

```
public ClassConstants(final Reader in){
    super(in);
}
public int read() throws IOException{
    int ch=-1;
    int Variable0=queuedData.length();
    if(queuedData!=null&&Variable0==0) {
        queuedData=null;
    }
    if(queuedData!=null) {
        ch=queuedData.charAt(0);
        queuedData=queuedData.substring(1);
```

passed org.apache.tools.ant.filters.SimpleFilterReader.read()

org.apache.tools.ant.filters.SuffixLines.read()

```
import java.io.IOException;
import java.io.Reader;
import org.apache.tools.ant.types.Parameter;
public final class SuffixLines extends BaseParamFilterReader imp
    private static final String SUFFIX_KEY="suffix";
    private String suffix=null;
    private String queuedData=null;
    public SuffixLines(){
        super();
    }
    public SuffixLines(final Reader in){
        super(in);
    }
    public int read() throws IOException{
        if(!getInitialized()) {
            initialize();
            setInitialized(true);
        }
        int ch=-1;
        int Variable0=queuedData.length();
        if(queuedData!=null&&Variable0==0) {
            queuedData=null;
        }
        if(queuedData!=null) {
            ch=queuedData.charAt(0);
            queuedData=queuedData.substring(1);
            int Variable1=queuedData.length();
            if(Variable1==0) {
                queuedData=null;
```

<< 4/10 >> pairID 4

図 11: 出力の例

5　評価実験

提案手法を実装したツールを用いて，以下の 2 つの実験を行った．

実験 1 複数の開発者による被験者実験

実験 2 オープンソースソフトウェアに対する実験

5.1　実験 1:複数の開発者による被験者実験

複数の開発者による被験者実験を行う目的は，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である．

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．このため，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．

5.1.1　実験対象

被験者実験に使用したソースコードの規模を表 1 に示す．これらのソースコードはすべて，異なる開発者が Java で実装したものである．

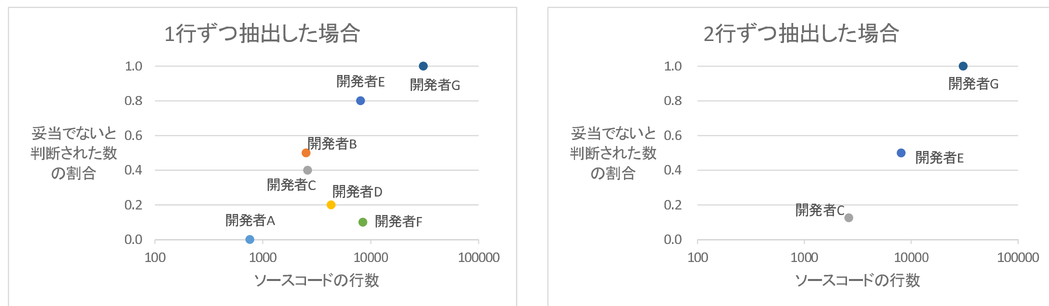
5.1.2　実験方法

実験の準備として，あらかじめ各ソースコードに対する本ツールの実行結果を用意しておく．このとき，コード片を抽出する行数を 1 から 3 までに設定し，実行結果を個別に保存する．また，実験にかかるコストを削減するため，検出されたコードが 10 箇所を超える場合は 10 箇所をランダムに抽出して使用する．

開発者には自身の開発したソースコードから検出されたコードを確認してもらい，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．妥当かを判断する基準は “処理

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 12: 被験者実験の結果：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

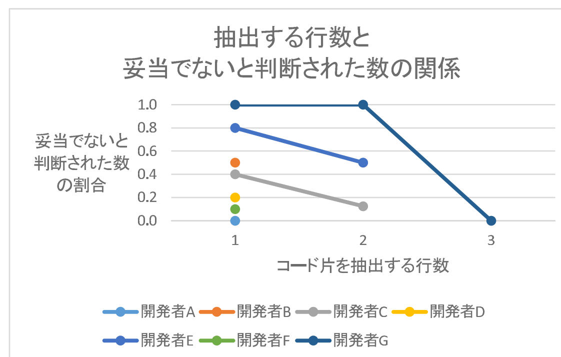


図 13: 被験者実験の結果：コード片を抽出する行数と妥当でないと判断された数の割合の関係

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とする．

判断してもらった結果を集計し，開発者ごと，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．

5.1.3　実験結果

実験結果を表 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，ソースコードによってはコードが検出されなかった．したがって，表 2 中の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コードが検出されなかった．

また，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 12 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

```
255:  String Variable55=sb.toString();
256:  int Variable56=node.getStartPosition();
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

```
082:  MethodInfo info=map.get(binding);
083:  if(info==null) {
084:  int Variable8=id.getAndIncrement();
085:  info= new MethodInfo(Variable8);
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:  }
094:  info.setNode(node);
```

(c) 検出された箇所

図 14: 妥当だと判断された検出結果の例

図 13 に示す．図 12(a) は 1 行ずつ抽出した場合で，図 12(b) は 2 行ずつ抽出した場合である．3 行ずつ抽出した場合は開発者 G からのみコードが検出されているため省略する．

この実験結果から，コード片を抽出する行数が増えるほど，妥当でないと判断された数の割合が低くなることが分かる．すなわち，コード片を抽出する行数を増やした方がリファクタリング対象として妥当なコードが検出できるといえる．

5.1.4 検出結果の例

検出されたコードの例を図 14，図 15 に示す．図 14 は開発者 F のソースコードに対して，コード片を抽出する行数を a に設定した場合の検出結果であり，リファクタリング対象として妥当だと判断された．また，図 15 は開発者 G のソースコードに対して，コード片を抽出する行数を 1 に設定した場合の検出結果であり，リファクタリング対象として妥当でないと判断された．

図 14 の例では，検出箇所は Block クラスのオブジェクトを取得した後，これが null であるかを確認している．図 14(a) ではオブジェクトに対する処理を図 14(b) に示す accept メソッドに委譲して実装しているが，図 14(c) ではメソッドに委譲しないで実装している．この例では検出箇所は類似した行っており，Block オブジェクトに対する処理をメソッドに抽出したほうが良いことを開発者に確認した．したがって，本研究の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リファクタリングの候補として提示できることが確認できた．

図 15 の例では，検出箇所は整数値の加算の後にオブジェクトが null であるかを確認し

表 2: 被験者実験の結果

	開発者	A	B	C	D	E	F	G
1 行ずつ抽出した場合	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 行ずつ抽出した場合	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 行ずつ抽出した場合	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

```
953:   Label[] values=new Label[len];
954:   u+=8;
955:   for(int i=0;i<len;++i) {
956:     keys[i]=readInt(u);
957:     int Variable98=readInt(u+4);
958:     values[i]=labels[offset+Variable98];
959:     u+=8;        括弧閉じは無視
960:   }
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;
304:    this.stack=null;
305:  }
```

(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 15: 妥当でないと判断された検出結果の例

ている．しかし，検出箇所の前を見ると，図 15(a) では for 文による配列の操作を行い，図 15(c) ではフィールド変数の加算を行っている．検出箇所の後を見ると，図 15(b) では visitLookupSwitchInsn メソッドを呼び出し，図 15(c) では visitInsn メソッドを呼び出している．このように，検出箇所の前後では別の処理が行われている．また，図 15 のコードは偶然一致しただけで，異なる処理を行っていることを開発者に確認した．このように，提案手法によって検出されたコードの中には，異なる処理を実装している箇所が存在することが確認できた．

5.2 実験 2:オープンソースソフトウェアに対する実験

オープンソースソフトウェアに対する実験を行う目的は，ある程度大きな規模のソースコードに対して提案手法を適用した場合に検出されるコードの数を調べるためである．

5.2.1 実験対象

本実験では，Apache Ant(v1.9.2)[21] を対象とした．このソフトウェアの規模は，総行数が 216,486 行でファイルの数が 853 個である．

5.2.2 実験結果

Apache Ant に対する実験結果を表 3 に示す．実験 1 と同様，4 行ずつ抽出した場合はコードが検出されなかった．この実験結果から，コード片を抽出する行数を増やすと検出される数が減少することが分かる．特に 1 行ずつ抽出した場合は 2 行ずつ，3 行ずつ抽出した場合に比べて検出された数が非常に多いことが確認できる．

表 3: Apache Ant に対する実験結果

	検出された数
1 行ずつ抽出した場合	37,117
2 行ずつ抽出した場合	55
3 行ずつ抽出した場合	10

6　考察

実験 1 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 15 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 の確認やメンバ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が挙げられる．

7　妥当性について

本研究の結果の妥当性について，以下で挙げる点に注意する必要がある．

コード片の変形方法 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，検出されるコードが変化すると考えられる．

実験対象 本研究での実験に使用したソースコードは，どれも個人によって開発されている．2.2.2 節で述べたように，複数人で開発したソースコードは開発者によって実装方針が異なるため，委譲の有無の有無が異なる可能性が高いと考えられる．このような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．

検出結果の抽出 本研究で行った被験者実験において，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．

8　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の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と，オープンソースソフトウェアに対する実験を行った．実験の結果，コード片を抽出する行数をできるだけ増やして検出された箇所から順番に調べることが望ましいことが確認できた．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コードが確認できた．

今後の課題としては，引数が null であるかの確認や，配列の添字が正常であるかの確認など，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に選択できるようにするなど，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．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました，楠本 真二 教授に心から感謝申し上げます．

本研究に関して，有益かつ的確なご助言を頂きました，岡野 浩三 准教授に深く感謝申し上げます．

本研究において，多大なるご助言を頂きました，井垣 宏 特任准教授に深く感謝申し上げます．

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました，肥後 芳樹 助教に深く感謝申し上げます．

本研究を行うにあたり，多大なるご助言，ご助力を頂きました，堀田 圭佑 氏に深く感謝申し上げます．

その他の楠本研究室の皆様からいただいたご助言やご協力に心より感謝致します．

また，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[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. In *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. In *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] Y. Higo, K. Hotta, and S. Kusumoto. Enhancement of crd-based clone tracking. In *Proc. of the 13th International Workshop on Principles of Software Evolution*, pp. 28–37, Aug. 2013.

[9] M. Balazinska, E. Merio, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. of the 6th international Software Metrics Symposium*, pp. 292–303, Nov. 1999.

[10] M. Weiser. Profram slicing. *IEEE Trans. on Software Engineering*, Vol. 10, No. 4, pp. 352–357, July 1984.

[11] 丸山勝久. 基本ブロックスライシングを用いたメソッド抽出リファクタリング. 情報処理学会論文誌, Vol. 43, No. 6, pp. 1625–1637, June 2002.

[12] 丸山勝久, 大森隆行. 利便性の高いリファクタリングの取り消しメカニズム. 情報処理学会論文誌, Vol. 48, No. 8, pp. 2663–2673, Aug. 2007.

[13] 若林智徳, 松浦佐江子. Java プログラミング初学者のクラス構成学習のための依存関係解析. 電子情報通信学会技術研究報告, Vol. 110, No. 386, pp. 55–60, Jan. 2011.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローンを対象としたリファクタリング支援環境. 電子情報通信学会論文誌, Vol. J88-D-I, No. 2, pp. 186–195, Jul. 2005.

[20] *Java development tools*. http://www.eclipse.org/jdt/.

[21] *Apache Ant(v1.9.2)*. http://ant.apache.org/.