# 並行開発支援のための機能的類似コード検出手法の提案

田島 諒[1]　名倉 正剛[2]　高田 眞吾[1]

**概要**：ソフトウェアを複数人で並行開発する場合，別々の開発者によって似た機能を提供する異なるコードを作り込むことがある．このようなコードの存在はメソッドの肥大化や変更コストの増大を招くため，減らすことは保守性向上のために有効である．本研究ではこの似た機能を提供する異なるコードを機能的類似コードと呼び，自動で検出する手法を提案する．提案手法はメソッドを単位にして，入出力の比較及び，プログラム依存グラフの比較に基づいて機能的類似コードの検出を行う．

**キーワード**：機能的類似コード検出，並行開発支援，ランダムテスト，プログラム依存グラフ

## 1. 序論

ソフトウェアを複数人で並行開発する場合，別々の開発者によって似た機能を提供する異なるコードを作り込むことがある．本研究ではこのような似た機能を提供する異なるコードを機能的に類似したコード（機能的類似コード）と呼ぶ．これは，Bellon らの研究 [1] で言及される 3 つのタイプのコードクローンのコード片に加え，同じ入力に対して同じ出力が得られるようなコード片のことを指す．

例えば，プロジェクト内で共通利用されるユーテリティ機能を複数の開発者が別個に実装してしまった場合，機能的類似コードがプロジェクト内に重複して存在するようになる．このようなコードの存在は，メソッドの肥大化や変更コストの増大を招くため，機能的類似コードを減らすことは保守性向上のために有効である．また，機能的類似コードがプロジェクトに存在する期間が長ければ長いほど，それらを利用する別のコードが開発される可能性が高くなる．機能的類似コードを減らす際に，類似コードを利用する別のコードが多いほど，必要とする保守開発コストは増大する．そのため，機能的類似コードは可能な限り早い段階で検出されることが望ましい．

本研究では，機能的類似コードを検出し開発者に提示することを目的として，コーディング時に機能的類似コードを検出し，提示する手法を提案する．提案手法では機能的類似コードをメソッド単位で検出する．具体的には開発者がメソッドに関するコーディングを終えた時点で，プロジェクト内の他のソースコード内から機能的類似コードのメソッドを検出する．メソッドの検出は，ランダムテストによるメソッドへの入力に対する出力の比較と，メソッド自体の構造の比較によって行う．

[1] 慶應義塾大学
Keio University
[2] 日本大学
Nihon University

## 2. 関連研究

Bellon らの研究 [1] で言及される 3 つのタイプのコードクローンの検出については，ソースコードを文字列とみなして比較する研究 [2] や，AST などの木構造を抽出して比較する研究 [3]，プログラム依存グラフ（PDG）を用いて元のコードに対して非連続的な似たコードの検出を可能にした研究 [4][5] など，様々な研究が存在する．しかし，Wang らの論文 [4] で言及されているように，これらの方法では同じ機能の実現に対して異なるアルゴリズムで実装した場合は，仮に似たようなコードが存在しても検出できない．

Deissenboeck ら [6] の研究ではランダムテストを利用した，機能的類似コードの検出手法を提案した．Deissenboeck らはこの研究の目的として，別々の開発者により独立して記述された類似コードの検出を挙げており，本研究の目的と同じ方向性である．この研究では入力に対する出力を比較することにより，コードの文の類似性に関わらずに機能的に似ているコードの検出が可能となっている．また検出対象とするコードの単位について実験を行った結果，メソッド単位での検出が最も効率的であると述べている．しかし，入力に対する出力の比較だけでは機能的類似コードを検出できない場合が存在する．例えば**図 1** のように，出力を一律的に変更するようなコードが挿入されたメソッドを検出できない．

```
double inchToCm(int inch){
   return (double) inch * 2.54;
}
```

```
double inchToMeter( int inch){
   return (double) inch * 2.54 / 100;
}
```

図 1: 入出力の比較では検出できないコード例

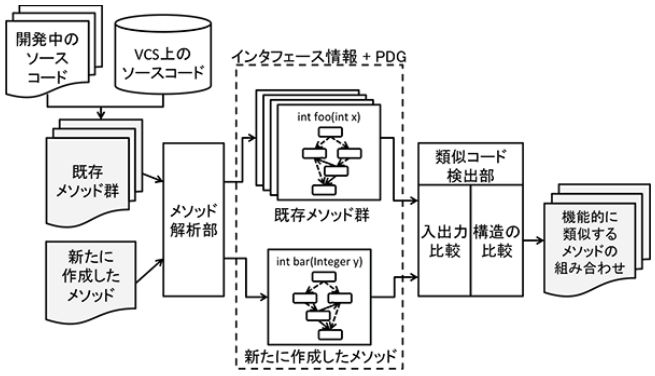



図 2: 提案手法の処理の流れ

## 3. 提案手法

我々は Java プログラムを対象に，開発中のコードに対する機能的類似コードを開発者に提示する手法を提案する．

### 3.1 概要

提案手法では機能的類似コードの検出をメソッド単位で行う．そして開発者がメソッドを作成した際に，開発中のソースコードや，そのソースコードが含まれるプロジェクトのバージョン管理システム (VCS) に含まれるプロジェクト全体の最新のソースコード群から，作成したメソッドに対して機能的に類似するメソッドを検出して提示する．

提案手法の構成と処理の流れを**図 2** に示す．提案手法は，メソッド解析部と類似コード検出部からなる．メソッド解析部は，開発中のソースコードやプロジェクト全体のソースコードから抽出したメソッドから，インタフェース情報と，構造を示すプログラム依存グラフ (PDG) を取得する．類似コード検出部は，既存ソースコードに含まれるメソッド群と新たに作成したメソッドについて，メソッド解析部によって抽出した情報を比較して，機能的に類似するコードを検出する．この類似コード検出は，比較対象のメソッドに対する，同一入力への出力の類似性と，構造の類似性によって行う．そして開発中のコードに対する類似コードを，開発者が類似性を理解できるように提示する．

本章では，まずメソッド解析部により抽出するメソッド情報について述べ，次に入出力の比較による類似コード判定と，構造の比較による類似コード判定について述べる．

### 3.2 類似性判定に利用するメソッド情報

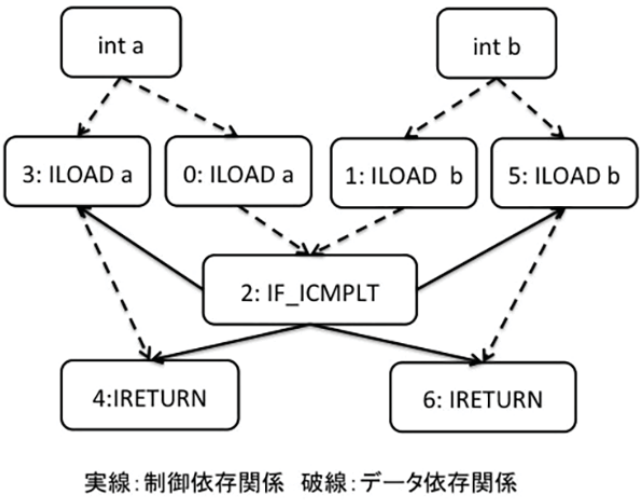
メソッド解析部は，メソッドの類似性の判定に用いるためのメソッド情報として，メソッドのインタフェース情報とメソッドの PDG を抽出する．まずメソッドのインタフェース情報としては，メソッドのシグネチャ情報を取得する．そして入出力の比較による類似コード判定に利用するため，メソッドの引数に関する情報と，メソッドの戻り値に関する情報を抽出する．メソッドの PDG はプログラム中の命令間に存在する制御依存関係やデータ依存関係について表した有向グラフであり，構造の比較による類似コード判定に利用する．メソッドとメソッド解析部により抽出される PDG の例を**図 3** に表す．図 3 では (a) で示したメソッドに対し，(c) のように表される PDG を抽出する．PDG の各ノードはバイトコードの命令単位で出力され，それぞれ (b) に示したバイトコード命令に対応する．そして，命令間の制御依存関係とデータ依存関係も抽出する．図 3(c) では，それぞれ実線と破線で示している．

```
public int max(int a, int b){
   if(a > b){
      return a;
   }
   return b;
}
```

(a) ソースコード

```
0: ILOAD a
1: ILOAD b
2: IF_ICMPLT
3: ILOAD a
4: IRETURN
5: ILOAD b
6: IRETURN
```

(b) (a) のバイトコード




(c) (b) から生成される PDG

図 3: PDG の例

### 3.3 機能的類似コード判定（1）：入出力の比較

比較対象のメソッドに対して，与えた入力値に対する出力値の類似性を判定する．これは次のステップで行う．

(1) 入力と出力の型の比較

(2) 入力値に対する出力値の比較

なお，同一の引数のセットを入力とする複数のメソッドを比較する場合に，引数の順番が異なったり，追加の引数が存在することで引数の数自体が異なったりするメソッドを

比較することが考えられる．このようなメソッドの比較を考慮するため，メソッドの入出力の類似性の比較はメソッドの引数群のすべての組み合わせに対して行う．

#### 3.3.1 入力と出力の型の比較

新たに作成したメソッドと既存メソッドの入力と出力の型を比較する．入出力の型の組み合わせがいずれも一致しない場合は，入出力が類似していないとみなし，入力値に対する出力値の比較を行わない．双方のメソッドのいずれかの引数の型が一致し，それぞれの出力の型が一致する場合に，入出力の型の組み合わせが一致するとみなし，入力値に対する出力値の比較を行う．

なお，入出力の型の組み合わせの比較は，実際の型の完全な一致ではなく，意味的な一致により行う．具体的には次の場合に型が一致したものとみなす．

- 同一のプリミティブ型や，同一のクラスの場合
- プリミティブ型と，そのラッパークラスの場合
  （例：boolean 型と Boolean 型，int 型と Integer 型）
- 互いに数値を示すプリミティブ型やクラスの場合
  （例：int 型と double 型，Integer 型と Double 型）
- 片方が他方のクラスの継承クラスの場合

#### 3.3.2 入力値に対する出力値の比較

新たに作成したメソッドと既存メソッドに対して，同一の入力を与えた場合に実行結果として同じ出力が得られる割合によって，類似性を判断する．

まず，入力として与える引数を生成する．プリミティブ型の場合は，乱数を用いて生成する．数値型のラッパークラスの場合は，プリミティブ型と同様に乱数を利用してインスタンスを生成する．それら以外のクラスの場合は，インスタンス生成のためにコンストラクタを呼び出す．コンストラクタ呼び出しに引数を必要とする場合は，再帰的に引数を生成する．なお，コンストラクタが複数存在する場合はすべてのコンストラクタに対し入出力の比較を行う．

次に，生成した引数を新たに作成したメソッドと既存メソッドに対して入力することによって，実行結果を得る．メソッドの返戻値の型がプリミティブ型やそのラッパークラスの場合は，プリミティブ値を取り出して比較する．その他のクラスの場合は，オブジェクトの等価性を判断するための e uals メソッドが実装されていれば，それを呼び出して比較を行う．実装されていない場合は出力を比較できないため，異なる実行結果とみなす．

複数回ランダムに生成した入力値に対して出力値が等しいかどうかを比較し，その等しい回数の割合を入出力の類似度として算出する．算出式を (1) に示す．

$$\frac{\text{出力が同じ値であった回数}}{\text{入出力の比較回数}} \quad (1)$$

### 3.4 機能的類似コード判定（2）：構造の比較

新たに作成したメソッドと既存メソッドの構造の類似性を判定する．これは，2 メソッドの PDG を比較し，共通部分グラフに含まれるノードが，新たに作成したメソッドのノードに占める割合により算出する．算出式を (2) に示す．

$$\frac{\text{PDG の共通部分グラフに含まれるノード数}}{\text{新たに作成したメソッドの PDG の総ノード数}} \quad (2)$$

## 4. 実装

本章では，3 章で述べた提案手法の実装について述べる．

### 4.1 全体の概要

提案手法を，Eclipse での編集中に機能的に類似するメソッドを検出し提示するように実装した．まず，新たに作成したメソッドの解析部分と，機能的類似メソッドの検出・提示部分を Eclipse プラグインとして実装した．プロジェクト全体で大量に存在する既存コードからメソッド情報を抽出するためには，長時間の解析時間を要することが想像できる．このため，VCS 上に存在するコードや，開発中の別クラスに対するソースコードに対する解析処理部分は，プラグインの処理とは別処理として実装した．そして，リポジトリや開発中のソースコードに対する最新情報を利用できるようにするため，一定時間間隔で定期的に実行し，データベースに保持するように実装した．

### 4.2 メソッド情報の抽出

メソッド解析部は開発中のソースコードや VCS 上のソースコードから，インターフェース情報と PDG を抽出する．

メソッドのインタフェース情報の抽出には，Java リフレクション API を利用して実装した．その際に，メソッド名や引数と返戻値の型といったリフレクション API によって取得した情報とともに，判定結果の表示に用いるためにパッケージ名とクラス名も取得するように実装した．また同様に判定結果の表示に利用するため，Eclipse の GUI ライブラリである JFace を利用して，メソッドのファイル内での位置を抽出するように実装した．

PDG の抽出は，SOBA[7] を利用し実装した．そして VCS 上に存在するコードや，開発中の別クラスに対するソースコードに対する解析処理の場合は，前述のように抽出したグラフデータをデータベースに保持する．この実装には，オープンソースのグラフデータベースである Neo4j[8] を利用した．この際に PDG のバイトコードの命令単位でノードに保存し，命令間の依存関係をノード間のエッジとして保存する．そして，エッジのラベルには命令間の依存関係の種類を，プロパティにはノードに保存した命令を含むメソッド名を設定する．

なお，メソッドシグネチャ等の PDG 以外の情報もこのデータベースに保持する．このため，PDG のノードのラベルには，メソッドのシグネチャに関する情報を保持し，

PDG で表現されるノード ID とバイトコード命令の内容については，プロパティに保持するように実装した．さらに，各メソッドごとに専用ノードを追加し，プロパティにメソッドのファイル内位置を保持するように実装した．

### 4.3 機能的類似コード判定（1）：入出力の比較

類似コード検出部は，新たに作成したメソッドと既存の各メソッドに対し，引数の順番を考慮して入出力の比較を行う．まず，比較対象のそれぞれのメソッドの引数と返戻値の型を比較する．その結果として入力値と出力値を比較できる場合は，比較対象のそれぞれのメソッドの引数のすべての組み合わせについて，入力値を生成する．そしてそれぞれの入力を与えた場合の出力を比較し，類似度を算出する．全ての引数の組み合わせに対する出力の類似度の中での最大値を，2 つのメソッドの出力の類似度とみなす．

#### 4.3.1 入力と出力の型の比較

Java リフレクションを用いてそれぞれのメソッドの引数と返戻値の型の比較を行う．この比較は，3.3.1 節に記述したように，型が意味的に一致するかどうかにより実施する．その結果，比較対象のメソッドの引数に型が一致する組み合わせが一つ以上存在する場合に，入力値に対する出力値の比較を行う．なお，一致する引数の型の組み合わせが複数存在する場合，全ての組み合わせについて比較を行う．

#### 4.3.2 入力値に対する出力値の比較

それぞれのメソッドの引数に対して入力値を生成したのち，入力させて実行することで得られた出力を比較する．

まず，4.3.1 節で導出した組み合わせごとに，引数として取りうる値の型を決定する．4.3.1 節では意味的に一致する型の場合に組み合わせを生成している．このため対応する引数の実際の型が異なることあるが，その場合どちらにも入力できる型を取り得る型として決定する．例えば，片方のメソッドの int 型の引数に対して，他方のメソッドでは double 型の引数を組み合わせとして導出した場合， 両方が取り得る型として int 型 （整数の数値）を決定する．

次に，入力値を生成する．数値については乱数生成によって行う．引数が特定クラスのインスタンスの場合は，コンストラクタを利用して行う．型が一致する引数の組に対しては同一の入力値を生成する．なお，コンストラクタの呼び出しに引数が必要な場合は再帰的に引数を生成する．実行時間を考慮して再帰回数を 2 回に制限しており，2 回を超える場合は 3 回目のコンストラクタの呼び出しに対して，引数が空のコンストラクタを呼び出す．

次に，生成した入力値をそれぞれのメソッドの対応する引数に入力し，メソッドを実行する．入力値に対する出力値の比較は，4.3.1 節で導出したすべての組み合わせに対しランダムな入力値を生成し，実行結果として同じ値を得られるか比較することで行う．現在の実装では，それぞれの組み合わせに対し入力値を 20 セット生成し，実行結果を比較する．そして実行した回数と，同じ値が得られた回数を用い，3.3.2 節 (1) 式から入出力の類似度を算出する．

### 4.4 機能的類似コード判定（2）：構造の比較

Neo4j に保存した既存のメソッドの PDG と新たに作成したメソッドの PDG を比較する．各メソッドの引数を表すノードと，どのノードからも依存先になっていないノードを起点ノードとして，起点とするノードから順に一致する部分グラフを探索する．

部分グラフ探索の手順を，次に示す．まず既存メソッドと新たに作成したメソッドの PDG に含まれる起点ノードに対して，すべての組み合わせを作成する．作成した組み合わせそれぞれに対して，起点ノードから順にノードとエッジを比較する．そして，ノード（バイトコード命令）とエッジ（依存関係）が一致する場合に部分グラフが一致しているとみなし，部分グラフを導出する．なお比較の際には 4.2 節で追加したメソッドのファイル内位置を表すノードを含めない．この比較を，ノードとエッジが一致しなくなるまで繰り返す．この導出手順をすべての起点ノードの組み合わせに対して実施する．そして新たに作成したメソッドの PDG に含まれるノードのうち導出された部分グラフに含まれるノードを一致したノードとして扱い，そのノード数をカウントする．この際に，複数の部分グラフに含まれるノードについては，重複せずにカウントする．この値を PDG 間で一致したノード数とし，新たに作成したメソッドの PDG に含まれる全ノード数を利用して，3.4 節 (2) 式によって構造の類似度を算出する．

### 4.5 検出結果の表示

提案手法は 2 章で言及したように，ランダムテストを利用した機能的類似コード検出手法や，構造による機能的類似コード検出手法のどちらかだけでは検出できないような類似コードを，開発者が検出できるような方法を提供することを目的としている．したがって，それぞれの評価尺度によりどの程度類似しているかを開発者が確認できるように，検出結果を 2 次元平面上に表示する．表示例を**図 4** に示す．X 軸に構造の類似度を， 軸に入出力の類似度をそれぞれパーセント表示で表している．図 4 は，test パッケージの Sandbox クラスの getID3 メソッドに対する機能的類似メソッドの検出結果である．三角のプロットはそれぞれ検出された類似メソッドを表している．プロット選択時には各メソッド実装を確認するためのポップアップウィンドウを表示する．このポップアップウィンドウには，開発者が類似する既存メソッドに対してリファクタリングを実施する際の参考情報として，次の情報を併せて表示する．

- 該当メソッドを含むファイルのディレクトリ階層
- 最後に変更を加えた開発者の名前と変更日時
- 該当メソッドの参照数，参照メソッド名

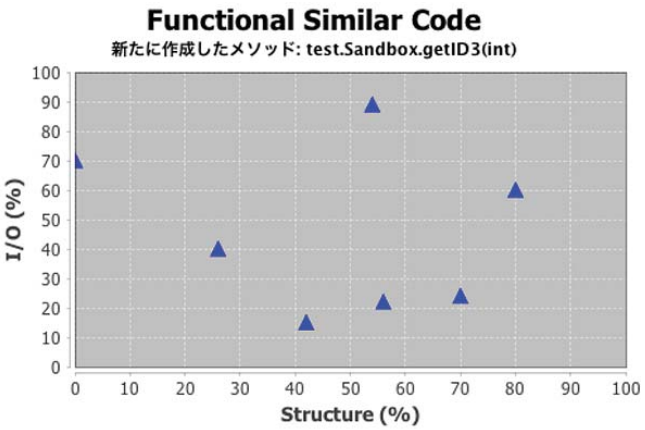



図 4: 判定結果の出力

## 5. ケーススタディ

本章では，具体的な開発事例を想定し，提案手法による処理の流れを示す．

### 5.1 事例 1：元コードの複製により開発を実施した場合

#### 5.1.1 概要

元コードのコピー&ペーストにより開発した事例を想定し，メソッドを**図 5** に示す．この 2 つのメソッドは，円の面積を求めて返却するという全く同一の処理である．開発者は circle1 メソッドを利用して円の面積を算出しようとしたが，メソッドの可視性（protected）により開発中のパッケージからアクセスできず利用できなかった．新たに作成しているメソッドから利用するためには，元メソッドの可視性を変更する必要があった．しかし，不要なパッケージに可視性が及ぶことによる既存コードへの影響により結果としてパッケージ構成を変更する必要があり，複製して別途 circle2 メソッドを用意した．

このような過程により開発者が circle2 メソッドを記述したことを想定し，実装したツールを実行した．

#### 5.1.2 メソッド情報の抽出

メソッド解析部は，定期的な実行によりリポジトリ内に存在するメソッドのシグネチャと PDG を抽出する．そして，package1.Class1.circle1 メソッド (引数 : *double*，返戻値 : *double*) のシグネチャと，**図 6** (a) の PDG を得る．

同様に，新規メソッドの開発終了時点でエディタ上から指示することにより，エディタ上で記述されたメソッドについてシグネチャと PDG を抽出する．その結果，package2.Class2.circle2 メソッド (引数 : *double*，返戻値 : *double*) のシグネチャと，図 6(b) の PDG を得る．

#### 5.1.3 機能的類似コード判定（1）：入出力の比較

package1.Class1.circle1 メソッドと package2.Class2.circle2 メソッドの入出力の型を比較する．引数の型と返戻値の型はともにそれぞれ double 型であるため，どちらも一致している．そこで，同一の入力値を生成し，出力値の比較を行った (**表 1**)．入力値のセットを 20 セット生成し入力した結果すべての出力が一致したため，入出力の類似度は 100%と算出された．

```
protected double circle1(double rad){
   return Math.PI * rad * rad;
}
```

(a) 既存のメソッド (package1.Class1 クラス内)

```
protected double circle2(double rad){
   return Math.PI * rad * rad;
}
```

(b) 新たに作成したメソッド (package2.Class2 クラス内)

図 5: 事例 1：複製により生じた類似コード

| | 入力値 | 出力値 (circle1) | 出力値 (circle2) |
|---|---|---|---|
| 1 | 0.9124179437 | 2.6153963173 | 2.6153963173 |
| 2 | 0.3182517071 | 0.3181935386 | 0.3181935386 |
| 3 | 0.6526229095 | 1.3380566565 | 1.3380566565 |
| ⋮ | (略) | ⋮ | ⋮ |
| 20 | 0.5474696404 | 0.9416075976 | 0.9416075976 |

表 1: 事例 1 にて生成した入力値と対応する出力値

#### 5.1.4 機能的類似コード判定（2）：構造の比較

図 6 に示した 2 つのメソッドの PDG の類似度を算出する．まず，図 6 に吹き出しで示すノードを，起点ノードとして決定した．次に 2 つのメソッドの 2 つずつの起点ノードの 4 通りの組み合わせに対して一致する部分グラフの探索を行うと，図 6(b) に示す全てのノードが部分ノードに含まれたため，構造の類似度は 100%と算出された．

### 5.2 事例 2：定型的なロジックを再開発した場合

#### 5.2.1 概要

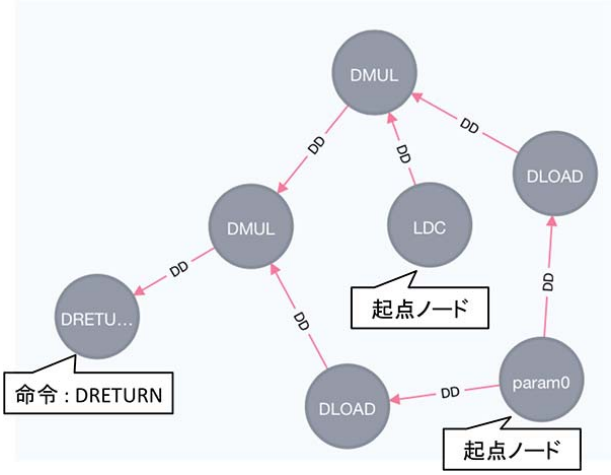
定型的なロジックを再開発した事例を想定し，メソッドを**図 7** に示す．この 2 つのメソッドはどちらもデータベースに接続し，クエリを実行した後で接続を切断する処理を行っている．開発者はデータベース接続から切断までの流れの定型的なロジックの存在を想定しつつも，特に意識せずに getDataCount メソッドを作成した．

このような過程により開発者が getDataCount メソッドを記述したことを想定し，実装したツールを実行した．

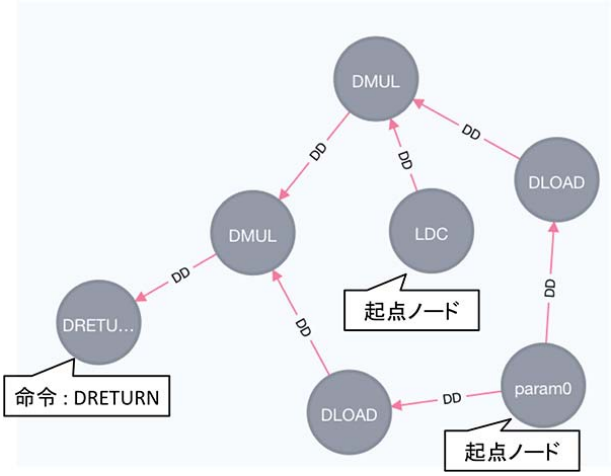
#### 5.2.2 メソッド情報の抽出

メソッド解析部は，定期的な実行によりリポジトリ内に存在するメソッドのシグネチャと PDG を抽出をする．その結果，package3.Class3.deleteData メソッド (引数 : *int*，返戻値 : *void*) のシグネチャと，**図 8** (a) の PDG を得る．

同様に，新規メソッドの開発終了時点でエディタ上か




(a) circle1 メソッドの PDG




(b) circle2 メソッドの PDG

図 6: 事例 1：抽出された PDG

ら指示することにより，エディタ上で記述されたメソッドについてシグネチャと PDG を抽出する．その結果，package4.Class4.getDataCount メソッド (引数：*String*, 返戻値：*int*) のシグネチャと，図 8(b) の PDG を得る．

#### 5.2.3 機能的類似コード判定（1）：入出力の比較

package3.Class3.deleteData メソッドと package4.Class4.getDataCount メソッドの入出力の型を比較する．引数の型はそれぞれ int 型と String 型であり，また返戻値の型はそれぞれ void 型と int 型であり，どちらも一致しない．このため，入出力が類似していないとみなし，入力に対する出力の比較を行わない．

#### 5.2.4 機能的類似コード判定（2）：構造の比較

図 8 に示した 2 つのメソッドの PDG の類似度を算出する．まず，図 8 に吹き出しで示すノードを，起点ノードとして決定する．次に 2 つのメソッドの 9 つずつの起点ノードの 81 通りの組み合わせに対して一致する部分グラフの探索を行うと，図 8(b) に示す 31 ノード中で 13 ノードが部分ノードに含まれたため，構造の類似度は 41.9%と算出された．

```
void deleteData(int id){
   String db=”jdbc:sqlite:tableA.db”;
   String sql = ”delete from table A where id = ” + id;
   Connection conn = DriverManager.getConnection(db);
   conn.createStatement().executeUpdate(sql);
   conn.close();
}
```

(a) 既存のメソッド (package3.Class3 クラス内)

```
int getDataCount(String name){
   String db=”jdbc:sqlite:tableB.db”;
   String sql = ”select * from table B where name =” + name;
   Connection conn = DriverManager.getConnection(db);
   ResultSet rs = conn.createStatement().executeQuery(sql);
   rs.next();
   int count = rs.getInt(1);
   conn.close();
   return count;
}
```

(b) 新たに作成したメソッド (package4.Class4 クラス内)

図 7: 事例 2：ロジックの再開発で生じた類似コード

### 5.3 事例 3：同一目的の別ロジックを開発した場合

#### 5.3.1 概要

同一目的で別のロジックを開発した事例を想定し，メソッドを**図 9** に示す．この 2 つのメソッドはどちらも入力として与えられた配列の中から最大値を抽出する処理を行っているが，異なるロジックで実装されている．開発者は類似コードである findMaxValue1 メソッドの存在を知らずに， findMaxValue2 メソッドを開発した．

このような過程により開発者が findMaxValue2 メソッドを記述したことを想定し，実装したツールを実行した．

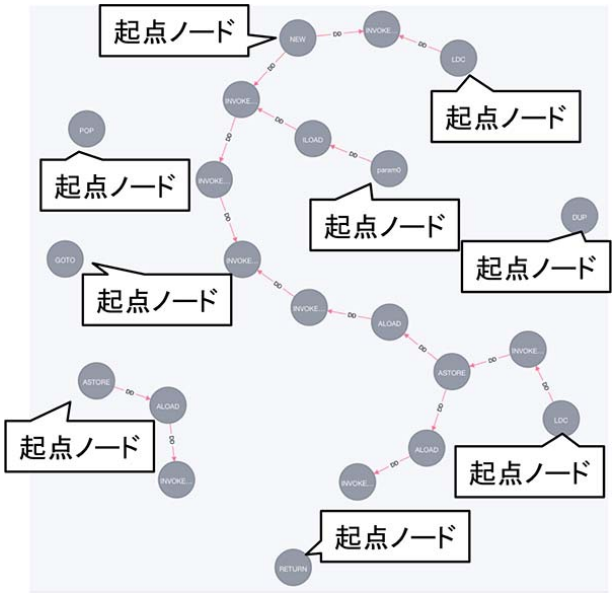
#### 5.3.2 メソッド情報の抽出

メソッド解析部は，定期的な実行によりリポジトリ内に存在するメソッドのシグネチャと PDG を抽出する．その結果として，package5.Class5.findMaxValue1 メソッド (引数：*double*[]，返戻値：*double*[]) のシグネチャと，**図 10** (a) の PDG を得る．

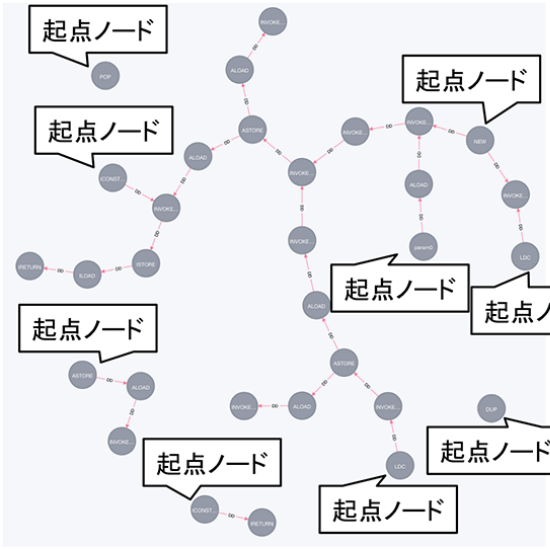
同様に，新規メソッドの開発終了時点でエディタ上から指示することにより，エディタ上で記述されたメソッドについてシグネチャと PDG を抽出する．その結果として，package6.Class6.findMaxValue2 メソッド (引数：*int*[]，返戻値：*int*[]) のシグネチャと，図 10(b) の PDG を得る．

#### 5.3.3 機能的類似コード判定（1）：入出力の比較

package5.Class5.findMaxValue1 メソッドと package6.Class6.findMaxValue2 メソッドの入出力の型を比較する．引数はそれぞれ数値型の配列である double 型の配列と int 型の配列であり一致している．また返戻値の型はそれぞれ数値型である double 型と int 型であり一致している．そこで，精度の低い int 型配列によって同一の入力値を生成




(a) deleteData メソッドの PDG




(b) getDataCount メソッドの PDG

図 8: 事例 2：抽出された PDG

| | 入力値 | 出力値 (findMaxValue1) | 出力値 (findMaxValue2) |
|---|---|---|---|
| 1 | {-82, 50} | 50.0 | 50 |
| 2 | {98, 82, -64} | 98.0 | 98 |
| 3 | {84, -13, 88, -18} | 88.0 | 88 |
| ⋮ | (略) | ⋮ | ⋮ |
| 20 | {22, -95} | 22.0 | 22 |

表 2: 事例 3 にて生成した入力値と対応する出力値

し，出力値の比較を行った（表 1）．なお紙面の都合上，ここでは配列の要素数を 5 個以内に，各要素の値を -100 から 100 までの間に制限している．入力値のセットを 20 セット生成し入力した結果すべての出力が一致したため，入出力の類似度は 100%と算出された．

```
static double findMaxValue1(double[] data){
  int max = -1;
  boolean isFirst = true;
  for (int x: data)
    if (isFirst || max < x){
      max = x;
      isFirst = false;
    }
  return max;
}
```

(a) 既存のメソッド (package5.Class5 クラス内)

```
static int findMaxValue2(int[] data){
  int i = 0, max = -1;
  for(i = 0; i < data.length; i++)
    max = i == 0 ? data[i]: Math.max(max, data[i]);
  return max;
}
```

(b) 新たに作成したメソッド (package6.Class6 クラス内)

図 9: 事例 3：同一目的の別ロジックで生じた類似コード

#### 5.3.4 機能的類似コード判定（2）：構造の比較

図 10 に示した 2 つのメソッドの PDG の類似度を算出する．まず，図 10 に吹き出しで示すノードを，起点ノードとして決定する．次に 2 つのメソッドのそれぞれ 7 個，4 個の起点ノードの 28 通りの組み合わせに対して一致する部分グラフの探索を行うと，図 10(b) に示す 25 ノード中で 8 ノードが部分ノードに含まれたため，構造の類似度は 32.0%と算出された．

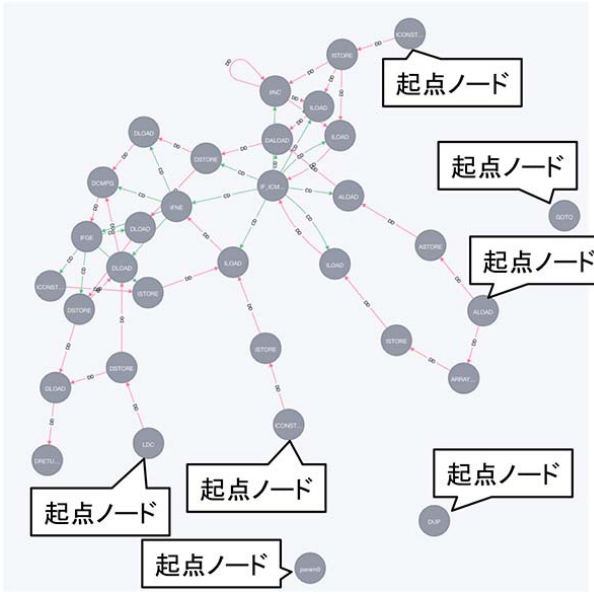
## 6. 考察

### 6.1 ケーススタディについて

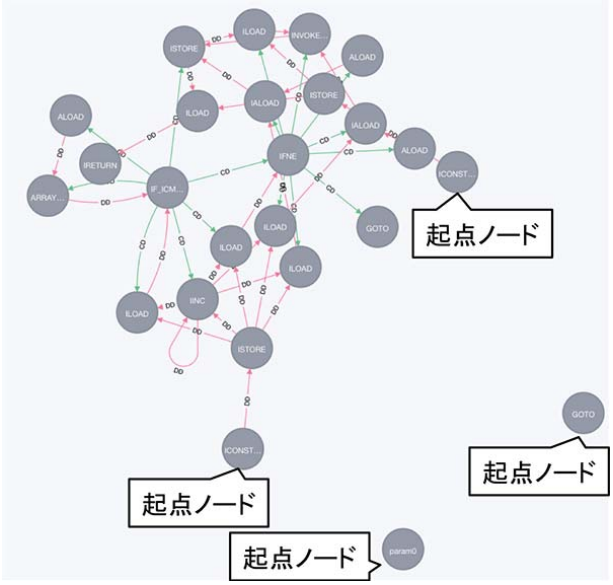
提案手法は，複数開発者が並行開発を実施する際に，似たような機能を提供するコードを作りこむことを防ぐために，機能的類似コードを検出することを目的としている．

事例 1 は，開発者が明らかに意図的に複製した場合の例を示した．この場合は入出力も構造も高い類似度を示す．図 4 の結果表示では，右上隅にプロットされることになる．事例 2 は，プロジェクト内でよく利用される定型的なロジックを再開発した場合の例を示した．この例ではロジックそのものの利用目的が異なるため，入出力の値を比較できない．しかし部分的に機能的類似コードが含まれるため，構造についてはある程度の類似性を示す．事例 3 は，同一目的のロジックが存在するにもかかわらず，それに気づかずに別開発した場合の例を示した．この例は元々存在したロジックとは別のロジックにより実装しており，構造の類似度は低くなる．しかし同じ目的で利用されるメソッドであるため，入力値と出力値の類似度が高くなる．

提案手法の目的を考えると，事例 1 のように同じ開発者により意図的に複製されたコードは，検出結果として表示




(a) findMaxValue1 メソッドの PDG




(b) findMaxValue2 メソッドの PDG

図 10: 事例 3：抽出された PDG

すべきでない可能性がある．またこの事例の範囲内では，事例 2 や事例 3 のように，開発者が意図的ではなく，または存在の可能性について想定しつつもそれを調べることなく作成した結果生じた類似コードについては，検出結果として効果的に提示すべきである．しかし，これらの事例では，入出力や構造の類似度のいずれかが極端に低くなり，図 4 の表示では，軸に寄った値として表示される．特に事例 2 については構造の類似度も突出して高い値ではなかった．現在の実装の検出結果表示では，結果表示を確認した開発者が，機能的類似コードの検出結果を見落としてしまい，リファクタリングにつながらない可能性もある．意図的ではなく作りこんだ類似コードであることを分類して，効果的に結果を提示することが，今後の課題である．

### 6.2 制限事項

提案手法の現状の実装では，次の制限がある．

- 入力値生成の方法について
  入力値を単純なランダム値によって生成しているため，開発者の意図に沿っていない値や，メソッドの処理に対して無意味な値を入力してしまう場合がある．
- PDG による構造の比較の方法について
  起点ノードから連続する一致する部分グラフを求める際に，両方のメソッドがほとんど一致する部分グラフを含んでいても，片方の部分グラフ内にノードが挿入されていた場合，一致する部分グラフと判定できない．

## 7. 結論

本論文では入力に対する出力と構造の比較を用いた，機能的類似コード検出手法を提案した．これにより，開発者がコード開発時に機能的類似コードの存在を意識でき，効果的なリファクタリング実施により，保守性を向上できる．

現在の実装では，検出結果をその類似度とともに表示しているのみであるため，検出結果を提示することがリファクタリング実施につながらない場合があり得る．このため，保守性向上につながると考えられる類似コードに対する効果的な表示方法を検討することが今後の課題である．

### 参考文献

[1] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591 (2007).

[2] Roy, C. K. and Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, *IEEE International Conference on Program Comprehension*, pp. 172–181 (2008).

[3] Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J. and Nguyen, T. N.: Clone Management for Evolving Software, *IEEE Transactions on Software Engineering*, Vol. 38, No. 5, pp. 1008–1026 (2012).

[4] Wang, T., Wang, K., Su, X. and Ma, P.: Detection of semantically similar code, *Frontiers of Computer Science*, Vol. 8, No. 6, pp. 996–1011 (2014).

[5] Higo, ., asushi, U., Nishino, M. and Kusumoto, S.: Incremental Code Clone Detection: A PDG-based Approach, *Working Conference on Reverse Engineering*, IEEE Computer Society, pp. 3–12 (2011).

[6] Deissenboeck, F., Heinemann, L., Hummel, B. and Wagner, S.: Challenges of the Dynamic Detection of Functionally Similar Code Fragments, *European Conference on Software Maintenance and Reengineering*, pp. 299–308 (2012).

[7] 秦野智臣，石尾　隆，井上克郎：SOBA:シンプルな Java バイトコード解析ツールキット，コンピュータソフトウェア， Vol. 33, No. 4, pp. 4–15 (2016).

[8] Technology, N.: Neo4j, the world s leading graph database - Neo4j Graph Database, https://neo4j.com/ [Accessed JUL. 3, 2017].