情報処理学会論文誌 Vol. 52 No. 2 801–816 (Feb. 2011)

不具合修正に関わるメソッド呼び出しの変更についての定量的分析

畑 秀明[†1] 森井亮介[†1]
水野 修[†2] 菊野 亨[†1]

ソフトウェアの開発において，プログラムの再利用や効率的なプログラム作成のために，API（Application Program Interface）が用いられる．しかし，使用方法や使用例についてのドキュメントが整備されていないことも多く，API を適切に使用することは難しい．API の設計や提供方法などについての定性的な議論は行われているが，これまで定量的な分析はほとんど行われていない．本稿では，不具合修正時に各メソッド呼び出しに対する変更が実際にどれほど行われているかを分析する．特に，不具合修正時に頻繁に修正される特徴的なメソッド呼び出しがあるか，複数のプロジェクトで共通して頻繁に変更されているメソッド呼び出しはあるか，プロジェクトの時間経過によって変更対象となるメソッド呼び出しに変化はあるか，を明らかにすることを目標とした．各変更がどのメソッド呼び出しに影響するかを明らかにするため，プログラム依存グラフに基づく分析を行った．7 つの Java のオープンソースプロジェクトに対してケーススタディを行った．分析結果から，プロジェクト特有の頻繁に変更されるメソッド呼び出しがあること，複数のプロジェクトにて頻繁に変更されているメソッド呼び出しもあること，時間経過によって変更対象となるメソッド呼び出しが変化することを明らかにした．

Quantitative Analysis of Method Call Changes Related to Bug Fixing

Hideaki Hata,[†1] Ryosuke Morii,[†1] Osamu Mizuno[†2] and Tohru Kikuno[†1]

In developing software, developers use APIs to reuse existing programs but APIs are said to be difficult to use properly because of insufficient documents or code examples. Though there are several notions about the difficulty of APIs, there are few studies conducting quantitative analysis of bug-related APIs. In this paper, we conduct quantitative analysis of method calls related to bug fixing. The main research questions in this paper are: Which method calls are more frequently changed, are there common method calls frequently changed over projects, and are there differences among periods in frequently changed method calls? To answer these questions, bug-fix changes in software repositories that affect method calls are explored. To capture the effects of change on method calls, analyzing only syntactic structure changes is not appropriate. We conducted program dependency graph based analysis to capture data dependency and control dependency. An empirical case study is conducted with seven open source projects and project-specific method calls that are frequently changed are obtained. Also, particular method calls are found to be changed among several projects. In addition, it reveals that particular method calls are frequently changed in a particular period.

1. はじめに

多くのソフトウェア開発プロジェクトにおいて API（Application Program Interface）が用いられる．API は提供される様々な機能へのインタフェースであり，プログラムの関係を抽象化することで，一貫性のあるプログラム作成やソースコードの再利用を促進し，生産性の向上が期待される．しかしながら，API は複雑でドキュメントが整備されていないことも多い．また API は時とともに大きく多様に拡張される．これらのことから，使い方を習得するのは難しいともいわれている[14),16)]．

Robillard は開発者へのインタビュを通して API 使用における問題を調査している[14)]．調査結果から，いくつかの課題が指摘されている．まず，API の設計における抽象的な考え方などの情報が提供され，理解されることが必要である．次に，API の使用例の意図するものと使用者の目的との間に食い違いがある場合，その使用例が理解への障害となることがある．また，抽象的な設計が他の API の設計と異なっているため，API の動作が予期したものと異なり，障害の原因となることがある．これに関しては，ユーザや開発者が最も自然に思えるように設計すべきであるという，驚き最小の原則に従うことが推奨されている[2),14)]．

このように API についての定性的な問題がいくつか提示されている．API 使用の難しさは不具合混入の原因となることもある[2)]．しかしながら，API 使用の問題についての定量的な分析に関する研究は十分には行われていない．本稿では，Java の API 使用におけるメ

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
†2 京都工芸繊維大学大学院工芸科学研究科
Graduate School of Science and Technology, Kyoto Institute of Technology

ソッド呼び出しに限定して定量的な分析を試みる．ただし，メソッド呼び出し使用の問題点を直接分析することは難しいので，各メソッド呼び出しが不具合修正時に変更される回数を測定し，分析する．本稿では，特に以下の 3 つの点について分析を行う．

- 各プロジェクトにおいて，不具合修正時に頻繁に変更される特徴的なメソッドはあるか．
- 不具合修正時に変更されるメソッド呼び出しに，複数のプロジェクトに共通するものはあるか．
- プロジェクトの開発時期によって，不具合修正時の変更対象となるメソッド呼び出しに変化はあるか．

本稿では，ソフトウェアリポジトリの履歴を調査し，分析を行う．Pan らは 27 の不具合修正パターンを定義し，オープンソースプロジェクトにおける不具合修正を調査している[11)]．調査の結果，不具合修正時に，MC-DAP（Method Call with Different Actual Values，メソッド呼び出しにおける引数の変更），IF-CC（Change in IF Condition，`if` 条件における変更），AS-CE（Change of ASsignment Expression，代入処理における変更）といったものが，頻繁に見られるパターンだと報告されている．MC-DAP パターンが頻出するということは，特定のメソッド呼び出しが不具合と関わりがあることをほのめかしている．IF-CC パターンと AS-CE パターンは，それぞれ制御構造の変更や変数の値の変更である．これらの変更も特定のメソッド呼び出しに関連しているかもしれないが，特定はできない．これは，彼らの分析手法が行単位の構造変化のみを分析していることによる．

プログラムの変更についての分析は非常に大切な技術であるため，様々な研究が行われている．例として，テキストベースの研究[3),17)]，グラフベースの研究[1),4),5),8)]，バイナリベースや[12),15)]，ドキュメントベース[9)] などの研究があげられる．どの手法が適当であるかは，変更分析の目的次第である．メソッド呼び出しに関連した変更を分析する際には，テキスト上の構造変化を分析するだけでは不十分である．どの変更がメソッド呼び出しへ影響するかを特定することが必要である．

Jackson らは，プログラムの 2 つのバージョン間の変更の影響を出力する手法を提案している[6)]．彼らは，変更の影響を把握するため，データの依存関係を分析している．しかしながら，制御の依存関係は分析されていない．Parnin らは，変更におけるコンテキストまでも把握する研究を行っている[12)]．たとえば，ある変数の値が変更され，その変数があるメソッド呼び出しの引数となる場合，その変更はメソッド呼び出しの引数に対する変更であると出力する手法を提案している彼らの研究の目的は，どんな変更についても適切な粒度のコンテキスト情報を提供することであるため，狭い範囲での影響だけを分析対象としている．

本稿では，Java のメソッド単位のプログラム依存グラフ（Program Dependency Graph，以降 PDG）から，データ依存と制御依存の関係を取得する．不具合修正時前後での差分から，メソッド内で変更が影響するメソッド呼び出しを特定する．本稿では，変更により影響を受けるメソッド呼び出しを変更されたメソッド呼び出しと呼ぶ．

プログラムの変更のうち，不具合修正時の変更を対象に分析を行う．しかし，不具合修正時に変更されたメソッド呼び出しのすべてに不具合があって修正されたとはいえない．また，変更されたメソッド呼び出しのうち，不具合修正の意図があったものを適切に検出することは難しい．本稿では，実際に不具合があったかどうかは特定しないが，これらの問題点をふまえて分析を行う．

Eclipse 関連の 7 つのオープンソースプロジェクトを対象としてケーススタディを行った．その結果次のような知見を得た．(i) 不具合修正時に変更されるメソッド呼び出しで頻繁に変更されるものは一部である．(ii) 複数のプロジェクトで，共通して不具合修正時に変更されるメソッド呼び出しが存在する．(iii) 時間の経過によって，不具合修正時に変更対象となるメソッド呼び出しは変化する．

以降，2 章ではどのようなプログラムの変更をメソッド呼び出しの変更とするかを説明し，3 章でプログラムの変更における分析手法を説明する．4 章で，行ったケーススタディとその結果を示し，5 章でその詳細について考察する．関連研究については 6 章で議論し，最後に 7 章でまとめと今後の課題について述べる．

2. 対象とするメソッド呼び出しの変更

本稿では，メソッド内での変更がどのメソッド呼び出しに影響を与えるかを特定する．メソッド呼び出しに関連した変更にはどのようなものがあるかを**図 1** を用いて説明する．以下のような変更がある場合，メソッド呼び出し mc に関する変更が行われたとし，メソッド呼び出し mc は変更されたと呼ぶ．

(1) メソッド呼び出し mc が挿入された，または削除された．
（メソッド mth_1 において，メソッド呼び出し mc_1 が挿入されている．）

(2) メソッド呼び出し mc の引数部分が変更された．
（メソッド mth_1 において，メソッド呼び出し mc_2 に対して引数部分が a から $a+1$ へ変更されている．）

(3) メソッド呼び出し mc へ直接引数として渡される変数に変更が加えられた．
（メソッド mth_2 において，変数 b に関する文が “`b++`” から “`b--`” へ変更されてい

る．その変数 b は，メソッド呼び出し mc_3 の引数となる．）

(4) メソッド呼び出し mc の引数計算に使われる変数に変更が加えられた．
（メソッド mth_2 において，変数 b に対して変更が加えられており，その影響が変数 c に及んでいる．その変数 c は，メソッド呼び出し mc_4 の引数となる．）

(5) メソッド呼び出し mc の実行を直接制御する条件が追加，除去，または変更された．
（メソッド mth_3 において，メソッド呼び出し mc_5 を制御する条件が “`if (e % 2 == 0)`” から “`if (e % 3 == 0)`” へ変更されている．）

(6) メソッド呼び出し mc の引数に直接または間接的に使われる変数への制御が追加，除去，または変更された．
（メソッド mth_3 において，変数 d の値を計算する文を制御する条件が追加されている．その変数 d は，メソッド呼び出し mc_5 の引数となる．）

変更前のクラス C

```
public class Cls {
  void mth1 (int a) {
    mc2(a);
  }
  void mth2 (int b) {
    b++;
    mc3(b);
    int c = b * 2;
    mc4(c);
  }
  void mth3 (int d, int e) {
    d--;
    if (e % 2 == 0) mc5(d+e);
  }
}
```

変更後のクラス C'

```
public class Cls {
  void mth1 (int a) {
    mc1(a);
    mc2(a+1);
  }
  void mth2 (int b) {
    b--;
    mc3(b);
    int c = b * 2;
    mc4(c);
  }
  void mth3 (int d, int e) {
    if (d > 0) d--;
    if (e % 3 == 0) mc5(d+e);
  }
}
```

図 1　修正前後のクラスの例
Fig. 1　Example of an original class and a modified class.

3. 分析手法

3.1 概　　要

本稿では Java を対象とし，メソッドレベルでの変更を分析する．**図 2** に分析手法の概要を示す．分析は変更前後の Java ファイル F と F' を，ソフトウェアリポジトリから用意することから始まる．それぞれのファイル中のクラスにおいて，クラス名の完全一致から対応するクラスペアを抽出する．それぞれのクラス中から，メソッドシグネチャの完全一致から対応するメソッドのペアを見つけだし，各メソッドペアごとに比較を行う．メソッド呼び出しへ影響する変更があった場合，そのメソッド呼び出しを変更されたメソッド呼び出しとして特定する．以上のように，本稿ではクラスペアとメソッドペアは完全一致のみを対象としており，不具合修正時にクラス名やメソッドシグネチャの変更があったものは分析の対象外としている．

2 章に示した，メソッド呼び出しへ影響を与える変更を検出することを目的とした研究は我々の知る限り行われていない．Pan らが行った行単位の構造変化の分析では，2 章に示し

入力： 修正前後のクラス F と F'
出力： 変更されたメソッド呼び出し集合 MC

F と F' から，対応するクラスを抽出
for それぞれのクラスペア (C, C') **do**
　C と C' から，対応するメソッドを抽出
　for それぞれのメソッドペア (mth, mth') **do**
　　mth と mth' 間での変更を分析
　　変更が影響するメソッド呼び出しを特定
　　特定したメソッド呼び出しを MC へ追加
　end for
end for
return MC

図 2　分析手法の概要
Fig. 2　Overview of change analysis.

た変更のうち (3) から (6) は検出できない[11]．また，Parnin らはメソッド呼び出しの引数となる変数に対する変更を分析しているが，その分析では直接引数となる変数のみを対象としている．そのため (4) と (6) は分析対象外である．本稿では，対象とする変更を検出するため，変数のデータ依存と制御の依存関係を考慮した，グラフに基づく分析手法を新たに考案した．

3.2　メソッドからのデータ取得

3.2.1　プログラム依存グラフ

メソッド呼び出しと変数とのデータ依存関係だけでなく，制御の依存関係も把握するため，まずメソッドレベルでプログラム依存グラフ（PDG）を構築する．PDG はプログラムの要素（文や条件節の式）をノードとした有向グラフである．有向グラフの辺は，ノード間のデータ依存と制御依存の関係を表す．

Java メソッドからの PDG 構築を**図 3** を用いて説明する．図 3 (a) は，例となるメソッドのソースコードである．文 “int $y = x * 10$”，条件節の式 “$x < 0$” などは PDG のノードとなる．ノード n_A とノード n_B との間に以下のデータ依存関係がある場合，n_A から n_B へデータ依存辺 d_{AB} が引かれる．

- n_A：変数 v を定義または変数 v に対して代入を行うノード
- n_B：ノード n_A において定義または代入された変数 v を参照しているノード．同様の参照をしている場合，すべてのノードに辺 d_{AB} が引かれる．

辺 d_{AB} には，変数 v がラベル付けされる．

ノード n_A とノード n_B との間に以下の制御依存関係がある場合，n_A から n_B へ制御依存辺 c_{AB} が引かれる．

- n_A：条件節の式であるノード
- n_B：n_A の条件下にあるブロックに含まれるノード．ブロック中に複数のノードが含まれる場合，すべてのノードに辺 c_{AB} が引かれる．

辺 c_{AB} には，ノード n_A の条件によって “true” か “false” かのラベル付けされる．

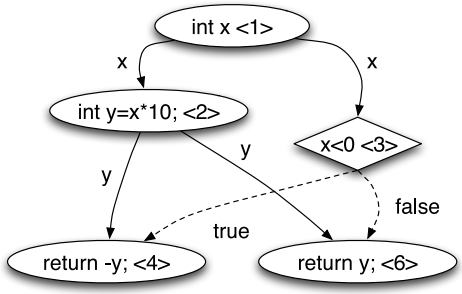
図 3 (b) はソースコード (a) から構築された PDG である．データ依存を表す辺は変数名のラベルを付けた実線を，制御依存を表す辺には “true” か “false” のラベル付けされた点線で，矢印を引いている．

3.2.2　依存関係の把握

PDG を構築することによって，プログラム要素間での依存関係を把握できる．さらに，メソッド呼び出しがどの文に含まれるか，変数がどのメソッド呼び出しの引数となるか，といった詳細な情報も取得する．

```
1    int mth_s (int x) {
2      int y = x * 10;
3      if (x < 0){
4        return -y;
5      }else{
6        return y;
7      }
8    }
```

(a) メソッド mth_s のソースコード
(a) Source code of a sample method mth_s



(b) メソッド mth_s から構築される PDG
(b) PDG constructed from mth_s

図 3　ソースコードから構築される PDG の例
Fig. 3　A PDG constructed from source code of a method.

ここで注意すべき点は，上記のように構築した PDG の 1 ノードは複数のメソッド呼び出しを含むことがあり，また複数の変数計算を含むことがあるということである．たとえば次に示した文は，変数 x が引数となるメソッド呼び出し mc_a と，変数 y が引数となるメソッド呼び出し mc_b を含む．またこの文は，変数 y の値を更新しており，変数 z の値を計算している．

$$\mathrm{z} = mc_a(\mathrm{x}) + mc_b(\mathrm{y{+}{+}});$$

例の文の前にて，変数 x に何らかの変更が加えられた場合，メソッド呼び出し mc_a と変数 z による以降の処理には影響があると考えられるが，変数 y が変数 x への変更後に値を参照しない限り，メソッド呼び出し mc_b には影響がない．

本稿では，上記の例を以下のように新たな変数 MC_A と MC_B を導入し，分割したプログラム要素として扱う．

$$\mathrm{Y} = \mathrm{y{+}{+}};$$
$$MC_A = mc_a(\mathrm{x});$$
$$MC_B = mc_b(\mathrm{Y});$$
$$\mathrm{z} = MC_A + MC_B;$$

つまり以降のプログラム要素に対して変数の参照元となる場合に，たかだか 1 種類の変数のデータ依存の参照元となり，たかだか 1 つのメソッド呼び出しを含むものをプログラム要素として扱う．これにより，各プログラム要素のデータ依存をたどることで，そのプログラム要素が影響を与えるメソッド呼び出しがただちに把握できる．

各プログラム要素のデータ依存関係から，参照するプログラム要素を順にたどり（図 3 (b) の矢印の方向），変数を直接引数とするメソッド呼び出しと，間接的に引数とするメソッド呼び出しを以下のように特定する．元のプログラム要素が変数 v の参照元とすると，変数 v が直接引数となるメソッド呼び出しと間接的に引数となるメソッド呼び出しは，以下のように定まる．

- 直接引数となるメソッド呼び出し：元のプログラム要素からのデータ依存辺をたどって到達可能なプログラム要素の中で，変数 v を引数とするメソッド呼び出し
- 間接的に引数となるメソッド呼び出し：元のプログラム要素からのデータ依存辺をたどって到達可能なプログラム要素の中で，変数 v 以外の変数を引数とするメソッド呼び出し

以下の例では，

```
x++;
mc_c(x);
y = x+1;
mc_d(y);
```

プログラム要素 `x++;` への変更は，変数 x が直接引数となるメソッド呼び出し mc_c へと，変数 y を介して間接的に引数となるメソッド呼び出し mc_d へ影響を与えると特定する．

また，プログラム要素がどの制御条件下にあるかという情報を，制御条件の順番の情報とともに保持する．プログラム要素から制御依存辺を逆にたどり，条件節の内容と制御辺のラベル，たどった順番を取得する．制御構造の情報について，次の例を用いて説明する．

```
if (obj == null) { }
else {
  if (str != null) {
    str = obj.toString(); }
}
```

例のプログラム要素 “`str = obj.toString();`” では，制御条件 “`str == null`” が true で 1 番目，制御条件 “`obj != null`” が false で 2 番目と情報を取得する．

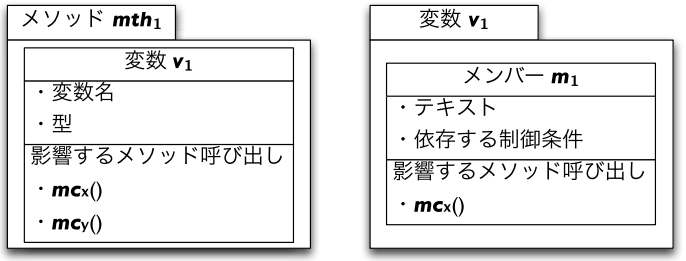
以上の情報は公開されているツール，MASU [19),⋆1] で取得した．

3.2.3　データ構造の構築

変更前後の対応する 2 つのメソッド mth と mth' を比較して，変更が影響するメソッド呼び出しを特定する．プログラム中の要素がそれぞれ対応するかどうかを調べ，変更によって対応する要素がなくなったものについて，データ依存や制御依存によって影響を受けるメソッド呼び出しがあるかどうかを調べる．

プログラム要素の依存関係から，各変数がどのプログラム要素の処理を経てメソッド呼び出しの引数となるかが把握できる．我々は，**図 4** に示すようなデータ構造を構築して分析を行う．まず，メソッド内で各変数ごとにデータをまとめる．図 4 (a) に示すように，各変数のデータには変数名と変数の型を属性として持たせる．さらに，この変数が直接的，間接的に引数となるメソッド呼び出しも影響するメソッド呼び出しとして保持する．このとき，プログラム要素への分解のために導入した変数は除外する．

また各変数ごとに，その変数のデータ依存があるプログラム要素をその変数のメンバと定義し，データをまとめる．図 4 (b) に示すように，各メンバのデータには，そのプログラム要素のテキスト情報と依存する制御条件を属性として持ち，影響するメソッド呼び出しも保



(a) 各変数のデータ構造
(a) Data structure of each variable

(b) 各メンバのデータ構造
(b) Data structure of each member

図 4　データ構造
Fig. 4　Data structure.

⋆1 http://sourceforge.net/projects/masu/

持する．テキスト情報においては，空白を統一したフォーマットで保持する．プログラム要素は複数の変数のメンバとなりうる．

次のソースコードを例として，どのようなデータ構造となるかを説明する．

```
int i = 10;
int j = i*3;
mc1(j);
mc2(i);
```

変数 i，j についてはデータ構造として，それぞれ変数名 i，j と型 `int` を属性として持つ．また，変数 i が影響するメソッド呼び出しとして $mc_2(int)$ を，変数 j が影響するメソッド呼び出しとして $mc_1(int)$ をそれぞれ保持する．また，変数 i，j それぞれにメンバとして文 "`j = i*3;`" を保持する．また文 "$mc_2(i)$`;`" は，変数 i のメンバであり，影響するメソッド呼び出し $mc_2(int)$ を保持する．同様に，文 "$mc_1(j)$`;`" は変数 j のメンバとなる．

次の例は，1 文中に複数のメソッド呼び出しを含んでいる．

```
k = mc1( mc2(j) + mc3(j) + j );
```

この文は，以下のプログラム要素に分解して扱う．

```
MC2 = mc2(j);
MC3 = mc3(j);
MC1 = mc1(MC2 + MC3 + j);
k = MC1;
```

この場合，変数 k においては `k =` MC_1 のみがメンバである．変数 j においては，上から 3 つのプログラム要素がメンバとなる．

3.3　アルゴリズム

3.1 節で，変更分析の全体の概要を示した．分析は変更前後の Java ファイルの入力から始まる．両ファイルから対応するメソッドを特定し，それぞれ変更を調査する．**図 5** にメソッドレベルでの変更分析アルゴリズムを示す．

まず，それぞれのメソッドペア mth と mth' において，すべての変数データが両メソッドに存在するかどうかを調査する．このフェーズにおいて，変数名と型という 2 つの属性がともに等しい変数データは対応する変数ペアとして特定する．対応する変数データがなく，

```
for それぞれのメソッドペア (mth, mth') do
  mth と mth' から，対応する変数データがあるか調査
  if 対応しない変数データがある and
  その変数データが影響するメソッド呼び出しがある then
    該当するメソッド呼び出しを MC へ追加
  end if
  for それぞれの変数ペア (v, v') do
    v と v' から，対応するメンバデータがあるか調査
    if 対応しないメンバデータがある and
    そのメンバデータが影響するメソッド呼び出しがある then
      該当するメソッド呼び出しを MC へ追加
    end if
  end for
end for
return MC
```

図 5　変更が影響するメソッド呼び出しを特定するアルゴリズム
Fig. 5　Algorithm detecting method calls affected by changes.

一方のメソッドのみに存在する変数データがある場合は，その変数の追加，除去，変数名や型の変更のいずれかがあったということである．その変数データが，影響されるメソッド呼び出しを保持している場合，そのメソッド呼び出しは変更によって影響されるメソッド呼び出しとして特定される．このフェーズでは，2 章で列挙した変数に関わる変更（(3) と (4) にあたる）を特定できる．

次に，それぞれの変数ペア v と v' において，すべてのメンバデータが両変数に存在するかどうかを調査する．このフェーズにおいて，テキストと依存する制御条件という 2 つの属性がともに等しいメンバデータがなく，影響するメソッド呼び出しがある場合，以下の変更が 1 つ以上あったということになる．

- 変数 v に関わるテキストが挿入，削除，変更された（テキストにメソッド呼び出しが含まれているものは，2 章の (1) や (2) に相当する．含まれていないものは，変数の値を変える可能性のある変更であり (3) や (4) にあたる）．
- 制御構造が追加，除去，変更された（(5) と (6) にあたる）．

3.4 問　題　点

変数にはローカル変数，パラメータ変数，インスタンスフィールド変数，クラスフィールド変数などがあるが，本稿では区別せず扱っている．そのためフィールドを介して影響する変更は対象外としている．配列に関しては，配列変数の依存関係のみを取得しており，配列の要素ごとのデータ依存は取得できていない．そのため，配列の要素に関連した変更は特定していない．

また PDG の情報に基づいているため，構文の順序による依存関係は無視している．

```
obj.mc_a(x);
obj.mc_b(x);
```

上記の例で，メソッド呼び出し mc_b の前にメソッド呼び出し mc_a があることに意味があっても，データ依存のない文の順序が入れ替わるような変更については，本手法では検出できない．

また，データの依存関係のない文の変更もメソッド呼び出しに関わる変更として検出しない．たとえば，“`break;`” や “`continue;`” といった文はデータの依存関係がない．加えて本手法では，引数となる変数のデータ依存から分析を行っているため，引数を持たないメソッド呼び出しの変更（挿入と削除）を検出できない．本稿では，引数を持たないメソッド呼び出しを対象外として以降の分析を行う．

本稿では，実際のソフトウェアプロジェクトの複数年のデータという大量のデータを解析するため，Java ファイルの分析ではファイル間の変更分析のみを行っている．依存関係のある他のソースコードやライブラリの情報がないため，型情報などが不明な場合がある．オーバライドのメソッド呼び出しに関しては，MASU によりインスタンスの型情報が取得できている場合，どのサブクラスのメソッドであるかは特定し区別している．しかし型情報が不明なメソッド呼び出しに関しては，本稿では無視している．この件に関しては 5.1 節で議論する．

4. ケーススタディ

対象プロジェクトには，**表 1** に示した 7 つのオープンソースプロジェクトを選んだ．Java のプロジェクトであり，CVS によってバージョン管理されている．実験データは 2009 年 10 月 1 日に Eclipse のサイト[⋆1]から取得した．

⋆1 http://archive.eclipse.org/arch/

表 1　対象プロジェクト
Table 1　Target projects.

プロジェクト	リポジトリのサイズ（MB）	期間（開始–取得）
BIRT	24.2	01/2005–10/2009
Datatools	9.2	09/2001–10/2009
Modeling	95.1	12/2006–10/2009
Technology	69.4	12/2002–10/2009
Tools	67.8	09/2001–10/2009
TPTP	17.3	12/2002–10/2009
Webtools	50.8	10-2003–10/2009

変更のうち特に不具合修正を調査するため，Mockus と Votta が行っているようにリポジトリのログに着目する[9]．変更ログに “fix” や “bug” といった不具合修正に関連したキーワードを含んでいる場合，その変更を不具合修正時の変更として扱う．

4.1 メソッド呼び出し変更回数の評価方法

定量的分析を行うため，不具合修正時のメソッド呼び出し変更回数を数える．ここで，各メソッド呼び出しは使用回数が異なる．使用回数の多いメソッド呼び出しは使用回数の少ないメソッド呼び出しに比べて，変更回数は多くなりやすいため，単純な変更回数での評価は適切ではない．そこで，開発履歴全体でのメソッド呼び出しの変更回数を，ある時点でのそのメソッド呼び出しの使用回数で割った値をそのメソッド呼び出しの平均変更回数として評価する．

本稿では，2006 年 1 月 1 日の時点での使用回数を基準とした．また，評価対象のメソッド呼び出しは，2006 年 1 月 1 日に存在し，かつ 2009 年 1 月 1 日においても存在するものに限定した．2 時点の間でメソッドシグネチャが変更され，また元に戻されるということもありうるが，この限定によって，大部分はメソッドシグネチャの変更がなかったメソッド呼び出しに限定した調査となる．

不具合修正時の変更回数が多いものであっても，同様に不具合修正以外の変更時にも頻繁に変更されるメソッド呼び出しもありうる．これらは，不具合修正時の特徴的なメソッド呼び出しとはいい難い．そのため，不具合修正時だけでなく他の変更時も同様に変更回数を分析する．本稿では各メソッド呼び出しにおいて，全変更時の変更回数のうち不具合修正時の変更回数（以降，不具合修正時変更率と呼ぶ）が 50%未満のものは除外して扱った．

表 2 に詳細情報を表す．表 2 の 2 列目に 2006 年 1 月 1 日でのファイル数を，3 列目に 2009 年 1 月 1 日でのファイル数を示す．また，不具合修正時に変更されたメソッド数を 4

表 2　プロジェクトごとの詳細情報
Table 2　Detail information in each project.

	ファイル数		変更メソッド数	
プロジェクト	2006 年 1 月 1 日	2009 年 1 月 1 日	不具合修正	不具合修正以外
BIRT	2,223	3,396	27,987	15,465
Datatools	1,122	481	1,510	4,230
Modeling	4,549	10,127	23,688	71,668
Technology	688	6,941	18,633	39,977
Tools	6,099	6,400	42,481	44,041
TPTP	2,475	898	13,232	1,633
Webtools	6,905	5,271	16,321	58,496

表 3　対象メソッドのうち主に不具合修正時に変更されたメソッド呼び出し
Table 3　Changed method calls in target method calls.

プロジェクト	対象メソッド呼び出し	主に不具合修正時に変更されたメソッド呼び出し
BIRT	5,972	1,702
Datatools	841	64
Modeling	7,567	327
Technology	1,021	186
Tools	6,871	1,508
TPTP	3,294	1,129
Webtools	9,160	447

列目に示し，5 列目に不具合修正以外の変更時に変更されたメソッド数を示す．

4.2　修正時に頻繁に変更されるメソッド呼び出し

表 3 にメソッド呼び出し数の情報を示す．4.1 節で示した対象メソッド呼び出しのうち，不具合修正時に 1 回以上変更され，不具合修正時変更率が 50%を超えたものを，主に不具合修正時に変更されたメソッド呼び出しとしてその総数を示した．プロジェクトによって，対象メソッドのうち主に不具合修正時に変更されたものが 30%を超えるものから（TPTP プロジェクト），5%に満たないものまで（Modeling プロジェクト）見られた．

以降は主に不具合修正時に変更されたメソッド呼び出しについて分析を行う．**図 6** は，BIRT プロジェクトにおけるメソッド呼び出しの平均変更回数についてのヒストグラムである．変更されたメソッド呼び出しのうち，平均変更回数が 5 回以下のメソッド呼び出しは 90%以上を占めた．それに対して，一部のメソッド呼び出しは 10 回以上も修正されているものがあることが分かった．他の 6 つのプロジェクトにおいても，変更回数の偏りに同様の傾向が見られた．この結果から，一部に頻繁に修正されるメソッド呼び出しが存在するが，

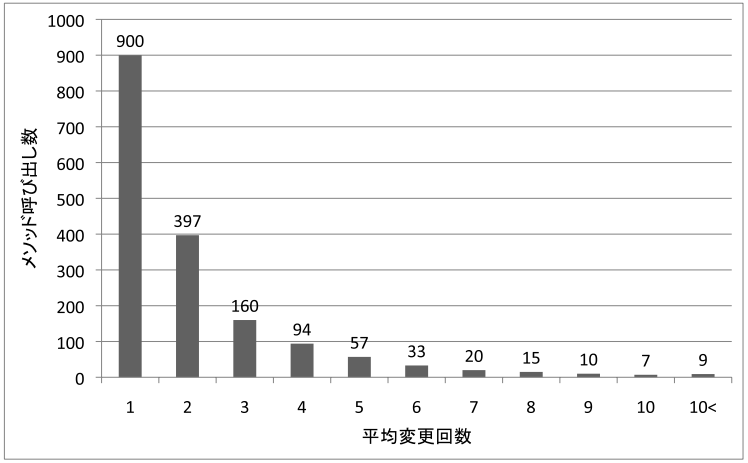


図 6　BIRT プロジェクトにおけるメソッド呼び出しの平均変更回数についてのヒストグラム
Fig. 6　Histogram of the changed times of method calls in the BIRT project.

多くは数回修正される程度ということが分かる．

最初に掲げた分析目標「各プロジェクトにおいて，不具合修正時に頻繁に変更される特徴的なメソッド呼び出し」を確認するため，**表 4** に各プロジェクトごとの，頻繁に修正されたメソッド呼び出し上位 5 つをそれぞれの平均変更回数とともにまとめた．また，不具合修正時変更率を 95%信頼区間とともに示す．不具合修正時変更率が 50%以上のものをまとめているが，標本数が少なく信頼区間が広いものがあることに留意する必要がある．各プロジェクトにおいて，ほとんどのメソッド呼び出しは数回しか変更されないのに対して，平均変更回数上位のメソッド呼び出しには 10 回を超えて変更されていることが分かる．

表 4 に見られるように，各プロジェクトにおいて修正されるメソッド呼び出しには，プロジェクトで作成されたものも，Java Platform，Standard Edition（Java SE）API で用意されているものも含まれている．プロジェクトで作成されたメソッド呼び出しに関しては，修正されやすい，不具合に関連しやすいメソッド呼び出しとして，プロジェクト特有のものと考えられる．

4.3　共通して変更されるメソッド呼び出し

表 3 で特定された，主に不具合修正時に変更されたメソッド呼び出しの中で，複数のプロジェクトで現れるものがあるかを調査した．7 つのプロジェクトでの平均変更回数の中央

表 4　頻繁に変更されたメソッド呼び出し上位 5
Table 4　Top 5 changed method calls.

プロジェクト	メソッド呼び出し	平均変更回数	不具合修正時変更率（95%信頼区間）（%）
BIRT	org.eclipse.birt.core.script.JavascriptEvalUtil.convertJavascriptValue(java.lang.Object)	21.0	70.0 (53.6–86.4)
	org.eclipse.birt.chart.model.ChartWithAxes.setSampleData(org.eclipse.birt.chart.model.data.SampleData)	20.0	95.2 (88.8–100)
	org.eclipse.birt.chart.model.attribute.impl.SizeImpl.create(double,double)	14.5	100 (90.2–100)
	org.eclipse.birt.report.designer.util.DEUtil.getExpression(java.lang.Object)	13.0	100 (92.6–100)
	org.eclipse.birt.report.designer.internal.ui.util.ExceptionHandler.handle(java.lang.reflect.InvocationTargetException)	12.0	100 (77.9–100)
Datatools	org.eclipse.swt.widgets.Text.setEnabled(boolean)	6.0	85.7 (59.8–100)
	java.util.Vector.add(java.lang.String)	5.0	55.5 (23.1–88.0)
	org.eclipse.datatools.sqltools.result.internal.utils.ILogger.error(java.lang.String,java.io.IOException)	4.0	100 (47.3–100)
	java.util.Map.remove(java.lang.String)	4.0	80.0 (44.9–100)
	org.eclipse.ui.dialogs.PropertyPage.setValid(boolean)	4.0	57.1 (20.5–93.8)
Modeling	junit.framework.Assert.assertTrue(java.lang.String,boolean)	10.0	53.6 (40.5–66.6)
	org.eclipse.gmf.runtime.common.core.command.CompositeCommand.compose(org.eclipse.gmf.runtime. 中略. commands.DuplicateViewsCommand)	6.0	85.7 (59.8–100)
	org.eclipse.gmf.runtime.notation.Edge.setSource(org.eclipse.gmf.runtime.notation.Node)	5.0	71.4 (38.0–100)
	org.eclipse.draw2d.Graphics.drawImage(org.eclipse.swt.graphics.Image,org.eclipse.draw2d.geometry.Point)	5.0	100 (54.9–100)
	org.eclipse.emf.common.util.URI.createURI(java.lang.String,boolean)	5.0	62.5 (38.8–86.2)
Technology	org.eclipse.swt.graphics.GC.drawLine(int,int,int,int)	28.0	73.7 (59.7–87.7)
	org.eclipse.jface.viewers.ISelectionChangedListener.selectionChanged(org.eclipse.jface.viewers.SelectionChangedEvent)	11.0	100 (76.2–100)
	org.eclipse.emf.ecore.resource.Resource.save(java.util.Map)	10.0	83.3 (62.2–100)
	java.lang.Object.extensibleElement2XML(org.eclipse.bpel.model.Activity,org.w3c.dom.Element)	9.0	75.0 (50.5–99.5)
	org.eclipse.swt.widgets.Table.setLayoutData(org.eclipse.swt.layout.GridData)	9.0	69.2 (44.1–94.3)
Tools	org.eclipse.ui.forms.widgets.FormToolkit.createComposite(org.eclipse.swt.widgets.Composite)	27.0	96.4 (89.6–100)
	org.eclipse.ui.forms.widgets.FormToolkit.paintBordersFor(org.eclipse.swt.widgets.Composite)	24.0	96.0 (88.3–100)
	org.eclipse.swt.widgets.TableItem.setText(int,java.lang.String)	18.0	85.7 (70.7–100)
	org.eclipse.swt.custom.StyledText.setStyleRange(org.eclipse.swt.custom.StyleRange)	16.0	100 (82.9–100)
	org.eclipse.cdt.core.dom.ast.IASTFunctionCallExpression.setFunctionNameExpression(org.eclipse.cdt.core.dom.ast.IASTExpression)	14.5	85.3 (73.4–97.2)
TPTP	org.w3c.dom.Element.appendChild(org.w3c.dom.Text)	16.0	100 (82.9–100)
	org.eclipse.osgi.util.NLS.bind(java.lang.String,java.lang.Object[])	14.0	89.4 (80.5–98.2)
	org.eclipse.jface.dialogs.Dialog.applyDialogFont(org.eclipse.swt.widgets.Composite)	12.5	100 (88.7–100)
	org.w3c.dom.Document.createTextNode(java.lang.String)	12.0	100 (77.9–100)
	org.eclipse.emf.ecore.impl.EObjectImpl.eBasicRemoveFromContainer(org.eclipse.emf.common.notify.NotificationChain)	9.8	100 (95.0–100)
Webtools	org.eclipse.wst.common.componentcore.internal.ModuleStructuralModel.addProjectModulesIfNecessary(org.eclipse.emf.ecore.xmi.XMIResource)	8.0	66.7 (40.0–93.3)
	org.eclipse.jface.viewers.CheckboxTreeViewer.setGrayed(java.lang.Object,boolean)	4.0	100 (47.3–100)
	org.eclipse.jst.j2ee.internal.actions.OpenJ2EEResourceAction.selectionChanged(org.eclipse.jface.viewers.IStructuredSelection)	4.0	100 (47.3–100)
	org.eclipse.wst.html.core.internal.validate.HTMLAttributeValidator.getErrorSegment(org.eclipse.wst.xml.core.internal.provisional.document.IDOMNode,int)	4.0	57.1 (20.5–93.8)
	java.lang.Math.round(double)	4.0	66.7 (28.9–100)

表 5　複数のプロジェクトで頻繁に変更されたメソッド呼び出し上位 5
Table 5　Top 5 method calls and packages that are frequently changed cross projects.

メソッド呼び出し	平均変更回数	
org.eclipse.swt.widgets.Text.setText(java.lang.String)	BIRT	2.0
	Datatools	2.3
	Modeling	0.1
	Technology	7.5
	Tools	1.1
	TPTP	0.3
	Webtools	—
java.lang.StringBuffer.deleteCharAt(int)	BIRT	3.5
	Datatools	—
	Modeling	—
	Technology	8.0
	Tools	0.6
	TPTP	1.0
	Webtools	0.8
org.eclipse.jface.viewers.TreeViewer.setSelection (org.eclipse.jface.viewers.StructuredSelection,boolean)	BIRT	4.0
	Datatools	—
	Modeling	—
	Technology	4.3
	Tools	0.7
	TPTP	0.7
	Webtools	—
java.lang.Boolean.toString(boolean)	BIRT	2.0
	Datatools	—
	Modeling	—
	Technology	—
	Tools	1.7
	TPTP	0.8
	Webtools	0.6
java.util.Map.get(org.eclipse.core.resources.IFile)	BIRT	—
	Datatools	—
	Modeling	2.0
	Technology	0.5
	Tools	—
	TPTP	4.0
	Webtools	0.7

値が高い順に**表 5** にまとめた．中央値の算出時に，不具合修正時変更率が 50%以下のものは平均変更回数を 0 回として算出したが，表 5 では — として表している．同じく，実際に平均変更回数が 0 回のものも — としている．

表 5 から，Java SE のメソッド呼び出しのほかに SWT（Standard Widget ToolKit）や JFace といった，GUI に関連したメソッド呼び出しが複数のプロジェクトで共通して不具合修正時に変更されたことが分かる．複数のプロジェクトで使用される API の中で，不具合修正に関連した特徴的なメソッド呼び出しがあることが確認できた．

4.4　期間ごとの頻繁に変更されるメソッド呼び出し

期間によって変更されるメソッド呼び出しに違いがあるかを調べるため，各年ごとにメソッド呼び出しの変更回数を調査した．**表 6** に Modeling プロジェクトの結果をまとめた．また，変更されたメソッド呼び出しの平均変更回数を，属するパッケージごとに集計している．このプロジェクトにおいては，上位 5 つのメソッド呼び出しに年をまたいで見られるものはなかった．多くのメソッド呼び出しは特定の期間に頻繁に修正されているようである．この傾向は他のプロジェクトにおいても見られた．パッケージごとの結果を見ると，修正対象となるメソッド呼び出しを含むパッケージが，期間によって大きく変わっていることが分かる．

プロジェクトごとにどんなメソッド呼び出しが期間ごとに修正対象となっているかの傾向を見るため，**表 7** にパッケージごとの変更回数上位 5 を 4 年間まとめた．各プロジェクトにおいて，4 年間通して変更対象となったパッケージを太字で示した．Datatools と Webtools プロジェクトにおいては，Java SE の `java.util` パッケージ以外には 4 年間通して変更対象となったパッケージは見られない．しかし他のプロジェクトでは，Java SE 以外のパッケージに 4 年間通して変更対象となったものがあることが分かる．また，いずれのプロジェクトでも特定の期間のみに修正対象となるパッケージがあることも分かる．

以上の結果から，期間によって修正対象となるメソッド呼び出しに変更があり，パッケージレベルで見ても期間によって修正対象に変化があることが分かる．またプロジェクトによっては，修正対象となるパッケージが期間によってまったく異なることがある．これはプロジェクトによって，機能追加などのためパッケージが次々と追加されていたり，逆に，あまり機能追加はなく保守が中心だったりと，開発の進め方が異なるためと考えられる．

5. 考　　察

5.1　分析手法の問題点について

3.4 節で述べたように，分析手法にはいくつか問題点がある．ここでは，特にケーススタディの妥当性への脅威となりうる，型情報の不明なメソッド呼び出しの問題を考察する．

ケーススタディと同様の条件で，型情報の不明なメソッド呼び出しも含めて分析を行う

表 6　Modeling プロジェクトにおける期間ごとの頻繁に変更された各メソッド呼び出しとパッケージ上位 5
Table 6　Top 5 changed method calls and packages in each period of the Modeling project.

期間	メソッド呼び出し	平均変更回数	パッケージ	変更回数小計
2004	org.eclipse.emf.ecore.util.FeatureMap.add(org.eclipse.emf.ecore.EStructuralFeature,java.lang.Object)	1.0	org.eclipse.emf.ecore	12.0
	org.w3c.dom.Document.createCDATASection(java.lang.String)	1.0	org.w3c.dom	4.4
	org.eclipse.emf.ecore.EFactory.convertToString(org.eclipse.emf.ecore.EDataType,java.lang.Object)	1.0	java.util	4.4
	org.eclipse.core.runtime.Platform.getFragments(org.osgi.framework.Bundle)	1.0	org.eclipse.xsd.impl	1.6
	org.eclipse.xsd.XSDSchema.getCorrespondingComponent(org.w3c.dom.Node)	1.0	java.io	1.0
2005	org.eclipse.draw2d.Graphics.drawImage(org.eclipse.swt.graphics.Image,org.eclipse.draw2d.geometry.Point)	5.0	org.eclipse.gmf.runtime	66.2
	org.eclipse.gmf.runtime.common.core.command.CompositeCommand.compose(org.eclipse.gmf. 中略．commands.DuplicateViewsCommand)	4.0	org.eclipse.gmf.codegen	47.7
	org.eclipse.gmf.codegen.util.Generator.generateNodeLabelEditPart(org.eclipse.gmf.codegen.gmfgen.GenNodeLabel)	3.0	org.eclipse.emf.ecore	23.1
	org.eclipse.emf.ecore.util.EcoreUtil.resolveAll(org.eclipse.emf.ecore.resource.Resource)	3.0	java.util	22.2
	org.eclipse.gmf.codegen.util.Generator.generateNodeItemSemanticEditPolicy(org.eclipse.gmf.codegen.gmfgen.GenNode)	2.0	org.eclipse.emf.codegen	13.3
2006	org.eclipse.gmf.runtime.diagram.ui.tools.TextDirectEditManager.show(char)	9.0	org.eclipse.gmf.runtime	51.2
	org.eclipse.gmf.runtime.notation.Edge.setSource(org.eclipse.gmf.runtime.notation.Node)	5.0	java.util	14.6
	org.eclipse.gmf.runtime.notation.Edge.setTarget(org.eclipse.gmf.runtime.notation.Node)	5.0	org.eclipse.emf.ecore	8.3
	org.eclipse.gmf.runtime.diagram.ui.requests.ArrangeRequest.setPartsToArrange(java.util.List)	3.0	org.eclipse.uml2.uml	6.6
	org.eclipse.gmf.examples.taipan.gmf.editor.part.TaiPanVisualIDRegistry.getVisualID(org.eclipse.gmf.runtime.notation.View)	3.0	org.eclipse.gmf.mappings	6.0
2007	java.lang.Character.isLetterOrDigit(char)	3.0	java.lang	7.1
	org.eclipse.emf.ecore.impl.BasicEObjectImpl.eDynamicIsSet(int,org.eclipse.emf.ecore.EStructuralFeature)	2.0	org.eclipse.emf.ecore	3.4
	org.eclipse.xsd.impl.XSDSimpleTypeDefinitionImpl.equalValues(java.lang.Object,java.lang.Object)	1.0	java.util	3.1
	org.eclipse.emf.validation.internal.service.TraversalStrategyManager.Descriptor.initializeStrategy(org.eclipse. 中略．IConfigurationElement)	1.0	org.eclipse.xsd.impl	2.0
	org.eclipse.xsd.impl.XSDSimpleTypeDefinitionImpl.compareValues(java.lang.Object,java.lang.Object)	1.0	org.eclipse.swt.widgets	1.5
2008	junit.framework.Assert.assertTrue(java.lang.String,boolean)	6.3	java.util	7.6
	org.eclipse.emf.edit.ui.provider.AdapterFactoryContentProvider.getChildren(java.lang.Object)	2.0	junit.framework	7.5
	java.util.Arrays.equals(byte[],byte[])	2.0	org.eclipse.emf.ecore	5.7
	org.eclipse.core.resources.IFile.create(java.io.InputStream,boolean,org.eclipse.core.runtime.IProgressMonitor)	2.0	org.eclipse.core.resources	5.7
	org.eclipse.core.resources.IProject.getFile(org.eclipse.core.runtime.Path)	2.0	org.eclipse.emf.edit	4.6
2009	org.eclipse.emf.common.util.URI.createURI(java.lang.String,boolean)	5.0	java.util	15.8
	junit.framework.TestCase.assertNotNull(org.eclipse.emf.ecore.EObject)	3.0	org.eclipse.emf.ecore	12.7
	junit.framework.Assert.assertTrue(java.lang.String,boolean)	2.0	org.eclipse.emf.common	8.0
	org.eclipse.emf.ecore.util.EcoreUtil.getURI(org.eclipse.emf.ecore.EObject)	2.0	org.eclipse.core.resources	6.1
	org.eclipse.jface.text.IDocument.getLineOfOffset(int)	2.0	junit.framework	6.0

と，ヒストグラムは図 6 と同様の傾向になることを確認した．すべての型情報が判明した状態で再計測を行うと平均変更回数や順位に変更はあると思われるが，大幅な変更ではないと思われる．

本稿では，部分的な情報による解析を行ったため型情報の取得できないメソッド呼び出しがあった．今後の課題として，型情報を適切に取得した分析を行いたい．また，本稿では対象外とした引数のないメソッド呼び出しや，フィールドを介した変更なども今後の課題である．

5.2　不具合修正の特有性

本稿では，不具合修正時に変更回数が多いメソッド呼び出しとそのメソッド呼び出しの使用に不具合が多いことと関係があるのではないかという仮説のもと，分析を行った．ケーススタディでは不具合修正時の変更を分析したが，特定されたメソッド呼び出しは不具合修正時特有の結果なのかといった疑問が残る．そこで本節では，我々の仮説を検証し，特定され

表 7　期間ごとの頻繁に変更された各メソッド呼び出しを含むパッケージ上位 5
Table 7　Top 5 packages containing frequently changed method calls in each period.

プロジェクト	2006	2007	2008	2009
BIRT	**org.eclipse.birt.report**	**org.eclipse.birt.report**	**org.eclipse.birt.report**	**org.eclipse.birt.report**
	org.eclipse.birt.chart	**org.eclipse.birt.chart**	**org.eclipse.birt.chart**	**org.eclipse.birt.chart**
	java.util	**java.util**	**java.util**	**java.util**
	java.lang	org.eclipse.birt.core	java.sql	java.lang
	org.eclipse.birt.data	java.lang	org.eclipse.birt.core	org.eclipse.swt.widgets
Datatools	**java.util**	org.eclipse.swt.widgets	org.eclipse.datatools.connectivity	org.eclipse.ui.dialogs
	org.eclipse.datatools.sqltools	**java.util**	org.eclipse.swt.widgets	**java.util**
	org.eclipse.datatools.connectivity	java.io	**java.util**	java.lang
	org.eclipse.jface.viewers	java.sql	java.lang	java.sql
	java.sql	java.lang	org.eclipse.core.runtime	org.eclipse.swt.widgets
Modeling	org.eclipse.gmf.runtime	java.lang	**java.util**	**java.util**
	java.util	**org.eclipse.emf.ecore**	junit.framework	**org.eclipse.emf.ecore**
	org.eclipse.emf.ecore	**java.util**	**org.eclipse.emf.ecore**	org.eclipse.emf.common
	org.eclipse.uml2.uml	org.eclipse.xsd.impl	org.eclipse.core.resources	org.eclipse.core.resources
	org.eclipse.gmf.mappings	org.eclipse.swt.widgets	org.eclipse.emf.edit	junit.framework
Technology	org.eclipse.bpel.model	org.eclipse.bpel.model	org.eclipse.swt.graphics	org.eclipse.swt.graphics
	org.eclipse.swt.widgets	**java.lang**	**org.eclipse.swt.widgets**	java.util
	java.lang	**org.eclipse.swt.widgets**	java.text	**org.eclipse.swt.widgets**
	java.util	org.eclipse.jface.viewers	**java.lang**	org.eclipse.emf.ecore
	org.eclipse.jface.viewers	java.util	org.eclipse.ui.plugin	**java.lang**
Tools	**java.util**	**org.eclipse.cdt.core**	**org.eclipse.cdt.core**	**org.eclipse.cdt.core**
	org.eclipse.cdt.core	**java.util**	**java.util**	**org.eclipse.cdt.internal**
	org.eclipse.cdt.internal	org.eclipse.cdt.debug	**org.eclipse.cdt.internal**	**java.util**
	org.eclipse.swt.custom	org.eclipse.swt.widgets	org.eclipse.swt.widgets	org.eclipse.swt.widgets
	org.eclipse.ajdt.internal	**org.eclipse.cdt.internal**	org.eclipse.ui.forms	org.eclipse.ui.forms
TPTP	org.eclipse.hyades.models	**org.eclipse.hyades.test**	**org.eclipse.hyades.test**	**org.eclipse.hyades.test**
	org.eclipse.hyades.test	**java.util**	org.eclipse.hyades.edit	org.eclipse.hyades.models
	java.util	java.lang	org.eclipse.swt.widgets	**java.util**
	org.eclipse.swt.widgets	org.eclipse.tptp.platform	**java.util**	org.eclipse.tptp.platform
	java.lang	org.eclipse.jface.dialogs	org.w3c.dom	org.eclipse.swt.widgets
Webtools	**java.util**	org.eclipse.wst.common	**java.util**	junit.framework
	org.eclipse.jst.j2ee	**java.util**	org.eclipse.jst.j2ee	java.lang
	org.eclipse.wst.common	org.eclipse.jface.viewers	org.eclipse.wst.common	org.eclipse.wst.sse
	org.eclipse.wst.web	org.eclipse.wst.wsdl	org.eclipse.wst.sse	org.eclipse.emf.ecore
	org.eclipse.wst.sse	org.eclipse.jst.j2ee	org.eclipse.jdt.core	**java.util**

たメソッド呼び出しが不具合修正に特有のものかどうかを考察する．

まず，不具合修正以外についても平均変更回数を調査した．ここでは，TPTP プロジェクトにおける平均変更回数の上位 5 つのメソッド呼び出しを，それぞれ不具合修正と不具合修正以外で**表 8** にまとめた．引数は省略してある．不具合修正時に変更された上位 5 つのメソッド呼び出しのうち，4 つは不具合修正時変更率が 100%であった．一方，不具合修正以外では，不具合修正時変更率が 0%のものが 3 つあり，変更の意図によって変更される

表 8　TPTP プロジェクトにおける変更意図ごとの頻繁に変更されたメソッド呼び出し上位 5
Table 8　Top 5 changed method calls and packages in the TPTP project.

(a) 不具合修正の変更
(a) Bug-fix changes

メソッド呼び出し	平均変更回数	不具合修正時変更率（95%信頼区間）（%）	検索件数
org.w3c.dom.Element.appendChild	16.0	100 (82.9 – 100)	85,500
org.eclipse.osgi.util.NLS.bind	14.0	89.4 (80.5 – 98.2)	1,100
org.eclipse.jface.dialogs.Dialog.applyDialogFont	12.5	100 (88.7 – 100)	486
org.w3c.dom.Document.createTextNode	12.0	100 (77.9 – 100)	17,700
org.eclipse. 中略. EObjectImpl.eBasicRemoveFromContainer	9.8	100 (95.0 – 100)	279

(b) 不具合修正以外の変更
(b) Not bug-fix changes

メソッド呼び出し	平均変更回数	不具合修正時変更率（95%信頼区間）（%）	検索件数
org.eclipse.hyades. 中略. datapool.IDatapoolIterator.dpInitialize	14.0	26.3 (6.5 – 46.1)	107
org.eclipse.emf.ecore.impl.EFactoryImpl.createFromString	3.0	0 (0 – 22.0)	346
org.eclipse.emf.ecore.impl.EFactoryImpl.convertToString	3.0	0 (0 – 22.0)	1,010
org.eclipse.hyades.loaders.util.XMLLoader.loadEvents	1.0	0 (0 – 95.0)	5
org.eclipse. 中略. control.AgentConfigurationEntry.setValue	1.0	33.3 (0 – 86.7)	6

メソッド呼び出しに偏りが見られることが分かる．

さらに，実際にメソッド呼び出しが不具合と関連があったかを確認するため，検索エンジン Google で調査した．おおまかな調査として，メソッド呼び出しのメソッド名と error または bug を含む記事を検索した．検索結果の件数を表 8 の最終列に示す．数値上の比較では，不具合修正以外の変更ではやや検索件数が少ないことが分かる．以上の，変更意図による偏りは他のプロジェクトでも同様に見られた．Web 検索での検索件数の差は，Java Platform，Standard Edition のメソッド呼び出しでは全体に検索件数が多いなどの例外は見られるが，同様の傾向が見られた．

表 8 (a) 中の `org.w3c.dom` パッケージ関連の Web 検索を行うと，TPTP プロジェクト以外でも様々な不具合が報告されていることが分かった．ここで，TPTP プロジェクトにおいて具体的な事例を調査する．TPTP プロジェクトにおいて，同パッケージ関連の DOM による XML 文書へのアクセスにおけるバグ ID163352 [⋆1] が報告されている．優先度は最高レベルが設定され，不具合修正を意図した複数回の変更後，不具合修正の完了が確認されている．`org.eclipse.tptp.monitoring.logui.internal.util.FilterTransformationHelper.java` の開発履歴から，主にメソッド呼び出し `appendChild` や `createTextNode` の前の条件分岐が増える不具合修正が行われていることを確認した．

これらの結果から，本稿の手法で特定されたメソッド呼び出しは主に不具合修正に特有のものであり，Web 検索の結果のもとでは，不具合修正以外の変更でメソッド呼び出しに比べ，不具合とも関連が高い傾向があることが確認できた．ただ，本稿の手法で特定した，不具合修正時に頻繁に変更されるメソッド呼び出しがただちに不具合につながりやすいと断定するまでには至らない．これについては，今後さらなる分析が必要となると考えている．本稿では不具合修正を一律に扱ったが，変更方法の違いを分析することで，各メソッド呼び出しが主にどのような不具合と関連し，どう修正されたかなどの分析も行えるのではないかと考えている．今後の方針としては，変更方法の分析と，各変更がどんな不具合の修正だったのかという分析について進めていきたい．

5.3　不具合修正関連のキーワードによる分析

ケーススタディにおいて，4 章の最初に述べたとおり，不具合修正に関連したキーワードをコミットログに含む変更を不具合修正として扱った．しかし，ログにキーワードを含んでいても，スタイルの変更のような PDG では変化のない変更も多く，すべてが信頼できる不具合修正とはいえない．Murphy-Hill らは，スタイルを整形するときや不具合を修正したときに，同時に何らかのリファクタリングを行うコミットが非常に多いことを明らかにしている[10]．これは，実際に不具合を修正した変更であっても，不具合修正以外の変更も含むことが多いということを意味する．

上述の議論から，不具合修正時に頻繁に変更されたメソッド呼び出しであっても，それが単純に不具合を混入しやすいメソッド呼び出しであるとは結論づけられない．しかしなが

⋆1 https://bugs.eclipse.org/bugs/show_bug.cgi?id=163352

ら，不具合修正時の平均変更回数に非常に大きな偏りが見られることから，一部のメソッド呼び出しは不具合修正やリファクタリングなどの意図で何度も変更を加えられることがあると考えられる．

5.4　メソッド呼び出しの修正

本稿では，不具合の修正でメソッド呼び出しに何らかの影響がある場合，そのメソッド呼び出しへ修正が行われたとしている．このメソッド呼び出しの修正には，いくつかの意図が考えられる．まず，メソッド呼び出しの使い方が不適切であるため，それを正そうという意図があげられる．ほかには，使うべきなのに使っていなかった（逆も），異なるメソッド呼び出しを使うべきだったため，呼び出すメソッド自体を変更する意図も考えられる．後者については，プロジェクトの進行によって，修正が求められることもある．本稿では，これらを区別せず扱っているが，区別した分析を行えば新しい知見が得られる可能性がある．

5.5　対象プロジェクトについて

本稿では，対象プロジェクトとして，Eclipse 関連のプロジェクトを選択した．これらは，ソフトウェア開発支援ツール関連のソフトウェアである．分析手法では，プロジェクトの特性に依存する解析は行っていないため，ドメインによらず同様の解析は可能であると考えている．

Pan らは，テキスト編集ソフトウェア，コンピュータゲームなど様々なドメインのソフトウェアを対象とし，行単位の構造変化に基づく不具合修正パターンを調査している[11]．Pan らは，開発履歴に分布する不具合修正パターンが，プロジェクト間でピアソンの相関係数が 0.85 を超え，ドメインの違いによらずよく起こる不具合修正パターンは似ていることを報告している．

すなわち，不具合修正における構造変化の様子ではドメイン間に大きな違いは見い出されていない．ただし，4.4 節で調査した期間ごとの推移は，同じソフトウェア開発支援というドメインでもプロジェクト間で差が見られた．これらの違いは，主にプロジェクトの開発がどう進められるかの違いによるのではないかと思われる．ドメインの特有性があるかどうかの調査は，今後の課題である．

6. 関 連 研 究

プログラムの変更分析には様々な研究がなされている．Canfora らは情報検索技術などを応用することで，*diff* ツールの出力を改善する手法を提案し，行レベルでの変更分析を行っている[3]．一方，ツリーレベルでの変更分析も研究されている[1),4)]．Fluri らは，抽象構文木の比較からノードの追加，削除，移動，更新を特定するアルゴリズムを提案している[4]．Apiwattanapong らはオブジェクト指向のプログラムにおける変更を分析するため，拡張したフローグラフに基づく変更分析を提案している[1]．また，個別のソースコード間の変更分析でない研究としては Kim らによる systematic change についての研究があげられる[7]．彼らの研究では，メソッドやクラスレベルでの追加や除去検出を目的としている．

変更の履歴に基づき，関連する変更を検出する研究も行われている．Zimmermann らは共通して変更される行を特定する手法を提案している[17]．Ren らは変更の与える影響を分析することで，追加的にどこまでテストすべきかを提示する手法を提案している[13]．また Mockus らはログを解析することでなぜプログラムが変更されたかを分析する手法を提案している[9]．

ソースコード間の変更の影響についての意味的な解析も行われている．Horwitz は PDG に基づき振舞いの変更を分析している[5]．Jackson らも依存関係を考慮することで意味的な振舞いの違いを把握する手法を提案している[6]．吉田らは構文要素の移動や変更の影響を受けた可能性の高い要素を出力する手法を提案している[18]．Parnin らは変更の影響を表現する手法を提案している[12]．彼らは，すべての変更に対して適切な粒度で変更の影響を記述することを目標としてるので，狭い範囲の影響を探索している．これに対して本稿では，メソッド呼び出しへ影響を与えるメソッド内の変更を広く検出している．

不具合修正についての研究として，Pan らは不具合修正についてのパターンを 27 種定義し，7 つのオープンソースプロジェクトのリポジトリに対して，どのパターンがよく見られるかを調査している[11]．その結果 MC-DAP（メソッド呼び出しにおける引数の変更），IF-CC（`if` 条件に関する変更），AS-CE（代入処理部分の変更）といったものが，頻繁に見られる不具合修正パターンとして特定されている．また相関分析によって，その傾向はプロジェクト横断的なものであると述べている．Pan らの研究は，不具合修正について，どのような構文的な変更が行われているかを明らかにした．一方本稿では，不具合修正について，どのメソッド呼び出しが変更されているかを分析している．

7. ま と め

本稿では，メソッド呼び出しの不具合修正に関して定量的な分析を行うため，実際のソフトウェアプロジェクトを対象に，メソッド呼び出しへ影響を与える変更の解析を行った．本稿では，(1) 不具合修正時に頻繁に変更される特徴的なメソッド呼び出しがあるか，(2) 複数のプロジェクトに共通して，不具合修正時の変更対象となるメソッド呼び出しがあるか，

(3) プロジェクトの時間経過によって修正されるメソッド呼び出しに変化はあるか，を分析目標とした．ケーススタディは，Java 言語で書かれた Eclipse 関連の 7 つのオープンソースプロジェクトを対象とした．

分析によって，以下のことを明らかにした．まず，各プロジェクト内にてほとんどのメソッド呼び出しは数回程度（5 回以下）の修正回数だが，一部のメソッド呼び出しは頻繁に（10 回以上）修正されている．また，複数のプロジェクトで使用される Java Platform, Standard Edition の API や Eclipse の GUI に関連した API のうち，共通して不具合修正時に変更されるメソッド呼び出しがある．さらに，時間経過によって修正対象となるメソッド呼び出しには変化がある．

本稿は，不具合修正における定量的な分析の初期段階であり，さらなる詳細な分析を行う必要がある．具体的には，どのような修正が頻繁に行われているか，どのような意図の修正なのかといった分析を進めていきたいと考えている．また，不具合修正か否かを判定する技術などは，精密な分析のために精度の向上が求められる．本稿を進めていくことで，頻繁に繰り返される似た不具合の混入を未然に防いだり，プロジェクトの進捗状況を把握したりすることで，低コストで高信頼なソフトウェア開発を支援することを目指したいと考えている．

謝辞　本稿の初版に対して，2 人の査読者，および，メタ査読者様より有益な助言とコメントをいただいたことを感謝する．また本研究を行うにあたり，PDG 構築のツールを提供してくださるとともに，貴重なコメントなどをいただいた大阪大学大学院情報科学研究科の肥後芳樹助教に深く感謝する．本研究は，文部科学省グローバル COE プログラム（アンビエント情報社会基盤創成拠点）の助成を得た．

参 考 文 献

1) Apiwattanapong, T., Orso, A. and Harrold, M.J.: JDiff: A differencing technique and tool for object-oriented programs, *Automated Software Engg.*, Vol.14, No.1, pp.3–36 (2007).
2) Bloch, J.: How to design a good API and why it matters, *Proc. 21th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, ACM, pp.506–507 (2006).
3) Canfora, G., Cerulo, L. and Penta, M.D.: Identifying Changed Source Code Lines from Version Repositories, *Proc. 4th International Workshop on Mining Software Repositories*, Washington, DC, USA, IEEE Computer Society, pp.14–22 (2007).
4) Fluri, B., Wuersch, M., PInzger, M. and Gall, H.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction, *IEEE Trans. Softw. Eng.*, Vol.33, No.11, pp.725–743 (2007).
5) Horwitz, S.: Identifying the semantic and textual differences between two versions of a program, *Proc. ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp.234–245 (1990).
6) Jackson, D. and Ladd, D.A.: Semantic Diff: A Tool for Summarizing the Effects of Modifications, *Proc. 10th International Conference on Software Maintenance*, Washington, DC, USA, IEEE Computer Society, pp.243–252 (1994).
7) Kim, M. and Notkin, D.: Discovering and representing systematic code changes, *Proc. 31st International Conference on Software Engineering*, Washington, DC, USA, IEEE Computer Society, pp.309–319 (2009).
8) Laski, J. and Szermer, W.: Identification of Program Modifications and its Applications in Software Maintenance, *Proc. 8th International Conference on Software Maintenance*, pp.10–13 (1992).
9) Mockus, A. and Votta, L.G.: Identifying Reasons for Software Changes Using Historic Databases, *Proc. 16th International Conference on Software Maintenance*, Washington, DC, USA, IEEE Computer Society, pp.120–130 (2000).
10) Murphy-Hill, E., Parnin, C. and Black, A.P.: How we refactor, and how we know it, *Proc. 31st International Conference on Software Engineering*, pp.287–297 (2009).
11) Pan, K., Kim, S. and Whitehead, Jr., E.J.: Toward an understanding of bug fix patterns, *Empirical Software Engineering*, Vol.14, No.3, pp.286–315 (2009).
12) Parnin, C. and Görg, C.: Improving change descriptions with change contexts, *Proc. 5th International Workshop on Mining Software Repositories*, New York, NY, USA, ACM, pp.51–60 (2008).
13) Ren, X., Shah, F., Tip, F., Ryder, B.G. and Chesley, O.: Chianti: a tool for change impact analysis of java programs, *Proc. 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, ACM, pp.432–448 (2004).
14) Robillard, M.P.: What Makes APIs Hard to Learn? Answers from Developers, *IEEE Software*, Vol.26, No.6, pp.27–34 (2009).
15) Wang, Z., Pierce, K. and McFarling, S.: BMAT: A binary matching tool for stale profile propagation, *The Journal of Instruction-Level Parallelism*, Vol.2, pp.99–83 (2000).
16) Xie, T. and Pei, J.: MAPO: Mining API usages from open source repositories, *Proc. 3rd International Workshop on Mining Software Repositories*, New York, NY, USA, ACM, pp.54–57 (2006).
17) Zimmermann, T., Kim, S., Zeller, A. and Whitehead, Jr., E.J.: Mining version archives for co-changed lines, *Proc. 3rd International Workshop on Mining Soft-*

ware repositories, New York, NY, USA, ACM, pp.72–75 (2006).
18) 吉田　敦，山本晋一郎，阿草清滋：意味を考慮した差分抽出ツール，情報処理学会論文誌，Vol.38, No.6, pp.1163–1171 (1997).
19) 三宅達也，肥後芳樹，楠本真二，井上克郎：多言語対応メトリクス計測プラグイン開発基盤 MASU の開発，電子情報通信学会論文誌 D，Vol.J92-D, No.9, pp.1518–1531 (2009).

(平成 22 年 2 月 1 日受付)
(平成 22 年 11 月 5 日採録)

畑　　秀明（学生会員）

平成 19 年大阪大学工学部電子情報エネルギー工学科卒業．平成 21 年同大学大学院博士前期課程修了．現在，同大学院博士後期課程に在学．ソフトウェアのバグ検出手法，ソフトウェアリポジトリのマイニングに関する研究に従事．電子情報通信学会，日本信頼性学会，IEEE 各会員．

森井　亮介

平成 20 年大阪大学基礎工学部情報科学科卒業．平成 22 年同大学大学院博士前期課程修了．在学中，ソフトウェアリポジトリのマイニングに関する研究に従事．現在，三菱電機株式会社に勤務．

水野　　修（正会員）

平成 10 年大阪大学大学院情報数理系専攻博士前期課程修了．平成 11 年 3 月同大学院博士後期課程中退．博士（工学）．同年 4 月より大阪大学大学院基礎工学研究科助手．その後，大阪大学大学院情報科学研究科助教を経て．平成 21 年 9 月京都工芸繊維大学准教授．主にソフトウェア開発プロセスの改善支援，ソフトウェアのバグ検出手法，ソフトウェアリポジトリのマイニングに関する研究に従事．電子情報通信学会，IEEE 各会員．

菊野　　亨（フェロー）

昭和 50 年大阪大学大学院博士課程修了．工学博士．同年広島大学工学部講師．同大学助教授を経て，昭和 62 年大阪大学基礎工学部情報工学科助教授．平成 2 年同大学教授．現在，大阪大学大学院情報科学研究科教授．大阪大学国際交流センター・センター長．主にフォールトトレラントシステム，ソフトウェア開発プロセスの定量的評価に関する研究に従事．電子情報通信学会，情報処理学会各フェロー．ACM，IEEE 各会員．日本信頼性学会前会長．