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.

プログラム依存グラフを用いたリファクタリング候補の特定と可視化

兼光　智子†　肥後　芳樹†　楠本　真二†

† 大阪大学 大学院 情報科学研究科

あらまし　リファクタリングは，保守の効率化において重要であるが，手動によるリファクタリングは誤りが混入しやすいため，その作業を自動化する手法及びツールが必要である．本稿では，メソッド抽出リファクタリングの候補を支援する提示する手法を提案する．メソッド抽出リファクタリングとは，メソッドの一部を別のメソッドとして切り出すことである．これまでの研究により，メソッド抽出リファクタリングは他のリファクタリングの前に頻繁に行われることが示されており，メソッド抽出リファクタリングを支援することは重要である．既存研究では行数や複雑さを基にメソッド抽出リファクタリングの候補を提示するが，本来メソッドは機能に基づいて分割することが望ましい．本稿では，文の間のデータの繋がりに着目する．データの繋がりの強い部分が一つの機能を表すと考え，メソッド抽出リファクタリングの候補を求め，自動的に提示する手法を提案する．提案手法では，リファクタリング可能なところを特定するだけでなく，リファクタリングすべきところを提示する．

キーワード　保守　リファクタリング プログラム依存グラフ

Identifying and Visualization Refactoring Candidates for Extract Method using Program Dependence Graph

Tomoko KANEMITSU†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University

Abstract　Refactoring is important for efficient software maintenance. However, tools supports are highly required for performing refactoring because manual operations in refactoring are troublesome and error prone. This paper proposes technique that indicates candidates of "Extract Method" refactoring automatically. Extract Method refactoring is to create a new method from a code fragment in an existing member. It is shown that the Extract Method refactoring is performed prior to other refactoring, and it is important to support Extract Method refactoring. The existing study proposes candidates of Extract Method refactoring based on linage and complexity. However it is originally desirable to divide methods based on their functionalities. This paper use the strong connection of data between sentences. We deem that strongly-connected data expresses a single function. This paper proposes technique that indicates candidates of Extract Method refactoring based on strongly-connected data.

Key words　Maintenance Refactoring Program Dependence Graph

1. まえがき

ソフトウェアの設計品質はソフトウェアの開発や保守を効率的に行うための重要な因子である．近年，ソフトウェアの応用分野の拡大と共にソフトウェアが大規模・複雑化し，ソフトウェアの保守に要するコストが増加してきている．このため，ソフトウェアの設計品質の重要度はますます高まっている．しかし，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．このような問題に対処するために，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ったまま，ソフトウェアの内部構造を改善することによりソフトウェアの設計品質を高める技術である [1]．

しかしながら，手動によるリファクタリングは次の問題を抱えている．どのようなときリファクタリングが必要であるかを示す厳密な基準が存在しないため，この判断には多くの経験や知識を必要とする [1]．大規模ソフトウェアから手動でリファクタリングすべき箇所を特定するのは非常に手間がかかる．特定した箇所をどのようにリファクタリングすべきか決定するには多くの経験や知識を必要とする [1]．これらの問題を軽減するためには，リファクタリング作業を支援する手法が必要であ

る．本稿では，代表的なリファクタリングパターンの一つであるメソッド抽出リファクタリングを支援する手法を提案する．メソッド抽出とは，既存のメソッドから適度な規模のコード片を新しいメソッドとして抽出する作業である．

本稿では，リファクタリング候補を提案するためプログラム依存グラフ (Program Dependence Graph) を利用する．メソッド抽出リファクタリングを必要とするメソッドの候補として，長すぎるメソッドや制御ロジックが複雑なメソッドなどがあげられるが，本来メソッドは行数や複雑さではなく，機能に基づいて分割することが望ましい．本稿では，一つの機能を構成する文の間ではデータのつながりも強いと考え，プログラム依存グラフを利用しデータの強さを定義した．それを用いてメソッド抽出リファクタリングを必要とするメソッドの候補を検出する．また，既存研究ではリファクタリング候補の提示がソースコード上に示すものが多いが，本稿ではグラフを可視化して表示することによりデータのつながりや強さを視覚的に確認することが可能である．

2. 準　　備

2.1 プログラム依存グラフ

プログラム依存グラフ (Program Dependence Graph) とは，ソースコードの文をノードとしノード間の依存関係をエッジで表したグラフである．PDG を用いることで，データの流れなどプログラムの振る舞いを表現することができる．PDG では以下の二種類の依存関係を表す．

データ依存　次の全てを満たす場合，二つの文 s_1 から s_2 の間にはデータ依存が存在する．

- 文 s_1 で変数 v が定義されている
- 文 s_2 で変数 v が参照されている
- 文 s_1 から文 s_2 までの間の実行パスに変数 v が再定義されていないパスが存在する

制御依存　次の全てを満たす場合，二つの文 s_3 から s_4 の間には制御依存が存在する．

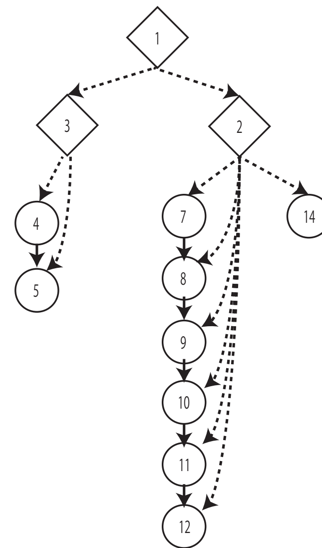
- 文 s_3 が条件節である
- 文 s_4 を実行するかどうかが，文 s_3 の結果に依存する

PDG の例を図 1 に示す．図 1(a) のソースコードを PDG で表したものが図 1(b) である．データ依存辺は実線で，制御依存辺は点線で表している．

通常の PDG では，オブジェクトのメソッド呼び出しは，全てそのオブジェクトへの参照となるが，本稿で用いる PDG は，オブジェクトの状態が変更となる文では，データの参照と代入が行われると考える．図 1 にて，例を示す．図 1(a) の 7 行目から 12 行目の文について考える．通常の PDG であれば，7 行目の文から，8 行目から 12 行目までそれぞれにデータ依存辺が存在する．つまり，7 行目から 5 本のデータ依存辺が出ることになる．しかし，本稿で用いる PDG では，8 行目から 11 行目のメソッド append の呼び出しにて，text の中の状態が変わるためそれぞれ変数 text の参照と代入と考える．図 1(b) のようにデータ依存辺が引かれる．

```
1:  String sample1(){
2:    if(this.trueOrFalse()){
3:      if(null == this.getPath()){
4:        Project proj = this.getProject();
5:        this.setPath(proj.getBaseDir());
6:      }
7:      StringBuilder text = new StringBuilder();
8:      text.append("String A");
9:      text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     return text.toString();
13:   }else{
14:     return "";
15:   }
16: }
```

(a) ソースコード



(b) PDG

図 1　ソースコードと PDG の例

2.2 メソッド抽出リファクタリング

メソッド抽出リファクタリングとは，メソッドの一部を新たなメソッドとして切り出すことであり，次の手順で実行される．

（1）　新たなメソッドとして抽出する文を選択する

（2）　選択した文をコピーし新たなメソッドとして定義する

（3）　元の文を削除し新たに定義したメソッドへの呼び出しに置き換える

メソッドが機能に基づき分解されることで保守性や再利用性が高まる．また，他のリファクタリングの前に実行されることも多く重要でありこれまで多くの研究がされている．

メソッド抽出可能な文の集合には以下の条件が必要である [1]．

- 選択範囲内から選択範囲外へのデータ依存辺が一本以下である
- 条件節内の return 文がない

```
01 public void method(int x){
02    int i = 0;
03    int t = 0;
04
05    i=x+1;
06    if (x>0){
07       i=i+2;
08       t=t+3;
09    }
10    System.out.print("i:"+i);
11    System.out.print("t:"+t);
12 }
```

(a) メソッド抽出前

```
01 public void method(int x){
02    int i = 0;
03    int t = 0;
04
05    i=x+1;
06    if (x>0){
07       t=t+3;
08    }
09    method2(x,i);
10    System.out.print("t:"+t);
11 }
12
13 public void method2(int x, int i){
14    if (x>0){
15       i=i+2;
16    }
17    System.out.print("i:"+i);
18 }
```

(b) メソッド抽出後

図 2 メソッド抽出の際，文が複製される場合

- break や continue などによる選択範囲外への実行フローがない

また，条件節の内外の文を同時に抽出対象とした場合，その条件節を元のメソッドと新しいメソッドの両方に複製する必要がある．図 2 に例を示す．図 2(a) がメソッド抽出する前とし，7 行目と 10 行目を新たなメソッドとして抽出する場合を考える．6 行目の if 文は，7 行目の実行に必要なので 6 行目も新たなメソッドとして抽出する範囲に含める．しかし同様に 6 行目の if 文は，8 行目の実行に必要なので，元のメソッドにも残す必要がある．よって，6 行目の if 文は元のメソッドと新しいメソッドの両方に複製されることとなり，メソッド抽出後は図 2(b) のようになる．

3. 関 連 研 究

Murphy-Hill らは，メソッド抽出リファクタリングを視覚的に補助するツールを提案した [1]．提案したツールは，Selection Assist，Box View，Refactoring Annotations の三つである．Selection Assist は，エディタで文を選択する際，if 文の全範囲など完全な文の範囲をハイライトで表示するだけであり，利用者が手動でソースコードを選択する必要がある．複数行にまたがる文や if 文の範囲を容易に確認することが可能である．Box View は，コードのネスト構造をボックスで表示する．エディタで文を選択すると対応するボックスの色が変化し，ボックスを選択すると対応する文が選択状態となる．Refactoring Annotations は，ソースコード上にコントロールとデータのフローを矢印で表示する．実験により，彼らの提案した視覚的な補助ツールを使用した方が，より早く正確にリファクタリングが行えた．しかし，それらのツールでメソッドとして抽出する文の選択は，人がソースコードを見て考える必要があり，またその選択はソースコード上で連続した文に限られている．

また，Murphy-Hill らは，開発時実際に行われたリファクタリングの内容を調査してまとめた [2]．その結果，ある程度まとまった開発を行った後集中してリファクタリングを行うより，開発の途中にリファクタリングを何度か行う場合の方が多いことが分かった．

丸山らは，基本ブロックに基づきメソッド抽出リファクタリングの候補を自動的に抽出する手法を提案した [3]．プログラム全体からではなく，プログラムの基本ブロックで構成される領域（部分ブロック）からコードを抽出する．この手法では，プログラマはリファクタリング対象のコードにおいて着目する変数を指定し，ツールにより生成された候補から適切な候補を選択する．実験により，単一の基本メソッドから異なる大きさや引数を持つさまざまなメソッドを抽出することが出来ることを確認した．

Tsantalis らは，丸山らの手法を発展させ，プログラム全体に対して適切なリファクタリング候補を自動的に提示する手法を提案した [4]．丸山らの手法では，メソッド抽出後のプログラムが振る舞いを保たない場合を指摘し，そのような候補を除外する．また，丸山らの手法では着目する変数を指定する必要があるが，Tsantalis らの手法では，プログラム中の全てのメソッド全ての変数に対してスライスの候補を計算する．そのため，開発者は解析前に変数を指定する必要がなく，解析後に出力された複数のリファクタリング候補から適切なものを選ぶだけでよい．実験により，システム設計者が実際にリファクタリングを行いたい候補を提示できたことを示した．

4. 提 案 手 法

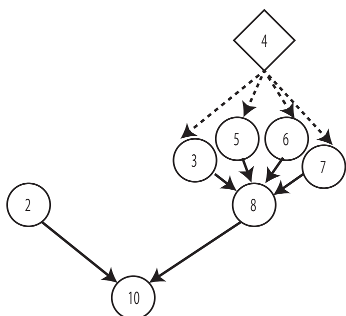
PDG の可視化によるメソッド抽出リファクタリングの候補を提示する手法を提案する．メソッドは機能別に分割されるべきである．ノード間のデータ依存辺が多いノードの集合が一つの機能を表すと考えた．PDG の形状から各ノード間のデータ依存の強さを推測し，それを可視化時のノード間の距離として反映させる．データのつながりが強いほどノード間の距離を短くする．そして，ノード間の距離が近く密集しているノードの集合を，メソッドとして抽出すべき文の集合とする．ノード間の距離を定義したグラフを表示することで，リファクタリングを実行して適切か視覚的に判断することが出来る．

図 3 に提案手法の例を示す．図 3(a) のソースコードを，PDG とし提案手法によりノードを配置したものが図 3(b) である．8 行目にはデータ依存辺が 4 本入っており，10 行目にはデータ依存辺が 2 本入っている．8 行目に入るデータ依存辺の本数の方が多いため，そのデータ依存辺の元の 3,5,6,7 行目のノードが 8 行目のノードに近く配置されている．10 行目へ入るデータ依存辺の元である 2,8 行目のノードは離れて配置されている．この場合，ノードが密に配置されている 3,5,6,7,8 行目をメソッド抽出リファクタリングの候補とする．

以下にノード間の距離として具体的に三種類のパラメータを定義する．これらの関係にあるノードは別のメソッドとして分けるべきではないと考えられる．$v_i (i = 1, 2, 3)$ は各パラメータの影響を表す定数であり，ユーザがツール上で変更すること

```
01 public void method(int x, objectA test){
02     int i = 0;
03     int t = 0;
04     if (x>0){
05         int a = test.getA();
06         int b = test.getB();
07         int c = test.getC();
08         i = t + a + b + c + 1;
09     }
10     System.out.println(i);
```

(a) ソースコード



(b) PDG

図 3　提案手法による PDG の配置

が可能である．

4.1 Atomic Data Dependency

ある文で定義された変数が一度しか参照されていない場合．

例えば，次のような変数 tmp を考える．

```
final java.lang.Object tmp = objectA;
objectA = objectB;
objectB = tmp;
```

データ依存辺が一つのみ存在する場合であり，そのデータ依存辺の両端のノードは同じメソッドにまとめておくと考えられる．この依存関係を Atomic Data Dependency(ADD) と定義し，この関係にあるデータ依存辺の長さを短くする．ADD のデータ依存辺の長さを以下のように定義する．

$$distance_{ADD} = v_1 \tag{1}$$

4.2 Spread Data Dependency

ある文で定義した変数を他の多くの文で使用している場合．

例えば，次のような変数 `tax_rate` を考える．

```
float tax_rate = 0.05;
int taxA = articleA.getPrice() * tax_rate;
int taxB = articleB.getPrice() * tax_rate;
int taxC = articleC.getPrice() * tax_rate;
```

よく参照される変数は重要な値であり，その参照関係はつながりが強いと考える．この依存関係を Spread Data Dependency(SDD) と定義し，この関係にあるデータ依存辺の長さを短くする．SDD のデータ依存辺の長さを以下のように定義する．n は，一つの文から出るデータ依存辺の数である．

$$distance_{SDD} = \frac{v_2}{n} \tag{2}$$

4.3 Gathered Data Dependency

多くの変数をある文で参照している場合．

例えば，次のような変数 a,b,c を考える．

```
int a = coefficients.getA();
int b = coefficients.getB();
int c = coefficients.getC();
Answer ans =
    QuadraticFormula.getAnswer(a, b, c);
```

このようにメソッド呼び出しの準備などまとまった機能を表していると考えられ，それらのつながりは強いと考える．この依存関係を Gathered Data Dependency(GDD) と定義し，この関係にあるデータ依存辺の長さを短くする．GDD のデータ依存辺の長さを以下のように定義する．m は，一つの文へ入るデータ依存辺の数である．

$$distance_{GDD} = \frac{v_3}{m} \tag{3}$$

5. 実　　装

提案手法をツール ReAF(Refactoring Automated Finding) として実装した．入力は，対象とするソースコードを含むフォルダであり，対象言語は，Java である．ツールを実行すると，メソッド単位の PDG が表示される．グラフ内の各エッジの長さは上記パラメータに基づいている．構文解析部には MASU(注1)を，グラフの可視化には Jung(注2)を使用している．

ツールの外観を図 4 に示す．左端にクラスとメソッドを階層表示しており，そこから一つメソッドを選択する．選択すると右端にそのメソッドの PDG グラフが表示され，中央にはソースコードが表示される．グラフ上部のパラメータで微調節し，ノードが密集している範囲を選択する．選択したノードの集合が新たなメソッドとして抽出可能な場合は「OK」と表示し，抽出不可能な場合は「NG」の表示とともに抽出不可能となる原因を強調表示する．また，選択したノードに対応するソースコード上の文を強調表示する．

グラフでのノードの大きさは，ネストが深いものほど小さく表示しており，制御構造も視覚的にわかりやすく表示している．

図 4 にて適用したソースコードは，M. Fowler によるリファクタリングに関する本の中でも使用されている例である [5]．クラス Customer の中のメソッド statement のグラフを表示して

(注1)：http://sourceforge.net/projects/masu/
(注2)：http://jung.sourceforge.net/

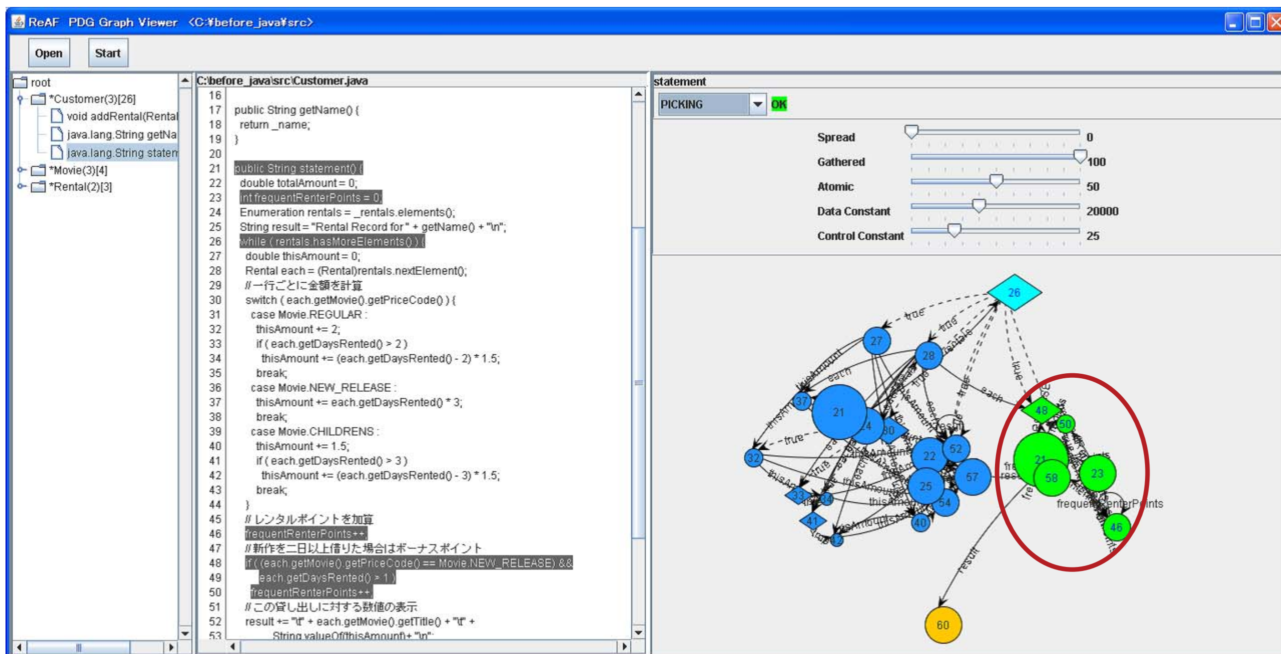


図 4 ReAF

いる．レンタルショップの料金計算のうち，レンタルポイントに関連する部分がグラフの右に密な部分として提示できた．

6. 実 験

6.1 実 験 対 象

コンピュータサイエンスを専攻している学部四年から博士後期課程三年の学生 14 人に対して ReAF と既存ツールとの比較実験を行った．半分の 7 人は自分が研究で作成している Java のソースコードに対して適用した．自分が作成したソースコードなので内容に熟知しており，開発の途中にリファクタリングを行うことを想定している．残りの 7 人は著者らで指定したソースコードに対して適用した．対象としたソースコードは他の学生が研究用に開発したソースコードであり，他人の書いたソースコードに対してリファクタリングすることを想定している．

6.2 比 較 対 象

比較対象とする既存ツールは，関連研究で述べた Tsantalis らの手法を用いた「JDeodrant」である [4]．基本ブロックに基づいたスライスにより，リファクタリング候補を提示する．Tsantalis らにより Eclipse のプラグインとして実装されている．対象とするプロジェクトやクラスを指定し，解析開始ボタンを押すとリファクタリング候補を表に一覧として提示する．その中から一つ候補を選択すると，その候補で抽出対象となる文がソースコード上でハイライト表示になる．さらにその状態でリファクタリングボタンを押すと，リファクタリング前後のソースコードを比較でき，よければそのままリファクタリングを実行することも可能である．

6.3 実 験 方 法

あらかじめ，各自がリファクタリングを行いたいメソッドを三つ用意し，その同じメソッドに対して二つのツールを適用する．その結果を見て，そのメソッドに対するリファクタリング候補は提示されたか，提示された場合は実際にリファクタリングを実行したいか考える．ツールの実行順序は，半分が先に ReAF を残りの半分が先にプラグインを実行した．

その後，各ツールの提示した候補についてやツールの使用感についてアンケートを行った．アンケートの項目は，以下の三点についてそれぞれ四段階で質問した．

- 可視化による提示がリファクタリング候補を見つけるのに役に立ったか
- ツールの操作性はよかったか
- リファクタリング候補を見つけるのにそのツールを再び利用したいか

6.4 実 験 結 果

6.4.1 リファクタリング候補の特徴

ツールにて表示された候補のメソッドごと内容を表 1 に示す．メソッド単位で提示されたリファクタリング候補について，被験者が実際にリファクタリングを行いたいものであった場合，有用な候補提示としている．ReAF は，無用な候補より有用な候補提示が多く，また JDeodorant より多く有用な候補を提示することが出来た．

また，ReAF が良い結果であったメソッド，JDeodorant が良い結果であったメソッドについて行数を比較した．ReAF が良い結果であったメソッドとは，ReAF が有用な候補を提示し

表 1 ツールの提示したリファクタリング候補内容ごとのメソッド数

ツール	提示された候補の内容	メソッド数
ReAF	有用な候補提示	27
	無用な候補提示	17
JDeodorant	有用な候補提示	24
	無用な候補提示	8
	候補提示なし	12

JDeodorant が無用な候補を提示または候補提示がなかったメソッドであり，JDeodorant が良い結果であったメソッドとは，ReAF が無用な候補を提示し JDeodorant が有用な候補を提示または候補提示がなかったメソッドである．結果を表 2 に示す．JDeodorant は全体では平均行数が 63.286 であったが，一つだけ外れて 270 行のものがあり，それを除くと残りは 103 行以下で平均は 47.385 となった．ReAF は，文をノードとして表示するためメソッドが長いほうがノード数が多くなる．そのため，ノードの疎密が明白になり良い候補を提示できたと考えられる．

6.4.2 使用感想

各ツールを使用した感想のアンケート結果を表 3 に示す．ReAF では，可視化の評価が高かったが操作性の評価が低い．可視化の評価については，14 人中 12 人がグラフによる可視化が直感的で候補を見つけやすいと回答した．操作性の評価が低い原因は，グラフ操作や範囲選択の問題など本質的な機能以外のユーザビリティによるものであり，それらの改善によって評価は上がると考えられる．JDeodorant では，可視化・操作性ともに評価が高かった．しかし，操作性の評価が高い理由は，eclipse の慣れた画面で操作が行えるなどプラグインに起因したものであった．さらに JDeodorant では，一文含むか含まないかだけで他は同じ文などの似たような候補が多く提示され，確認する際大変であるという意見もあった．ReAF では，同じ範囲はグラフで確認できるためそのような候補を提示することはない．

6.4.3 時間

各ツールでの解析時間と候補を提示してからリファクタリングを行うか決定するまでの時間を比較した．結果を表 4 に示す．JDeodorant の時間には，候補提示がないメソッドに関する時間は含んでいない．ReAF の方が解析時間は短く，決定時間と時間合計は JDeodorant の方が短い．ReAF の解析時間はソースコード全体の PDG 構築時間であるが，JDeodorant ではさらに各メソッド内の変数ごとにスライスを計算するので時間がかかる．どちらも解析時間より決定時間の方が長かった．時間がかかる原因として，ReAF はエッジの長さを決めて初めからノードを見やすい位置に配置できないので，グラフを整えるのに時間がかかる．グラフを整えるための時間は，グラフの操作性の悪さからさらに時間がかかっていると考えられる．JDeodorant は，メソッドによっては出力候補が多く全てを確認するのに時間がかかる．

表 2 ツールによる差があるメソッドの比較

メソッド行数	平均値	最小値	最大値
ReAF	68.833	25	178
JDeodorant	63.286(47.385)	16	270(103)

表 3 アンケート評価結果

	ReAF			JDeodorant		
評価数値	可視化	操作性	継続利用	可視化	操作性	継続利用
4	4	1	2	4	5	4
3	9	3	7	10	6	6
2	1	9	4	0	3	3
1	0	1	1	0	0	1
数値平均	3.214	2.286	2.714	3.286	3.143	2.9286

7. あとがき

本稿では，プログラム依存グラフを利用しメソッド抽出リファクタリングの候補を提示する手法を提案した．具体的には，データ依存辺を用いデータのつながりの強さを定義し，データのつながりの強い文の集合をメソッド抽出の候補とする．提案手法を用いてリファクタリングの支援を行うツールを開発し，学生に対して実験を行った．その結果，グラフ操作など操作性の点で問題がありリファクタリング決定時間は長かったが，短い解析時間で結果を出力することができ，14 人中 12 人からグラフによるリファクタリング候補提示は直感的でわかりやすいという意見が得られた．

今後の課題として，グラフ操作や柔軟な選択方式などユーザビリティの改良があげられる．また，グラフ上で密な部分やユーザーが別のメソッドとして分けたくない文同士のノードを一つにまとめる機能を追加することを考えている．

謝辞

本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，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) および (C)(課題番号:20500033)，文部科学省科学研究費補助金若手研究 (B)(課題番号:22700031) の助成を得た．

文献

[1] E. Murphy-Hill and A. Black: “Breaking the barriers to successful refactoring: Observations and tools for extract method”, Proceedings of the 30th international conference on Software engineeringACM New York, NY, USA, pp. 421–430 (2008).

[2] E. Murphy-Hill, C. Parnin and A. Black: “How we refactor, and how we know it”, Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference onIEEE, pp. 287–297 (2009).

[3] K. Maruyama: “Automated method-extraction refactoring by using block-based slicing”, Proceedings of the 2001 symposium on Software reusability: putting software reuse in contextACM, pp. 31–40 (2001).

[4] N. Tsantalis and A. Chatzigeorgiou: “Identification of extract method refactoring opportunities”, European Conference on Software Maintenance and ReengineeringIEEE, pp. 119–128 (2009).

[5] M. Fowler and K. Beck: “Refactoring: improving the design of existing code”, Addison-Wesley Professional (1999).

表 4 ツールの解析時間とリファクタリング決定時間

時間	ReAF			JDeodorant		
(s)	解析時間	決定時間	合計	解析時間	決定時間	合計
最大値	35.3	1162	1170	223	600.8	621.2
最小値	0.02	20	23.7	0.01	12.4	16.0
平均	8.2	305.1	313.4	35.6	142.8	178.4