社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Java プロジェクトの変更履歴に含まれる本質的でない変更についての調査

前島 葵[†] 肥後 芳樹[†] 松本淳之介[†] 楠本 真二[†]

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{a-maejim,higo,j-matumt,kusumoto}@ist.osaka-u.ac.jp

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．本研究では，前者を本質的な変更とし，後者を本質的でない変更とする．本質的でない変更は，プログラムの振る舞いに影響を与えないため，本質的な変更が行われた時と同様の費用をテストにかける必要がない．本質的でない変更の割合を調査することで，無駄となっているテストの費用を推定できる．本研究では，Java プロジェクトを対象とし，バイトコードの変化の有無による本質的な変更と本質的でない変更の分類方法を提案する．コンパイルによって失われる情報だけを含む変更は本質的でないとみなし，ソースコードを変更した際にバイトコードが変わらない変更を本質的でない変更とする．調査の結果 4 つの Java プロジェクトに含まれる Java ファイルの変更がなされたコミットのうち，8.9%~22.3% のコミットは本質的でない変更のみで構成されていた．

キーワード ソフトウェアリポジトリマイニング，ソフトウェア進化

1. まえがき

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 [1]．変更履歴に記録された変更をマイニングする多くの手法が存在し，マイニングされた変更の情報は，欠陥予測 [2], [3]，コード間の依存関係の検出 [4], [5] に利用されている．

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のプログラムの振る舞いが変わる変更が存在する一方で，コメントの変更やソースコードのフォーマット，変数名の変更などのプログラムの振る舞いに影響を与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，プログラムの振る舞いは保持される．本研究では，プログラムの振る舞いが変わる変更を本質的とし，プログラムの振る舞いが変わらない変更を本質的ではないとする．

コミットごとに大きな費用を費やしてテストが行われている．しかし，コミットの中には本質的な変更で構成されたコミットがある一方で，フォーマットやコメントの変更のみで構成されたコミットも存在する．本質的でない変更しかなされていないコミットは，本質的な変更がされているコミットと同様の費用をかけてテストを行う必要がない．

Kawrykow らによって，本質的でない変更についての調査が行われた [6]．彼らは本質的でない変更を，プログラムの振る舞いが保持される変更であり，リポジトリマイニングにおいて分析の必要がない変更と定義した．また，その定義から本質的でない変更例を挙げた．コメントの変更や，変数名の変更などがあたる．調査の結果，メソッドの変更の最大で 15.5% が本質的でない変更であるという結果が報告された．しかしながら，調査の対象となった本質的ではない変更は彼らの定義による主観的な分類である．そのため，彼らの調査で発見されていない本質的で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，バイトコードの変化の有無による本質的な変更と本質的でない変更の分類方法を提案する．変更の情報がコンパイルによって失われるか失われないのかで分類を行うため，分類者の主観が入らない分類が可能になる．

提案手法では，既存の Java プロジェクトリポジトリから，Java ファイルとバイトコードを逆コンパイルしたファイルを持つリポジトリを作成する．作成されたリポジトリの変更履歴を辿り Java ファイルの変更のみがなされているコミットを本質的でない変更のみで構成されたコミットとして調査を行う．

調査では以下に示す 2 つの Research Question に答えることを目的とする．

RQ 1 本質的でない変更がどの程度存在するか
RQ 2 本質的でない変更にはどのような変更があるか

調査の結果，4 つの Java プロジェクトにおいて，Java ファイルの変更がなされたコミットのうち，8.9%~22.3% のコミットは本質的でない変更のみで構成されたコミットであることを確認した．また，先行研究と比較し新たに本質的でない変更と定義された変更として，アノテーションの変更や final 修飾子の付与など 19 の変更が存在することを確認した．

2. 研究背景

Kawrykow らは本質的でない変更について調査を行い，メ

ソッドの変更の最大で 15.5% が本質的でない変更であるという結果を報告した [6]．

彼らは，本質的ではない変更の定義を，以下の 3 項目を満たす変更と定義した．

- 表面的な変更である
- プログラムの振る舞いが保持される
- 変更箇所の役割や関係から，意味のある情報を得られそうにない

これらの定義から本質的でない変更を以下の 6 つとした．詳しい内容については 6. で述べる．

- 完全限定名から単純名への変更
- 一時変数への式の抽出
- 名前の変更およびその参照箇所の変更
- 些細なキーワードの更新
- ローカル変数名の変更
- 空白文字およびコメントの更新

しかしながら，これらの 6 つの変更は彼らの定義による主観的な分類である．そのため，彼らの調査で発見されていない本質的でない変更が存在するのではないかと著者らは考えた．

3. 調 査 方 法

3.1 分　類　法

本研究では本質的な変更と本質的ではない変更を，バイトコードの変化の有無で分類する．

バイトコードとは，Java ソースコードのコンパイル時に生成される中間コードのことである．バイトコードの変化の有無で分類することで，ソースコードがコンパイルされる際に失われるコメントやフォーマットなどの情報は本質的ではない情報と定義できる．

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本質的な変更とし，ソースコードが変化したにも関わらずバイトコードが変化しなかった変更は本質的ではない変更とする．

3.2 rJava ファイル

本研究では，バイトコードを逆コンパイルした結果得られるファイルを rJava ファイルと呼ぶ．

逆コンパイルには，`javap -p -c [file]` コマンドを用いた．オプションには，`-p -c` を指定した．`-p` オプションにより，プライベートメソッドも含めすべてのメソッド，フィールドを逆コンパイルの対象と指定する．`-c` オプションにより，メソッドごとに，逆コンパイルされたすべてのバイトコード命令を出力する．

3.2.1 変換リポジトリの作成方法

調査を行うにあたって，前準備として既存の Java プロジェクトリポジトリから，Java ファイルと rJava ファイルを持つリポジトリを作成する．変換リポジトリの作成方法の概要を図 1 に示す．変換リポジトリの作成方法の入力は，Java ソースコードを含む Git リポジトリである．出力は，Git リポジトリに含まれる全てのコミットに対して以下の 4 つのステップを適用し作成された Java ファイルと rJava ファイルで構成される Git リポジトリである．

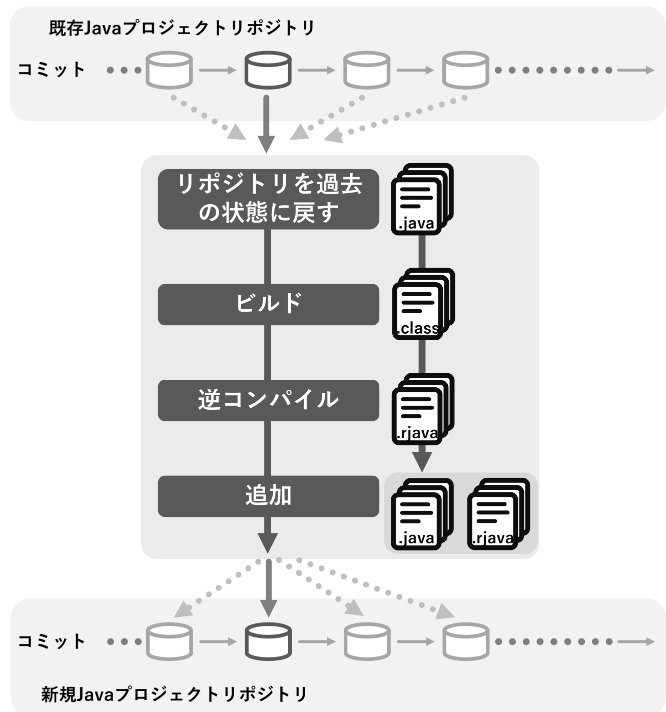

図 1　調 査 方 法

Step 1　リポジトリを特定のコミットの状態に復元
Step 2　ビルド
Step 3　逆コンパイル
Step 4　Java, rJava ファイルを新規リポジトリに追加

以降，各 Step について説明する．

Step1 ではリポジトリを特定のコミットの状態に復元する．Step2 ではビルド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．対象のプロジェクトごとにビルドツールは異なっており，本研究では Gradle および Ant を用いた．Step3 ではクラスファイルを逆コンパイルする．逆コンパイルの結果 rJava ファイルが作成される．Step4 では Java ファイルと rJava ファイルを対応付け，新規リポジトリに追加する．

3.3 調 査 方 法

作成された変換リポジトリの変更履歴を調べ，Java ファイルが変更された全てのコミットを調査する．Java ファイルの変更にともない rJava ファイルが変更されていれば本質的な変更がなされたコミット，rJava ファイルが変更されていなければ本質的でない変更のみで構成されたコミットであると分類する．

4. Research Question

本研究では，Java プロジェクトリポジトリにどの程度本質的でない変更が存在するかを調査する．調査を行うにあたり以下に示す 2 つの Research Question を設定した．

RQ1: 本質的でない変更がどの程度存在するか

Java プロジェクトのコミットの何割が本質的でない変更のみで構成されているか調査する．コメントの変更やフォーマットの変更のような本質的でない変更しかなされないコミットが実際に存在するのか，またその割合を調査する．本質的で

ない変更のみで構成されたコミットは，プログラムのテストを行う必要がないため，どの程度のテストの費用が削減できるかの推定に繋がる．

RQ2: 本質的でない変更にはどのような変更があるか

バイトコードの変化の有無で分類することで，どういった変更が本質的でない変更に分類されるか調査する．先行研究では，Kawrykow らによって本質的でない変更の定義がなされていたが，バイトコードによる分類法を用いることで，本質的でないと定義できる変更の範囲がどの程度広がるのか調査を行う．

5. RQ1 ，RQ2 の調査

本章では，バイトコードの変化の有無による分類法を用いて行った調査と，その結果について述べる．

5.1 調査対象

調査対象は Java ソースコードを含む Java プロジェクトのリポジトリである．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，先行研究が対象にしたプロジェクト（Apache ant, Azureus, Hibernate, JDT-Core, JDT-UI, Spring Framework, Xerces）のうち，Java のバージョンの違いなどの理由によりビルドに失敗したプロジェクトを除いたプロジェクト（Apache ant, Hibernate, Spring Framework) を加えた 4 つのプロジェクトで調査を行った．

調査対象のプロジェクト，調査の対象期間を表 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，調査対象に含まなかった．

5.2 RQ1 の調査結果

調査結果を表 2 に示す．4 つの Java プロジェクトにおいて，Java ファイルの変更がなされたコミットのうち，8.9%～22.3% のコミットは本質的でない変更のみで構成されたコミットであることを確認した．

5.3 RQ2 の調査結果

また，4 つのプロジェクトで本質的でない変更と分類された結果全てに対して目視確認を行った．本質的ではない変更の分類，その割合を表 3 に示す．先行研究と比較し新たに本質的でない変更と定義された変更として，アノテーションの変更や final 修飾子の付与など 19 の変更が存在することを確認した．本質的ではない変更の中で最も多かったものは，コメントの変更であり続いてフォーマットであった．

表 1 調査対象プロジェクト，対象期間

プロジェクト名	調査開始	調査終了
kGenProg	2018-04-09	2019-11-04
Apache ant	2017-01-01	2019-05-22
Spring Framework Core	2016-08-15	2019-11-13
Hibernate ORM Core	2016-12-16	2019-07-05

```
-     java.util.List list = new ArrayList<>;
+     List list = new ArrayList<>;
```

図 2 完全限定名から単純名への置換

6. 考察

本章では，本質的でない変更を以下の 3 種類に分類し，述べる．また先行研究と差が生じた割合についても述べる．

- 先行研究と同様に本質的でないと定義された変更
- 先行研究では本質的であるが提案手法では本質的でないと定義された変更
- 先行研究では本質的でないが提案手法では本質的と定義された変更

6.1 先行研究と同様に本質的でないと定義された変更

コメントの変更 コメントの変更には，1 行，もしくは複数行に渡るコメント，および Javadoc の変更が挙げられる．これらはコンパイル時に情報が失われるため，バイトコードに変化を与えない．調査の結果，Apache ant，Spring Framework Core では本質的でない変更の半分がコメントの変更であった．

フォーマットの変更 フォーマットには，インデントの調整，等号や括弧の前後のスペースの挿入削除，中括弧後の改行の調整，行の折り返しの調整などが存在した．コメントと同様に可読性や保守性を高めるフォーマットの変更であるが，バイトコードに変化を与えない．コメントに次いで割合が多く，12.2%-31.0% を占めた．

完全限定名から単純名への置換 例を図 2 に示す．パッケージ名を指定してクラスを呼び出す完全限定名の記述から，import 文にパッケージ名を記述しクラス名のみを用いて呼び出しを行う単純名の記述への変更はバイトコードに変化を与えない．割合としては少なく，0.0%-1.7% ほどであった．

ローカル変数名の変更 ローカル変数名の変更，および変更にともなう参照箇所の変更は，バイトコードに変化を与えなかった．ローカル変数名の変更は，変更によって与えらえる影響がそのメソッド内で閉じている．そのため，他のクラスやメソッドに影響を及ぼすことのない変更だといえる．

パラメータ名の変更 パラメータ名の変更，および変更にともなう参照箇所の変更は，ローカル変数名の変更と同じくバイトコードに変化を与えなかった．

this の付与，削除 this の付与例を図 3 に示す．this の付与，削除についてはバイトコードが変化しないことが調査対象のプロジェクトから確認できた．

return 文の付与，削除 return 文の付与例を図 4 に示す．戻り値が void 型であるメソッドでは，メソッドの処理の最後に return 文を記述しても記述しなくても良い．これらは，バイトコード中では return 文のない記述に統一される．

super() の付与，削除 super() によるスーパークラスの引数

表 2 調査結果

プロジェクト名	Java ファイルの変更回数	rJava ファイルの変更回数	本質的でない変更の割合
kGenProg	808	651	19.4%
Apache ant	323	259	19.8%
Spring Framework Core	803	624	22.3%
Hibernate ORM Core	1,178	1.073	8.9%

```
-     hoge();
+     this.hoge();
```

図 3　this の付与

```
-     void hoge() {};
+     void hoge() { return; };
```

図 4　return の挿入

```
-     void hoge() {};
+     void hoge() { super(); };
```

図 5　super() の挿入

なしコンストラクタの呼び出しの有無について．super() の付与例を図 4 に示す．サブクラスでは，自動的にスーパークラスの引数なしコンストラクタの呼び出しがコンパイラによって行われる．そのため，super() を記述する必要がない．

6.2　先行研究では本質的であるが提案手法では本質的でないと定義された変更

バイトコードの変化の有無による分類によって，先行研究に対し新たに本質的ではない変更と定義される例が存在した．

final 修飾子の付与，削除　final 修飾子は変数の代入を無効にするために使用されるが，final 修飾子を付与してもバイトコードが変化しない例が存在した．メソッド引数への final 修飾子の付与例を図 6(a) に，catch 節への final 修飾子の付与例を図 6(b) に，拡張 for 文の要素への final 修飾子の付与例を図 6(c) に，ローカル変数の ArrayList への final 修飾子の付与例を図 6(d) に示す．これらはいずれもバイトコードに変化を与えなかった．また，ArrayList の宣言のように，参照型のローカル変数への final 修飾子の付与はバイトコードに変化を与えなかった．今回の調査対象プロジェクトの中では，kGenProg のみ final 修飾子を多用していた．

```
-     void hoge(int fuga){
+     void hoge(final int fuga){
```

(a) メソッド引数への final 修飾子付与

```
-     } catch (Exception e) {
+     } catch (final Exception e) {
```

(b) catch 節への final 修飾子付与

```
-     for(int hoge : list){
+     for(final int hoge : list){
```

(c) 拡張 for 文の要素への final 修飾子付与

```
-     List<String> list= new ArrayList<>;
+     final List<String> list= new ArrayList<>;
```

(d) ローカル変数の ArrayList への final 修飾子の付与

図 6　final 修飾子付与の本質的でない変更

import 文の並び替え，削除　先行研究では，メソッド単位で本質的ではない変更が調査されていたためメソッド外の文は注目されていなかった．しかし，変更の中には，import 文の並べ換えを行う変更や，未使用の import 文を削除する変更も存在し，これらの変更はバイトコードに変化を与えなかった．また，調査対象 4 つのプロジェクト全てにおいて import 文の変更は行われ，5.2%-11.1% 存在した．

アノテーションの変更　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．バイトコードが変化しないとされたアノテーションの例を図 7 に載せる．先行研究に対し新たに本質的ではない変更と定義された変更の中で最も割合が高く，7.8%-16.3%

表 3　調 査 結 果

本質的ではない変更例	kGenProg	Apache ant	Spring Framework Core	Hibernate ORM Core
コメントの挿入，変更，削除	56 (34.6%)	42 (54.5%)	124 (54.1%)	53 (41.1%)
フォーマットの変更	26 (16.0%)	16 (20.8%)	28 (12.2%)	40 (31.0%)
完全限定文，単純名間の変更	2 (1.2%)	1 (1.3%)	4 (1.7%)	
ローカル変数名の変更	10 (6.2%)	1 (1.3%)	4 (1.7%)	
パラメータ名の変更	3 (1.9%)	1 (1.3%)	2 (0.9%)	1 (0.8%)
this の挿入，削除	2 (1.2%)		4 (1.7%)	
final 修飾子の付与	17 (10.5%)		1 (0.4%)	
import 文の挿入，削除，並べ替え	18 (11.1%)	5 (6.5%)	12 (5.2%)	8 (6.2%)
アノテーションの挿入，削除，編集	25 (15.4%)	6 (7.8%)	30 (13.1%)	21 (16.3%)
インターフェース中のメソッドの public 修飾子の削除	1 (0.6%)			2 (1.6%)
キャストの挿入，削除	1 (0.6%)		1 (0.4%)	
空文の削除	1 (0.6%)		1 (0.4%)	1 (0.8%)
条件式の書き換え			1 (0.4%)	
括弧の挿入，削除			8 (3.5%)	
if 文のブロックの挿入，削除			1 (0.4%)	2 (1.6%)
if 文の連結		1 (1.3%)		
else if 文から if 文への書き換え			1 (0.4%)	
総称型と Object 型		1 (1.3%)		
総称型のワイルドカードの挿入削除		1 (1.3%)	3 (1.3%)	
総称型の型宣言の挿入削除		1 (1.3%)		
ダイヤモンド演算子の使用		1 (1.3%)		
配列宣言の変更			1 (0.4%)	
修飾子の入れ替え			1 (0.4%)	
newFactory, newInstance の書き換え			1 (0.5%)	
呼び出しのないフィールド変数の初期化の変更			1 (0.5%)	1 (0.8%)

```
+       @Override
+       @Deprecated
+       @Expose
+       @SuppressWarnings("unused")
+       @Incubating
+       @Target({ FIELD, METHOD })
```

図 7　アノテーションの変更

```
-       final ProductSourcePath path =
                (ProductSourcePath) ast.getProductSourcePath();
        //getProductSourcePath()はProductSourcePath型を返す
+       final ProductSourcePath path =
                    ast.getProductSourcePath();
```

図 8　キャストの削除

```
-       if(!hoge || !fuga){
+       if(!(hoge && fuga)){
```

図 9　条件式の書き換え

```
-       if( hoge==null && fuga ){
+       if( (hoge==null) && fuga ){
```

図 10　括弧の挿入

```
-       else
-               hoge();
+       else {
+               hoge();
+       }
```

図 11　if 文のブロックの挿入

存在した．

インターフェース中のメソッドの public 修飾子の削除　インターフェースでは，メソッドは暗黙的に public メドッドになるため，メソッド宣言において public 修飾子が取り除かれてもバイトコードが変化しない．

キャストの削除　例を図 8 に示す．キャストの挿入や削除によって，バイトコードが変わらない例が存在した．

空文の削除　セミコロンのみが記述された空文は命令を実行しないため削除されてもバイトコードが変化しない．

条件式の書き換え　例を図 9 に示す．調査対象の Spring Framework Core において変更が確認された．ド・モルガンの法則に従って OR 演算子を AND 演算子で書き換えた条件式はバイトコードに変化を与えなかった．また，バイトコード中では OR 演算子を用いた記述に統一されていた．

括弧の挿入．削除　例を図 10 に示す．可読性の向上のため，括弧を用いて数式や条件式を括り出す変更や，括弧を取り除く変更が存在した．これらの括弧の挿入削除はバイトコードに変化を与えなかった．

if 文のブロックの挿入，削除　例を図 11 に示す．調査対象の Spring Framework Core において変更が確認された．if 文および else 文の後に 1 行で命令を記述する形式から，ブロックを用いる記述への変更はバイトコードに変化を与えなかった．

if 文の連結　例を図 12 に示す．調査対象の Apache ant において変更が確認された．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 演算子で結合し，if 文を一つにまとめた記述への変更はバイトコードに変化を与えなかった．

else if 文から if 文への書き換え　例を図 13 に示す．調査対象の Spring Framework Core において変更が確認された．直前の if 文の処理に return 文が記述されていたならば，その直

```
-       if(hoge){
-               if(fuga){
-                       continue;
-               }
+       if(hoge && fuga){
+                       continue;
        }
```

図 12　if 文の連結

```
        if(hoge){
                return;
        }
-       else if(fuga){
+       if(fuga){
```

図 13　else if 文から if 文への書き換え

```
        protected <T> T hoge() {
                Class<?> c;
-               T rv = (T) c.newInstance();
+               Object rv = c.newInstance();
                ......
-               return rv;
+               return (T) rv;
}
```

図 14　総称型と Object 型

```
-       Class<?> superclass = ((Class) element).getSuperclass();
+       Class<?> superclass = ((Class<?>) element).getSuperclass();
```

図 15　総称型のワイルドカードの挿入

```
-       List<String> list = new ArrayList<String>();
+       List<String> list = new ArrayList<>();
```

図 16　ダイヤモンド演算子の使用

```
-       public final static int hoge = 0;
+       public static final int hoge = 0;
```

図 17　修飾子の入れ替え

```
-       char chars[] = new char[5];
+       char[] chars = new char[5];
```

図 18　配列宣言の変更

後の else if 文が if 文に変更されてもバイトコードは変化しなかった．

総称型と Object 型　例を図 14 に示す．コンパイル時に総称型は Object 型としてコンパイルされるため，総称型を Object 型に書き換えてもバイトコードは変化しなかった．

総称型のワイルドカードの挿入削除　例を図 15 に示す．総称型のワイルドカードの挿入削除はバイトコードに変化を与えなかった．

ダイヤモンド演算子　例を図 16 に示す．ダイヤモンド演算子は Java 7 から導入された記述であり，インスタンス生成時に型推論が行われるため，リスト宣言の右側のデータ型名の省略が可能になる．ダイヤモンド演算子を導入してもバイトコードは変化しない．

修飾子の入れ替え　例を図 17 に示す．final 修飾子と static 修飾子を入れ替えてもバイトコードが変化しない例が存在した．

配列宣言の変更　例を図 18 に示す．char 型修飾子と，char 型配列修飾子を入れ替えてもバイトコードが変化しない例が存在した．

```
-     return hoge(XMLInputFactory::newFactory);
+     return hoge(XMLInputFactory::newInstance);
```

図 19　newFactory, newInstance の書き換え

```
-     hoge(list.size());
+     int size = list.size();
+     hoge(size);
```

図 20　一時変数の導入

newFactory, newInstance の書き換え　例を図 19 に示す．newFactory, newInstance は共に XMLInputFactory クラスのメソッドであり，どちらも新しいインスタンスを作成する処理を行う命令として実装されているため，バイトコードが変化しなかった．

呼び出しのないフィールド変数の初期化の変更　呼び出しの行われないフィールド変数が初期化された場合には，バイトコードが変化しない．

6.3　先行研究では本質的でないが提案手法では本質的と定義された変更

一時変数の導入　例を図 20 に示す．一時変数を導入し，後続のプログラムで式の代わりに一時変数を参照する変更はコードの可読性を向上させる．しかし，先行研究では本質的ではない変更とされていたが，本研究では一時変数の作成はバイトコードが変化してしまうため本質的な変更と判定された．

名前の変更と参照箇所の変更　クラス名，フィールド名，メソッド名の変更は，それにともなう参照箇所の変更も引き起こす．ローカル変数名や，パラメータ名，それにともなう参照箇所の変更はバイトコードに変化を与えなかったが，クラス名，フィールド名，メソッド名は，バイトコードが変化してしまうため本質的な変更と判定された．

6.4　先行研究との差異

バイトコードの変化の有無で分類することにより先行研究に比べ，新たに本質的でないと定義された変更の割合を表 4 に示す．すべての対象プロジェクトにおいて，先行研究と比較し新たに本質的でないと定義できる変更が存在した．さらに，アノテーションの変更や final 修飾子の付与など新たに 19 の変更が本質的でない変更と定義された．また，final 修飾子の付与やアノテーションの変更が多く存在した kGenProg は，他のプロジェクトに比べ新たに本質的でないと定義された変更の割合が多くなっていた．対して，本質的でない変更のうちコメントやフォーマットの変更が大部分を占めた Apache ant は新たに本質的でないと定義された変更の割合が低くなった．

7.　妥当性への脅威

本研究では本質的でない変更をプログラムの振る舞いを変えない変更と定義した．また，プログラムの振る舞いを変えない変更を，ソースコードが変更された際にバイトコードが変化しない変更とした．しかし，プログラムの振る舞いが保持され，ソースコードが変更された際にバイトコードが変化する変更が存在する．先行研究において本質的でない変更とされていたクラス名，メソッド名の変更とそれにともなう参照箇所の変更や，一時変数の導入は，本研究の分類方法ではバイトコードが変化するため本質的な変更と分類されてしまう．また，メソッドの移動についてもバイトコード中の命令の順序が変わってしまうため，本研究の分類方法では本質的な変更と判定されてしまう．このように，バイトコードが変化するが，プログラムの振る舞いは保持される変更が存在する．

8.　あ と が き

本研究では，バイトコードの変化の有無によって本質的な変更と本質的でない変更を分類する手法を提案し，調査を行った．調査の結果，4 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，8.9%〜22.3% のコミットは本質的でない変更のみで構成されていることを確認した．また，先行研究と比較し新たに 19 の変更が本質的でない変更と定義された．

今後の課題としては，本質的でない変更のみがなされているコミットのテストを削減することで，テスト時間がどの程度短縮されるか推測を行うことや，先行研究と比較し新たに本質的ではない変更とみなせる例が本論文で挙げた他に存在しないかの調査が挙げられる．

表 4　新たに本質的ではない変更と定義された変更の割合

プロジェクト名	本質的でない変更の数	先行研究同様に本質的でないと定義された変更の数	新たに本質的でないと定義された変更の数	新たに本質的でないと定義された変更の割合
kGenProg	162	99	63	38.9%
Apache ant	77	61	16	20.8%
Spring Framework Core	229	166	63	27.5%
Hibernate ORM Core	129	94	35	27.1%

文　　献

[1] M.J. Rochkind, “The source code control system,” IEEE Transactions on Software Engineering, vol.SE-1, no.4, pp.364–370, Dec. 1975.

[2] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, “Predicting fault incidence using software change history,” IEEE Transactions on Software Engineering, vol.26, no.7, pp.653–661, July 2000.

[3] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” Proceedings of the 27th International Conference on Software Engineering, pp.284–292, ICSE ’05, ACM, New York, NY, USA, 2005. http://doi.acm.org/10.1145/1062455.1062514

[4] H. Gall, M. Jazayeri, and J. Krajewski, “Cvs release history data for detecting logical couplings,” Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings., pp.13–23, Sep. 2003.

[5] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, “Predicting source code changes by mining change history,” IEEE Transactions on Software Engineering, vol.30, no.9, pp.574–586, Sep. 2004.

[6] D. Kawrykow and M.P. Robillard, “Non-essential changes in version histories,” 2011 33rd International Conference on Software Engineering (ICSE), pp.351–360, May 2011.