Javaプロジェクトに含まれる振る舞いを変えない変更の検出

前島　葵[†a)]　肥後　芳樹[†]　松本淳之介[†]　楠本　真二[†]

Detecting Non-Behavioral Changes in Java Projects

Aoi MAEJIMA[†a)], Yoshiki HIGO[†], Junnosuke MATSUMOTO[†], and Shinji KUSUMOTO[†]

あらまし　ソースコードの変更には，プログラムの振る舞いを変える変更がある一方，振る舞いを変えない変更も存在する．既存研究による調査の結果，最大で 15.5% の変更が振る舞いを変えない変更であるという結果が報告された．しかし，調査された変更は 12 種類だけである．そのため，他にも振る舞いを変えない変更は存在するのではないかと著者らは考えた．本研究では，Java プロジェクトを対象とし，バイトコードを利用した振る舞いを変えない変更の検出方法を提案する．提案手法では，バイトコードを逆アセンブリした結果が変わらない変更を振る舞いを変えない変更とした．六つのオープンソース Java プロジェクトに対して振る舞いを変えない変更の検出を行った．その結果，Java ファイルが変更されたコミットのうち，8.6% ~ 22.4% は振る舞いを変えないコミットであった．また，今回の調査では先行研究と比較し新たに 25 種類の振る舞いを変えない変更を検出した．

キーワード　ソフトウェアリポジトリマイニング，ソフトウェア変更分析，開発履歴，Git

1. まえがき

ソフトウェアの開発において，変更の理解は重要である．Git に代表されるバージョン管理システムを利用したオープンソースソフトウェア開発が広がり，ソースコードの様々なバージョンが記録され，バージョン間の変更が追跡されている．ソフトウェアリポジトリマイニングとは，開発履歴が蓄積されたソフトウェアリポジトリに対するデータマイニングである．ソフトウェア開発における有用な情報を抽出することを目指し，近年多くの研究が行われている [1]〜[8]．リポジトリに蓄積されたソースコードの開発履歴を理解することで，いつバグが混入したのか，なぜ変更が行われたのか，変更はプログラムにどのような影響を与えたかなどの分析が行われている [1]〜[4]．

バージョン管理システムを用いた開発において，コードレビューや変更が競合した場合にソースコードの差分を理解する必要がある．このような場合に開発者にソースコードの変更をわかりやすく表現することが目指され，抽象構文木を用いた変更理解の研究が行われている [9], [10]．抽象構文木を利用した分析では，コメントやフォーマットの変更は無視される．この戦略の背後には，これらの変更は単純な開発作業であり，ソースコードの理解や分析に意味のある情報を生み出す可能性が低いという仮定がある．

Kawrykow らは，更に多くの種類の変更が意味のある情報を生み出す可能性が低く，変更分析を誤る原因にもなることを明らかにした [11]．例えば，開発者がある変数名の変更を行うと，その変数名への参照を含む全ての行も変更される．しかし，この変更は単純な開発作業を行ったにもかかわらず変更が何行ものコードや複数のメソッド，ファイルに渡っていることがある．そのため，変更がバグを発生している可能性が高いと推測されたり [12]，コードの劣化であると推測されたりする可能性がある [13]．また，コード間の依存関係を誤って検出してしまうかもしれない [14]．このように変更を誤って分析してしまう可能性がある．

ソースコードの変更には，新たな機能の実装や既存の機能の修正などプログラムの振る舞いを変える変更が存在する一方で，コメントの変更やフォーマットの変更，変数名の変更などプログラムの振る舞いを変えない変更も存在する．これらの変更は可読性や保守性などの品質向上のために行われることが多い [15]．

Kawrykow らによって，振る舞いを変えない変更に

[†] 大阪大学，吹田市
Osaka University, Suita-shi, 565–0871 Japan
a) E-mail: a-maejim@ist.osaka-u.ac.jp
DOI:10.14923/transinfj.2020PDP0031

ついての調査が行われた [11].

彼らは振る舞いを変えない変更として，コメントの変更や変数名の変更など 12 種類の変更を検出対象に選び調査を行った．Java プロジェクトリポジトリに対する調査の結果，最大で 15.5% の変更が振る舞いを変えない変更であるという結果が報告された．

しかしながら，調査された変更は 12 種類のみであり，彼らの調査で発見されていない振る舞いを変えない変更が存在するのではないかと著者らは考えた．

本研究では，Java プロジェクトを対象とし，バイトコードを利用した振る舞いを変えない変更の検出方法を提案する(注1)．先行研究に対し別の手法で検出を行い，振る舞いを変えない変更の検出数の増減を議論する．

本研究では以下に示す三つの Research Question に答えることを目的とする．

RQ 1 先行研究に比べ本研究における振る舞いを変えない変更の検出数は増加するか

RQ 2 先行研究が網羅できていなかった振る舞いを変えない変更はどのような変更だったか

RQ 3 先行研究と比較しどの程度振る舞いを変えない変更の種類が増加するか

六つの Java プロジェクトのオープンソースリポジトリに対して調査を行った．その結果，Java ファイルの変更がなされたコミットのうち，8.6% ~ 22.4% はプログラムの振る舞いを変えないコミットであった．また，今回の調査では先行研究と比較し新たに 25 種類の振る舞いを変えない変更を検出した．新たに発見された振る舞いを変えない変更は，先行研究では見落とされていた出現回数の少ないマイナーな変更が多かった．

2. 研究背景

Kawrykow らは Java プロジェクトリポジトリを対象に調査を行い，最大で 15.5% の変更が振る舞いを変えない変更であるという結果を報告した [11].

彼らは以下に示す 12 種類の変更を検出対象の振る舞いを変えない変更とした．

C1 完全限定名から単純名への変更
C2 一時変数の導入
C3a クラス名の変更
C3b フィールド変数名の変更

(注1)：本論文は文献 [11] で行われた振る舞いの変わらない変更の検出を別の解法で解いた論文である．

```
  for (final ArgumentProcessor processor :
        processorRegistry.getProcessors()) {
    final List<String> extraArgs =
        extraArguments.get(processor.getClass());
-   if (extraArgs != null) {
-     if (processor.handleArg(extraArgs)) {
-       return;
-     }
+   if (extraArgs != null && processor.handleArg(extraArgs)) {
+     return;
    }
```

図 1　if 文の統合（先行研究で網羅されていない振る舞いを変えない変更例）
Fig. 1　If-Condition Consolidation (not covered in previous studies).

C3c メソッド名の変更
C3d パラメータ名の変更
C4a this の挿入，削除
C4b オペランドをもたない return 文
C4c 引数なし親コンストラクタの呼び出し
C5 ローカル変数名の変更
C6a コメントの変更
C6b フォーマットの変更

しかしながら，振る舞いを変えない変更とされたのはこれらの 12 種類だけである．そのため，彼らの調査で発見されていない振る舞いを変えない変更が存在するのではないかと著者らは考えた．そこで本研究では，先行研究とは別の手法を用いて振る舞いを変えない変更の検出を行う．

図 1 は if 文を統合した変更を示す．この変更は，本研究において検出された実際の振る舞いを変えない変更である．また，この変更は先行研究では網羅されていなかった．

3. 振る舞いを変えない変更の調査方法

3.1 振る舞いを変えない変更の検出方法

本研究では，先行研究と別の解法を用いて振る舞いを変えない変更の検出を行う．本研究では，振る舞いを変えない変更をバイトコードを利用して検出する．バイトコードとは，ソースコードがコンパイルされたときに生成される中間コードのことである．変更が行われる前のソースコードのバイトコードと変更が行われた後のバイトコードを逆アセンブリした結果を比較し，その変化の有無に注目することで，振る舞いを変えない変更を検出する．ソースコードの変更にともないバイトコードも変化する変更はプログラムの振る舞いを変える変更とし，ソースコードが変化したにもかかわらずバイトコードが変化しなかった変更は振る舞いを変えない変更とする．コンパイルによって失われるコメントやフォーマットなどの情報はプログラムの振る舞いに変化を与えないとみなすことができる．

```
public class Sample {
    public int bar = 10;
    public int foo(int n) {
        if (n > bar) n--;
        else n++;
        return n;
    }
}
```

図 2　Sample.java
Fig. 2　Sample.java.

```
Compiled from "Sample.java"
public class sample {
  public int bar;
  public sample();
  public int foo(int);
}
```

図 3　javap Sample.java コマンドの実行結果
Fig. 3　Result of javap Sample.java.

3.2　rJava ファイル

本研究では，バイトコードを逆アセンブルした結果得られるファイルを rJava ファイルと呼ぶ．逆アセンブルには，javap -p -c [file] コマンドを用いた．

javap コマンドは，Java ファイルのコンパイルにより生成されるクラスファイルを逆アセンブルするコマンドである．javap コマンドはオプションを使用しない場合，指定されたファイルに含まれるクラスの protected 及び public 修飾子が付いたフィールドとメソッドの一覧を出力する．本研究において javap コマンドのオプションには-p -c を指定した．-p オプションにより，private 修飾子も含め全てのメソッド，フィールドを逆アセンブルの対象に指定する．-c オプションにより，逆アセンブルされた各メソッドのバイトコード命令列を出力する．

Java ソースコードを図 2 に示し，それに対する javap Sample.java コマンドの実行結果を図 3 に示す．同様に，javap -p -c Sample.java コマンドの実行結果を図 4 に示す．

3.3　変換リポジトリの作成

調査を行うにあたって，前準備として調査対象の Java プロジェクトリポジトリから，Java ファイルと rJava ファイルをもつリポジトリを作成する．

Step 1　コミットの状態を復元
Step 2　コンパイル
Step 3　逆アセンブル
Step 4　Java, rJava ファイルをコミット

以降，各 Step について説明する．

Step 1 では git reset -hard <commit>コマンドを用いてリポジトリを対象のコミットの状態に復元する．Step 2 ではコンパイルを行いバイトコードであるクラスファイルの作成を行う．本研究ではプロジェクトのビルドに Gradle 及び Ant を用いた．Step 3 ではクラスファイルを逆アセンブルする．その結果，rJava ファイルが作成される．Step 4 では Java ファイルと rJava ファイルをリポジトリに追加する．

```
Compiled from "Sample.java"
public class sample {
  public int bar;

  public sample();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: aload_0
       5: bipush        10
       7: putfield      #2 // Field bar:I
      10: return

  public int foo(int);
    Code:
       0: iload_1
       1: aload_0
       2: getfield      #2 // Field bar:I
       5: if_icmple     14
       8: iinc          1, -1
      11: goto          17
      14: iinc          1, 1
      17: iload_1
      18: ireturn
}
```

図 4　javap -p -c Sample.java コマンドの実行結果
Fig. 4　Result of javap -p -c Sample.java.

3.4　調 査 方 法

git log -name-stats コマンドを用いて **3.3** で作成したリポジトリの変更履歴を調査する．Java ファイルが変更されたが rJava ファイルが変更されていないコミットを検出することで，振る舞いを変えないコミットを検出できる．振る舞いを変えないコミットとは，コミットで行われた全ての Java ファイルの変更に対して，rJava ファイルの変更が一切行われていないコミットである．

4.　Research Question

調査を行うにあたり以下に示す三つの Research Question を設定した．

RQ1 : 先行研究に比べ本研究における振る舞いを変えない変更の検出数は増加するか

Java プロジェクトリポジトリに存在するコミットのどの程度が振る舞いを変えないコミットであるか調査を行い，検出数を先行研究と比較する．

RQ2 : 先行研究が網羅できていなかった振る舞いを変えない変更はどのような変更だったか

先行研究においては 12 種類の変更が調査対象となる振る舞いを変えない変更とされていた．それに対し，本研究ではバイトコードを用いた検出を行う．先行研究が網羅できていなかった振る舞いを変えない変更は

どのような変更だったか調査を行う．

RQ3 : 先行研究と比較しどの程度振る舞いを変えない変更の種類が増加するか

バイトコードによる検出法を用いることで振る舞いを変えない変更の種類が先行研究と比較しどの程度増加するのか調査する．

5. Research Question の調査

5.1 調査対象

調査対象は Java プロジェクトの Git リポジトリである．本研究では，著者らの所属している研究室で開発されている Java プロジェクトである kGenProg [16]，先行研究 [11] で調査対象とされていた Apache Ant, Hibernate ORM Core, Spring Framework Core, 更に MSR の調査対象として頻繁に利用されている Apache プロジェクトである Apache Tomcat, Apache POI を加えた六つのプロジェクトを対象に調査を行った．先行研究では七つのプロジェクトが調査対象とされていたが，本研究においてビルドが必須のため Java やビルドツールのバージョンの違いでビルドに失敗したプロジェクトを除外した．また，古くから開発が続くプロジェクトについては現在の環境でビルドできない期間があったため対象期間を制限した．

調査対象のプロジェクト，調査の対象期間，調査対象とする Java ファイルが変更されたコミット数を表 1 に示す．ただし，テストコードは対象となるソフトウェアを構成しないため，調査対象に含まない．全てのプロジェクトにおいて Java ファイルが変更されたコミットが 200 コミット以上であり，1,800 コミットにおよぶプロジェクトも存在した．

5.2 RQ1: 先行研究に比べ本研究における振る舞いを変えない変更の検出数は増加するかの調査結果

六つのオープンソース Java プロジェクトリポジトリに対する調査結果を表 2 に示す．表 2 には，対象プロジェクト名と Java ファイルが変更されたコミット，Java ファイルが変更されたが rJava ファイルの変更のないコミットを示している．括弧内に示す数値がプログラムの振る舞いを変えないコミットの割合である．

先行研究では，Java プロジェクトリポジトリに存在する変更の中で，2.6% ~ 15.5% は振る舞いを変えない変更であると調査された．

RQ1 への回答として，「Java ファイルの変更が行われたコミットの中で，8.6% ~ 22.4% は振る舞いを変えないコミットであり，振る舞いを変えない変更の検出数は先行研究に比べ増加している」といえる．

表 1　調査対象プロジェクト，調査対象期間
Table 1　Target projects and target periods.

プロジェクト名	対象コミット数	調査開始	調査終了
kGenProg	808	2018-04-09	2019-11-04
Ant	323	2017-01-01	2019-05-22
Spring	851	2016-08-15	2020-01-24
Hibernate	1,253	2016-12-16	2020-01-31
Tomcat	1,800	2017-07-04	2020-01-15
POI	255	2018-01-26	2020-01-13

表 2　RQ1 の調査結果
Table 2　Results of RQ1.

プロジェクト名	Java ファイルの変更コミット数	rJava ファイルの変更のないコミット数 (振る舞いを変えないコミットの割合)
kGenProg	808	157 (19.4%)
Ant	323	64 (19.8%)
Spring	851	191 (22.4%)
Hibernate	1,253	114 (9.1%)
Tomcat	1,800	278 (15.4%)
POI	255	22 (8.6%)
Total	5,287	826 (15.6%)

5.3 RQ2: 先行研究が網羅できていなかった振る舞いを変えない変更はどのような変更だったかの調査結果

六つのプロジェクトから検出された振る舞いを変えないコミット全てに対して目視調査を行い，実際にどのような変更が行われているのか確認した．表 4 は，振る舞いを変えない変更をその変更が検出されたコミット数とともに示している．また，括弧内に示す数値は全調査対象プロジェクトまたは各プロジェクトの検出総数に占めるその割合である．例えば，コメントの変更が検出された割合は六つの調査対象プロジェクト全体の中では 46.7% を占め，kGenProg の中では 34.1% を占める．

全調査対象プロジェクトから見つかった振る舞いを変えない変更について Jaccard 係数の分析を行った．Jaccard 係数とは，ある二つのプロジェクトで検出された振る舞いを変えない変更全体のなかで，共通して検出された振る舞いを変えない変更の割合を指す．

例として，kGenProg と Ant から検出された変更は全部で 15 種類であり，その中で共通し検出された変更はコメントやフォーマットの変更など 7 種類存在した．よって，Jaccard 係数は 7/15 = 0.47 となる．表 3 には各二つのプロジェクト間の Jaccard 係数を示す．

表 3　Jaccard 係数
Table 3　Jaccard index.

	kGenProg	Ant	Spring	Hibernate	Tomcat	POI
kGenProg		0.47	0.50	0.62	0.39	0.56
Ant			0.40	0.36	0.48	0.44
Spring				0.35	0.46	0.53
Hibernate					0.38	0.47
Tomcat						0.43
POI						

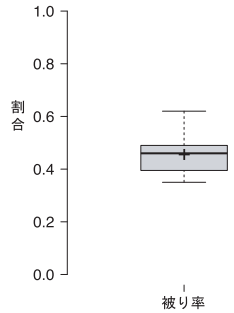

図 5　Jaccard 係数の箱ひげ図
Fig. 5　Box plot of Jaccard index.

図 5 には表 3 の結果を箱ひげ図で示す．Jaccard 係数の最大値は 0.62 であり，中央値は 0.46 であり，最小値は 0.35 であり，平均値は 0.46 である．この結果から，どの二つのプロジェクト間においても共通して検出された振る舞いを変えない変更の種類数は約半分にしか及ばず，片方のプロジェクトのみから検出された変更も多く存在することがわかる．

そのため，各プロジェクトでは異なる種類の振る舞いを変えない変更が行われていることがわかる．本研究では六つのプロジェクトを対象としたが，プロジェクトの数を増加を行っても，追加されたプロジェクトに応じた結果が得られ，新たな振る舞いを変えない変更が検出される可能性があるといえる．

RQ2 への回答として，「開発者は表 4 に示す振る舞いを変えない変更を行う．先行研究では網羅されておらず本研究で新しく検出された振る舞いを変えない変更は，表 4 の上部に示す 8 種類の変更を除いた残り 25 種類の変更である．アノテーションの変更やド・モルガンの法則による条件式の書き換えなどが含まれる．先行研究で網羅されておらず本研究で新たに検出可能となった変更は，先行研究では見落とされていた出現回数の少ない変更である」といえる．

5.4　RQ3: 先行研究と比較しどの程度振る舞いを変えない変更の種類数が増加するかの調査結果

六つの調査対象プロジェクトから検出された振る舞いを変えない変更には，表 4 に示す 33 種類の変更がある．表 4 の上部に示す 8 種類の変更(注2)は先行研究 [11] においても振る舞いを変えない変更と分類されていたが，表の残り 25 種類は提案手法で新しく検出された振る舞いを変えない変更である．

図 7 は先行研究，本研究のそれぞれから若しくはその両方から振る舞いを変えない変更と定義や検出された変更についてベン図で示している．先行研究でのみ振る舞いを変えないと定義された変更は 4 種類であり，本研究でのみ振る舞いを変えないと検出された変更は 25 種類である．また，先行研究と本研究で共に振る舞いを変えないと定義や検出された変更は 8 種類である．

先行研究と本研究では共通した定義の振る舞いを変えない変更の検出を行った．しかし，検出法の違いにより先行研究で検出された 4 種類は本研究では検出できなかった．これら 4 種類はクラス名，フィールド変数名，メソッド名の変更と一時変数の導入である．図 6 に一時変数の導入の例を示す．これらの変更は，本研究の検出方法においてバイトコードの逆アセンブリ後の結果が変化するため振る舞いを変える変更と判定される．このように，手法の違いによって検出される変更数の減少も生じた．

更に，表 5 に振る舞いを変えない変更の検出数について先行研究と本研究の比較を示す．表 5 には，対象プロジェクト名と，各プロジェクトの振る舞いを変えない変更の検出数，本研究でのみ検出可能な振る舞いを変えない変更の数を示している．括弧内の数値は，本研究でのみ検出可能な振る舞いを変えない変更の割合を示す．

六つのプロジェクトの合計数に注目すると，振る舞いを変えない変更全体の 26.4% を本研究で新たに検出された変更が占めた．表 4 に示したとおり，振る舞いを変えない変更の中では先行研究において挙げられていたコメントの変更やフォーマットの変更が大部分を占める．対して，本研究で新たに検出された変更された変更は出現回数の少ないマイナーな変更であり全体に占める割合も低くなっている．

RQ3 への回答として，「先行研究では検出対象とされた振る舞いを変えない変更は 12 種類のみであった．

(注2)：オペランドをもたない return 文と引数なし親コンストラクタの呼び出しについては調査対象プロジェクトからは検出されなかったが，著者が手元で確認したところ逆アセンブリ結果を変化させなかった．そのため本研究で検出可能な変更に含める．

表 4　振る舞いを変えない変更一覧
Table 4　Patterns of Non-behavioral Changes.

振る舞いを変えない変更	Total	kGenProg	Ant	Spring	Hibernate	Tomcat	POI
C6a コメント	457 (46.7%)	56 (34.1%)	42 (56.0%)	133 (55.1%)	61 (43.9%)	150 (46.3%)	15 (42.9%)
C6b フォーマット	212 (21.7%)	26 (15.9%)	15 (20.0%)	30 (12.4%)	40 (28.8%)	93 (28.7%)	8 (22.9%)
C1 完全限定名，単純名間の変更	11 (1.1%)	2 (1.2%)	1 (1.3%)	4 (1.7%)		3 (0.9%)	1 (2.9%)
C5 ローカル変数名	22 (2.2%)	10 (6.1%)	1 (1.3%)	5 (2.1%)		5 (1.5%)	1 (2.9%)
C3d パラメータ名	12 (1.2%)	3 (1.8%)	1 (1.3%)	2 (0.8%)	1(0.7%)	4 (1.2%)	1 (2.9%)
C4a this の挿入，削除	7 (0.7%)	2 (1.2%)		4 (1.7%)			1 (2.9%)
C4b オペランドをもたない return 文	0 (0.0%)						
C5c 引数なし親コンストラクタの呼び出し	0 (0.0%)						
final 修飾子	22 (2.2%)	19 (11.6%)		1 (0.4%)	1 (0.7%)	1 (0.3%)	
import 文	56 (5.7%)	18 (11.0%)	5 (6.7%)	14 (5.8%)	9 (6.5%)	8 (2.5%)	2 (5.7%)
アノテーション	100 (10.2%)	25 (15.2%)	5 (6.7%)	31 (12.9%)	22 (15.8%)	16 (4.9%)	1 (2.9%)
インターフェース中のメソッドの public 修飾子の削除	3 (0.3%)	1 (0.6%)			2 (1.4%)		
キャスト	1 (0.1%)	1 (0.6%)					
ラムダ式の引数の型	1 (0.1%)			1 (0.4%)			
空文	5 (0.5%)	1 (0.6%)		1 (0.4%)	1 (0.7%)	1 (0.3%)	1 (2.9%)
ドモルガンの法則による条件式の書き換え	1 (0.1%)			1 (0.4%)			
括弧	11 (1.1%)			7 (2.9%)		3 (0.9%)	1 (2.9%)
ブロック	11 (1.1%)				2 (1.4%)	8 (2.5%)	1 (2.9%)
if 文の統合	1 (0.1%)		1 (1.3%)				
else if 文から if 文への書き換え	1 (0.1%)			1 (0.4%)			
else 文の削除	1 (0.1%)			1 (0.4%)			
総称型	8 (0.8%)		3 (4.0%)	3 (1.2%)		2 (0.6%)	
ダイヤモンド演算子	4 (0.4%)		1 (1.3%)			3 (0.9%)	
配列宣言	1 (0.1%)			1 (0.4%)			
修飾子の並び替え	2 (0.2%)			1 (0.4%)		1 (0.3%)	
空文字列	1 (0.1%)					1 (0.3%)	
static イニシャライザ	1 (0.1%)					1 (0.3%)	
Iterator の使用から拡張 for 文の使用への変更	1 (0.1%)					1 (0.3%)	
オートボクシング，アンボクシング	19 (1.9%)					19 (5.9%)	
整数型リテラル	1 (0.1%)						1 (2.9%)
定数	1 (0.1%)					1 (0.3%)	
Enum クラスの修飾子	3 (0.3%)					3 (0.9%)	
Enum 定数列挙末尾のセミコロン	1 (0.1%)						1 (2.9%)

```
    int a = 1;
    int b = 2;
-   System.out.println("sum " + (a+b));
+   int sum = a + b;
+   System.out.println("sum " + sum );
```

図 6　一時変数の導入
Fig. 6　Local Variable Extractions.

対して本研究の手法では全部で 33 種類の振る舞いを変えない変更が検出可能となった．このことから，本研究のバイトコードを用いた手法では検出可能な振る舞いを変えない変更の種類数を大きく増加させられる」といえる．

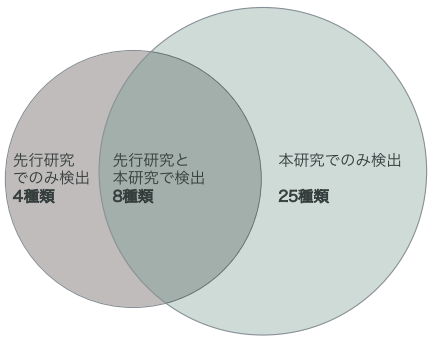

図 7　検出種類数のベン図
Fig. 7　Venn diagram of the number of patterns.

表 5　新たに検出された振る舞いを変えない変更の割合
Table 5　Ratio of newly-detected non-behavioral changes.

プロジェクト名	検出数全体	本研究でのみ検出
kGenProg	164	65 (39.6%)
Ant	75	15 (20.0%)
Spring	245	67 (27.3%)
Hibernate	139	37 (26.6%)
Tomcat	324	69 (21.3%)
POI	35	8 (22.9%)
Total	979	258 (26.4%)

```
+   @Override
+   @Deprecated
+   @Expose
+   @SuppressWarnings("unused")
```

図 8　アノテーション
Fig. 8　Annotation.

```
  public void store(Writer writer, String comments)
      throws IOException {
    for (String line : contents.split(EOL)) {
-     if (!this.omitComments || !line.startsWith("#")) {
+     if (!(this.omitComments && line.startsWith("#"))) {
        writer.write(line + EOL);
      }
    }
  }
```

図 9　ド・モルガンの法則による条件式の書き換え
Fig. 9　De Morgan in Expressions.

6.　振る舞いを変えない変更

本章では，提案手法で新たに検出された振る舞いを変えない変更の幾つかを紹介する．

アノテーション

アノテーションには，コンパイル時に情報が残るアノテーションと残らないアノテーションが存在する．振る舞いを変えないと検出されたアノテーションの一部を図 8 に載せる．また，図に示す以外にも振る舞いを変えないプロジェクト独自のアノテーションも存在した．

ド・モルガンの法則による条件式の書き換え

実際の変更を図 9 に示す．調査対象の Spring Framework Core において，ド・モルガンの法則に従った OR 演算子から AND 演算子の書き換えが振る舞いを変えない変更であると検出された．

if 文の統合

実際の変更を図 10 に示す．調査対象の Apache Ant において if 文の統合が振る舞いを変えない変更と検出された．if 文の中で二重に if 文を記述する形式から二つの条件を AND 演算子で統合し一つの if 文で記述する形式への変更は振る舞いを変えない．

総　称　型

総称型の型宣言の挿入，削除を振る舞いを変えない変更と検出した．総称型の挿入を図 11 (a) に，総称型のワイルドカードの挿入を図 11 (b) に，総称型から Object 型への変更を図 11 (c) に示す．

```
  for (final ArgumentProcessor processor :
        processorRegistry.getProcessors()) {
    final List<String> extraArgs =
        extraArguments.get(processor.getClass());
-   if (extraArgs != null) {
-     if (processor.handleArg(extraArgs)) {
-       return;
-     }
+   if (extraArgs != null && processor.handleArg(extraArgs)) {
+     return;
    }
```

図 10　if 文の統合
Fig. 10　If-Condition Consolidation.

```
    } else {
-     ArrayList cipherList = new ArrayList<String>(1);
+     ArrayList<String> cipherList = new ArrayList<>(1);
      cipherList.add(sm.getString("managerServlet.notSslConnector"));
      result.put(connector.toString(), cipherList);
    }
```

(a) 総称型の挿入

```
  public Constructor<?> getConstructor() {
-   return (this.executable instanceof Constructor ?
-             (Constructor) this.executable : null);
+   return (this.executable instanceof Constructor ?
+             (Constructor<?>) this.executable : null);
  }
```

(b) 総称型のワイルドカードの挿入

```
  protected <T> T loadClass(...) {
    ...
    Class<?> clazz;
-   T rv = (T) clazz.newInstance();
+   Object rv = clazz.newInstance();
    if (!type.isInstance(rv)) {
      throw new BuildException(
        "Specified class (%s) %s", classname, msg);
    }
-   return rv;
+   return (T) rv;
  }
```

(c) 総称型から Object 型への変更

図 11　総　称　型
Fig. 11　Generics.

修飾子の並び替え

フィールド変数宣言やメソッド宣言における修飾子の並び替えを振る舞いを変えない変更と検出した．Spring Framework Core において検出されたフィールド変数宣言の final 修飾子と static 修飾子の並び替えを図 12 (a) に示す．また，Apache Tomcat において検出されたメソッド宣言の public 修飾子と synchronized 修飾子の並び替えを図 12 (b) に示す．

Iterator の使用から拡張 for 文の使用への変更

実際の変更を図 13 に示す．拡張 for 文は J2SE 5.0 で導入された記述であり，コレクションの全ての要素にアクセスするために用いられる．Iterator を使用した記述から拡張 for 文を使用した記述への変更を振る舞いを変えない変更と検出した．

```
  public class TypePath {
    ...
    */
-   public final static int ARRAY_ELEMENT = 0;
+   public static final int ARRAY_ELEMENT = 0;
```

(a) フィールド変数宣言時の修飾子の並び替え

```
    * /
    @Override
-   synchronized public PooledObject<PoolableConnection> makeObject()
-     throws Exception {
+   public synchronized PooledObject<PoolableConnection> makeObject()
+     throws Exception {
      Connection conn = getConnectionFactory().createConnection();
```

(b) メソッド宣言時の修飾子の並び替え

図 12　修飾子の入れ替え
Fig. 12　Reorder Modifiers.

```
   Set<String> keys = compositeType.keySet();
-  for (Iterator<String> iter = keys.iterator(); iter.hasNext();) {
-    String key = iter.next();
+  for (String key : keys) {
     Object value = data.get(key);
```

図 13　Iterator の使用から拡張 for 文の使用への変更
Fig. 13　Iterator to Enhanced For-Block.

```
  public static XMLInputFactory createDefensiveInputFactory() {
-   return createDefensiveInputFactory(XMLInputFactory::newFactory);
+   return createDefensiveInputFactory(XMLInputFactory::newInstance);
  }
```

図 14　誤検出：メソッド参照におけるメソッド名の変更
Fig. 14　False Positive: a Change in Method Reference.

7. 誤　検　出

本章では，本研究の提案手法において誤検出された変更とその理由について述べる．

誤検出された変更はメソッド参照におけるメソッド名の変更である．実際の変更を図 14 に示す．メソッド参照とは JavaSE 8 で導入された構文である．「クラス名::メソッド名」と記述し，メソッドを引数なしで呼び出す処理が行われる．

図 14 において，呼び出されるメソッド名が newFactory から newInstance に変更されている．メソッド名が変更され，変更前とは別のメソッドの処理が行われるためプログラムの振る舞いを変える変更である．しかし，本研究の手法では振る舞いを変えない変更と誤検出された．その理由は，バイトコードを逆アセンブルする `javap` コマンドのオプションに`-p -c` のみを指定し，`-v` を指定していないためである．`-v` オプション使用時に逆アセンブルされる定数プールの情報を提案手法では無視していた．しかし，メソッド参照において呼び出されるメソッドの情報は定数プールに保存されるため，誤検出された．

提案手法において，`-v` オプションを指定しなかった理由は，定数プールに変数名などの情報が載ってしまうためである．振る舞いを変えない変更とみなすべきである変数名などの変更が，`-v` オプションを用いると振る舞いを変える変更であると判定されてしまう．そのため，著者らは`-v` オプションは使用すべきでないと考えた．

8. 妥当性への脅威

本研究の提案手法ではソースコードが変更されてもバイトコードの逆アセンブリ後の結果が変化しない変更を振る舞いを変えない変更とした．しかし，バイトコードが変化するがプログラムの振る舞いを変えない変更も存在する．先行研究において振る舞いを変えない変更とされていたクラス名やフィールド変数名，メソッド名の変更などがあたる．これらは本研究の検出方法ではバイトコードが変化するため振る舞いを変える変更と判定されてしまう．このように，現時点ではプログラムの振る舞いを変えない変更全てを提案手法において正しく検出することはできない．

また，本研究の調査では一つのコンパイラを用いた調査しか行っていない．違うコンパイラを使用することで本研究の結果とは違うバイトコードが得られ，調査結果が変化する可能性がある．例えば，本研究ではド・モルガンの法則に従った AND 演算子から OR 演算子の書き換えは振る舞いを変えない変更と検出された．これは，AND 演算子から OR 演算子への変更前後で同じバイトコードが生成されたためである．しかし，コンパイラによっては AND 演算子，OR 演算子の記述によってそれぞれ違うバイトコードを生成する可能性がある．そのため，振る舞いを変える変更と検出される可能性がある．

9. む　す　び

本研究では，バイトコードを利用した振る舞いを変えない変更を検出する手法を提案し，調査を行った．調査の結果，六つの Java プロジェクトにおいて，Java ファイルの変更が行われたコミットのうち，8.6% ~ 22.4% は振る舞いを変えないコミットであった．また，今回の調査では先行研究と比較し新たに 25 種類の振る舞いを変えない変更を検出した [11].

今後の課題としては，Java 以外の VM 言語での実験を行い，振る舞いを変えない変更の言語による違いはあるのかを調査をしていきたい．優れたコンパイラが設計されている言語の場合，Java に比べプログラムの振る舞いを変えない変更がより発生しにくいと考えられる．

文　　献

[1] B. Fluri and H.C. Gall, “Classifying change types for qualifying change couplings,” Proc. IEEE Int. Conf. Program Comprehension, pp.35–45, 2006.

[2] R. Purushothaman and D.E. Perry, “Toward understanding the rhetoric of small source code changes,” IEEE Trans. Softw. Eng., vol.31, no.6, pp.511–526, 2005.

[3] A. Hindle, D.M. German, and R. Holt, “What do large commits tell us? a taxonomical study of large commits,” Proc. Int. Working Conf. Mining Software Repositories, pp.99–108, 2008.

[4] S. Rastkar and G.C. Murphy, “Why did this code change?,” Proc. Int. Conf. Software Engineering, pp.1193–1196, 2013.

[5] T. Baum, S. Herbold, and K. Schneider, “An industrial case study on shrinking code review changesets through remark prediction,” Dec. 2018.

[6] H. Gall, M. Jazayeri, and J. Krajewski, “Cvs release history data for detecting logical couplings,” Proc. Int. Workshop Principles of Software Evolution, pp.13–23, 2003.

[7] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, “Predicting source code changes by mining change history,” IEEE Trans. Softw. Eng., vol.30, no.9, pp.574–586, 2004.

[8] M. Wen, R. Wu, and S. Cheung, “Locus: Locating bugs from software changes,” Proc. Int. Conf. Automated Software Engineering, pp.262–273, 2016.

[9] B. Fluri, M. Wursch, M. PInzger, and H. Gall, “Change distilling:tree differencing for fine-grained source code change extraction,” IEEE Trans. Softw. Eng., vol.33, no.11, pp.725–743, 2007.

[10] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” ACM/IEEE Int. Conf. Automated Software Engineering, ASE ’14, Vasteras, Sweden - Sept. 15 - 19, 2014, pp.313–324, 2014.

[11] D. Kawrykow and M.P. Robillard, “Non-essential changes in version histories,” Proc. Int. Conf. Software Engineering, pp.351–360, 2011.

[12] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” Proc. Int. Conf. Software Engineering, pp.284–292, 2005.

[13] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, “Does code decay? Assessing the evidence from change management data,” IEEE Trans. Softw. Eng., vol.27, no.1, pp.1–12, 2001.

[14] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” IEEE Trans. Softw. Eng., vol.31, no.6, pp.429–445, 2005.

[15] X. Wang, L. Pollock, and K. Vijay-Shanker, “Automatic segmentation of method code into meaningful blocks to improve readability,” Proc. Working Conference on Reverse Engineering, pp.35–44, 2011.

[16] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, “kGenProg: A high-performance, high-extensibility and high-portability apr system,” Proc. Asia-Pacific Software Engineering Conference, pp.697–698, 2018.

（2020 年 5 月 29 日受付，9 月 30 日再受付，
12 月 2 日早期公開）

前島　葵

令 2 大阪大学基礎工学部情報科学科卒．現在，大阪大学大学院情報科学研究科博士前期課程在学中．リポジトリマイニングに関する研究に従事．

肥後　芳樹　（正員）

2002 大阪大学基礎工学部情報科学科中退．2006 同大学大学院博士後期課程了．2007 同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助教．2015 同准教授．博士（情報科学）．ソースコード分析，特にコードクローン分析，リファクタリング支援，ソフトウェアリポジトリマイニング及び自動プログラム修正に関する研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE 各会員．

松本淳之介

平 30 大阪大学基礎工学部情報科学科卒．令 2 大阪大学大学院情報科学研究科博士前期課程卒．リポジトリマイニングに関する研究に従事．

楠本　真二　（正員）

1988 大阪大学基礎工学部卒．1991 同大学大学院博士課程中退．同年同大学基礎工学部助手．1996 同講師．1999 同助教授．2002 同大学大学院情報科学研究科助教授．2005 同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．情報処理学会，IEEE，IFPUG 各会員．