特別研究報告

題目

Java プロジェクトに含まれる本質的でない変更の検出

指導教員

楠本 真二 教授

報告者

前島 葵

2020 年 2 月 10 日

大阪大学 基礎工学部 情報科学科

2019 年度 特別研究報告

Java プロジェクトに含まれる本質的でない変更の検出

前島 葵

内容梗概

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なテストを行う必要がない．本質的でない変更を客観的かつ自動で判断できれば，不必要なテストを削減できる．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でない変更の検出方法を提案する．提案手法では，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 6 つの Java プロジェクトに含まれる Java ファイルが編集されたコミットのうち，8.6%~22.4% のコミットは本質的でない変更のみで構成されていた．また，先行研究と比較し新たに 25 種類の本質的でない変更を検出可能であることを確認した．さらに，Java のファイル変更が行われたコミットのテスト時間のうち，10.4% のテスト時間が削減できることを確認した．

主な用語

ソフトウェアリポジトリマイニング，ソフトウェア進化，継続的インテグレーション，版管理，Git

目次

図目次

表目次

1 はじめに

ソフトウェアの開発は，ソースコードの変更の繰り返しである．Git [1] や Subversion [2] 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 [3]．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が目指され，近年多くの研究が行われている [4–13]．

変更履歴に記録された情報をマイニングをすることで，変更にはどのような種類があるのか，どのような変更が行われたのか，その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 [4–7]．変更履歴の理解性の向上は，保守性の向上に繋がる．そのため，変更の重要度を測ることで，変更履歴の理解を助ける研究も行われている [8]．Fluri らは，ある変更が，機能を変更するための変更であるのか，機能を保持するための変更であるかに従って変更の重要度の評価が行われた [4] ．また，過去に同時に変更された情報を用いて，ソースコードの依存関係の分析が行われている [9]．その情報を利用し，同時に変更される可能性の高い箇所の予測を行い，開発を助ける研究も行われている [10] ．また，ソースコードの変更は欠陥との関連が高いと言われ [11, 14]，マイニングされた変更の情報から欠陥箇所を予測をする研究も行われている [11–13]．このように，開発履歴に記録された変更を分析する研究や，その分析を助ける研究，変更の情報を用いた研究が盛んに行われている．

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，コメントの変更やフォーマットの変更，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 [15, 16]，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な変更とし，プログラムの振る舞いが変わらない変更を本質的でない変更とする．

ソフトウェア開発では，継続的インテグレーション (Continuous Integration, 以降 CI と呼ぶ) が盛んに用いられている [17, 18]．CI とは，ソフトウェア開発において，ビルドやテストをコミット単位で頻繁に繰り返し行なうことで，問題を早期に発見し，開発の効率化や品質の維持を狙いとする手法である．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，フォーマットの変更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，テスト結果のフィードバックを待つ必要がなく，本質的な変更がされているコミットと同様のテストを行う必要がない．本質的でない変更で構成されたコミットを自動で判定することができれば，不要なテストの削減が可能になる．

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 [19]．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，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でない変更の検出方法を提案する．コンパイルされる際に失われる変更を本質的でない変更とするため，主観が入らない検出が可能になり，自動で検出が行える．

調査では以下に示す 3 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： どのような変更を本質的でないと検出するか
RQ 3： コミットのテスト時間がどの程度削減可能か

6 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% 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 25 種類の本質的でない変更を検出可能であることを確認した．さらに，Java のファイル変更が行われたコミットのテスト時間のうち，10.4% のテスト時間が削減できることを確認した．

以降，2 章で研究動機として背景研究とその問題点について述べる．3 章で調査方法について説明する．4 章で Research Question について述べ，5 章でその結果について示す．6 章で先行研究と提案手法の結果を比較し，考察を行う．7 章で発見された本質的でない変更について示す．また，8 章で誤検出について 9 章で妥当性の脅威について述べ，最後に 10 章で本研究のまとめと今後の課題について述べる．

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，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 [19] ．彼らは本質的でない変更を表面的であり，プログラムの振る舞いが保持され，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変更であると定義した．また，彼らはそ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 章で述べる．

- 完全限定名から単純名への変更
- 一時変数の導入
- 名前の変更
- 些細なキーワードの変更
- 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 つの変更が本質的でない変更とされたのは，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　検出法

本研究では，本質的でない変更をバイトコードの変化の有無により検出する．

バイトコードとは，ソースコードがコンパイルされた際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に注目することで，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ない情報とみなすことができる．

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` コマンドを用いた．このコマンドは，Java ファイルがコンパイルされた際に生成される Class ファイルを逆アセンブルするコマンドである．オプションを使用しない場合、`javap` コマンドは、指定されたファイルに含まれるクラスの protected および public なフィールドとメソッドを出力する．本研究において，オプションには`-p -c` を指定した．`-p` オプションにより，private な指定も含めすべてのメソッド，フィールドを逆アセンブルの対象に指定できる．`-c` オプションにより，メソッドごとに逆アセンブルされたバイトコード命令列を出力する．

3.3　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．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．

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 ファイルを新規リポジトリにコミット

以降，各 Step について説明する．

Step 1 では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

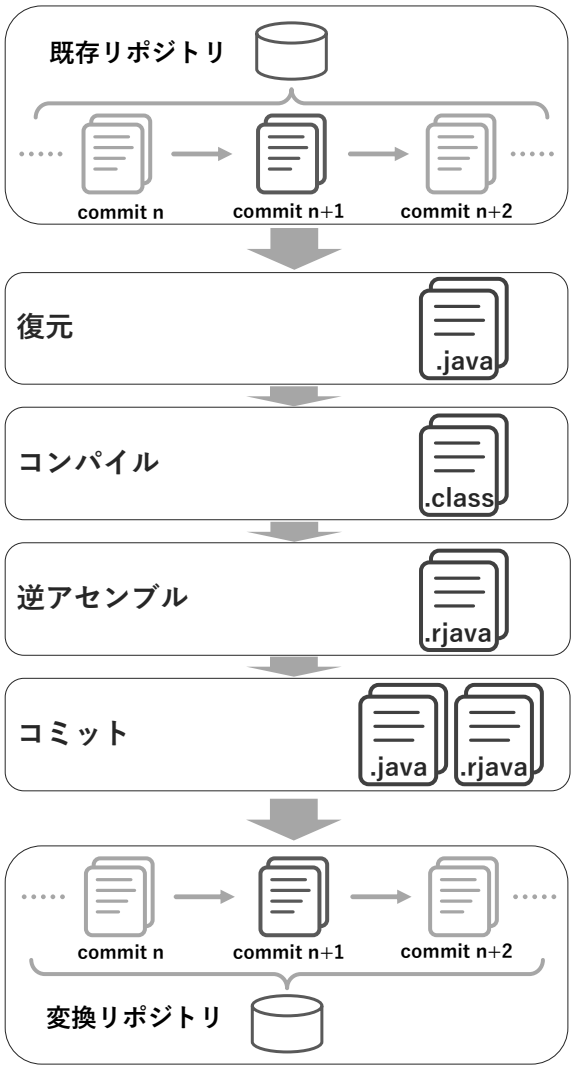

図 1　調査方法

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．Step 3 ではクラスファイルを逆アセンブルする．逆アセンブルの結果 rJava ファイルが作成される．Step 4 では Java ファイルと rJava ファイルを対応付け，新規リポジトリに追加する．

3.4　調査方法

`git log` コマンドを用いて作成された変換リポジトリの変更履歴を調べる．Java ファイルが変更されたコミットを調査し，その中から，本質的でない変更のみで構成されたコミットの検出を行う．本質的でない変更のみで構成されたコミットとは，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 1 つも行われていないコミットである．複数ファイルの変更があるコミットにおいては，1 つでも rJava ファイルの変更があれば，本質的な変更がなされたコミットとみなす．

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のみで構成されたコミット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 3 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 プロジェクトリポジトリに存在するコミットのどの程度が本質的でない変更のみで構成されているか調査する．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，どの程度のテストが削減できるかの推定に繋がる．

RQ2 : どのような変更を本質的でないと検出するか

バイトコードの変化の有無で分類することで，どのような変更を本質的でないと検出するか調査する．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検出法を用いることで本質的でないと定義できる変更の種類がどの程度増加するのか調査を行う．

RQ3 : コミットのテスト時間がどの程度削減可能か

RQ1 で得られた本質的でない変更のみで構成されたコミットのテストを削減することで，どの程度テスト時間が削減可能か調査を行う．

5 Research Question の調査

本章では，バイトコードの変化の有無により，本質的でない変更のみで構成されたコミットを検出した結果について述べる．

5.1 調査対象

調査対象は Java ファイルを含む 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 [20]，先行研究で調査対象とされていた Apache Ant, Hibernate ORM Core, Spring Framework Core．さらに Apache プロジェクト [21] である Apache Tomcat, Apache POI を加えた 6 つのプロジェクトを対象に調査を行った．先行研究では 7 つのプロジェクトが調査対象とされていたが，提案手法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一部のプロジェクトを除外した．

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，調査対象に含まない．

5.2 RQ1 : 本質的でない変更がどの程度存在するかの調査結果

6 つのオープンソース Java プロジェクトリポジトリに対して行った調査結果を表 2 に示す．表 2 には，プロジェクト名と Java ファイルが変更されたコミット，その中に含まれる本質的でない変更のみで構成されたコミットを示している．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，多いもので 1800 コミットのプロジェクトも存在した．

RQ1 への回答として，**「ファイルの変更が行われたコミットの中で，8.6%∼22.4% が本質的でない変更のみで構成されたコミットである」**といえる．

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	2020-01-24
Hibernate ORM Core	2016-12-16	2020-01-31
Apache Tomcat	2017-07-04	2020-01-15
Apache POI	2018-01-26	2020-01-13

5.3 RQ2：どのような変更を本質的でないと検出するかの調査結果

調査を行った 6 つのプロジェクトにおいて，検出された本質的でない変更のみで構成されたコミット全てに対して目視確認を行った．また，コミット内において行われた変更を変更の種類に基づいて排他的に数え，分類を行った．その際，1 つのコミット内で複数箇所に渡って同じ種類の変更がなされていても重複して数えない．コミット内で行われた本質的でない変更の分類，およびその変更が占める割合を表 3 に示す．

RQ2 への回答として，**「本質的でない変更には，表 3 に示す 31 種類の変更がある．また，表に上部に示す 6 種類の変更は先行研究 [19] においても本質的でない変更と分類されていたが，表の残り 25 種類は，提案手法で新しく発見された本質的でない変更である」**といえる．

本質的でない変更それぞれについて詳しくは，7 章で述べる．

割合に注目したところ，コメントの変更が全体の 46.7% の最も高い割合を占めた．続いてフォーマットの変更が 21.7% を占めた．提案手法により新たに発見された変更の中では，アノテーションの変更が全体の 10.2% であり，import 文の変更が 5.7% という結果になった．

表 2　RQ1 の調査結果

プロジェクト名	Java ファイルが変更されたコミット	本質的でない変更のみで構成されたコミット
kGenProg	808	157 (19.4%)
Apache Ant	323	64 (19.8%)
Spring Framework Core	851	191 (22.4%)
Hibernate ORM Core	1,253	114 (9.1%)
Apache Tomcat	1,800	278 (15.4%)
Apache POI	255	22 (8.6%)
Total	5,287	826 (15.6%)

表 3: RQ2 の調査結果

本質的でない変更	Total	kGenProg	Apache Ant	Spring Framework Core	Hibernate ORM Core	Apache Tomcat	Apache Poi
コメントの挿入，削除，変更	457 (46.7%)	56 (34.1%)	42 (56.0%)	133 (55.1%)	61 (43.9%)	150 (46.3%)	15 (42.9%)
フォーマットの変更	212 (21.7%)	26 (15.9%)	15 (20.0%)	30 (12.4%)	40 (28.8%)	93 (28.7%)	8 (22.9%)
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%)
ローカル変数名の変更	22 (2.2%)	10 (6.1%)	1 (1.3%)	5 (2.1%)		5 (1.5%)	1 (2.9%)
パラメータ名の変更	12 (1.2%)	3 (1.8%)	1 (1.3%)	2 (0.8%)	1(0.7%)	4 (1.2%)	1 (2.9%)
this の挿入，削除	7 (0.7%)	2 (1.2%)		4 (1.7%)			1 (2.9%)
final 修飾子の挿入，削除	22 (2.2%)	19 (11.6%)		1 (0.4%)	1 (0.7%)	1 (0.3%)	
import 文の挿入，削除，並べ替え	56 (5.7%)	18 (11.0%)	5 (6.7%)	14 (5.8%)	9 (6.5%)	8 (2.5%)	2 (5.7%)
アノテーションの挿入，削除，編集	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%)		
キャストの挿入，削除	1 (0.1%)	1 (0.6%)					
ラムダ式の引数の型の挿入，削除	1 (0.1%)			1 (0.4%)			
空文の削除	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%)			
括弧の挿入，削除	11 (1.1%)			7 (2.9%)		3 (0.9%)	1 (2.9%)
ブロックの挿入，削除	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%)			

本質的でない変更	Total	kGenProg	Apache Ant	Spring Framework Core	Hibernate ORM Core	Apache Tomcat	Apache Poi
総称型の挿入，削除	8 (0.8%)		3 (4.0%)	3 (1.2%)		2 (0.6%)	
ダイヤモンド演算子の使用	4 (0.4%)		1 (1.3%)			3 (0.9%)	
配列宣言の変更	1 (0.1%)			1 (0.4%)			
修飾子の並び替え	2 (0.2%)			1 (0.4%)		1 (0.3%)	
空文字列の挿入，削除	1 (0.1%)					1 (0.3%)	
static イニシャライザの挿入，削除	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%)	
整数型リテラルの宣言の変更	1 (0.1%)						1 (2.9%)
定数の使用	1 (0.1%)					1 (0.3%)	
Enum クラスの修飾子の変更	3 (0.3%)					3 (0.9%)	
Enum 定数列挙末尾の セミコロンの挿入，削除	1 (0.1%)						1 (2.9%)

5.4 RQ3：コミットのテスト時間がどの程度削減可能か の調査結果

調査対象としたプロジェクトの Spring Framework Core について，どの程度のテスト時間が削減可能か調査を行った．リポジトリに対して `git log` コマンドを用いて取得された履歴に含まれるコミットを，調査対象期間内の全てのコミット，Java ファイルの変更が行われたコミット，本質的でない変更のみで構成されたコミットに分類した．

表 4 は，分類それぞれについての，コミットの数およびテストにかかった総時間を示している．本質的でない変更のみで構成されたコミットはテストの必要がない．

RQ3 への回答として，**プロジェクト全体で 1.1%，Java のファイル変更が行われたコミットに対しては 10.4% のテスト時間が削減できる**といえる．

表 4　RQ3 の調査結果

	コミット数	テスト時間
全てのコミット	7,195	58,195s
Java のファイル変更が行われたコミット	788	6,575s
本質的でない変更のみで構成されたコミット	70	687s

6 考察

本章では考察を行うにあたり，本質的でない変更を以下の 3 種類に分類した．

- 先行研究でのみ本質的でないと定義された変更
- 本研究でのみ本質的でないと定義された変更
- 先行研究と本研究で共に本質的でないと定義された変更

また，2 章 で述べた先行研究において本質的でない変更と定義された 6 つの変更を以下のように細分化し 12 の変更として扱う．

- 完全限定名から単純名への変更
- 一時変数の導入
- クラス名の変更
- フィールド変数名の変更
- メソッド名の変更
- パラメータ名の変更
- ローカル変数名の変更
- this の挿入，削除
- return 文の付与，削除
- super() の付与，削除
- コメントの変更
- フォーマットの変更

また，RQ2 では 31 の変更が検出されたという結果だったが，それらの変更に加え return 文の付与，super() の付与の 2 種類の変更を検出可能な変更に含めて述べる．return 文の付与，super() の付与の変更は対象プロジェクトからは確認されなかったが，バイトコードが変化しない本質的でない変更である．そのため先行研究と本研究共に本質的でないと定義された変更の種類数に含める．

本研究で検出された本質的でない変更の種類数を先行研究と比較し述べる．

図 2 は先行研究，本研究もしくはその両方から本質的でない変更と定義された変更についてベン図で表している．先行研究でのみ本質的でないと定義された変更は 4 種類であり，本研究でのみ本質的でないと定義された変更は 25 種類である．また，先行研究と本研究共に本質的でないと定義された変更は 8 種類である．

バイトコードによる本質的でない変更の検出は，33 の変更を本質的でないと定義でき，先行研究と比

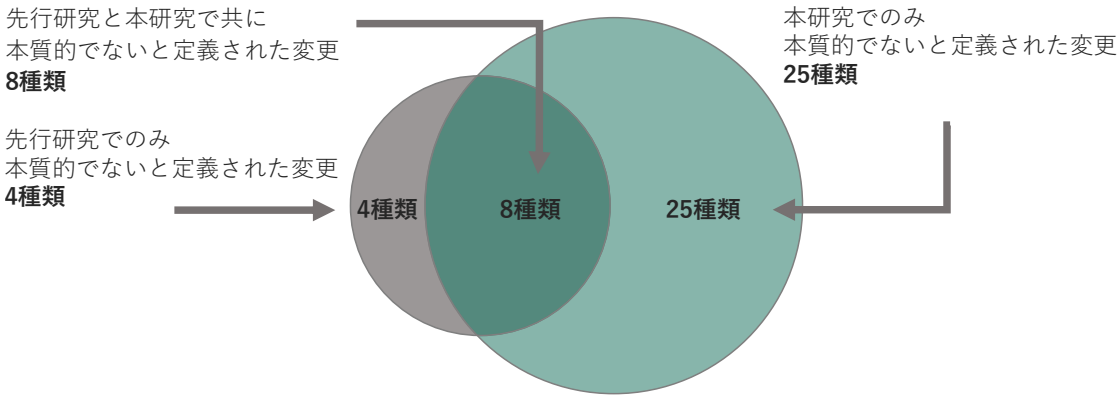

図 2　先行研究と提案手法の結果のベン図

較し，新たに 25 の変更を本質的でないと定義できる結果になった．また，先行研究で挙げられた全ての本質的でない変更を，本研究において本質的でないと定義できなかった．その理由については，9 章で述べる．

さらに，表 5 に，6 つのプロジェクトから調査された本質的でない変更について，発見された変更数の割合の比較を示す．プロジェクト全体では，本質的でない変更全体の 26.4% を本研究で新たに発見された本質的でない変更が占めた．表 3 に示した通り，本質的でない変更の中では，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．条件式の書き換えや，型宣言の変更など，新たに見つかった変更は個数が少なく全体に占める割合も低いという結果になった．プロジェクト別にみると，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 39.6% と多くなった．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 や Apache Tomcat はその割合が低くなった．

表 5　新たに本質的で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	新たに本質的でないと定義された変更
kGenProg	164	65 (39.6%)
Apache Ant	75	15 (20.0%)
Spring Framework Core	245	67 (27.3%)
Hibernate ORM Core	139	37 (26.6%)
Apache Tomcat	324	69 (21.3%)
Apache POI	35	8 (22.9%)
Total	979	258 (26.4%)

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

図 3　完全限定名，単純名間の変更

7　本質的でない変更

本章では，本質的でない変更と検出された変更について述べる．はじめに，先行研究と本研究で共に本質的でないと定義された変更について，次に本研究で新たに発見された本質的でない変更について述べる．最後に，先行研究でのみ本質的でないと定義された変更について述べる．

7.1　先行研究と本研究で共に本質的でないと定義された変更

コメントの挿入，削除，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．これらはコードの視認性や可読性向上のため行われるが．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，対象とした 6 つのプロジェクト全てにおいてコメントの変更が行われたコミットが多数存在し，本質的でない変更の中で 46.7% の最も高い割合を占めることが確認された．また，Apache Ant，Spring Framework Core においては本質的でない変更の半数以上がコメントの変更であった．

フォーマットの変更

フォーマットの変更には，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．コメントに次いで割合が多く，本質的でない変更全体の 21.7% を占めた．

完全限定名，単純名間の変更

例を図 3 に示す．クラス名が記述される際に，クラスが所属するパッケージも含めて記述された名前が完全限定名であり，import 文にパッケージ名が記述され，クラス名のみが記述された名前が単純名である．完全限定名，単純名間の変更はバイトコードに変化を与えない．

ローカル変数名の変更

ローカル変数名の変更は，バイトコードに変化を与えない．. ローカル変数名の変更の中には，拡張 for 文の要素名の変更や，catch 節の引数名の変更も存在した．

パラメータ名の変更

パラメータ名の変更は，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い．

this の挿入，削除

省略可能な this の付与，削除は，バイトコードに変化を与えない．

return 文の挿入，削除

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 文を記述しなくても良い．return 文の付与や削除が行われてもバイトコードは変化しない．調査対象とした 6 つのプロジェクトからは，return 文の付与，削除の変更は確認されなかった．

super() の挿入，削除

super() を記述することで，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行われる．しかし，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く，super() の付与や削除が行われてもバイトコードは変化しない．調査対象とした 6 つのプロジェクトからは，super() の付与，削除の変更は確認されなかった．

7.2 本研究でのみ本質的でないと定義された変更

final 修飾子の挿入，削除

final 修飾子は変数の再代入を無効にするために使用される．final 修飾子が付与され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 4(a) に，catch 節の引数への final 修飾子の付与例を図 4(b) に，拡張 for 文の要素への final 修飾子の付与例を図 4(c) に，List 型のローカル変数への final 修飾子の付与例を図 4(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．調査対象プロジェクトの中では，kGenProg において final 修飾子の付与が多用されていた．

import 文の挿入，削除，並び替え

import 文の並べ換えを行う変更や，import 文を追加する変更，未使用の import 文を削除する変更はバイトコードに変化を与えない，また，6 つの調査対象としたプロジェクト全てにおいて import 文

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

(a) メソッド引数への final 修飾子の挿入

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

(b) catch 節の引数への final 修飾子の挿入

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

(c) 拡張 for 文の要素への final 修飾子の挿入

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

(d) ローカル変数の ArrayList への final 修飾子の挿入

図 4　final 修飾子の挿入

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

図 5　アノテーションの挿入

の挿入，削除，並び替えの変更は行われ，本質的でない変更の全体で 5.7% を占めた．

アノテーションの挿入，削除，編集

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検出されたアノテーションの例を図 5 に載せる．また，図に示す以外にも，バイトコードが変化しないプロジェクト独自のアノテーションも存在した．新たに本質的でない変更と定義された変更の中で最も割合が高く，本質的でない変更の全体で 10.2% を占めた．

インターフェース中のメソッドの public 修飾子の削除

インターフェースにおいて，メソッドは暗黙的に public メソッドになるため，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は変化しない．

```
  for (final GeneratedAST ast :
              variant.getGeneratedSourceCode().getAsts()) {
     final String code = ast.getSourceCode();
     // ast.getProductSourcePath() は ProductSourcePath 型を返す
-    final ProductSourcePath path =
-                (ProductSourcePath) ast.getProductSourcePath();
+    final ProductSourcePath path = ast.getProductSourcePath();
     final int lastLineNumber = countLines(code);
       for (int line = 1; line <= lastLineNumber; line++) {
```

図 6　キャストの削除

```
  public Flux<DataBuffer> encode(...) {
    ...
-   return Flux.from(inputStream).map(bytes -> {
+   // The following (byte[] bytes) lambda signature
+   // declaration is necessary for Eclipse.
+   return Flux.from(inputStream).map((byte[] bytes) -> {
      encodeValue(bytes, bufferFactory, elementType, mimeType));
  }
```

図 7　ラムダ式の引数の型の挿入

キャストの挿入，削除

例を図 6 に示す．省略可能なキャストの挿入や削除はバイトコードに変化を与えない．

ラムダ式の引数の型の挿入，削除

例を図 7 に示す．ラムダ式ではコンパイル時に型推論が行われるため，引数の型が省略可能である．省略可能なラムダ式の引数の型の挿入，削除は，バイトコードに変化を与えない．

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，削除によってバイトコードが変化しない．

ド・モルガンの法則による条件式の書き換え

例を図 8 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 演算子で書き換える変更はバイトコードに変化を与えない．

```
  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 8　ド・モルガンの法則による条件式の書き換え

```
  public class ResolvableType implements Serializable {
    ...
-   return (((this.type instanceof Class &&
-             ((Class<?>) this.type).isArray())) ||
+   return ((this.type instanceof Class &&
+             ((Class<?>) this.type).isArray()) ||
      this.type instanceof GenericArrayType ||
               resolveType().isArray());
  }
```

図 9　括弧の挿入

括弧の挿入，削除

例を図 9 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．省略可能である括弧の挿入，削除はバイトコードに変化を与えない．

ブロックの挿入，削除

if 文や else 文，ラムダ式などにおいて，1 行で命令が記述できる場合，ブロックの省略が可能である．省略可能なブロックの挿入，削除はバイトコードに変化を与えない．ラムダ式中のブロックが削除された例を図 10 に示す．

if 文の統合

例を図 11 に示す．調査対象の Apache Ant において変更が確認された．一般的にネストが深いコードは可読性が低下すると言われる．そのため，if 文の統合が行われることがある．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い．

```
  public static Flux<DataBuffer> write(...) {
    ...
    Flux<DataBuffer> flux = Flux.from(source);
-   return Flux.create(sink -> {
-     flux.subscribe(new
-        AsynchronousFileChannelWriteCompletionHandler(
-          sink, channel, position));
-     });
+   return Flux.create(sink ->
+     flux.subscribe(new
+        AsynchronousFileChannelWriteCompletionHandler(
+          sink, channel, position)));
  }
```

図 10　ラムダ式中のブロックの削除

```
  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 11　if 文の統合

else if 文から if 文への書き換え

例を図 12 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い．

else 文の削除

例を図 13 に示す．不要な else 文が削除され処理が else 文の外に記述されても，バイトコードは変化しない．

```
  public static boolean canConvertElements(...) {
    if (conversionService.canConvert(
          sourceElementType, targetElementType)) {
      return true;
    }
-   else if (sourceElementType.getType().
-              isAssignableFrom(targetElementType.getType())) {
+   if (sourceElementType.getType().
+              isAssignableFrom(targetElementType.getType())) {
      return true;
    }
  }
```

図 12　else if 文から if 文への書き換え

```
  public static boolean canConvertElements(...) {
    ...
    if (sourceElementType.getType().isAssignableFrom(
          targetElementType.getType())) {
      return true;
    }
-   else {
-     return false;
-   }
+   return false;
  }
```

図 13　else 文の削除

総称型の挿入，削除

称型はデータ型名をクラスやメソッドに付与する記述である．型をコンパイラに通知できるため、型の判定を行うことが可能になる．また，型が明示されることにより可読性が向上する．総称型の型宣言の挿入，削除はバイトコードに変化を与えない．

総称型の挿入例を図 14(a) に，総称型のワイルドカードの挿入例を図 14(b) に，総称型から Object 型への変更例を図 14(c) 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い．

ダイヤモンド演算子の使用

例を図 15 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用いてデータ型を省略する変更は，バイトコードに変化を与えない．

```
   } 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 14　総称型の挿入，削除

```
   public Set<SocketWrapperBase<S>> getConnections() {
-    return new HashSet<SocketWrapperBase<S>>(connections.values());
+    return new HashSet<>(connections.values());
   }
```

図 15　ダイヤモンド演算子の使用

修飾子の並び替え

フィールド変数宣言時や，メソッド宣言時の修飾子の順番が並び替えられてもバイトコードが変化しない例が存在した．Spring Framework Core においてフィールド変数宣言時の final 修飾子と static 修飾子の並び替えが行われたが，バイトコードが変化しなかった例を図 16(a) に示す．また，Apache

```
  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 16　修飾子の入れ替え

```
  private static char[] encodeHex(byte[] bytes) {
-  char chars[] = new char[32];
+  char[] chars = new char[32];
   for (int i = 0; i < chars.length; i = i + 2) {
```

図 17　配列宣言の変更

Tomcat においてメソッド宣言時の public 修飾子と synchronized 修飾子の並び替えが行われたが，バイトコードは変化しなかった例を図 16(b) に示す．

配列宣言の変更

例を図 17 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は変化しない．

空文字列の挿入，削除

例を図 18 に示す．省略可能な空文字列の挿入や削除はバイトコードに変化を与えない．

static イニシャライザの挿入，削除

例を図 19 に示す．static イニシャライザの挿入，削除はバイトコードに変化を与えなかった．

```
  for (Entry<String,Integer> mrVersion : mrVersions.entrySet()) {
-   mrMap.put(mrVersion.getKey() ,
-             "META-INF/versions/" +mrVersion.getValue().toString()
-             + ""
-             + "/" + mrVersion.getKey());
+   mrMap.put(mrVersion.getKey() ,
+             "META-INF/versions/" + mrVersion.getValue().toString()
+             + "/" +  mrVersion.getKey());
   }
```

図 18　空文字列の削除

```
-   private static final CharsetCache charsetCache;
+   private static final CharsetCache charsetCache =
+      new CharsetCache();
-    static {
-        charsetCache = new CharsetCache();
-    }
```

図 19　static イニシャライザの削除

```
   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 20　Iterator の使用から拡張 for 文の使用への変更

Iterator の使用から拡張 for 文の使用への変更

例を図 20 に示す．拡張 for 文は Java 5 で導入された記述であり、コレクションの走査が行える．拡張 for 文の内部では Iterator を暗黙的に使用するため，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は，バイトコードに変化を与えない．

オートボクシング，アンボクシング

例を図 21 に示す．オートボクシング，アンボクシングとは Java 5 で導入されたプリミティブ型とラッパークラス間の変換を自動で行う機能である．型変換が自動で行われるため，Integer.valueOf の挿入や，longValue() ，intValue() の挿入による明示的に型変換を行う変更は，バイトコードに変化を与えない．

```
  import org.apache.jasper.compiler.Localizer;
    ...
    //getMessageは，String型，Object型を引数に持つ
    int len = (int)input.length();
-   throw new IOException(Localizer.getMessage(
-                       "jsp.error.readContent", len));
+   throw new IOException(Localizer.getMessage(
+                       "jsp.error.readContent",Integer.valueOf(len)));
```

図 21　オートボクシング，アンボクシング

```
  import org.eclipse.jdt.internal.compiler.impl.CompilerOptions;
    ...
-   settings.put(CompilerOptions.OPTION_Source, "12");
+   settings.put(CompilerOptions.OPTION_Source,
+                              CompilerOptions.VERSION_12);
```

図 22　定数の使用

```
  public class SSLHostConfigCertificate implements Serializable {
-   public enum Type {
+   public static enum Type {
        UNDEFINED,
        RSA(Authentication.RSA),
```

図 23　Enum クラスの修飾子の変更

数値リテラルの型宣言の変更

数値の宣言時に型を指定したい場合，数値末尾に型を明示する．例として，数値末尾に `L` または `l` で long 型リテラル，`F` または `f` で long 型リテラルとなる．これらのアルファベットの大文字と小文字間の変更は，バイトコードに変化を与えない．

定数の使用

例を図 22 に示す．定数の使用は，バイトコードに変化を与えない．

Enum クラスの修飾子の変更

例を図 23 に示す．Enum クラスの宣言において，static 修飾子は省略可能である．Enum クラスの static 修飾子の付与と削除はバイトコードに変化を与えない．

```
  public interface PlaceholderDetails {
    enum PlaceholderSize {
-     quarter, half, full;
+     quarter, half, full
  }
```

図 24　Enum 定数列挙末尾のセミコロンの削除

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

図 25　一時変数の導入

Enum 定数列挙末尾のセミコロンの挿入，削除

例を図 24 に示す．Enum クラスにおいてメソッドを定義しない場合，定数の列挙の末尾のセミコロンは記述してもしなくても良い．省略可能なセミコロンの付与と削除はバイトコードに変化を与えない．

7.3　先行研究でのみ本質的でないと定義された変更

一時変数の導入

例を図 25 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．先行研究では本質的でない変更とされていたが，本研究では一時変数の導入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

先行研究では，名前の変更はすべて本質的でない変更とされていた．しかし，クラス名，フィールド変数名，メソッド名の変更は，バイトコードに変化を与えてしまうため，本研究では本質的な変更と定義された．

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

図 26　メソッド参照におけるメソッド名の変更

8　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．実際の例を図 26 に示す．メソッド参照とは Java 8 で導入された構文であり，**クラス名::メソッド名** と記述し，すでに定義済みのメソッドを引数なしで呼び出す処理を行える．

図 26 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．この変更は，メソッド名が変更され，プログラムの振る舞いが変化する本質的な変更である．しかし，本研究の提案手法では本質的で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際の `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．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．`-v` オプションを用いると本質的でない変更とみなすべきである変数名などの変更が，本質的な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．

9　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，フィールド変数名，メソッド名の変更や，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判定された．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，本研究では本質的な変更と判定される．このように，現時点ではプログラムの振る舞いは保持される変更全てを，提案手法において正しく検出することはできない．

また，本研究では 1 つのコンパイラを用いた調査しか行っていない．そのため，違うコンパイラを使用することにより生成されるバイトコードに違いが生まれ調査結果が変わる可能性がある．

10　おわりに

本研究では，バイトコードの変化の有無によって本質的で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，6 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% 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，31 種類の本質的でない変更を検出可能であり，先行研究と比較し新たに 25 種類の本質的でない変更を検出可能であることを確認した [19]．

今後の課題としては，対象プロジェクトを増加し，新たに本質的で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調査を行うことや，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，無駄となっているテストにかかる時間や費用がどの程度存在するのか，本論文で述べた以上に詳しく調査を行うことが挙げられる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました楠本 真二 教授に心より感謝申し上げます．

本研究の全過程を通し，終始熱心なご指導を頂きました，肥後 芳樹 准教授に深く感謝申し上げます．

本研究に関して，貴重で有益な助言をして頂きました，柗本 真佑 助教に心より感謝申し上げます．

本研究に関して，的確で丁寧なご助言を頂き，また日々の研究室生活を盛り上げて頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年 松本 淳之介 氏に心より感謝申し上げます．

本研究を進めるにあたり，様々な形で励まし，協力を頂きました楠本研究室の皆様に心より感謝申し上げます．

本研究に至るまでに，講義，演習，実験等でお世話になりました，大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[1] GitHub. https://github.com/.

[2] Apache Subversion. https://subversion.apache.org.

[3] B. de Alwis and J. Sillito. Why are software projects moving from centralized to decentralized version control systems? In *Proc. ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pp. 36–39, 2009.

[4] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proc. IEEE International Conference on Program Comprehension*, pp. 35–45, 2006.

[5] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 511–526, 2005.

[6] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us? a taxonomical study of large commits. In *Proc. International Working Conference on Mining Software Repositories*, pp. 99—108, 2008.

[7] S. Rastkar and G. C. Murphy. Why did this code change? In *Proc. International Conference on Software Engineering*, pp. 1193–1196, 2013.

[8] Tobias Baum, Steffen Herbold, and Kurt Schneider. An industrial case study on shrinking code review changesets through remark prediction. *CoRR*, 2018.

[9] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Proc. International Workshop on Principles of Software Evolution*, pp. 13–23, 2003.

[10] 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, 2004.

[11] 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, 2000.

[12] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. International Conference on Software Engineering*, pp. 284–292, 2005.

[13] M. Wen, R. Wu, and S. Cheung. Locus: Locating bugs from software changes. In *Proc. International Conference on Automated Software Engineering*, pp. 262–273, 2016.

[14] 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 Transactions on Software Engineering*,

Vol. 27, No. 1, pp. 1–12, 2001.

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

[16] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. Vol. 26, No. 11, pp. 861–867, 1983.

[17] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proc. International Conference on Automated Software Engineering*, pp. 426–437, 2016.

[18] D. Ståhl and J. Bosch. Industry application of continuous integration modeling: A multiple-case study. In *Proc. International Conference on Software Engineering Companion*, pp. 270–279, 2016.

[19] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proc. International Conference on Software Engineering*, pp. 351–360, 2011.

[20] 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. In *Proc. Asia-Pacific Software Engineering Conference*, pp. 697–698, 2018.

[21] Apache Project. https://www.apache.org.