

製品コードのバグ混入予測に向けた テストコードの影響に関する定量分析

大西 真輝 阿萬 裕久 川原 稔

ソフトウェア製品のソースコード（製品コード）の更新は頻繁に行われる最も基本的な活動であるが、場合によっては製品コードの更新が新たなバグを混入してしまうこともある。それゆえ、コード更新時にバグ混入のリスクを評価する（バグ混入予測を行う）さまざまな手法が提案されてきている。一方、製品コードとともにテストコードも作成・更新されることが多いが、テストコードを主たる対象としたバグ混入予測については、まだ十分には検討されていない。そこで本論文では製品コードではなく、それに対するテストコードに着目したデータ分析を行っている。Apache Camel のデータを対象とした分析の結果、次の知見が得られている：(1) テストコードが製品コードの追加時点で存在するよりも後で追加される方が当該製品コードがバグ修正を経験する可能性が高い。(2) 共進化が発生している変更の方がそうでない場合よりもバグ率が高い。(3) テストコードの規模が大きいほどバグ修正の発生率が低くなるとはいえない。

1 はじめに

ソフトウェア開発において、製品のソースコード（製品コード）の更新は頻繁に行われる作業である。これらの変更は機能追加や仕様変更、品質向上を目的としているが、同時に意図せずバグを混入させてしまうリスクを伴う [4]。そのため、製品コードの更新内容を開発プロジェクトのリポジトリへコミットする時点でバグ混入の有無を即時に予測する“Just-In-Time バグ混入予測”が有用な手段として期待され、数多くの手法が提案されてきている。具体的には、バージョン管理システムや課題管理システム等から得られる各種変更メトリクスを特徴量として用い、機械学習モデル等によってバグ混入の有無を予測するという手法が広く研究されている [5][10]。さらに近年では、深層学習技術の発展に伴い、コミット時のファイル変更内容（差分）をそのまま（メトリクスを介さずに）予測

モデルへの入力として扱い、End-to-End 学習 [7] により予測を行う手法も提案されている [3]。

これまで提案・活用されている Just-In-Time バグ混入予測（以降、特に断らない限り、“バグ混入予測”と呼ぶ）手法では製品コードに対する変更の特徴量（変更メトリクス）やコード差分等が予測のための主たる情報源として利用されているが、筆者らの知る限り、テストコードに着目した検討は十分に行われていない。テストコードは被テスト対象である製品コードの動作検証を担い、製品コードそのものの特徴を表しているわけではないが、製品コードの品質向上に直結する重要な役割を果たす。それゆえ、製品コードの変更におけるバグ混入予測でも、当該製品コードをテストしているテストコードの作成時期であったり、製品コードとテストコード変更の因果関係、テストコードそのものの規模といった情報もまた注目に値するのではないかと筆者らは考える。

そこで、本論文では製品コードに対して“テストコードがどう影響を与えるのか？”という観点から以下に示す 3 つの研究課題（RQ）を掲げ、これらの下でデータ分析と結果に対する考察を行う。

Masaki Onishi, 愛媛大学大学院理工学研究科, Graduate School of Science and Engineering, Ehime University.

Hirohisa Aman and Minoru Kawahara, 愛媛大学総合情報メディアセンター, Center for Information Technology, Ehime University.

RQ1: テストコードの作成時期によって製品コードのバグ修正経験率は変化するか？

RQ2: 製品コードとテストコードの共進化はバグ混入防止に寄与するか？

RQ3: テストコードの規模は製品コードのバグ修正発生率に影響するか？

その結果、以下の知見が得られた：

1. テストコードが製品コード追加後に導入された場合は、初期から存在していた場合よりも、製品コードのバグ修正経験率が高い。
2. 共進化が発生している変更の方がそうでない場合よりもバグ率が高い。
3. テストコードの規模が大きいほどバグ修正発生率が低くなるとはいえない。

これらの知見は、テストコードの有無や質、更新の仕方といった情報が、製品コードの変更におけるバグ混入のリスクを捉える上で有効な指標となり得ることを示唆している。以下では、これらの知見を得るに至ったデータ分析の内容と結果について説明する。

本論文の構成は以下の通りである。2節で関連研究と研究課題を述べ、3節でテストコードの影響を分析する方法について説明する。そして、4節でデータ分析を行い、得られた結果に対して考察を行う。最後に、5節で本論文のまとめと今後の課題を述べる。

2 関連研究と研究課題

本節では、2.1節で関連研究としてバグ混入予測についてふれる。そして、2.2節で本研究の研究動機ならびに研究課題を述べる。

2.1 関連研究

従来より、バグ混入予測についてはさまざまな手法が提案されており、そこでは変更メトリクスが一般に用いられてきた[5]。つまり、メトリクスによってコミットの特徴をソースコードの変更内容やそれを行った人物の経験といったさまざまな視点から定量的にとらえ、実際にバグが混入したコミットの特徴を機械学習することで、新たなコミットが与えられた際にそれがバグ混入になっているかどうかを予測するというものである。しかしながら、変更メトリクスに基づいた

バグ混入予測では、特徴の抽出が使用するメトリクスの視点に限定されてしまうという課題がある。即ち、メトリクスの設計や選定が一種のバイアスとなり、対象のソフトウェアや開発プロジェクト次第では適切なバグ混入予測を実現できない恐れがある。

そこで近年では、深層学習モデルを用いた End-to-End 学習によるバグ予測手法が提案されている[3]。ここでいう深層学習モデルを用いた End-to-End 学習とは、入力データが与えられてから結果を出力するまでの処理を複数の層を備えた1つの大きなニューラルネットワークに置き換えて学習を行うものをいう[7]。つまり、End-to-End 学習ではコミットからの特徴量抽出も学習の過程に含めることになり、“コミットの特徴抽出からバグ混入予測まで”の一連の流れがすべて深層学習に集約される。したがって、上述したメトリクスの設計や選定によるバイアスを排除でき、なおかつ予測モデルに入力するための特徴データの前処理も不要になるという利点がある。この利点を活かした研究として、例えば、変更のあったソースコード片（変更された行とその前後の行のみを切り出したもの）からバグ混入を予測するといった手法がある[6]。一方、その種のモデルを使ったバグ混入予測における欠点として、深層学習モデルの構造が大規模で複雑になりがちであったり、学習に大量のデータや時間が必要となったりすることが挙げられる。即ち、バグ混入予測過程の理解が難しい、バグ混入コミットと予測された理由の説明が難しい、学習コストが高いといった問題がある。このようにバグ混入予測を行うにあたって End-to-End 学習が常に優れているとは断言できないが、予測精度という点では有用なことが多いことから、現在活発に研究されている手法の1つとなっている。

2.2 研究動機と研究課題

従来のバグ混入予測手法では、変更されたソースコードのうち製品コードを主たる入力データとしてモデルを構築し、その結果からバグ混入の有無を予測する手法が一般的である。しかし、テストコードの変更にもバグ混入予測上の有益な情報が含まれている可能性が高いと筆者は考える。

その1つとして“共進化”と呼ばれる製品コードとテストコードの変更の因果関係を示す概念がある。共進化とは、テストコードの変更と製品コードの変更が同一コミットで発生する、またはテストコードが製品コードの変更後に短時間で発生することである[11]。しかし、Sunらは実証実験を通して従来の共進化の定義では両者の変更内容に意味的関連がない場合があることを示した。そして、共進化の発生を自動で判定する新たな手法としてCHOSENを提案した[9]。この手法を用いることで共進化の判定を高精度で行うことができる。

テストコードがリポジトリ上に存在するが製品コードが変更されていく中で共進化しない場合や、製品コードに対してそれをテストするテストコードがリポジトリ上に見当たらない場合について考える。それらにはさまざまな理由があると思われる、当該製品コードの内容や変更が単純なものとなっていてわざわざテストコードを作成したり変更するまでもなかったケースもあれば、テストが十分に行われていなかったという深刻なケースもあるかもしれない。前者のような理由であれば、製品コードに対するバグ混入予測には直接的には影響しないことになるが、もしも後者のような理由であれば、バグ混入の見逃しを誘発することになるため、バグ混入予測の成否に及ぼす影響は小さくない。

上述したようにテストコードに着目することもまたバグ混入予測においては有用となる可能性があるが、筆者らの知る限り、テストコードを主たる対象としたかたちでのバグ混入事例の分析はまだ十分に行われていない。これが本論文の研究動機である。これに基づき、本論文では以下に示す3つの研究課題(RQ)を掲げ、これらの下でデータ分析を実施する。

RQ1： テストコードの作成時期によって製品コードのバグ修正経験率は変化するか？

製品コードの追加時にテストコードが存在する場合、当該製品コードはテストを意識した設計がなされていることからバグになりにくいと考える。しかしながら、テストコードが製品コードの追加時に見当たらないような場合も現実には少なくない。

それゆえ、RQ1はテストコードの作成時期が与える影響に関する基本的な問いである。仮に、製品コードの追加時点でテストコードが存在することがバグ防止に重要であると明らかになれば、テストコードの追加が遅れていることに対して警告を出すことが、バグの混入を防ぐうえで有効であるといえる。

なお、本論文ではテストコードが製品コードの追加時点ですでに存在している場合を“同時追加型”、テストコードが製品コード追加後に導入された場合を“事後追加型”と定義する。さらに、同時追加型にはテストコードが製品コードの追加以前に存在する場合も含める。

RQ2： 製品コードとテストコードの共進化はバグ混入防止に寄与するか？

製品コードが更新される際、バグの修正や機能の追加・変更が行われていた場合であれば、更新内容に対応したテストコードの追加・変更も必要となる。^{†1}

RQ2は、そのような共進化の影響に関する問いである。共進化の重要性が確認されれば、コミット時にテストコードの更新漏れを警告することがバグ混入防止に有効であるといえる。一方、そうでない傾向が見られた場合は、それに応じた対応策を講じなければならないことになる。

RQ3： テストコードの規模は製品コードのバグ修正発生率に影響するか？

テストコードの規模が大きいということは、それだけ製品コードに対する検証が十分に行われている可能性が高く、結果としてバグ防止に寄与していると考えられる。

そのため、RQ3はテストコードの規模が製品コードのバグ修正の発生に与える影響を明らかにする狙いがある。テストコードの規模の大きいほどバグ修正の発生率が低下していれば、バグ混入予測に有効なメトリクスとなる可能性がある。

^{†1} バグ修正の場合は機能の変更は起こっていないが、それまでに存在していたテストコードでは当該バグを見逃していたわけであり、それゆえテストコードはそのままでは不適切である。

3 テストコードの影響分析方法

本節では、前節で述べた研究課題の下で行うデータ分析のための準備を以下の流れで説明する。データセットの都合上、ここでは Java プロジェクトを対象とした説明を行う。なお、本研究では製品コードとテストコードの対応付けを機械的に行うため、静的解析のみで特定可能な単体テストコードに着目する。

1. 製品コードとテストコードのデータ取得
2. コミットで変更されたコード情報を取得
3. CHOSEN を用いた共進化の判定

3.1 製品コードとテストコードのデータ取得

本研究は先行研究でラベル付けされたデータセットを使用する。これに伴い、データセットに存在する最新コミットで製品コードとそれに対応する単体テストコードの変更履歴を取得する。例として、Apache Camel プロジェクトの 14904a というコミットに存在する `camel-spring/src/main/java/org/apache/camel/spring/CamelContextFactoryBean.java` および `camel-spring/src/test/java/org/apache/camel/spring/CamelContextFactoryBeanTest.java` という 2 つのソースコードを挙げる。

(1a) リポジトリのチェックアウト

リポジトリをデータセットに存在する最新のコミットに切り替える。

(2a) Java ソースコードのパスを取得

当該コミット時点でリポジトリに含まれるすべての Java ソースコードのパスを取得する。

(3a) ソースコードを製品コードとテストコードに分類

取得したソースコードがテストコードであるとは、次のいずれかの条件を満たすことである。

- ファイルパスに `/test/` が含まれる
- ファイル名の接頭語または接尾語に `Test` が付いている

これにより、テストコードと判定されなかったソースコードを製品コードとする。例えば、`camel-spring/src/test/java/org/apache/camel/spring/CamelContextFactoryBeanTest.java` は、ファイルパスに `/test/`、ファイル

名 `CamelContextFactoryBean` の接頭語および接尾語に `Test` が含まれないため製品コードと判定される。一方、`camel-spring/src/test/java/org/apache/camel/spring/CamelContextFactoryBeanTest.java` は、`/test/` を含むパスにあり、ファイル名も `Test` で終わっていることから、テストコードと判定される。

(4a) 製品コードと単体テストコードのペアを取得

手順 (2a) によってソースコードは製品コード群とテストコード群に分類される。そして、テストコード群に含まれるテストコードをもとにそのペアとなる製品コードを製品コード群から取得する。具体的には、手順 (3a) のソースコード分類で用いた 2 つの条件を両方とも満たす場合、単体テストコードと判定し、そのパスに対して以下の処理を行う。ただし、製品コードに対してテストコードが 1 つ存在するものを取得する。

● 完全一致

単体テストコードのファイルパスに含まれる `/src/test/` を `/src/main/` に置換する。さらに、ファイル名の接頭語および接尾語から `Test` を除去したパスを製品コード群の中から探す。これは、`camel-spring/src/test/java/org/apache/camel/spring/CamelContextFactoryBeanTest.java` から `camel-spring/src/main/java/org/apache/camel/spring/CamelContextFactoryBean.java` を探すことを意味する。

● 部分一致 (完全一致で該当しない場合)

`/src/main/` の前後に `**` (ワイルドカード) を挿入し、テストコードのファイル名の接頭語および接尾語から `Test` を除去したファイル名を最後尾に加えたパスを製品コード群から探す。例で示すと、`camel-spring/src/test/java/org/apache/camel/spring/CamelContextFactoryBeanTest.java` に対して処理を行った結果、`**/src/main/**/CamelContextFactoryBean.java` を製品コード群から探すことになる。

(5a) 製品コードとテストコードの変更履歴を取得

手順 (4a) で取得した単体テストコードとそのペアである製品コードそれぞれにおいて、最新コミットから遡ることで変更履歴を取得していく。このとき、`git log --first-parent --name-status` によって取得したログ履歴を使用する。

3.2 コミットで変更されたコード情報を取得

本研究において、変更内容がコメントや空白類のみであるものは変更とはみなさない。そのため、3.1 節の手順 (5a) で取得したソースコードの変更がコメントや空白類のみであるか否かの情報が必要となる。本節では、プロジェクトリポジトリの main ブランチでコミットされたものにおいてその情報を取得する手順を説明する。ここでも、Apache Camel プロジェクトの 14904a を例として用いる。

(1b) 変更されたソースコードを取得

変更されたファイルからソースコードのみを取得するために、ソースコードであるかを拡張子で判断する。例えば、拡張子が java であれば、それらのファイルは Java のソースコードであると考えられることから、この基準でもってソースコードのみを収集する。例に挙げたコミットでは、5 つのファイル (表 1) が変更されていた。これらの中で `camel_context_factory_bean_test.xml` を除いた 4 つのファイルがソースコードとなる。

(2b) ソースコードを製品コードとテストコードに分類

3.1 節の手順 (3a) で述べた方法で手順 (1b) で収集したソースコードを製品コードとテストコードに分類する。例の 4 つのソースコードは、`CamelContext.java`, `DefaultCamelContext.java`, `CamelContextFactoryBean.java` が製品コード、`CamelContextFactoryBeanTest.java` がテストコードに分類される。

(3b) 変更内容がコメントや空白類のみを除外

手順 (1b) で収集したコードの変更からコメントや空白類のみの変更を除外する手順を以下に示す。

- (a) コミット前後でのソースコードの内容を `git show` を使って取得

表 1 変更されたファイル一覧

種別	パス
M	camel-core/src/main/ java/org/apache/camel/ CamelContext.java
M	camel-core/src/main/ java/org/apache/camel/impl/ DefaultCamelContext.java
M	camel-spring/src/main/ java/org/apache/camel/spring/ CamelContextFactoryBean.java
R	camel-spring/src/test/ java/org/apache/camel/spring/ SpringClassPathRouteLoaderTest.java ↓ camel-spring/src/test/ java/org/apache/camel/spring/ CamelContextFactoryBeanTest.java
R	camel-spring/src/test/ resources/org/apache/camel/spring/ findRouteBuildersOnClassPath.xml ↓ camel-spring/src/test/ resources/org/apache/camel/spring/ camel_context_factory_bean_test.xml

- (b) 取得した内容からコメントを削除したファイルを作成
- (c) `GumTreeDiff` [1] (パラメータはデフォルト) を用いてファイル間での AST 差分を取得
この手順によって AST 差分が無いと判定された場合は変更とみなさない。例に挙げた 4 つのソースコードは全て AST 差分が存在したため全て変更が発生したとみなす。

3.3 CHOSEN を用いた共進化の判定

3.1 節で取得した製品コードとテストコードの変更履歴、および 3.2 節で得られたコミットごとに変更されたコードの情報を用いて、CHOSEN による共進化の判定を行う。

CHOSEN は製品コード変更とテストコード変更のペアを“共進化あり (Positive)”または“共進化なし (Negative)”のいずれかに分類するために、2 段階の処理を行う。1 段階では、製品コード変更が発生してから 12 時間以内に対応するテストコードの変更が行われた場合に Positive、それ以外は Negative

表 2 CHOSEN の第 2 段階と戦略

戦略	内容
戦略 1	変更タイプが ADD / DELETE などの非修正型であり、かつ両コミットの間には他の製品 / テストコード変更が存在しない場合は Positive と判定
戦略 2	製品コード変更コミットとテストコード変更コミットの間には別の製品コード変更が存在する場合は Negative と判定
戦略 3	変更内容が import 文のみであり、製品コード側とテストコード側の差分に共通の import が含まれていない場合は Negative と判定
戦略 4	コメント・フォーマット・import を除いた本体部分（クラス本文）の差分トークンに共通項がない場合は Negative と判定
戦略 5	テストコード側の変更がアノテーションの追加、修飾子の変更、または単なるリファクタリングにとどまる場合は Negative と判定

と判定する。続く 2 段階では、表 2 で示す 5 つの戦略に基づき、第 1 段階目でのラベル付けを補正する。なお、各戦略の詳細については、Sun らの研究 [9] を参照されたい。

4 データ分析

本節では、製品コードに対して単体テストコードが与える影響に着目し、それらと製品コードのバグ修正およびバグ混入の関係について実施したデータ分析について報告する。

4.1 目的

本分析の目的は、次の研究課題に対して、データ分析を通じて回答することである。

RQ1: テストコードの作成時期によって製品コードのバグ修正経験率は変化するか？

RQ2: 製品コードとテストコードの共進化はバグ混入防止に寄与するか？

RQ3: テストコードの規模は製品コードのバグ修正発生率に影響するか？

4.2 対象

本分析では、Java を主要開発言語とする Apache Camel プロジェクトを分析対象とする。データ分析

表 3 分析に使用するコミットデータの概要

データ収集期間	コミット		
	総数	バグ混入数	バグ率
2007/03 ~ 2019/11	20,097	2,926	14.56%

にあたっては、各コミットがバグ混入またはバグ修正であるか否かを判定するためのラベルが必要になる。そのため本研究では、先行研究でコミット単位のラベル付けが行われた公開データセット^{†2}を利用する。ただし、分析にあたって前処理を施している。前処理の詳細については、付録 A.1 を参照されたい。分析に使用するコミットデータ（前処理後）の概要を表 3 に示す。

4.3 手順

データ分析の手順を以下に示す。

1. RQ1 に関するデータ分析

(1c) 製品コードとテストコードのデータ取得

3.1 節で説明した手順に従ってデータセットに存在する最新コミットから製品コードとテストコードのペアおよびそれぞれの変更履歴を収集する。

(2c) コミットで変更されたコード情報を取得

コミットごとにコメントや空白類を除いたコードの変更情報を 3.2 節で説明した手順で取得する。

(3c) 製品コードの分類

製品コードを同時追加型と事後追加型に分類する。具体的には、手順 (1c) の変更履歴からテストコードが製品コードの追加時点で存在するかを調べ、存在する場合は同時追加型へ、そうでない場合は事後追加型へ分類する。

(4c) 変更履歴の抽出

同時追加型および事後追加型に分類された製品コードに対して、手順 (1c) で取得した変更履歴から手順 (2c) で取得したコミットで変更されたコード情報に存在するものを抽出する。

^{†2} <https://github.com/snaraya7/early-defect-prediction-icse21>

- (5c) **バグ修正経験率の算出**
各製品コードについて、その変更履歴の中に“バグ修正に該当するコミット”が1回以上含まれているかを判定する。このとき、1回でもバグ修正が確認された製品コードは“バグ修正を経験した”とみなす。
判定後、同時追加型と事後追加型それぞれでバグ修正を経験した製品コード数を分類内の全製品コード数で割り、バグ修正経験率を算出することで両者を比較する。

2. RQ2 に関するデータ分析

- (1d) **製品コードとテストコードのデータ取得**
手順 (1c) と同様に、製品コードとテストコードのデータを取得する。
- (2d) **コミットで変更されたコードの情報を取得**
手順 (2c) と同様に、コミットごとにコメントや空白類を除いたコードの変更情報を取得する。
- (3d) **CHOSEN を用いて共進化のデータを取得**
3.3 節で説明した共進化の判定方法を用いて、手順 (1d) で取得した製品コードの変更履歴に共進化が発生したか否かのラベルを付ける。
- (4d) **製品コードの分類**
手順 (3c) と同様にして、製品コードを同時追加型と事後追加型に分類する。
- (5d) **変更履歴の絞り込み**
同時追加型と事後追加型に含まれる製品コードの変更履歴を手順 (4c) 同様に処理し、バグ修正が発生している変更に限定する。さらに、製品コードが初めて追加された変更履歴も除外する。理由は、製品コードの初回追加時は設計や実装の不確実性が高く、そもそもバグを含みやすいことからバグ混入予測における共進化の有無の影響を評価する上で、ノイズとなる可能性があるためである。
- (6d) **バグ率の算出**
同時追加型と事後追加型に含まれる製品コードの変更を共進化が発生しているか否かでさらに分類する。これによって分類された 4

つのグループそれぞれで変更がバグ混入コミットに含まれていた割合（以降，“バグ率”と呼ぶ）を算出し、グループ間で比較する。

3. RQ3 に関するデータ分析

- (1e) **製品コードとテストコードのデータ取得**
手順 (1c) と同様に製品コードとテストコードのデータを取得する。
- (2e) **テストコード規模の収集**
手順 (1e) で取得した変更履歴においてテストコードの規模を収集する。ただし、規模メトリクスとしては実行可能行数を用い、測定には Lizard^{†3} を使用する。
- (3e) **コミットで変更されたコード情報を取得**
手順 (2c) と同様に、コミットごとのコメントや空白類を除いたソースコードの変更情報を取得する。
- (4e) **テストコードの規模情報を収集**
手順 (1e) で取得した変更履歴を手順 (3e) に含まれる変更に絞る。そして、製品コードの変更が発生する 1 つ前のコミットのテストコードの規模を収集する。
- (5e) **変更履歴の分類**
最新コミットにおいてテストコードの規模の最小値と最大値を取得する。そして、手順 (4e) で収集したデータを取得した最小値から最大値の範囲で 10 分割したグループに分類する。
- (6e) **バグ修正率の算出**
手順 (5e) を経て分類された 10 のグループごとにバグ修正が発生した割合（以降，“バグ修正発生率”と呼ぶ）を算出し、グループ間で比較する。

4.4 結果

Apache Camel プロジェクトから収集された単体テストコードが存在する製品コードのデータに対し、4.3 節に示した手順で各 RQ についてデータ分析を行った結果を示す。

^{†3} <https://github.com/terryyin/lizard>

4.4.1 RQ1 に関する結果

データセット上の最新コミットにおいて、単体テストコードが存在する製品コードを同時追加型と事後追加型で分類し、製品コードのバグ修正経験率を算出した。その結果を表4に示す。

表4から分かるように、同時追加型よりも事後追加型の方がバグ修正経験率は高いという結果が得られた。製品コード全体でのバグ修正経験率が44.22%に対して、同時追加型の製品コードでは26.33%、事後追加型は64.96%となっていた。即ち、事後追加型の方がバグ修正を経験する可能性が高いという傾向が見られた。

4.4.2 RQ2 に関する結果

RQ2に関する分析として、RQ1と同様に製品コードを分類した後、さらに共進化が発生したものとそうでないもので分類した。そして、各分類ごとにバグ率を算出し、比較した結果を表5に示す。

結果として、同時追加型では共進化が発生していない（共進化なし）場合は23.35%であるのに対して、発生している（共進化あり）場合は37.61%となっており、14.26%ほど高くなることがわかった。また、事後追加型の場合においても“共進化なし”が28.73%、“共進化あり”が31.69%となっており、共進化が発生している場合においてバグ率が2.96%ほど高くなることを確認した。即ち、同時追加型および事後追加型の両方において共進化が発生している方がバグ率が上昇する傾向が見られた。

表4 テストコードの作成時期の違いによるバグ修正経験率

分類	製品コード数	バグ修正数	経験率
全体	1,217	456	44.22%
同時追加型	866	228	26.33%
事後追加型	351	228	64.96%

表5 共進化の有無によるバグ率の比較

分類	共進化	総数	バグ数	バグ率
全体		1,455	405	27.84%
同時追加型	なし	501	117	23.35%
	あり	109	41	37.61%
事後追加型	なし	703	202	28.73%
	あり	142	45	31.69%

表6 テストコードの規模に基づいたバグ修正発生率の推移

規模の範囲	総数	バグ修正数	発生率
6-165	9,626	1,107	11.50%
165-324	1,394	216	15.49%
324-483	473	62	13.11%
483-642	86	5	5.81%
642-801	109	20	18.35%
801-960	29	3	10.34%
960-1,119	0	0	—
1,119-1,278	0	0	—
1,278-1,437	0	0	—
1,437-1,596	2	0	0.00%

4.4.3 RQ3 に関する結果

RQ3に関する分析として、製品コードの変更が起きる1つ前の変更のテストコードの規模を収集し、これらを最新コミットで取得したテストコードの規模の最小値と最大値の範囲で10グループへ分類した。最後に、それぞれのグループでバグ修正発生率を算出した。その結果を表6に示す。

表6を見ると、中規模帯（165-324行）では15.49%とやや高い発生率を示す一方で、大規模帯（483-642行）では5.81%と急激に低下し、その後再び上昇するといった変動パターンが見られる。つまり、テストコードの規模が大きいほどバグ修正発生率が単調に減少するような傾向は無いといえる。

4.5 考察

以下では、4.4節で示した各RQの結果について考察を述べるとともに、それぞれの回答を述べる。

4.5.1 RQ1に関する考察

RQ1（テストコードの作成時期によって製品コードのバグ修正経験率は変化するのか？）という問いの下、データ分析を実施した結果、表4に示すように同時追加型よりも事後追加型の方がバグ修正経験率が高いという傾向が見られた。この傾向は、テストコードの作成時期という観点において事後追加型よりも同時追加型の方が製品コードの品質管理という面で適していることを示す。

傾向の裏付けとして、製品コードの規模と複雑度を製品コードが追加した初期コミットと最新コミットで測定した結果に加え、製品コードの変更回数も用いて

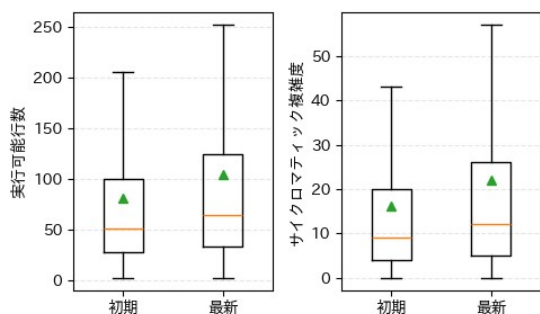


図 1 同時追加型における製品コードの初期と最新コミットのメトリクス比較

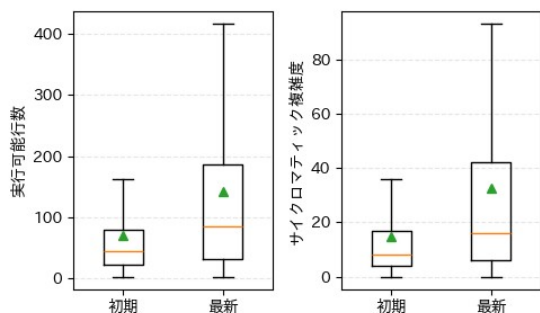


図 2 事後追加型における製品コードの初期と最新コミットのメトリクス比較

追加分析を行った。ただし、規模メトリクスとしては実行可能行数を、複雑度メトリクスとしてはマイクロマティック複雑度をそれぞれ使用している。なお、測定には 4.3 節 で述べた Lizard を使用した。メトリクスの比較結果を図 1、図 2 及び図 3 に示す。

この結果から、同時追加型では事後追加型と比較して初期から最新コミットまで規模や複雑さの変化が小さいだけでなく、変更回数も少ないということを確認した。つまり、テストコードの作成時期が同時追加型の場合は開発初期からテストコードが導入されることで、品質管理が一貫して行われ、設計の安定化や変更の抑制につながったためバグ修正経験率が低くなったといえる。

以上より、RQ1（テストコードの作成時期によって製品コードのバグ修正経験率は変化するか？）に対して以下の通り回答する。

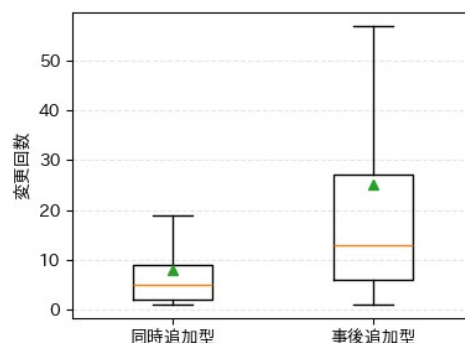


図 3 同時追加型と事後追加型における製品コードの変更回数の比較

テストコードの作成時期によって製品コードのバグ修正経験率が変化する傾向が見られた。具体的には、同時追加型の方が事後追加型より製品コードのバグ修正経験率が低くなることが確認された。追加分析を行った結果、事後追加型の方が同時追加型よりも規模や複雑さの変化が大きく、変更回数が多い傾向にあった。つまり、製品コードが追加された時点でテストコードが存在する方が品質管理が一貫して行われたことによりバグ修正経験率が低くなったと考える。

4.5.2 RQ2 に関する考察

RQ2（製品コードとテストコードの共進化はバグ混入防止に寄与するか？）という問いの下、データ分析を実施した結果、表 5 に示したように、共進化が発生している方が同時追加型では 14.26%、事後追加型では 2.96% バグ率が上昇することを確認した。直感的には、製品コードとテストコードの変更が共進化している方が最も適切かつ慎重に変更管理がなされていたため、バグ混入となる可能性は低くなるのではないかと筆者らは考えていたが、興味深いことにこの仮説とは真逆の結果がデータ分析によって得られた。

この理由について考察するため、それぞれのグループに含まれる製品コードの変更時の実行可能行数やマイクロマティック複雑度を測定し、比較を行った。その結果を図 4 及び図 5 に示す。

結果として、同時追加型および事後追加型はともに共進化が発生していない（共進化なし）場合と比較して、発生している（共進化あり）場合の方が実行可能

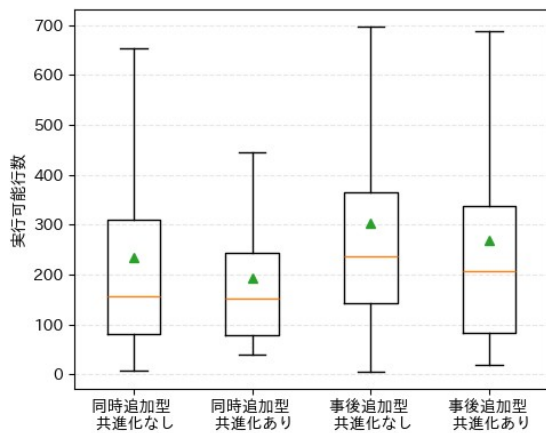


図 4 製品コード変更時の実行可能行数の比較

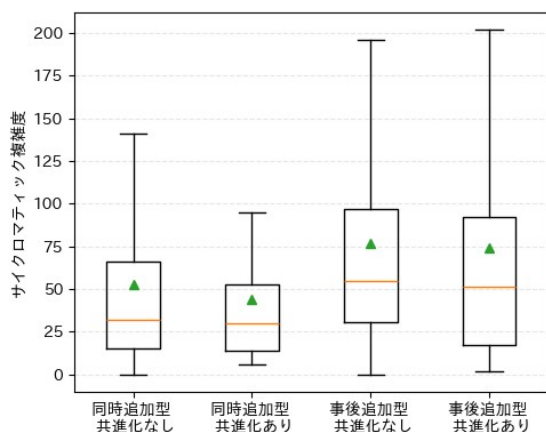


図 5 製品コード変更時のサイクロマティック複雑度の比較

行数およびサイクロマティック複雑度が小さくなる傾向となった。即ち、共進化が発生している場合のバグ率の高さは、製品コードの規模や複雑さが交絡因子となっているとはいえない。別の要因としてテストケースも一緒に変更しているが、残念ながら十分なテストができていない可能性であったり、テストスメルが存在していたりする可能性[2]も挙げられるが、それについてはまだ明らかにはできていない。この点に関するさらなる分析は今後の課題としたい。

以上より、RQ2（製品コードとテストコードの共進化はバグ混入防止に寄与するか？）に対して以下の通り回答する。

今回のデータ分析の範囲では、製品コードとテストコードが共進化することがバグ混入防止につながるという傾向は見られなかった。むしろ逆の兆候にあり、共進化している方がバグ混入となる可能性が高い傾向にあった。なお、これに関しては同時追加型および事後追加型はともに実行可能行数およびサイクロマティック複雑度が要因であるとはいえなかった。別の要因として、テストの不十分性やテストスメルの存在が要因となっている可能性もあり、さらなる分析が必要であると考えられる。

4.5.3 RQ3 に関する考察

RQ3（テストコードの規模は製品コードのバグ修正発生率に影響するのか？）という問いの下、データ分析を実施した結果、表 6 に示すようにテストコードの規模とバグ修正発生率の間に単調性は見られなかった。

この理由について模索するため、製品コード規模とテストコード規模の間の相関を調べた。相関係数には Pearson の相関係数と Spearman の相関係数の 2 種類を使用した。その結果を表 7 に示す。

結果として、Pearson の相関係数は 0.291, Spearman の相関係数は 0.354 となり、製品コード規模とテストコード規模の間には弱い正の相関しか見られなかった。すなわち、両者の規模が連動しているとは言い難い結果となった。

このことから、今後は単にテストコードの行数だけではなく、テストカバレッジやテストスメルの有無といった他の側面にも着目したさらなる分析が必要であると考えられる。

以上より、RQ3（テストコードの規模は製品コードのバグ修正発生率に影響するのか？）に対して以下の通り回答する。

テストコードの規模が製品コードのバグ修正発生率に影響するという傾向は確認できなかった。つまり、単にテストコードが多く書かれているだけでは、

表 7 製品コードとテストコードの規模の相関係数

指標	相関係数
Pearson	0.291
Spearman	0.354

それがバグの見逃し防止につながるとは言い難い結果となった。それゆえ、バグ混入予測技術の向上に向けては、テストカバレッジやテストスメルの有無といった、テストコードの品質にも着目したさらなる分析が必要である。

5 まとめと今後の課題

本論文では、バグ混入予測に対する新たな視点として、製品コードではなくそれをテストするテストコードに着目した。そして、テストコードの作成時期の違いによる影響、共進化発生の有無、テストコードの規模が与える影響という3つの視点からデータ分析を行った。

その結果、以下の傾向が確認された：(1) 同時追加型よりも事後追加型の方が当該製品コードがバグ修正を経験する割合が高い。(2) 共進化が発生している方が発生していない場合と比較してバグ混入になりやすい。(3) テストコードの規模が大きくなるほどバグ修正の発生率が低くなるという傾向は見られない。

追加調査の結果、(1)に関しては事後追加型から同時追加型と比較して、製品コードの規模や複雑さの変化が大きく、変更回数が多い傾向が見て取れた。それゆえ、同時追加型の場合は品質管理が十分に行われ、バグ修正経験率が低くなったといえる。(2)に関しては、同時追加型および事後追加型はともに共進化が発生している場合の方がそうでない場合と比較して、実行可能行数およびサイクロマティック複雑度が小さいことがわかった。つまり、製品コードの規模や複雑さが交絡因子として働いているわけではなかった。別の要因として、テストの不十分性やテストスメルの存在が要因となっている可能性もあり、さらなる分析が必要であると考えた。(3)に関しては、製品コードとテストコードの規模に弱い正の相関しか見られなかった。即ち、両者の規模が連動しているとは言い難い結果となった。そのため、テストカバレッジやテストスメルの存在などの他の側面に着目したさらなる分析が必要であると考えた。

以上の分析から、本論文の貢献はテストコードに関する以下の点がバグ混入予測の新たな警告要素として有用である可能性を示唆した点にある：

- 事後追加型の製品コードはバグ修正を経験する可能性が高いため、特に注視する必要があること。
- 製品コードとテストコードが共進化している場合こそ、警戒を強めるべきであること。
- テストコードの規模のみではバグの抑止効果を測れないため、他の側面にも注目し依存しないようにすること。

今後はテストコードの設計品質や保守状況を示す情報を組み込んだモデルを構築することで、バグ混入予測の精度向上が期待される。

謝辞 本研究の一部は JSPS 科研費 23K11382, 25K15059, 25K15062 の助成を受けたものです。

参考文献

- [1] Falleri, J. and Martinez, M.: Fine-grained, accurate and scalable source differencing, *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, ACM, 2024, pp. 231:1–231:12.
- [2] Fushihara, Y., Aman, H., Amasaki, S., Yokogawa, T., and Kawahara, M.: Fault-Proneness of Python Programs Tested By Smelled Test Code, *Proc. 50th Euromicro Conf. Softw. Eng. & Adv. App.*, August 2024, pp. 373–378.
- [3] Hoang, T., Khanh Dam, H., Kamei, Y., Lo, D., and Ubayashi, N.: DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction, *Proc. 16th Int'l Conf. Mining Softw. Repo.*, May 2019, pp. 34–45.
- [4] Jones, C.: *Applied Software Measurement: Global Analysis of Productivity and Quality*, McGraw-Hill, New York, 3rd edition, 2008.
- [5] Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance, *IEEE Trans. Softw. Eng.*, Vol. 39, No. 6(2013), pp. 757–773.
- [6] 近藤将成, 森啓太, 水野修, 崔銀恵: 深層学習による不具合混入コミットの予測と評価, ソフトウェアエンジニアリングシンポジウム 2017 論文集, Vol. 2017, August 2017, pp. 35–45.
- [7] 西田京介, 井島勇祐, 田島周平: エンドツーエンド深層学習のフロンティア, 電子情報通信学会誌, Vol. 101, No. 9(2018), pp. 920–925.
- [8] Sujay, R.: *Python Testing with Selenium: Learn to Implement Different Testing Techniques Using the Selenium WebDriver*, Apress, New York, 1st edition, 2020.
- [9] Sun, W., Yan, M., Liu, Z., Xia, X., Lei, Y., and Lo, D.: Revisiting the Identification of the Co-evolution of Production and Test Code, *ACM*

Trans. Softw. Eng. Methodol., Vol. 32, No. 6(2023).

- [10] Tourani, P. and Adams, B.: The Impact of Human Discussions on Just-in-Time Quality Assurance: An Empirical Study on OpenStack and Eclipse, *Proc. 23rd Int'l Conf. Softw. Analysis, Evol., & Reeng.*, Vol. 1, March 2016, pp. 189–200.
- [11] Wang, S., Wen, M., Liu, Y., Wang, Y., and Wu, R.: Understanding and Facilitating the Co-Evolution of Production and Test Code, *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 272–283.

A 付録

本論文では、先行研究で公開されているデータセットを利用しているが、それらを詳細に確認すると誤ってバグ混入とラベル付けされたコミットが含まれていた。そのため、コミットデータの前処理を行った。

A.1 データセットの再ラベリング

先行研究の公開データセットについて、全コミットにおいてバグ混入コミットの占める割合が30%超とやや高かったことから、筆者らはその正確性に疑問を持った。そこで、公開データセットの“fixes”欄で管理されていたバグ修正コミットを追跡調査した。具体的には、リポジトリの最新(2024年12月現在)からすべてのコミットのコミットメッセージを調べ、そこで記載されているIssueIDを取得した。そして、Selenium[8]を用いてITS^{†4}上でIssueの種類を確認した。すると、fixesに含まれるコミットの中に“バグ修正ではない”コミットが含まれていることがわかった。そのため、バグ修正ではないコミットにも関わらず、バグありと判定されていたコミットを“バグではない”として再ラベリングを行った。

^{†4} <https://issues.apache.org/jira/issues>