修士学位論文

題目

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自動特定と可視化

指導教員
楠本 真二 教授

報告者
兼光 智子

平成 23 年 2 月 7 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 22 年度 修士学位論文

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
自動特定と可視化

兼光 智子

内容梗概

リファクタリングとはソフトウェアの外部的な振る舞いを保ちつつ，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．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．本研究では，メソッド抽出リファクタリングの候補を支援し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．本研究では，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 1 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部分を特定するだけでなく，リファクタリングすべき部分を提示する．

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，他のツールでは提示することのできなかった有用な候補を提示することができ，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．また，ツール改良の結果，自動的に有用なリファクタリング候補を提示することができ，ユーザビリティも向上した．

主な用語

保守
リファクタリング
プログラム依存グラフ
可視化

目 次

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な振る舞いを保ちつつ，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]．ソフトウェアシステムの運用開始後も，バグ修正や新たな機能の追加などでソースコードはしばしば書き換えられる．Eick らは，そのような繰り返しのソースコード変更によりソースコードの保守性が低下し，そのため保守コストが多くなると報告している [2]．リファクタリングにより保守性が低下することを防ぐことができる．

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 [1]．

- 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と豊富な知識を必要とする
- 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

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．本研究では，代表的なリファクタリングパターンの 1 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．メソッド抽出リファクタリングとは，既存メソッドの一部を新しいメソッドとして抽出する作業である．メソッドが適切に分割されることで保守性を高めることができ，メソッドの再利用性も向上し，ソフトウェア資産としての価値も高まる．また，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行われることが多く重要なリファクタリングである．

メソッド抽出リファクタリングの候補を示す際に利用されるいくつかのソフトウェアメトリクスがある．例えば，コード行数 (LOC) はそのようなメトリクスの 1 つである．コード行数の小さいメソッドの方が，より可読性や再利用性が高い傾向にある．複雑なコントロールフローを含むメソッドも同様にメソッド抽出リファクタリングの候補となる．McCabc は，コントロールフローの複雑さを表すメトリクスとしてサイクロマチック数を提案した [3]．しかし，メソッドとは本来機能別に分割されるべきであり，行数の長いメソッドや複雑度の高いメソッドが分割されるべきとは限らない．

本研究では，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．本研究では，1 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する．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，本研究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，以降 PDG) 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 [4]. 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 2 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，2 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

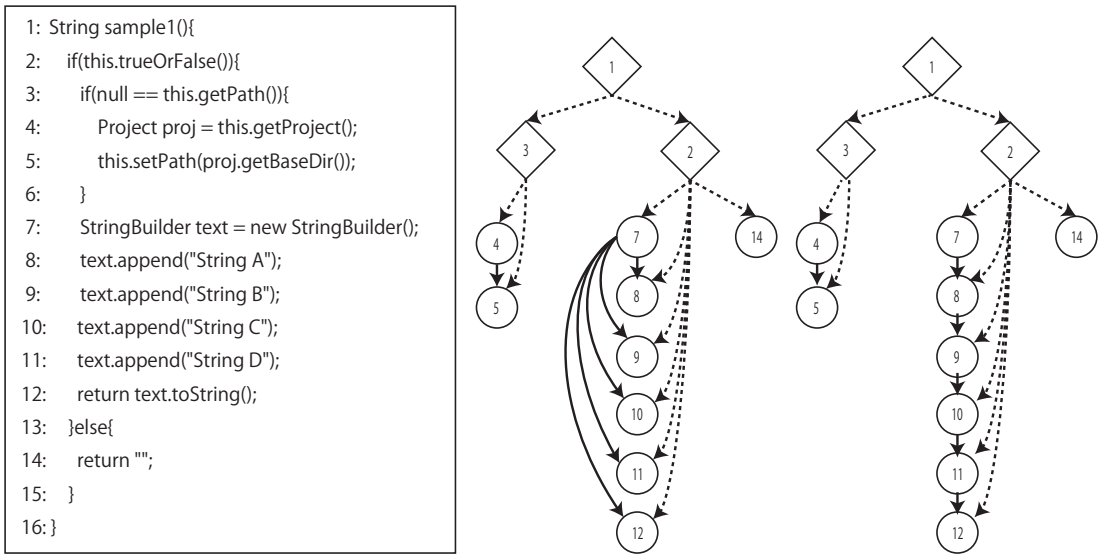
制御依存　次の全てを満たす場合，2 つの文 s_3 から s_4 の間には制御依存が存在する．

- 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) 及び図 1(c) 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．ノードのラベルは，対応するソースコードの行番号である．

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，本研究で用いる 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．通常の PDG の例を図 1(b)，本研究で用いる PDG の例を図 1(c) に示す．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，本研究で用いる 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(c) のようにデータ依存辺が引かれる．

例えば，9 行目を除き 8,10,11 行目をメソッドとして抽出することを考える．通常の PDG であれば，9 行目とそれらの行にデータ依存辺は存在しないため，メソッド抽出を行っても振る舞いを保つことができると考える．しかし，実際には 8,10,11 行目をメソッドとして抽出すると，12 行目での結果がメソッド抽出前と異なることになる．そこで，本研究での PDG を用いれば，9 行目にはメソッド抽出先からのデータ依存辺が存在するため，メソッド抽出を行うと振る舞いを保つことができないと正しく判断できる．



(a) ソースコード　(b) 通常の PDG　(c) 本研究で用いる 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．

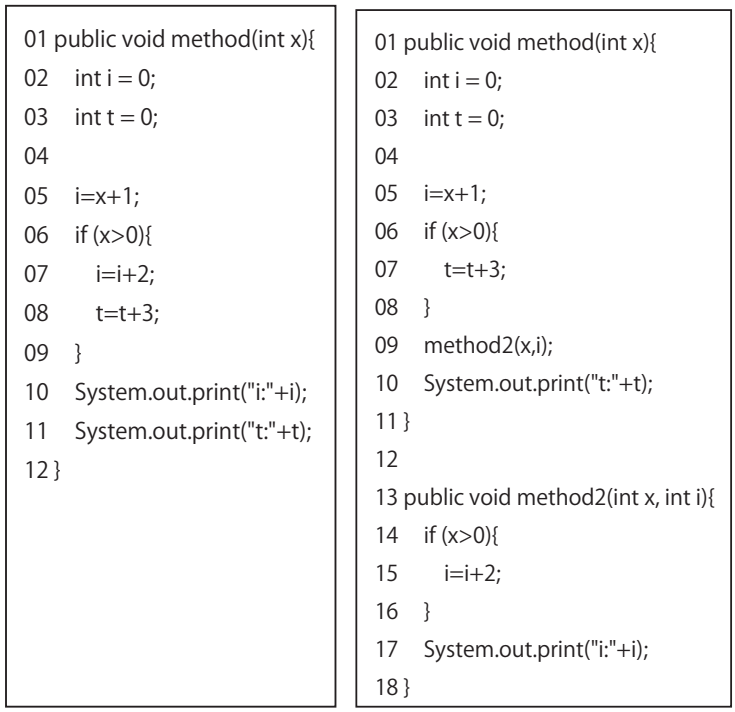
1. 新たなメソッドとして抽出する文を選択する
2. 選択した文をコピーし新たなメソッドとして定義する
3. 元の文を削除し新たに定義したメソッドへの呼び出しに置き換える

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 1 本以下である
- 条件付きの return 文がない
- break や continue などによる選択範囲外への実行フローがない

1 つ目の条件は，メソッドの返り値として指定できる変数は多くても 1 つだからである．2 つ目と 3 つ目の条件は，それらの文がメソッド抽出候補に含まれていると，抽出した新しいメソッドから返る



```
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

直前の状態によって，新しいメソッドから返った直後に，それらの return 文や break 文を実行するか判断する必要があるからである．

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に残したまま，新しいメソッド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

3.1 グラフ変形をリファクタリングに利用

Mens らは，ソースコードをグラフで表現し，リファクタリングをグラフの変形によって定義した [5]．グラフを変形することによって，プログラムの振る舞いを保つことを保証できる [6][7]．Heckel と Habel らは，プログラムをグラフとして表現した場合，リファクタリングをグラフ変形，リファクタリングの事前条件と事後条件をグラフ変形の事前条件と事後条件として表現できることを証明した [8][9]．Tip らは，グラフ表現に加えソースコードの型制限をつけることによっていくつかのリファクタリングツールを実装した [10]．Bottoni らは，プログラムとデザインの一貫性を保つために，それら 2 つのグラフを協調して変形させるスキーマを提案した [11]．

丸山らは，プログラムの構文を XML でデータベースに保存し，XML の変形によってリファクタリングを行うツールを開発した [12]．リファクタリング手順を XML の変形手順で示すことにより，より柔軟で簡単に拡張や新たなリファクタリングの実装ができる．しかし，プログラムを XML に変換するため，変換時間が長く多くの保存領域を必要とする．

3.2 メソッド抽出リファクタリング手法

Komondoor らは，C 言語プログラムから別のメソッドに抽出できる範囲を特定する手法を提案した [13]．コントロールフローグラフ (CFG) を入力とし，抽出すべきノードのセットを出力する．この手法では，特定のノードを含む抽出できる全てのノードの集合を全自動で提示する．

Harman らは，Komondoor らの手法を発展させた [14]．それまでは文をそのまま抽出するため，振る舞いを保つために元々は抽出対象となっていなかった文も抽出対象に加える必要があった．しかし，抽出する文を変形することによって，できるだけ抽出対象を増やさずにメソッドを抽出する手法を提案した．この手法により，より簡単にメソッド抽出を適用できるようになった．

Ducasse らは，ソースコードが複製された部分を特定し，リファクタリングによって取り除く手法を提案した [15]．

丸山は，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 [16]．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できることを確認した．

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 [17]．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．

3.3 リファクタリングを視覚的に補助するツール

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 3 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．

Mens らは，複数のリファクタリングの適用可能な順序をグラフで表示するツールを開発した [18]．プログラムの構造をグラフで表し，リファクタリング適用前後のグラフを表示することでリファクタリング結果を視覚的にわかりやすく提示できた．このツールが使用するグラフは，独自に定義したオブジェクト指向言語用の型グラフである，

3.4 メソッドの凝集度

Bieman らは，メソッドの凝集度を文単位ではなく字句単位で計算し分割すべきメソッドを特定する手法を提案した [19]．Harman らは，1 つの式に含まれる変数の数を使用して，より細かい粒度の情報に基づいてメソッドの凝集度を計算できるよう発展させた [20]．Ott らは，トークンに基づいた凝集度をオブジェクト指向言語に使用できるよう発展させた [21]．オブジェクト指向言語では，メソッド呼び出しによってオブジェクトの状態が変わる可能性がある．メソッド呼び出しがある文では，呼び出したメソッドの中で使用される変数も考慮して凝集度を計算する．しかし，これらはメソッド単位の凝集度を計算しており，具体的なメソッドの分割方法を示していない．

Krinke は，文単位での凝集度を表すメトリクスを提案した [22]．メソッド単位ではなく，文単位でのメトリクスを計算することで，凝集度の高い文を特定することができた．また，そのような文をソースコード上で色分けすることで視覚的に提示した．

3.5 他のリファクタリングの半自動化

Tourwé らは，論理メタプログラミング (LMP) を用い，リファクタリングすべき部分を半自動的に特定する手法を提案した [23]．Simon らは，オブジェクト指向言語用のメトリクスを用いてリファ

クタリングすべき部分を特定する手法を提案した [24]．しかし，これらの手法が特定するリファクタリングは，パラメータ削除やクラス抽出であり，メソッド抽出リファクタリングではない．

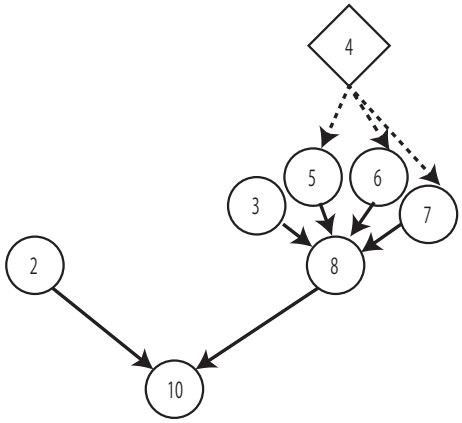
3.6 リファクタリング調査

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 [25]．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，バグの修正や機能追加を行うときにリファクタリングを同時に行うことが多いことがわかった．

Jiang らは，分割ができるメソッドを評価するため 6 つのオープンソースソフトウェアに対する実証的研究を行った [26]．多くのメソッドは分割することはできなかったが，分割できるメソッドは 2 つか 3 つに分割することができた．

```
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

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．すでに述べたようにメソッドは機能別に分割されるべきである．本研究では，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できる．

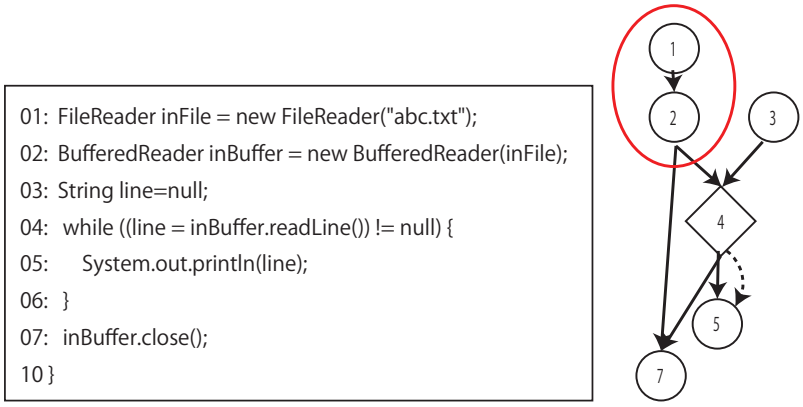
図 3 に提案手法の例を示す．図 3(a) のソースコードを，PDG とし提案手法によりノードを配置したものが図 3(b) である．8 行目にはデータ依存辺が 4 本入っており，10 行目にはデータ依存辺が 2 本入っている．8 行目に入るデータ依存辺の本数の方が多いため，そのデータ依存辺の元の 3,5,6,7 行目のノードが 8 行目のノードに近く配置されている．10 行目へ入るデータ依存辺の元である 2,8 行目のノードは離れて配置されている．この場合，ノードが密に配置されている 3,5,6,7,8 行目をメソッド抽出リファクタリングの候補とする．

以下にノード間の距離として具体的に 3 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が可能である．

4.1 Atomic Data Dependency

ある文で定義された変数が 1 度しか参照されていない場合を考える．

データ依存辺が 1 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



(a) ソースコード　　(b) PDG

図 4: 提案手法による PDG の配置 (Atomic Data Dependency)

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．

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

例を図 4 に示す．図 4(a) のソースコードを提案手法により配置した PDG を図 4(b) に示す．この例のうち 1 行目の変数 inFile に着目する．この変数は 2 行目だけで使用され，その後に使用されることはない。このような変数を使用する 1 行目と 2 行目は別のメソッドとして分けたくないと考えられるため，この間のエッジの長さは短く指定されている．

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を考える．

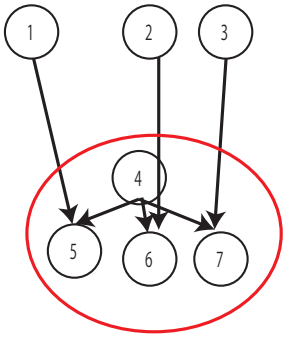
よく参照される変数は重要な値であり，その参照関係はつながりが強いと考える．この依存関係を Spread Data Dependency(SDD) と定義し，この関係にあるデータ依存辺の長さを短くする．SDD のデータ依存辺の長さを以下のように定義する．n は，1 つのノードから出るデータ依存辺の数である．

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

図 5(a) のソースコードを提案手法により配置した PDG を図 5(b) に示す．この例のうち 4 行目に着目する．4 行目から出るデータ依存辺は 3 本であり他に比べて多い．よって，それらのデータ依存辺の長さは，他の辺に比べて短く指定されている．これにより，関連する 4 行目から 7 行目のノードが近くに配置されている．

```
01:  Article articleA = stock.getArticle(itemID_A);
02:  Article articleB = stock.getArticle(itemID_B);
03:  Article articleC = stock.getArticle(itemID_C);
04:  float  tax_rate = 0.05;
05:  int  taxA = articleA.getPrice() * tax_tate;
06:  int  taxB = articleB.getPrice() * tax_tate;
07:  int  taxC = articleC.getPrice() * tax_tate;
10 }
```

(a) ソースコード



(b) PDG

図 5: 提案手法による PDG の配置 (Spread Data Dependency)

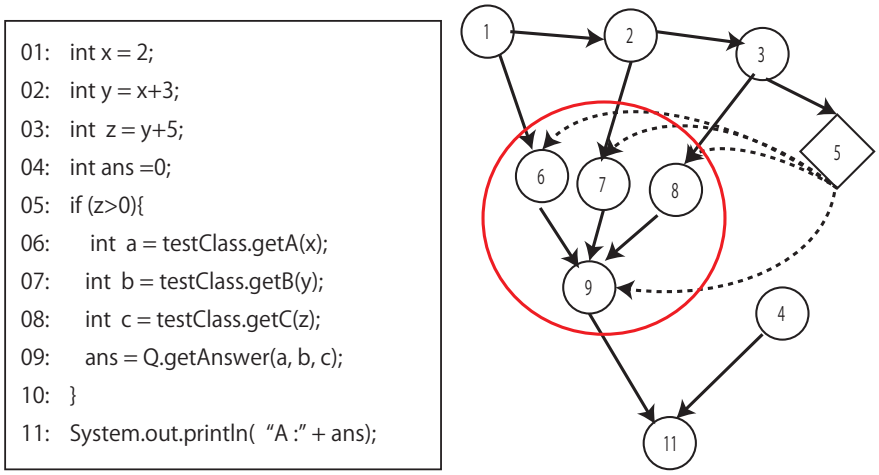
4.3 Gathered Data Dependency

多くの変数をある文で参照している場合を考える．

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，1 つのノードへ入るデータ依存辺の数である．

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

図 6(a) のソースコードを提案手法により配置した PDG を図 6(b) に示す．この例のうち 9 行目に着目する．9 行目に入るデータ依存辺は 3 本であり他に比べて多い．よって，それらのデータ依存辺の長さは，他の辺に比べて短く指定されている．これにより，関連する 6 行目から 9 行目のノードが近くに配置されている．



(a) ソースコード　　(b) PDG

図 6: 提案手法による PDG の配置 (Gathered Data Dependency)

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[27] を，グラフの可視化には Jung[28] を使用している．

ツールの外観を図 7 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 1 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 が表示され，中央にはソースコードが表示される．グラフの上のパネルには，グラフ操作やパラメータ調節のためのボタンが集約されている．

5.1 機能

5.1.1 クラス・メソッド一覧表示

クラスはパッケージごとの木構造で階層表示され，メソッドも含まれるクラスの葉として表示される．これにより，対象メソッドが探しやすくなっている．また，メソッド名の横には含まれるノードの数，クラス名の横には中に含まれるメソッド数と含まれるノードの最大値も共に表示している．これらの情報を参考にして対象メソッドを探すことも可能である．

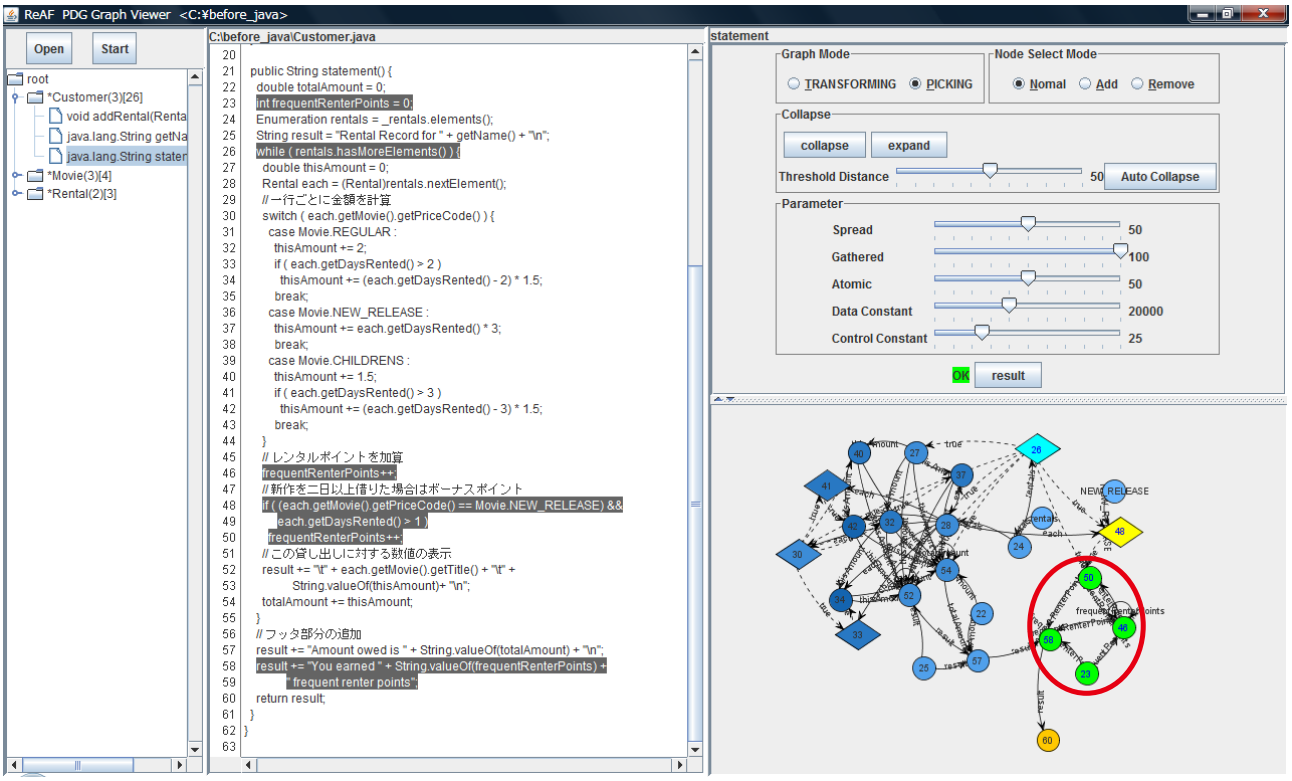


図 7: ReAF のスナップショット

5.1.2 パラメータ

表示するグラフは提案手法で定義された長さを保ってノードが配置される．提案手法によるパラメータ $distance_{ADD}$, $distance_{SDD}$, $distance_{GDD}$ の影響度をツール上でも変更可能にした．$distance_{ADD}$ が Atomic，$distance_{SDD}$ が Spread，$distance_{GDD}$ が Gathered のスライダーの値を変更することで，各パラメータの値を変更することができる．この値が大きいほど，そのパラメータにより影響されるエッジの長さが短くなる．

全体的なエッジの長さを変更する値として，データ依存辺の全体の長さを決定する”Data Constant”，制御依存辺の全体の長さを決定する”Control Constant”という値を設定した．スライダーの値を変更することでこれらの値も変更することができる．この値が大きいほど，エッジ全体の長さが長くなる．全体的にノード間の距離が短すぎて詳細を把握しづらい場合，全体的にノード間の距離が長すぎてノード配置の密な部分を把握しづらい場合に，この値により調節できると考えている．

これら 5 つの値は全てグラフを表示しながら動的に変更することができるため，より柔軟にリファクタリング候補を提示できると考えている．

5.1.3 ノード表示

形 条件文の場合ひし形，その他の文の場合丸型である．

色 return 文はオレンジ色，選択範囲は緑色，その他の文は青色となっている．ネストが深いものほどメソッド抽出の候補になりやすいと考えられるため，ネストの情報も表示するようにした．その他の文は，同じ青色でもネストが深いものほど濃い色となっている．

また，メソッド抽出後に振る舞いを保つようにするため，条件文にはメソッド抽出先に移動するもの，メソッド抽出先と抽出元の両方に複製するものがある．ノードの選択により，選択外のノードもメソッド抽出先に移動するノードは黄色，メソッド抽出先と抽出元の両方に複製するノードは水色で表示する．

ラベル ノードが引数やフィールド変数のときは，ノードのラベルとしてその変数を表示し，その他の場合はノードのラベルとしてその文の行数を表示している．

ツールチップ 選択状態にしなくても各ノードの上にマウスを乗せることで，そのノードの文をツールチップで表示する．

5.1.4 グラフ操作

グラフの操作モードが”TRANSFORMING”のときはグラフ全体の移動が，操作モードが”PICKING”のときは個々のノードを移動させることができる．個々のノードの位置を変更しても，指定したエッジの長さを保つように他のノードも移動する．また，この”PICKING”モードのときにマウスで範囲を指定することで，範囲内のノードを選択することができる．

5.1.5 選択範囲のソースコード強調表示

グラフ上でマウスによりノードを選択する．選択範囲のノードに対応するソースコードは，ハイライト表示される．また，ソースコード上の文を選択すると，対応するノードが選択状態になる．

どちらの方法で選択しても，選択範囲が正しくなるように条件文を選択範囲に追加するなど，選択範囲の修正が行われる．

5.1.6 抽出可否の判定

メソッド抽出を行うには，2.2 章で述べた条件が必要である．選択範囲がその条件を満たすかを判定する．条件を満たし抽出可能な場合は「OK」と表示する．抽出不可能な場合は「NG」と表示し，グラフ上で抽出不可能となった原因のエッジとノードを強調表示する．これにより，抽出不可能の場合でも抽出可能となるように選択範囲を変更することが容易である．

5.2 適用事例

図 7 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 [29]．クラス Customer の中のメソッド statement のグラフを表示している．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

提案手法により，既存手法では発見できなかった有用なリファクタリングを提示することが出来るか，また，実装したツールの操作性が良いかを調べるため，実験を行った．

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 4 年から博士後期課程 3 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．被験者の半数は自身が研究用に作成している Java のソースコードに対して適用した．自身が作成したソースコードなので内容を熟知しており，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．残りの半数は，他の学生が研究用に作成したソースコードに対して適用した．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 [17]．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[30] 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 1 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 3 つ用意し，その同じメソッドに対して 2 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，半数が先に ReAF を残りの半数が先にプラグインを実行した．被験者のグループ分けをまとめたものを表 1 に示す．

表 1: 実験グループ

	グループ A		グループ B	
人数	7 人		7 人	
対象ソースコード	自身のソースコード		他人のソースコード	
対象メソッド	被験者自身が決定		著者が指定	
先に実行するツール	ReAF	JDeodorant	ReAF	JDeodorant

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 3 点についてそれぞれ 4 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内容別メソッド数を表 2 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，有用な候補を提示することができたメソッドが，JDeodorant より多かった．また，ReAF では無用な候補を提示したが JDeodorant では有用な候補を提示したメソッドは 11 個あった．JDeodorant では無用な候補提示または候補提示なしであったが，ReAF では有用な候補を提示したメソッド数は $6 + 7 = 13$ 個あった．

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 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 3 に示す．JDeodorant は全体では平均行数が 63.286 であったが，1 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

表 2: 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

メソッド数		JDeodorant			
		有用な候補	無用な候補	候補提示なし	合計
ReAF	有用な候補	14	6	7	27
	無用な候補	11	3	3	17
	合計	25	9	10	44

表 3: 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)

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 4 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，1 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，確認作業が煩雑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 所要時間

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 5 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多くの時間を必要とする．どちらも解析時間より決定時間の方が長かった．決定時間が長くなる原因として，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多くの時間が必要となることが考えられる．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いため，全てを確認するには多くの時間を必要とする．

表 4: 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　ツールの改良

実験の結果，ツールの操作性が悪く，その影響でリファクタリングを決定する時間も長くなっていることがわかった．それらの問題を解消し，操作性を高めるためツールの改良を行った．改良点は以下の 4 点である．

- ノード選択
- 抽出結果表示
- 複数のノードを 1 つに集約
- 自動的に候補を提示

以下の節で各項目について説明する．

7.1　ノード選択

今までだと，ノード選択はマウスで示した四角形の中に含まれるノードしか選択できなかった．そのため，意図しない不要なノードを選択してしまったり，選択したいノードのうち 1 つだけ選択できなかったりした．

ノード選択をより容易に行うため，グラフ選択のモードを”Normal”，”Add”,”Remove”の 3 つ用意した．以下に各モードの説明を述べる．このモード選択の変更は，キーボードニーモニックを設定しており，キー操作でも変更が可能である．

Normal マウスで選択した範囲内のノードを選択することができる．別の範囲を選択した場合，元の選択範囲だったノードは選択が解除される．

Add 現在選択状態のものに加えて，新しくマウスで選択した範囲内のノードを選択状態にする．別の範囲を選択した場合でも，元の選択範囲だったノードの選択状態は維持する．

Remove 現在選択状態のものから，新しくマウスで選択した範囲のノードの選択状態を解除する．新しく選択したノードが，選択状態になっていなかった場合は，選択範囲は変わらない．

表 5: 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

この機能追加により，例えば，ノードを選択したが選択範囲から複数のデータ依存辺が出ているために抽出不可能となった場合，エラーとして赤く強調表示されたエッジの先のノードを選択範囲に加えることで，簡単にエラーを修正することができる．

7.2 抽出結果表示

実験で用いた JDeodorant には抽出結果表示の機能が実装されているため，実験の際の要望としてもこの機能の追加は多く挙げられていた．選択範囲がメソッド抽出可能な場合，メソッド抽出後の新しいメソッドのソースコードを表示する．これは，元のメソッドのソースコードから選択範囲のノードに対応する文を抜き出したものである．これにより，メソッド抽出を行うかがより判断しやすくなると考えられる．

7.3 複数のノードを 1 つに集約

ノードの数を減らし見やすくするため，複数のノードを 1 つのノードに集約することができるようにした．選択状態のノードを 1 つにまとめることができる．別のメソッドに分けたくないノードを選択し 1 つのノードにまとめることで，より簡単にメソッド抽出の候補を探すことが可能である．複数のノードを 1 つのノードに集約した場合は，集約したノードの中に含まれるノードの数が多いほど，ノードを大きく表示する．

また，集約したノードを選択し”expand”ボタンを押すことで，集約を解除する．

7.4 自動的に候補を提示

グラフ上のノードの疎密によりメソッド抽出候補を特定するが，エッジが複雑な場合は特定しにくい．そこで，エッジの長さが短いものを基準にメソッド抽出候補を 1 つのノードに集約することで提示できるようにした．グラフ上のすべてのエッジから最も短いものを基準に，閾値以下のエッジをもつノードを 1 つに集約する．自動的にノードを集約するアルゴリズムは以下の通りである．このアルゴリズムは，エッジの長さの閾値 k とプログラム依存グラフ P を入力とし，集約したノードを出力する．

1. P の中から最も長さの短いエッジ e を取得する
2. e の長さが k よりも長ければ，アルゴリズムを終了する
3. e の長さが k よりも短ければ，e の両端のノードを 1 つに集約する
4. 集約したノードに入るエッジと出て行くエッジの中から最も長さの短いエッジを取得し e とする．そして 2. に戻る

出力は，最後に集約してできたノードである．

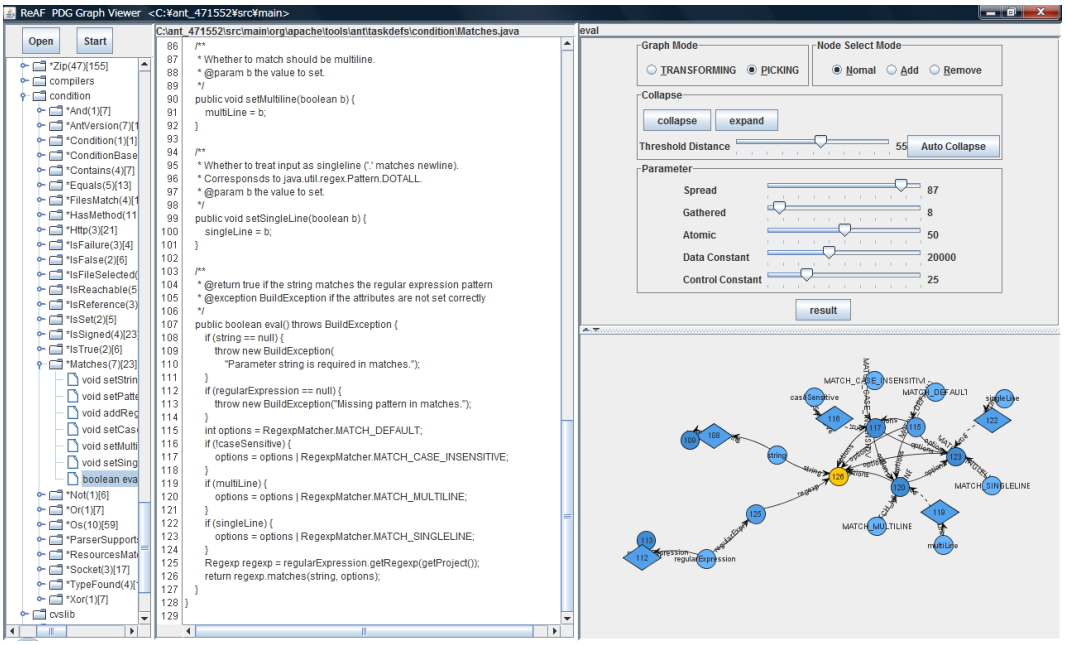
この機能により，メソッド抽出の候補をより簡単に確認できる．

7.5 ツールの評価

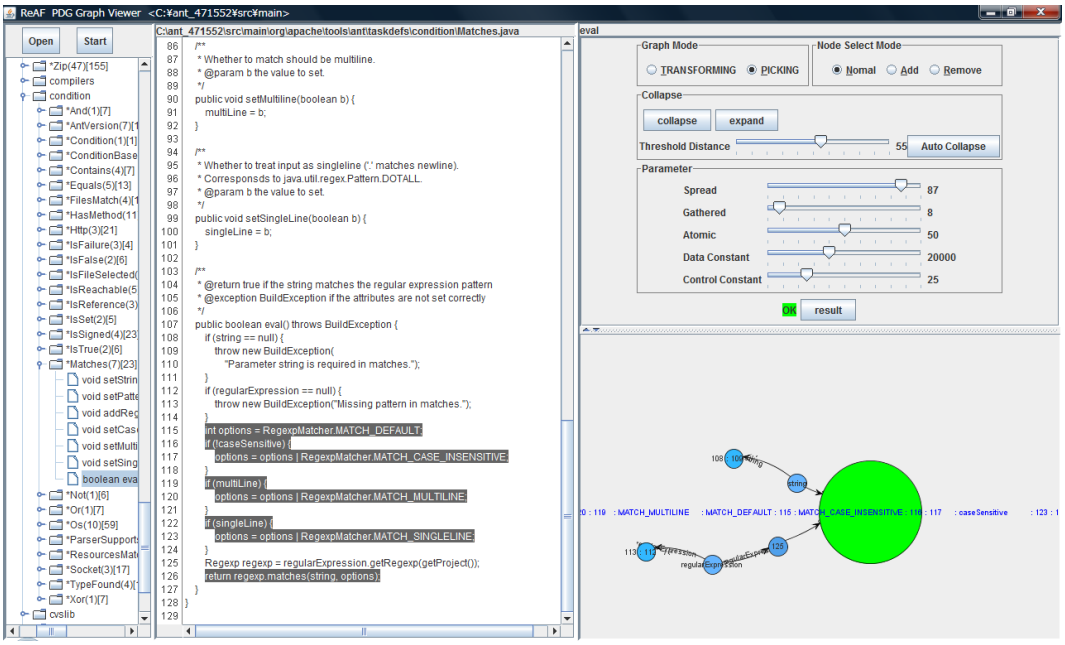
改良したツールを評価するため，オープンソースソフトウェアへの適用と追加実験を行った．

7.5.1 オープンソースソフトウェアへの適用

オープンソースソフトウェアへツールを適用した．対象ソースコードは，ant のバージョン 1.8.1 である．クラス org.apache.tools.ant.taskdefs.condition.Matches のメソッド eval() への適用結果を図 8 に示す．図 8(a) はグラフを表示直後であり，どのノードも集約させていない．ここでメソッド抽出候補を自動的に提示するためのボタン”Auto Collapse”を押した．これにより，グラフの一部を 1 つに自動的に集約させたものを図 8(b) に示す．自動的に 1 つのノードとなった範囲は，option の設定を行う処理である．実際の ant のソースコードを図 9 に示す．図 9(a) のうち「＋＋」で始まる行がリファクタリング候補として提示されている．図 9(b) にバージョン 1.8.2 でのリファクタリング後のソースコードと図 9(c) にリファクタリングによって抽出された新しいメソッドのソースコードを示す．リファクタリング候補として提示した範囲が，実際にバージョン 1.8.2 にて，新たなメソッドとして抽出されている．また，この例は他のメソッド抽出リファクタリング支援ツールである JDeodorant では提示できなかった．このように，他のツールでは提示できなかった有用なリファクタリング候補を自動的に提示することができた．



(a) グラフ表示直後



(b) リファクタリング候補の自動提示

図 8: ReAF への適用例 (ant)

```
  107: public boolean eval() throws BuildException {
  108:   if (string == null) {
  109:     throw new BuildException(
  110:       "Parameter string is required in matches.");
  111:   }
  112:   if (regularExpression == null) {
  113:     throw new BuildException("Missing pattern in matches.");
  114:   }
++115:   int options = RegexpMatcher.MATCH_DEFAULT;
++116:   if (!caseSensitive) {
++117:     options = options | RegexpMatcher.MATCH_CASE_INSENSITIVE;
++118:   }
++119:   if (multiLine) {
++120:     options = options | RegexpMatcher.MATCH_MULTILINE;
++121:   }
++122:   if (singleLine) {
++123:     options = options | RegexpMatcher.MATCH_SINGLELINE;
++124:   }
  125:   Regexp regexp = regularExpression.getRegexp(getProject());
  126:   return regexp.matches(string, options);
  118: }
```

(a) バージョン 1.8.1 のソースコード

```
107: public boolean eval() throws BuildException {
108:   if (string == null) {
109:     throw new BuildException(
110:       "Parameter string is required in matches.");
111:   }
112:   if (regularExpression == null) {
113:     throw new BuildException("Missing pattern in matches.");
114:   }
115:   in options = RegexUtil.asOptions(caseSensitive, multiLine, singleLine);
116:   Regexp regexp = regularExpression.getRegexp(getProject());
117:   return regexp.matches(string, options);
118: }
```

(b) バージョン 1.8.2 にてリファクタリング実行後のメソッド

```
95: public static in asOptions(boolean caseSensitive, boolean multiLine,
96:                            boolean singleLine){
97:   int options = RegexpMatcher.MATCH_DEFAULT;
98:   if (!caseSensitive) {
99:     options = options | RegexpMatcher.MATCH_CASE_INSENSITIVE;
100:  }
101:  if (multiLine) {
102:    options = options | RegexpMatcher.MATCH_MULTILINE;
103:  }
104:  if (singleLine) {
105:    options = options | RegexpMatcher.MATCH_SINGLELINE;
106:  }
107:  return options;
108: }
```

(c) バージョン 1.8.2 にて抽出されたメソッド

図 9: ReAF への適用例 (ant)

7.5.2 追加実験

改良したツールを評価するため，前回の実験の被験者数名に対して追加実験を行った．追加実験の被験者は，前回の実験にて ReAF の評価が低かった 3 人である．被験者には，改良したツールを使用してもらい，アンケートに答えてもらった．アンケート項目は前回と同じである，

実験の結果を表 6 に示す．追加機能はユーザインタフェースの改善であり，主に操作性の向上が期

待できる．実験結果によると，操作性に対する評価が大幅に向上した．自由記述による使用感想でも，追加機能により使いやすくリファクタリング候補を選択しやすくなったとの意見が得られた．

表 6: アンケート評価結果

	ReAF		
被験者	可視化	操作性	継続利用
A	3 → 3	1 → 3	2 → 2
B	3 → 4	2 → 3	2 → 3
C	2 → 3	2 → 2	1 → 2

8　あとがき

本研究では，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，他のツールでは提示することのできなかった有用な候補を提示することができた．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．また，ツール改良の結果，自動的に有用なリファクタリング候補を提示することができ，ユーザビリティも向上した．

今回，学生の作成したソフトウェアとオープンソースソフトウェアに対して適用した．今後の課題として，より多くの種類のソースコードに適用し，その効果と有用性を評価することを考えている．それにより，どういった場合にどのパラメータの値が有効であるかが得られ，パラメータの適切な初期値を設定することでよりツールが使いやすくなると考えている．

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，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.

その他の楠本研究室の皆様のご協力に心より感謝致します.

最後に，本学基礎工学部所属時より現在に至るまで，講義，演習，実験等を通じてお世話頂きました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] E. Murphy-Hill and A.P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th international conference on Software engineering*, pp. 421–430, 2008.

[2] S.G. Eick, T.L. Graves, A.F. Karr, JS Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, pp. 1–12, 2002.

[3] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pp. 308–320, 1976.

[4] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pp. 439–449, 1981.

[5] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. *Graph Transformation*, pp. 286–301, 2002.

[6] W.F. Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[7] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object systems*, Vol. 3, No. 4, pp. 253–263, 1997.

[8] R. Heckel. Algebraic graph transformations with application conditions. *Master's thesis, TU-Berlin*, 1995.

[9] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, Vol. 26, No. 3/4, pp. 287–313, 1996.

[10] F. Tip, A. Kiezun, and D. Baumer. Refactoring for generalization using type constraints. *ACM SIGPLAN Notices*, Vol. 38, No. 11, pp. 13–26, 2003.

[11] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated Distributed Diagram Transformation for Software Evolution1. *Electronic Notes in Theoretical Computer Science*, Vol. 72, No. 4, pp. 59–70, 2003.

[12] K. Maruyama and S. Yamamoto. Design and implementation of an extensible and modifiable refactoring tool. In *13th International Workshop on Program Comprehension*, pp. 195–204, 2005.

[13] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension*, pp. 33–42, 2003.

[14] M. Harman, D. Binkley, R. Singh, and R.M. Hierons. Amorphous procedure extraction. In *Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 85–94, 2005.

[15] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *26th IEEE International Conference on Software Maintenance*, p. 109, 1999.

[16] K. Maruyama. Automated method-extraction refactoring by using block-based slicing. In *Symposium on Software reusability: putting software reuse in context*, pp. 31–40, 2001.

[17] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *13th European Conference on Software Maintenance and Reengineering*, pp. 119–128, 2009.

[18] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, Vol. 6, No. 3, pp. 269–285, 2007.

[19] J.M. Bieman and L.M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, Vol. 20, No. 8, pp. 644–657, 2002.

[20] M. Harman, S. Danicic, B. Sivagurunathan, B. Jones, and Y. Sivagurunathan. Cohesion metrics. In *8 th International Quality Week, May 29th*, 1995.

[21] L.M. Ott and J.M. Bieman. Program slices as an abstraction for cohesion measurement. *Information and Software Technology*, Vol. 40, No. 11-12, pp. 691–699, 1998.

[22] J. Krinke. Statement-Level Cohesion Metrics and their Visualization. In *7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 37–48, 2007.

[23] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Seventh European Conference on Software Maintenance and Reengineering*, pp. 91–100, 2003.

[24] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *5th European Conference on Software Maintenance and Reengineering*, p. 30, 2001.

[25] E. Murphy-Hill, C. Parnin, and A.P. Black. How we refactor, and how we know it. In *IEEE 31st International Conference on Software Engineering*, pp. 287–297, 2009.

[26] T. Jiang, M. Harman, and Y. Hassoun. Analysis of procedure splitability. In *15th Working Conference on Reverse Engineering*, pp. 247–256, 2008.

[27] MASU. `http://sourceforge.net/projects/masu/`.

[28] jung. `http://jung.sourceforge.net/`.

[29] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. 1999.

[30] eclipse. `http://eclipse.org/`.