NAIST-IS-MT1551105

修士論文

OSS 開発における欠陥数とカバレッジの関係に基づく継続的インテグレーションの有効性検証

南 智孝

2017 年 3 月 16 日

奈良先端科学技術大学院大学
情報科学研究科 情報科学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士（工学）授与の要件として提出した修士論文である．

南 智孝

審査委員：

松本 健一 教授　（主指導教員）
安本 慶一 教授　（副指導教員）
伊原 彰紀 助教　（副指導教員）

OSS 開発における欠陥数とカバレッジの関係に基づく継続的インテグレーションの有効性検証*

南 智孝

内容梗概

本論文では，オープンソースソフトウェア（OSS）を対象に，継続的インテグレーション（CI）におけるテストでの欠陥検出漏れを明らかにする．CI とは，開発者が加えた変更や修正をコミットするたびに，自動化されたビルド（コンパイル，テスト，デプロイ）が実行することにより，欠陥を早期発見し，ソフトウェアの品質を維持する手法である．近年，OSS 開発では，ソフトウェアの品質向上を図るために，CI を実施するプロジェクトが増加している．従来研究では，テスト実行結果の失敗の回数を欠陥検出数とし，CI の有効性を評価した．実際に OSS に混入している欠陥数には着目していない．本論文では，テストにより見逃された欠陥を見逃し欠陥と定義し，テストカバレッジと欠陥数，見逃し欠陥数の関係を分析し，CI の有効性を検証する．ケーススタディを行った結果，テストカバレッジが高いプロジェクトほど欠陥混入数が少なく，また，テストカバレッジが高いプロダクトコードほど，見逃し欠陥数が多いことが明らかとなった．CI はソフトウェアに混入している欠陥を発見するのに有効であるが，混入している全ての欠陥を発見することは不可能である．

キーワード

オープンソースソフトウェア，ソフトウェアテスト，継続的インテグレーション，見逃し欠陥，テストカバレッジ

*奈良先端科学技術大学院大学 情報科学研究科 情報科学専攻 修士論文, NAIST-IS-MT1551105, 2017 年 3 月 16 日.

An Analysis of effectiveness of CI in OSS based on relationship between the number of bugs and coverage*

Tomotaka Minami

Abstract

The goal of this thesis is to reveal the actual state of test effectiveness at Continuous Integration (CI) in Open Source Software (OSS) development. CI is a development practice that execute automated build (compile, test and deployment) in each commit. By practicing CI, developers can detect defects earlier and make the software quality higher. Recently, many OSS projects have practiced CI. Previous researches have eveuated the effectiveness of CI by comparing the number of defects induced in CI practicing projects and not CI practicing projects. In this thesis, we define post build defect as defects which could not be detected by testing and then we investigate the test effectiveness of CI by analyzing test coverage, number of defects and number of post build defects. As the result of case study, we found that the number of defects is small when the test coverage is high. We found that CI is effectiveness to detect defects but developers can't detect all defects by CI.

Keywords:

Open source software, software testing, Continuous Integration, post build bug, test coverage

*Master's Thesis, Department of Information Science, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT1551105, March 16, 2017.

目次

図目次

表目次

第 1 章　はじめに

ソフトウェア開発では，完成したソフトウェアに対して品質保証活動を実施する．品質保証活動には，開発者が主体のテスト活動と不特定多数の開発者や利用者が主体の検証活動がある．開発者主体のテスト活動には，ソフトウェアテストがある．ソフトウェアテストとは，完成したソフトウェアが正しく動作するか，完成したソフトウェアに欠陥が混入していないかを確認するための作業である．オープンソースソフトウェア （OSS）開発における品質保証活動は，テスト活動よりも短期間でリリースを行い，リリース時に OSS プロジェクトによって公開されたソースコードを不特定多数の開発者や利用者が検証，欠陥報告し，開発者が欠陥を修正する形態が主流である．OSS 開発でテスト活動が十分に実施されていない理由として，テスト活動に関心のある開発者の不足やソフトウェアテストを熟知してる開発者の不足が挙げられる．このような限られたリソースにおけるテスト活動は困難である [1].

近年，OSS 開発では，ソフトウェアの品質向上とテスト活動の効率化を図るために，継続的インテグレーション（CI）を実施してる．CI とは，開発者が加えた変更や修正をコミットするたびに，自動化されたビルド（コンパイル，テスト，デプロイ）が実行される手法である．CI の利点は，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行なうことにより欠陥を早期に発見し，ソフトウェアの品質を維持できることである．Hilton らは，約 34,000 件の OSS プロジェクトを対象に CI 実施プロジェクト数を調査した．約 4 割のプロジェクトで CI に取り組んでいた [11]．CI に関する既存研究について，Bller らは，1108 件の CI を実施している OSS プロジェクトにおいて，少なくとも 1 回はテストが失敗しているビルドの割合を調査している．その結果，全ビルド実行回数の約 1 割でテストが失敗していた [3]．また，CI 実施 OSS プロジェクトでは，CI 未実施 OSS プロジェクトと比較して，より多くの欠陥を検出できている [24]．しかし，全ての欠陥を特定することは現実的に不可能である．見逃し欠陥は，OSS の品質を低下させる．エンピリカルなソフトウェア工学の研究では，OSS 開発におけるテスト分野の研究は実施されていないのが現状である．特に，継続的インテグレーションにおけるテストの研究は全く実施されていないといっても過言ではない．一方で，ソフトウェア工学の研究

では，テスト分野の研究は以前から実施されている．近年，ソフトウェア工学のテスト分野の研究がエンピリカル分野に適用されてきている．我々は，カバレッジ，欠陥数と見逃し欠陥数の関係を調査し，CI の有効性を検証する．

本論文では，1300 件の OSS プロジェクトのビルドログのデータセット Travis-Torrent [*]，ソースコードホスティングサイト GitHub の issue tracker [†]の欠陥報告履歴，バージョン管理システム Git [‡]の開発履歴を用いて，テスト実施による欠陥数，見逃し欠陥数を明らかにする．

続く 2 章では，OSS 開発におけるテスト活動について説明する．続く 3 章では，分析手法を説明する．4 章では，ケーススタディの概要とその結果を述べる．5 章では，ケーススタディについての考察を行い，6 章では，関連研究を述べ本論文の立場を明らかにする．7 章では，本論文の制約を述べる．最後に 8 章で本論文のまとめを述べる．

[*] https://travistorrent.testroots.org/
[†] https://github.com/issues
[‡] https://git-scm.com/

bb

第 2 章　OSS 開発におけるテスト

2.1　ソフトウェアテスト

ソフトウェアテストとは、プログラムから仕様にない振舞いまたは欠陥を見つけ出す作業である．ソフトウェアの開発フェーズでテストを分類すると，テストは単体テスト，統合テスト，システムテスト，アルファテスト，ベータテスト，受け入れテスト，回帰テストに分類される．表 2.1 に，各テストの説明を示す．

表 2.1　ソフトウェアテストの種類

テスト	説明
単体テスト	作成したプログラムを構成する小さなユニット（メソッドや関数）が設計どおりに機能しているかを検証するテストのこと
統合テスト	単体テストを行ったユニット（メソッドや関数）を複数結合することで正しく動作するかを検証するテストのこと
システムテスト	要件通りの機能、性能であり、致命的なバグがないことやプログラム間の連携が正しく取れているかを検証するテストのこと
アルファテスト	開発者以外の人が操作して，不具合がないことを確認するテストのこと
ベータテスト	発売・リリース前の製品を開発者以外の一般ユーザーが操作して，不具合がないことを確認するテストのこと
受入れテスト	実際に仕様することが可能なシステムが開発ベンダーより正しく納品されたかどうかを検証するテストのこと
回帰テスト	欠陥を修正・変更した後に，変更箇所が正しく動作するかを検証するテストのこと

2.2　継続的インテグレーション

継続的インテグレーション（CI）とは，開発者が加えた変更や修正をコミットするたびに，自動化されたビルド（コンパイル，テスト，デプロイ）が実行されるこ

とであり，ビルドやテストをコミット単位で繰り返し行うことにより，欠陥を早期発見し，開発の効率化や納期の短縮を図る手法である．CI の目的は，欠陥を早期に発見して対処すること，ソフトウェアの品質を高めること，そしてソフトウェアの更新を検証してリリースするためにかかる時間を短縮することである．CI は，表 2.1 の単体テストや結合テストに当てはまる．近年，OSS 開発において注目されている品質保証活動の手法の 1 つである [11].

2.3 OSS 開発におけるテスト

ソフトウェアテストは，ソフトウェアに欠陥が混入していないかやソフトウェアが正しく動作するかを確認するために実施する．商用ソフトウェア開発における品質保証活動には，開発者が主体のテスト活動がある．商用ソフトウェアは，品質が重要視されるため，厳密なソフトウェアテストが実施されている．OSS 開発における品質保証活動には，開発者が主体のテスト活動（ソフトウェアテスト）と不特定多数の開発者や利用者が主体の検証活動（コードレビュー）がある．OSS 開発では，開発者主体のテスト活動より，アルファテストやベータテストのような利用者による欠陥報告を受けて，開発者が欠陥を修正する検証活動が主流である．その理由を以下に述べる．OSS 開発者はボランティアで開発に取り組んでおり，開発への参加，離脱がいつでも可能である．また，OSS 開発者には，ソフトウェアの品質を保証する責任がない．さらに，OSS の開発コミュニティには，ソフトウェアテストに熟知している開発者が不足していることが挙げられる．

Singh らは，OSS 開発においてソフトウェアテスト実施プロジェクト数を調査した．その結果，OSS プロジェクト 20,000 件のうち，約 4 割のプロジェクトでテストが実施されていないことがわかった [24]．また，Singh らは，テストを実施している OSS プロジェクトにおいて，プロジェクトが持つテストコードのカバレッジを調査している．テストが実施されている OSS プロジェクト 327 件におけるテストカバレッジの平均は 41.96% であった．一方で，OSS 開発者はソフトウェアテストを軽視しているわけではなく，OSS 開発者の 8 割は，ソフトウェアテストの実施が必要であると主張している [24]．このように，OSS 開発におけるテスト活動は十分に実施されておらず，実施されているプロジェクトにおいても，プロダクトコー

ドを高い精度で網羅してるテストコードは作成されておらず，OSS 開発におけるテスト活動に向上の余地はある．OSS 開発におけるテスト活動の不足の原因の 1 つとして挙げられるのは，OSS 開発コミュニティには，テスト活動に関心のある開発者やソフトウェアテストを熟知している開発者が不足していることである．その理由は，OSS 開発では開発者がボランティアで開発に取り組んでいることや開発者に OSS の品質を保証する責任がないことから，OSS プロジェクトに参加する開発者は品質保証活動よりもソフトウェアの開発自体に関心があるからである．

近年，OSS 開発におけるソフトウェアの品質向上やテスト活動の効率化を図るために，CI を実施する OSS プロジェクトが増えてきている．Michael らは，OSS プロジェクト 34,000 件を対象に調査を実施し，約 4 割の OSS プロジェクトで CI を実施していることがわかった [11]．また，Laura らは，CI を実施しているプロジェクトと CI を実施していないプロジェクトでは，CI を実施しているプロジェクトでより多くの欠陥を検出できていることをテスト実行結果の失敗の回数に基づき明らかにしている [24]．CI の実施が OSS に混入する欠陥の早期発見や OSS 開発における開発コストの削減に寄与していることがわかる．しかし，既存研究では，CI のテストにおける欠陥検出数に着目していない．本論文では，CI を実施している OSS プロジェクトを対象にテストカバレッジと欠陥数，見逃し欠陥数の関係に基づき CI の有効性を分析する．

2.4 Research Questions

本論文では，CI を実施している OSS プロジェクトのテスト活動を見逃し欠陥という観点から分析し，テストカバレッジと欠陥数，見逃し欠陥数の関係を明らかにすることで，CI の有効性を示す．見逃し欠陥とは，欠陥が混入しているソースコードを対象にテストを実施し，テスト結果が成功（対象ソースコードに欠陥が混入していなかったこと）を示すことで，テストにより発見されなかった欠陥と定義する．目的を達成するために，5 つのリサーチクエスチョン（RQ）に答える．

RQ1: CI を実施している OSS プロジェクトにおいて欠陥, 見逃し欠陥は存在するか？

CI を実施している OSS を対象に欠陥と見逃し欠陥の有無を確認するために，欠

陥数と見逃し欠陥数を調査する．

RQ2: CI を実施している OSS プロジェクトのテストカバレッジはどの程度あるのか？

CI を実施している OSS プロジェクトでは，ソフトウェアテストでプロダクトコードがどの程度網羅されているかを明らかにするために，CI を実施している OSS プロジェクトのテストカバレッジを調査する．

RQ3: CI によるビルドの実行回数と欠陥数, 見逃し欠陥数に関係はあるか？ CI の利点は，繰り返しビルドを実行することで欠陥を早期発見できることである．そこで，CI のビルドの実行回数と欠陥数，見逃し欠陥数の関係を明らかにするために，CI のビルドの実行回数と欠陥数，見逃し欠陥数の関係を分析する．CI のビルド実行回数と欠陥数，見逃し欠陥数に一定の関係があれば，欠陥，見逃し欠陥の混入要因の手がかりを得ることが可能である．

RQ4: テストカバレッジと欠陥数，見逃し欠陥数に関係はあるか？

プロダクトコードにおいてテストで網羅されていない部分に欠陥が混入しているかを明らかにするために，テストカバレッジと欠陥数，見逃し欠陥数の関係を調査する．

RQ5: CI によるテストの実行結果（pass, fail）と見逃し欠陥の混入有無に関係はあるか？

テストコードの品質と欠陥混入有無の関係を明らかにするために，テストの実行結果と欠陥混入数，見逃し欠陥混入数の関係を分析する．

;

第 3 章　分析手法

本章では，分析対象プロジェクトの選出方法と欠陥，見逃し欠陥を特定するための手法について説明する．また，カバレッジの算出方法を説明する．

3.1　分析対象 OSS の選出

本論文では，CI 実施によるビルド結果のデータセットである TravisTorrent の OSS プロジェクトの中で，JAVA 言語で記述されている，かつ，Maven を使用している OSS プロジェクトを対象に分析を行う [3]．TravisTorrent の対象プロジェクトの使用言語は JAVA 言語と Ruby 言語である．既存研究より，JAVA 言語を使用している OSS プロジェクトの方が Ruby 言語を使用している OSS プロジェクトよりテストが成功している割合が高いことが明らかになっている．JAVA 言語を使用しているプロジェクトのソフトウェアテストは欠陥を検出できていない，または，テストが通過するようにテストコードが書かれている可能性がある．JAVA 言語のプロジェクトのソフトウェアテストの有効性を検証するために JAVA 言語を分析対象とする．また，本論文の調査では，テスト自動化ツール Jmockit でテストを実行し，テスト結果を取得するため，JAVA 言語，かつ，Maven を使用している OSS プロジェクトを分析対象とする．

3.2　テストで網羅されているプロダクトコードの箇所（行）の特定

Jmockit はテストを自動で実行可能なツールであり，Junit 中でモックオブジェクトを容易に作成し，利用できるライブラリである．Jmockit は，実行したテスト結果をカバレッジレポートとして出力することが可能である．図 3.1 はプロジェクトのカバレッジレポートの一例である．図 3.2 はプロダクトコードのカバレッジレポートの一例である．ffffff

出力されたカバレッジレポートには，ソフトウェアテストの対象となった各プロダクトコードのカバレッジ，行カバレッジとプロジェクト全体のカバレッジが出力される．出力されたカバレッジレポートを分析することで，プロダクトコードのど

の行がテストコードにより網羅されているかを調査する．テストで網羅されているプロダクトコードを特定し，テストで網羅されているプロダクトコードの行数を取得する．また，テスト対象のプロダクトコードのカバレッジ、各プロジェクトの全体のカバレッジを取得する．

3.3 欠陥，見逃し欠陥の特定

3.3.1 見逃し欠陥

図 3.3 は見逃し欠陥の概略図である．見逃し欠陥とは，欠陥が混入しているプロダクトコードを対象にテストを実施し，テスト結果が成功（対象プロダクトコードに欠陥が混入していなかったこと）を示すことで，テストにより発見されなかった欠陥のことである．次に，見逃し欠陥の特定方法を以下の 6 つの手順で示す．

1. Git のコミットログから欠陥修正コミットを特定し，欠陥修正日時を取得する．
2. Sliwerski らが提案した SZZ アルゴリズム [21] をもとに，Git のコミットログから欠陥が混入したコミットを特定する（欠陥の特定）．
3. (2) で特定した欠陥混入コミットで変更されたプロダクトコードの変更行数（例：xx 行目から yy 行目）を取得する．
4. Jmockit で出力されたカバレッジレポートからテストで網羅されているプロダクトコードを特定し，テストで網羅されているプロダクトコードの行数（例：aa 行目から bb 行目）を取得する．
5. (3) で特定した欠陥混入コミットで変更されたプロダクトコードの変更行数と (4) で特定したテストで網羅されているプロダクトコードの行数を比較する．
6. 欠陥混入コミットで変更されたプロダクトコードの変更行数とテストで網羅されているプロダクトコードの行数が一致すれば，(2) で特定した変更は見逃し欠陥であると考える．

各手順の詳細について説明する．

3.3.2 欠陥修正コミットの特定

本手法では，Sliwerski らが提案した SZZ アルゴリズムをもとに、バージョン管理システム Git のコミットメッセージとソースコードホスティングサイト GitHub の issue tracker に保存されている欠陥報告履歴を紐づける．GitHub の issue tracker では報告された欠陥報告票にバグ ID とタグ（bug）が割り当てられる．また、OSS 開発では，欠陥修正時にコミットメッセージに修正した欠陥の ID と修正内容を記載する習慣がある．コミットメッセージの例を図 3.4，図 3.5 に示す．図 3.4 はバグ ID がコミットメッセージに記載されているため，欠陥修正コミットを特定可能である．図 3.5 はバグ ID がコミットメッセージに記載さていないため，欠陥修正コミットを特定できない．本論文では，この習慣を利用し欠陥報告履歴と欠陥修正コミットを紐づける．まず，以下の条件のうち 1 つ以上を満たすコミットメッセージを抽出する：

- 以下のような正規表現に該当するコミットメッセージ:
  - `bug[# \t]*[0-9]+`
  - `issue[# \t]*[0-9]+`
  - `fix[# \t]*[0-9]+`
- 数字を含むコミットメッセージ．
- キーワードを含むコミットメッセージ．例えば，以下のような正規表現に該当するコミットメッセージ:
  - `fix(e[ds])?|bugs?|defects?|patch for?|solve?|close`

抽出されたコミットメッセージが以下の条件を満たす時，当該コミットメッセージを持つコミットは，欠陥修正コミットであると考える：

- コミットメッセージ内の数値が，バグ ID と一致している．
- 欠陥報告票のラベルが bug である．
- コミットメッセージが（“fix”，“bug”，“バグ ID” または “数値” のような）キーワードを含んでいる．
- 欠陥報告票の状態が，1 度以上修正完了になっている．
- 当該コミットのパッチ作成者が，欠陥の修正担当者になっている．

3.3.3　欠陥混入コミットの特定

欠陥混入コミットを特定するために，欠陥修正コミットで変更されたファイルと変更行数を特定する．次に，欠陥修正コミットで変更されたファイルの変更箇所が最後に変更されたコミットを特定する．ソースコードホスティングサイト GitHub の issue tracker に存在するタグが"bug"の欠陥報告履歴から欠陥報告日，欠陥修正日，バグ ID，author を取得する．図 3.6 は，タグ"Bug"が付与されている欠陥報告表の一例である．そして，そのコミットが実施された日付と欠陥報告日を比較し，コミットが実施された日付が欠陥報告日よりも前である場合，そのコミットを欠陥混入コミットとする．

3.3.4　見逃し欠陥の特定

見逃し欠陥は，Jmockit でテストを実行し，出力されたカバレッジレポートによりプロダクトコードが網羅されている場所（行）と欠陥混入コミットで変更されたファイルの変更行数を比較し，プロダクトコードが網羅されている場所が欠陥混入コミットで変更されたファイルの変更行数と一致する場合，欠陥混入コミットでの変更箇所は見逃し欠陥となる．テストの実行によりプロダクトコードが網羅されている場所（行）と欠陥混入コミットで変更されたファイルの変更行数の紐付け方法について説明する．テストで網羅されているプロダクトコードのファイルパスと欠陥混入コミットで変更されたファイルのファイルパスの紐付けにより，テストで網羅されているプロダクトコードの行数と欠陥混入コミットで変更された行数を比較し，一致していれば欠陥混入コミットでの変更箇所は見逃し欠陥である．

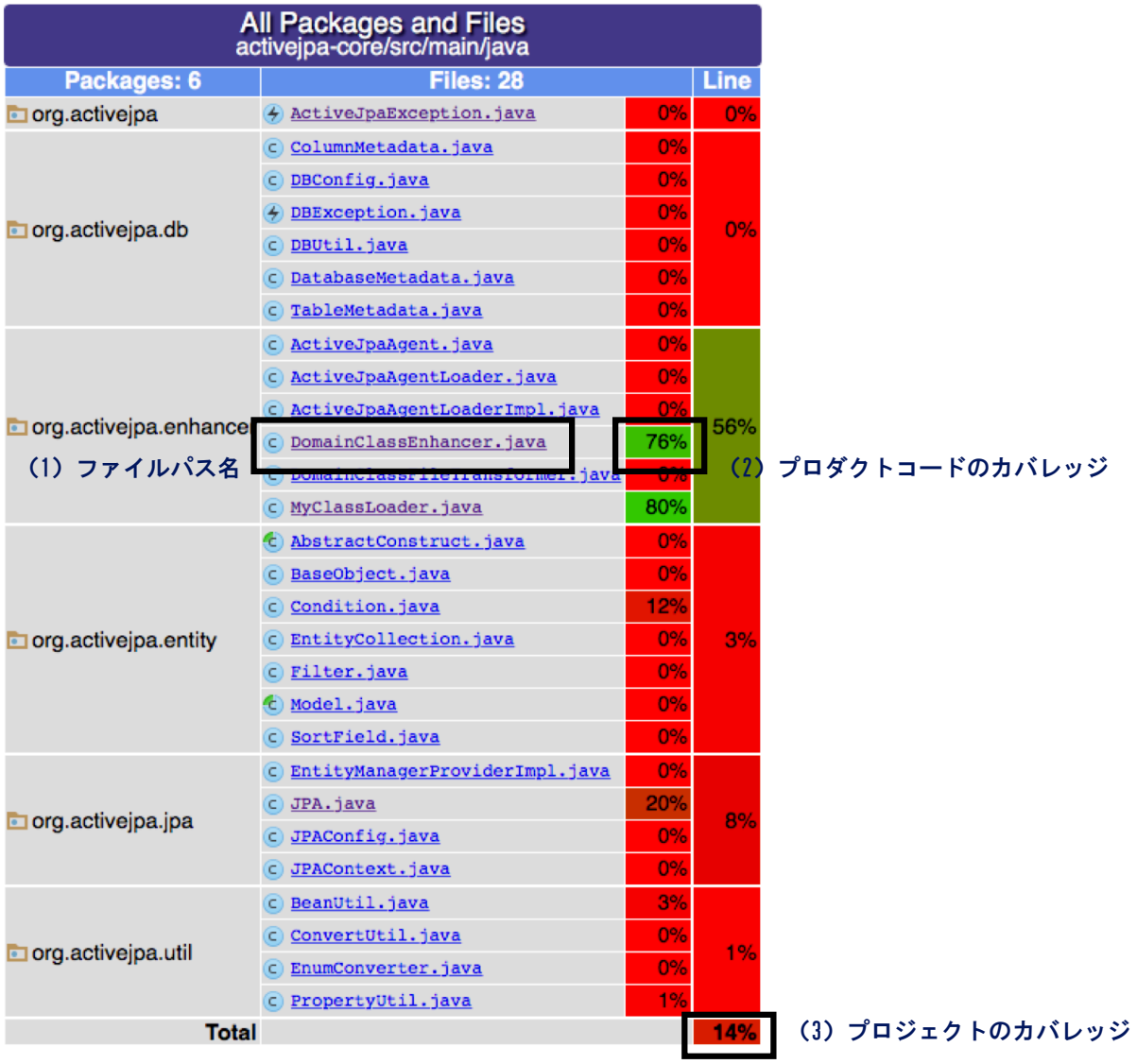

図 3.1　プロジェクトのカバレッジレポートの一例

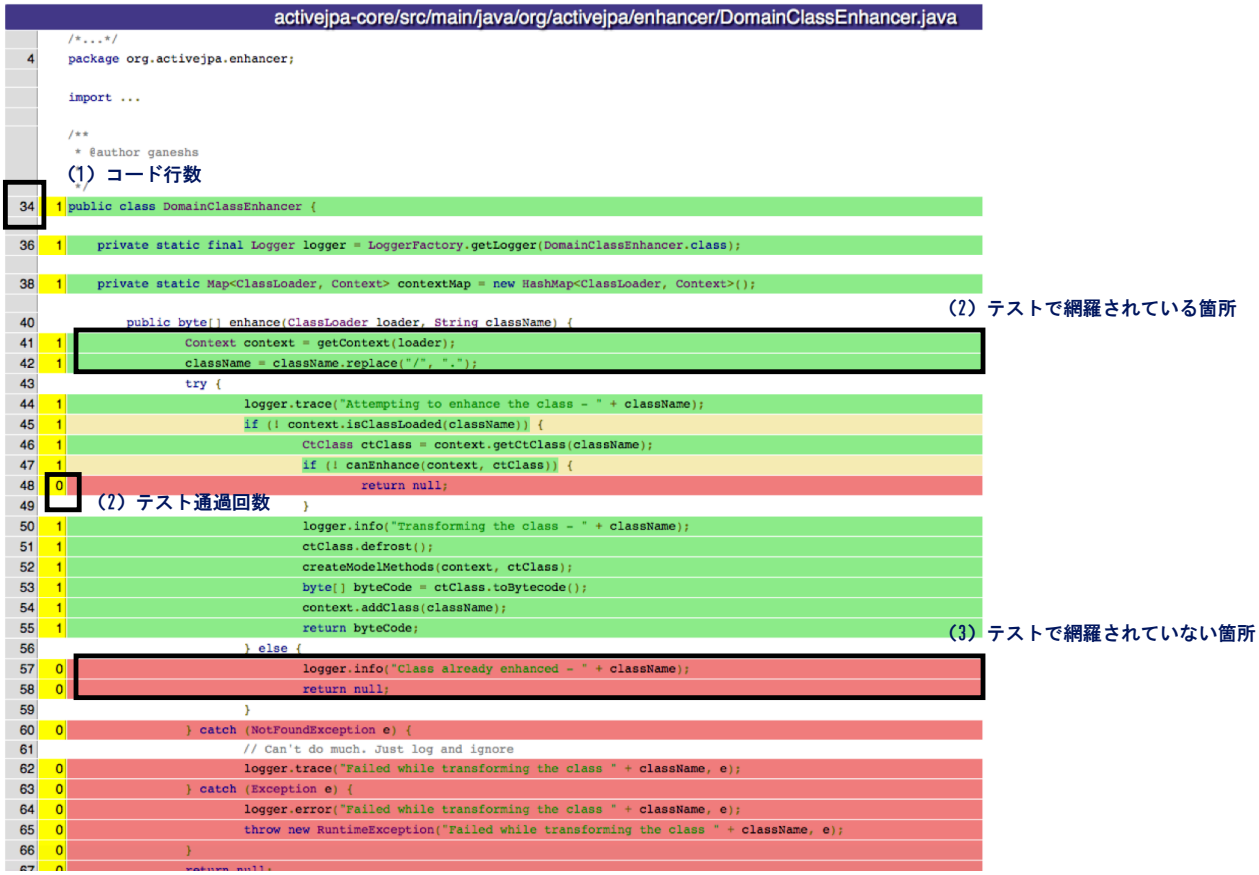

図 3.2　プロダクトコードのカバレッジレポートの一例

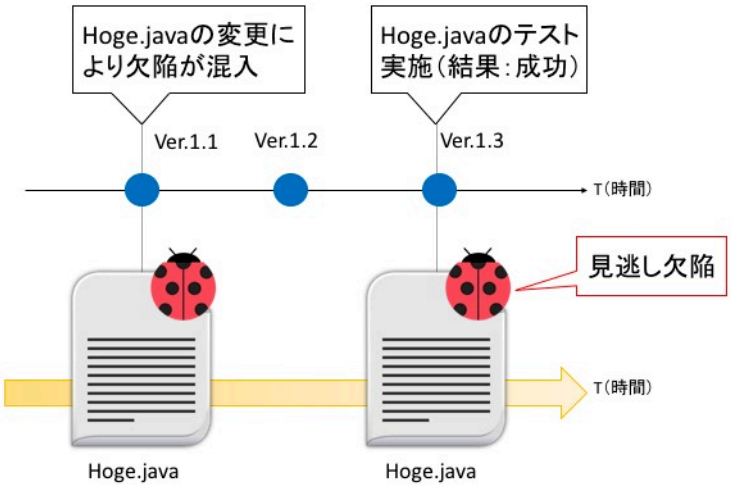

図 3.3　見逃し欠陥の概略図

```
commit 26cd0e64b484659d50d4456e26443eda20097647
Author: Ganesh Subramanian <ganesh.subramanian@hightail.com>
Date:   Fri Sep 26 17:17:08 2014 +0530

    Fixed issue #26
```

(1) ハッシュ値
(2) コミッター
(3) コミットした日付
(4) Bug IDが記載されているコミットメッセージ

図 3.4　欠陥修正コミットの特定可能なコミットメッセージの一例

```
commit 9fcf4bedcf78d4c1e6764943594c42c984d9a376
Author: gayatri.mali <gayatri.mali@olacabs.com>
Date:   Mon Feb 15 00:33:51 2016 +0530

    Fix for supporting 5.x version of hibernate-entitymanager
```

(1) ハッシュ値
(2) コミッター
(3) コミットした日付
(4) Bug IDが記載されていないコミットメッセージ

図 3.5　欠陥修正コミットの特定不可能なコミットメッセージの一例

図 3.6　欠陥報告票の一例

第 4 章　ケーススタディ

本章では，1,300 件の OSS プロジェクトのビルドログのデータセット TravisTorrent *[4] を分析対象とする．

4.1　データセット

TravisTorrent は，Beller[4] らにより，2015 年 8 月末に公開された CI 自動化ツール Travis CI で実行されたビルドの実行結果のデータセットである．Travis CI は GItHub と連動しており，指定したリポジトリ上にあるソースコードを自動的に取得し，ビルドを実行するツールである．TravisTorrent の詳細について説明する．TravisTorrent は，2015 年 8 月時点で，GitHub 上でアクティブな状態の 17,313,330 件のリポジトリから 1300 件の OSS プロジェクトのソースコード，開発プロセス，依存状態などの分析結果を含む．データセットの対象となった 1300 件の OSS プロジェクトは，GitHub 上で 2 番目,3 番目に多く使用されているプログラミング言語 Ruby（898 件）,Java（402 件）を使用しているプロジェクトの中で，CitHub 上での人気度（GitHub で 10 人以上が閲覧している）と Travis CI の利用数（ビルドの実行回数が 50 回以上）により選出されている．TravisTorrent が開発された目的について説明する．近年，OSS 開発において CI を実施するプロジェクトが増加する中で，CI の有効性は定量的に示されていなかった．TravisTorrent の公開により，様々な観点から CI の有効性を検証することが可能となる．また，TravisTorrent の利点は，CI 環境（Travis CI），バージョン管理システム（Git）とソースコードホスティングサービス（GitHub）が連動しており，データ取得，データ分析が容易にできることである．本論文では，TravisTorrent の中から，使用プログラミング言語が Java，かつ，Maven を使用している 246 件の OSS プロジェクトを分析対象とし，テストカバレッジ，欠陥数，見逃し欠陥数の関係を調査する．

Git は，ソースコードなどの変更履歴を記録，追跡することができる分散型のバージョン管理システムである．Git に存在するコミットメッセージで欠陥の修正

*https://travistorrent.testroots.org/

が特定可能なコミットを本論文では分析対象する．

GitHub は，ソフトウェア開発プロジェクトのためのソースコードホスティングサービスである．GitHub の issue tracker では，欠陥報告票にタグを付与することができる．本分析では，タグが"bug"のプロジェクトを対象に欠陥報告票を収集した．

Jmockit はテストを自動で実行可能なツールであり，Junit 中でモックオブジェクトを容易に作成し, 利用できるライブラリである．Jmockit は，実行したテスト結果をカバレッジレポートとして出力することができる．プロダクトコードとテストコードが存在している OSS プロジェクトを対象にカバレッジレポートを収集する．

本論文では，2016 年 7 月 9 日時点に公開されている Travistorrent のデータセットを用いて分析を行った．

表 4.1[4][10] は，TravisTorrent に存在するメトリクスを示す．表 4.1 のメトリクスを分析し，分析対象 OSS プロジェクト 246 件のプロジェクトの規模，ビルドの実行回数，ビルドの成功回数，ビルドの失敗回数を表 4.2 に示す．また，上記と同様にプロジェクトの規模，テストの実行回数，テストの成功回数，テストの失敗回数を表 4.3 に示す．

また，表 4.1 のメトリクスを分析し，分析対象 OSS プロジェクト 246 件のプロジェクトの規模，ビルドの実行回数，ビルドの成功回数，ビルドの失敗回数を表 4.2 に示す．また，上記と同様にプロジェクトの規模，テストの実行回数，テストの成功回数，テストの失敗回数を表 4.3 に示す．

表 4.1　TravisTorrent のメトリクスの一覧

Column Name	Description	Unit	Example
row	Unique identifier for a build job in TravisTorrent	Integer	1543966
git_commit	SHA1 Hash of the commit which triggered this build (should be unique world-wide)	String	c1d9c11cbe3d20f2...
git_merged_with	If this commit sits on a Pull Request (gh_is_pr true), the SHA1 of the commit that merged said pull request	String	
git_branch	Branch git_commit was committed on	String	4-1-stable
git_commits	All commits included in the push that triggered the build, minus the built commit	List of Strings	87a2f02199d21a2aa...
git_num_commits	The number of commits in git_commits, to ease efficient splitting	String	1
git_num_committers	Number of people who committed to this project	Integer	1
gh_project_name	Project name on GitHub (in format user/repository)	String	rails/rails
gh_is_pr	Whether this build was triggered as part of a pull request on GitHub	Boolean	false
gh_lang	Dominant repository language, according to GitHub	String	ruby
gh_first_commit_created_at	Timestamp of first commit in the push that triggered the build	ISO Date (UTC+1)	2014-04-18 20:12:32
gh_team_size	Size of the team contributing to this project within 3 months of last commit	Integer	168
gh_num_issue_comments	If git_commit is linked to a PR on GitHub, the number of comments on that PR	Integer	0
gh_num_commit_comments	The number of comments on git_commits on GitHub	Integer	0
gh_num_pr_comments	If gh_is_pr is true, the number of comments on this pull request on GitHub	Integer	0
gh_src_churn	How much (lines) production code changed by the new commits in this build	Integer	4
gh_test_churn	How much (lines) test code changed by the new commits in this build	Integer	8
gh_files_added	Number of files added by the new commits in this build	Integer	0
gh_files_deleted	Number of files deleted by the new commits in this build	Integer	0
gh_files_modified	Number of files modified by the new commits in this build	Integer	3
gh_tests_added	Lines of testing code added by the new commits in this build	Integer	0
gh_tests_deleted	Lines of testing code deleted by the new commits in this build	Integer	0
gh_src_files	Number of production files in the new commits in this build	Integer	
gh_doc_files	Number of documentation files in the new commits in this build	Integer	
gh_other_files	Number of remaining files which are neither production code nor documentation in the new commits in this build	Integer	
gh_commits_on_files_touched	Number of unique commits on the files included in this build within 3 months of last commit	93	
gh_sloc	Number of executable production source lines of code, in the entire repository	Integer	53421
gh_test_lines_per_kloc	Test density. Number of lines in test cases per 1,000 gh_sloc	Double	2191.011
gh_test_cases_per_kloc	Test density. Number of test cases per 1,000 gh_sloc	Double	188.3342
gh_asserts_cases_per_kloc	Assert density. Number of assertions per 1,000 gh_sloc	Double	535.0143
gh_by_core_team_member	Whether this commit was authored by a core team member	Boolean	true
gh_description_complexity	If gh_is_pr is true, the total number of words in the pull request title and description	Integer	
gh_pull_req_num	Pull request number on GitHub	Integer	
tr_build_id	Unique build ID on Travis	String	23298954
tr_status	Build status (pass, fail, errored, canceled)	String	passed
tr_duration	Overall duration of the build	Integer (in seconds)	23389
tr_started_at	Start of the build process	ISO Date (UTC)	2014-04-18 19:12:32
tr_jobs	Which Travis jobs executed this build (number of integration environments)	List of Strings	[23298955, ...]
tr_build_number	Build number in the project	Integer	15459
tr_job_id	This build job' s id, one of tr_jobs	String	23298981
tr_lan	Language of the build, as recognized by BUILDLOGANALYZER	String	ruby
tr_setup_time	Setup time for the Travis build to start	Integer (in seconds)	0
tr_analyzer	Build log analyzer that took over (ruby, java-ant, java-maven,java-gradle)	String	ruby
tr_frameworks	Test frameworks that tr_analyzer recognizes and invokes (junit, rspec, cucumber, ...)	List of Strings	testunit
tr_tests_ok	If available (depends on tr_frameworks and tr_analyzer): Number of tests passed	Integer	310
tr_tests_fail	If available (depends on tr_frameworks and tr_analyzer): Number of tests failed	Integer	1
tr_tests_run	If available (depends on tr_frameworks and tr_analyzer): Number of tests were run as part of this build	Integer	311
tr_tests_skipped	If available (depends on tr_frameworks and tr_analyzer): Number of tests were skipped or ignored in the build	Integer	
tr_failed_tests	All tests that failed in this build	List of strings	SerializedAttributeTest
tr_testduration	Time it took to run the tests	Double (in seconds)	28.2
tr_purebuildduration	Time it took to run the build (without Travis scheduling and provisioning the build)	Double (in seconds)	
tr_tests_ran	Whether tests ran in this build	Boolean	true
tr_tests_failed	Whether tests failed in this build	Boolean	true
tr_num_jobs	How many jobs does this build have (length of tr_jobs)	Integer	30
tr_prev_build	Serialized link to the previous build, by giving its tr_build_id	String	39557888
tr_ci_latency	Latency induced by Travis (scheduling, build pick-up, ...)	Integer (in seconds)	1408

表 4.2　TravisTorrent の分析対象プロジェクトのプロジェクト数，プロジェクトの規模，ビルドの実行回数，ビルドの成功回数，およびビルドの失敗回数

プロジェクト数	プロジェクトの規模	ビルドの実行回数	ビルドの成功回数	ビルドの失敗回数
246	12.75	1048,75	85,313	11,909

表 4.3　TravisTorrent の分析対象プロジェクトのプロジェクト数，プロジェクトの規模，テストの実行回数，テストの成功回数，およびテストの失敗回数

プロジェクト数	プロジェクトの規模	テストの実行回数	テストの成功回数	テストの失敗回数
246	12.75	97,866	8,286	89,580

4.2　RQ1: CI を実施している OSS プロジェクトにおいて欠陥，見逃し欠陥は存在するか？

分析方法

CI を実施している 246 件の OSS プロジェクトに混入している欠陥，見逃し欠陥の有無を確認するために，Sliwerski らが提案した SZZ アルゴリズム [21] をもとに，欠陥混入コミットを特定し，欠陥数を算出する．見逃し欠陥数は，欠陥混入コミットで変更されたファイルと行を特定し，テスト自動化ツール Jmockit を実行し，出力されたカバレッジレポートから変更されたファイルの行がテストで網羅されているかを紐づける．もし，欠陥混入コミットで変更された行がテストで網羅されている場合，欠陥混入コミットは見逃し欠陥であると考える．

分析結果

表 4.4 に欠陥が混入していたプロジェクト数，見逃し欠陥数が混入していたプロジェクト数，欠陥を発見することができなかったプロジェクト数を示す．分析対象の 246 件の OSS プロジェクトのうち 28 件（11.4%）のプロジェクトで欠陥を発見

することができた．欠陥を特定できた 28 件のプロジェクトのうち，見逃し欠陥を特定できたプロジェクトは 7 件（25%）であった．一方で，プロジェクト上の制約のため，218 件（88.6%）のプロジェクトでは，欠陥混入コミットを特定することができなかった．表 4.5 に欠陥混入コミットを特定することができなかった原因を示す．次に，表 4.6 に分析により発見された欠陥数，見逃し欠陥数を示す．SZZ アルゴリズムにより特定できた欠陥数は 332 件であった．また，特定できた欠陥 332 件のうち，見逃し欠陥数は 23 件（6.9%）であった．結果より，見逃し欠陥は存在する．欠陥数に占める見逃し欠陥数の割合が 6.9% であることから，CI のテストで欠陥が見逃されることは非常に少ないといえる．

RQ1 の回答: 欠陥が発見されたプロジェクトは 28 件（11.4%）で欠陥の数は 332 件であった．また，見逃し欠陥が発見されたプロジェクトは 7 件（25%）で見逃し欠陥の数は 17 件であった．

表 4.4　欠陥混入プロジェクト数，見逃し欠陥混入プロジェクト数，および欠陥の発見が不可能なプロジェクト数

プロジェクトの種類	件数
分析対象プロジェクト	246
欠陥混入有りプロジェクト	28
見逃し欠陥混入有りプロジェクト	7
欠陥発見不可プロジェクト	218

表 4.5　欠陥と見逃し欠陥を特定できなかった原因とプロジェクト数

要因	内容	件数
欠陥報告票	欠陥報告票に “bug” のラベルが割り当てられていない	122
コミットメッセージ	コミットメッセージにバグ ID が未記載	79
Jmockit	テストコードの削除によるテストの実行が不可能	48

表 4.6　欠陥数と見逃し欠陥数

欠陥の種類	件数
欠陥	332
見逃し欠陥	23

4.3　RQ2: CI を実施している OSS プロジェクトのテストカバレッジは？

分析方法

RQ1 で特定した欠陥が混入してる OSS プロジェクト 28 件を対象に，テスト自動化ツール Jmockit でテストを実行し，出力されたカバレッジレポートを分析し，各プロジェクトのカバレッジを算出する．また，各プロジェクトに存在するプロダクトコードのカバレッジを各プロジェクト単位で算出する．

分析結果

表 4.7 は分析対象プロジェクト 28 件におけるテストカバレッジの最大値，最小値，中央値，平均を示す．図 4.2 は 28 件のプロジェクト全体のテストカバレッジを示す．また，図 4.3 は各プロジェクトに存在するプロダクトコードのカバレッジを各プロジェクト単位で示す．表 4.7 より，テストカバレッジ最大は 92.77%，テストカバレッジが最小のプロジェクトは 0%，テストカバレッジの中央値は 22%，テストカバレッジの平均は 32.65% であった．また，図 4.3 より，プロジェクトによりカバレッジに偏りがあった．

RQ2 の回答: 分析対象 OSS プロジェクト 28 件の平均的なテストカバレッジは 32.65% であった．また，プロジェクトごとにカバレッジに偏りがあることがわかった．

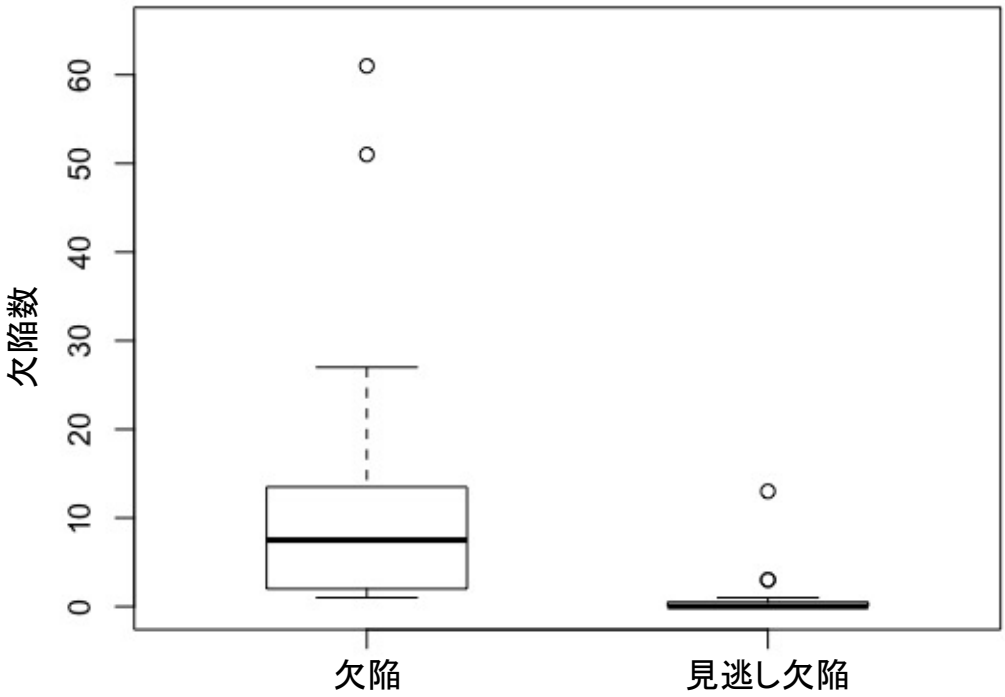

図 4.1　分析対象プロジェクトに混入している欠陥数，見逃し欠陥数

4.4　RQ3: CI によるビルドの実行回数と欠陥数，見逃し欠陥数に関係はあるか？

分析方法

RQ1 で特定した欠陥が混入してる OSS プロジェクト 28 件を対象に，TravisTorrent より各プロジェクトのビルドの実行回数を算出する．RQ1 で算出した欠陥数，見逃し欠陥数とビルドの実行回数の関係を分析する．

表 4.7　分析対象プロジェクトのテストカバレッジ

プロジェクト数	カバレッジの最大値	カバレッジの最小値	カバレッジの中央値	カバレッジの平均
28	92.77	0	22	32.65

分析結果

図 4.4 はビルドの実行回数と欠陥数の関係を示す．図 4.5 はビルドの実行回数と見逃し欠陥数の関係を示す．図 4.4 において，相関係数が 0.007 から，ほとんど相関はない．図 4.4 より，ビルドの実行回数と欠陥数に相関はないことがわかった．図 4.5 において，相関係数が-0.07 から，ほとんど相関はない．図 4.5 より，ビルドの実行回数と見逃し欠陥数に相関はないことがわかった．

RQ3 の回答: ビルドの実行回数と欠陥数に相関はない．また，ビルドの実行回数と見逃し欠陥数に相関はない．

4.5　RQ4: テストカバレッジと欠陥数，見逃し欠陥数に関係はあるか？

分析方法

RQ1 で特定した欠陥数，見逃し欠陥数と RQ2 で算出したカバレッジを用いて，プロジェクトのテストカバレッジを 0-9，10-19，20-29，30-39，40-49，50-59，60-69，70-79，80-89，90-99，100 の 11 段階に分類し，各段階のカバレッジを持つプロジェクト数，欠陥数，見逃し欠陥数を算出する．また，プロダクトコードのテストカバレッジを 0-9，10-19，20-29，30-39，40-49，50-59，60-69，70-79，80-89，90-99，100 の 11 段階に分類し，各段階のカバレッジを持つプロダクトコード数，欠陥数，見逃し欠陥数を算出する．テストカバレッジと欠陥数，見逃し欠陥数の関係を分析する．

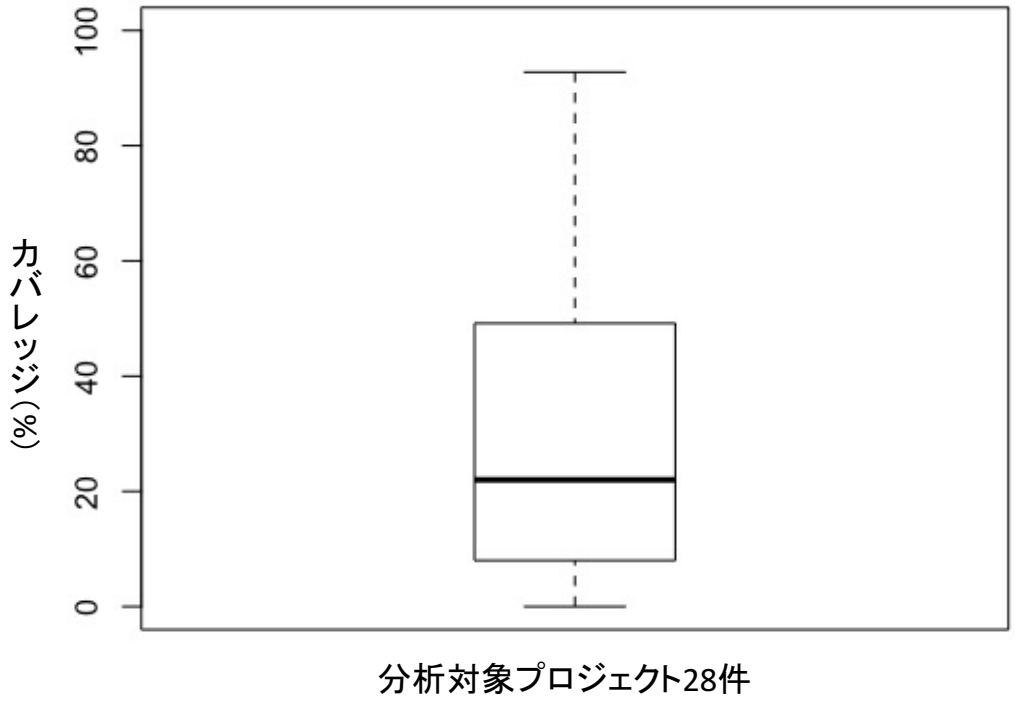

図 4.2　分析対象プロジェクトのテストカバレッジ

分析結果

表 4.8 はプロジェクトのテストカバレッジを 0-9，10-19，20-29，30-39，40-49，50-59，60-69，70-79，80-89，90-99，100 の 11 段階に分類し，各段階のカバレッジを持つプロジェクト数，欠陥数，見逃し欠陥数を算出した結果である．

また，表 4.9 はプロダクトコードのテストカバレッジを 0-9，10-19，20-29，30-39，40-49，50-59，60-69，70-79，80-89，90-99，100 の 11 段階に分類し，各段階のカバレッジを持つプロダクトコード数，欠陥数，見逃し欠陥数を算出した結果である．

表 4.8 より，プロジェクトのテストカバレッジと欠陥数に着目すると，テストカ

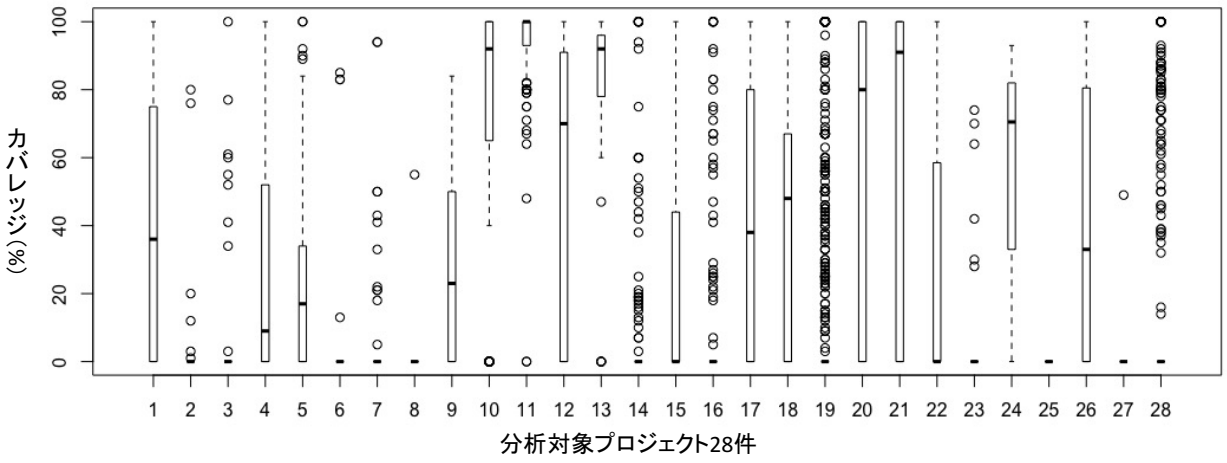

図 4.3　各プロジェクトに存在するプロダクトコードのテストカバレッジ

バレッジが 0-9 のプロジェクトに欠陥が 132 件，40-49 のプロジェクトに欠陥が 81 件混入していることから，カバレッジが低いプロジェクトに多くの欠陥が混入している．また，見逃し欠陥に関して，カバレッジが 40-49 のプロジェクトに最も多く混入していることがわかった．プロジェクトのテストカバレッジと欠陥数の相関係数は-0.71 から，高い相関がある．また，P 値により統計的有意差（有意水準 5%）が認められた．プロジェクトのテストカバレッジと見逃し欠陥数の相関係数は-0.04 から，ほとんど相関がない．表 4.8 より，プロジェクトのテストカバレッジと欠陥数には高い相関があることがわかった．また，プロジェクトのテストカバレッジと見逃し欠陥数に相関がないことがわかった．

表 4.9 より，プロダクトコードのテストカバレッジとプロダクトコード数に着目すると，プロダクトコードのテストカバレッジが 0-9 のプロダクトコードが 2850 件と多数を占めている．また，プロダクトコードのテストカバレッジと欠陥数に着目すると，プロダクトコードのテストカバレッジが 0-9 のプロダクトコードに 133 件の欠陥が混入している．一方で，見逃し欠陥数は，テストカバレッジが高いプロダクトコードに混入していることがわかった．プロダクトコードのテストカバレッジと欠陥数の相関係数は-0.49 であるが，P 値により統計的有意差（有意水準 5%）

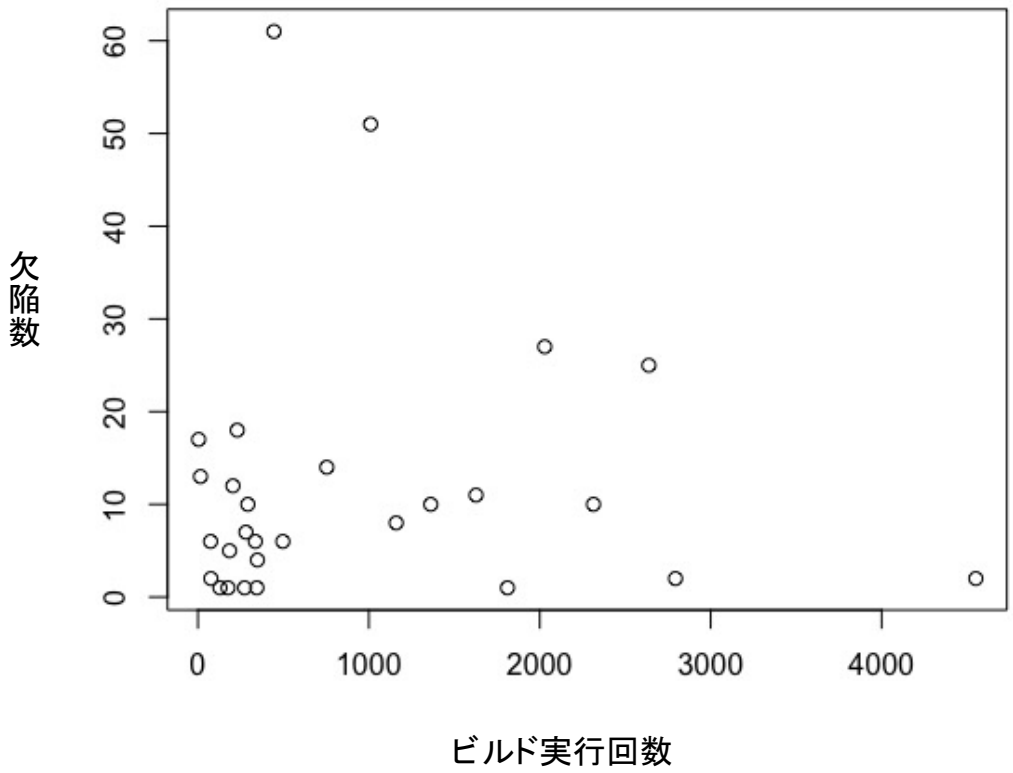

図 4.4　ビルド実行回数と欠陥数

が認められなかった．プロダクトコードのテストカバレッジと見逃し欠陥数の相関係数は 0.84 から，高い相関がある．また，P 値により統計的有意差（有意水準 5%）が認められた．表 4.9 より，プロダクトコードのテストカバレッジと見逃し欠陥数には高い相関があることがわかった．

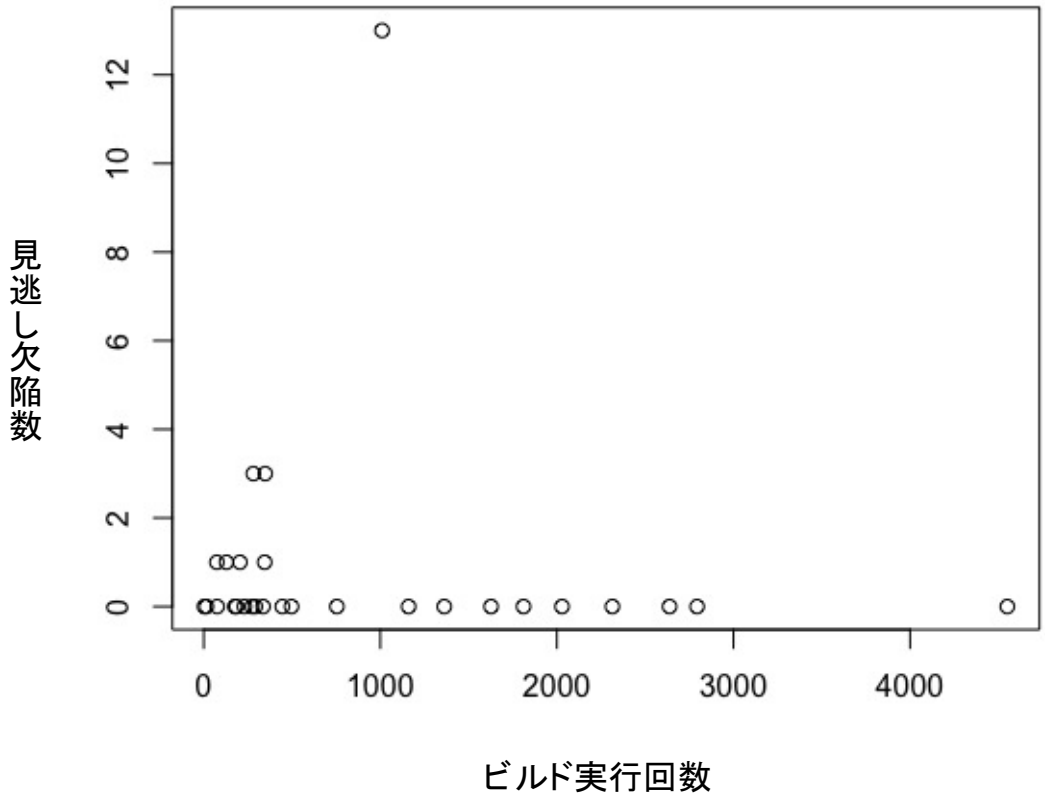

図 4.5　ビルド実行回数と見逃し欠陥数

RQ4 の回答: プロジェクトのテストカバレッジと欠陥数には高い相関がある．また，プロジェクトのテストカバレッジと見逃し欠陥数に相関がない．さらに，プロダクトコードのテストカバレッジと見逃し欠陥数には高い相関がある．つまり，プロジェクトのテストカバレッジが高いプロジェクトほど欠陥混入数が少ないので，品質が高い．テストカバレッジが高いプロダクトコードほど見逃し欠陥数が多いことから，混入している全ての欠陥を発見することは不可能である．

表 4.8　プロジェクトのカバレッジ，プロジェクト数，欠陥数，および見逃し欠陥数

プロジェクトのテストカバレッジ	プロジェクト数	欠陥数	見逃し欠陥数
100	0	0	0
90-99	1	2	0
80-89	1	4	3
70-79	2	7	1
60-69	2	17	3
50-59	1	1	1
40-49	5	81	13
30-39	1	1	1
20-29	3	41	0
10-19	5	46	1
0-9	7	132	0

4.6　RQ5:　CI によるテストの実行結果（pass, fail）と見逃し欠陥の混入有無に関係はあるのか？

分析方法

TravisTorrent より，RQ1 で特定した欠陥が混入してる OSS プロジェクト 28 件のテストの実行結果を抽出し，RQ1 で算出した欠陥数，見逃し欠陥数とテストの実行結果の関係を分析する．

分析結果

表 4.10 より，テストの実行結果が成功で混入していた欠陥の数は 51 件，混入していた見逃し欠陥の数は 4 件であった．テストの実行結果が失敗で混入していた欠陥の数は 10 件，混入していた見逃し欠陥数は 0 件であった．また，TravisTorrent と SZZ アルゴリズムで特定した欠陥 332 件を完全に紐づけることはできなかった．

表 4.9　プロダクトコードのカバレッジ，プロジェクト数，欠陥数，および見逃し欠陥数

プロダクトコードのテストカバレッジ	プロダクトコード数	欠陥数	見逃し欠陥数
100	432	7	4
90-99	143	8	5
80-89	215	5	2
70-79	202	6	4
60-69	132	9	4
50-59	133	5	2
40-49	102	5	1
30-39	89	2	0
20-29	132	16	0
10-19	44	2	0
0-9	2850	133	1

表 4.10　テストの実行結果と欠陥数，見逃し欠陥数

テストの実行結果	混入していた欠陥数	混入していた見逃し欠陥数
成功	51	4
失敗	10	0

RQ5 の回答: テストが成功しているにも関わらず，プロダクトコードに混入している欠陥の数は多い.

第 5 章　議論

5.1　CI を実施している OSS の品質

本節では，RQ1,　RQ2,　RQ3 について考察する．RQ1 で欠陥を特定できたプロジェクト数が分析対象プロジェクト 246 件のうち 28 件（11.4%） であった．各プロジェクトにより，コミットメッセージの書き方，欠陥報告票のタグ付け方法，ソースコードの管理方法が異なり，データの収集が不可能であったため，分析対象プロジェクト数が減少した．

RQ2 で分析対象プロジェクト 28 件のテストカバレッジの平均が 32.65% であることから，CI を実施している OSS の品質は低いと言える．また，テストカバレッジが 92.77% のプロジェクトが存在する，一方で，テストカバレッジが 0% のプロジェクトも存在することから，プロジェクトによりカバレッジに偏りがあることがわかる．カバレッジが低いプロジェクトが存在する原因として，OSS 開発者には品質を保証する責任がないことから，プロジェクトに存在する全コンポーネントをテストせず，重要なコンポーネントのみをテストしていると考えられる．また，高い品質が要求されるミッションクリティカルな OSS ではカバレッジが高くなるというように，OSS の種類によりカバレッジが高くなる，低くなるいうことが生じる可能性がある．そこで，表 5.1 は，分析対象プロジェクトをソフトウェアの種類に分類し，テストカバレッジの平均を示す．表 5.1 より，カバレッジの平均が最大プロジェクトの種類は，UI 系の OSS であり，カバレッジの平均が最小プロジェクトの種類は，データベース系の OSS であった．

RQ3 で，一般的に CI は短期間に繰り返しビルドを実行することで，品質の向上やテストの効率化を図ることが可能である．RQ3 の結果から，ビルドの実行回数と欠陥数に相関関係はないことがわかった．ソフトウェアの品質を向上するためには，テストコードの品質，つまり，テストカバレッジが重要であることが言える．

表 5.1　欠陥混入プロジェクト数，見逃し欠陥混入プロジェクト数，および欠陥の発見が不可能なプロジェクト数

OSS の種類	プロジェクト件数	カバレッジの平均
サーバー・クライアント・API	6	38.7
ウェブアプリケーション	1	23
ライブラリ・フレームワーク	8	31.2
データベース	2	1
UI	1	92.8
プラットフォーム	2	18
プラグイン	2	39.3
ツール	6	35.6

5.2　テストカバレッジと欠陥数，見逃し欠陥数の関係

本節では，RQ4 の分析結果から，プロジェクトのカバレッジが最も低いランク（0-9）のプロジェクトに最も多くの欠陥が混入していることがわかった．また，プロダクトコードのカバレッジが最も低いランク（0-9）のプロジェクトに最も多くの欠陥が混入していることがわかった．プロジェクトのテストカバレッジが高くなるほど，混入している欠陥数は少ないことがわかった．また，テストカバレッジが低いプロダクトコードにおいても，テストで網羅されている行からは欠陥はほとんど発見されず，テストで網羅されていない行から欠陥は発見されていることが調査でわかった．カバレッジはソフトウェアの品質を評価する上で重要な指標である．一方で，プロダクトコードのテストカバレッジが高いプロジェクトほど見逃し欠陥数が多いことがわかった．つまり，テストコードのカバレッジが高いことが必ずしもテストコードの品質が高いとは言い切れない．

5.3　研究の展望

本研究で特定した見逃し欠陥の特徴について分析することで，CI の精度を向上することが可能となる．また，本研究で分析したテストカバレッジと欠陥数，見逃

し欠陥数をクラスタリングする．開発者がソフトウェアを開発し，テストを実施する時，テストカバレッジをどの程度まで高めることで，テストを完了することができるのかという基準を作成することが可能である．

第 6 章　関連研究

近年，OSS を利用するユーザが増えており，OSS の品質に関心が集まっており，OSS の品質や品質保証活動に関する研究が盛んに行われている [8]．これまで，OSS 開発におけるテスト研究は，テスト結果などデータの不足により実態調査が主流であった．近年，TravisTorrent のようなビルドの実行結果などが公開されてきている．

6.1　OSS 開発におけるソフトウェアテストに関する研究

ソフトウェア工学の分野では，テスト手法やコードクローンなどソフトウェアテストの研究は盛んに行われている [17][2]．エンピリカルな分野では，OSS の開発記録を分析対象とし，ソフトウェアテストの実態調査などが行われている [20][7]．OSS 開発では，開発者にソフトウェアの品質を保証する責任がないことから，開発者主体のテスト活動（ソフトウェアテスト）より，利用者主体の検証活動（コードレビュー）が主流である．OSS 開発では，十分にソフトウェアテストが実施されていないのが現状である．Kochhar らは，20000 件の OSS プロジェクトのうち，約 4 割のプロジェクトでテストが実施されていないことを明らかにしている [14][18]．また，Kochhar らは 327 件の OSS を対象にテストカバレッジを調査した．その結果，テストカバレッジの平均は 41.96% であった [16]．このように，ソフトウェアテストが実施されている OSS において，十分にテストは実施されていないことがわかる．一方で，80% 以上の OSS 開発者はテスト活動が不足していることを認めており，テストが必要であると考えている [16]．Tosi ら [23] は，33 件の OSS プロジェクトから実践されているテスト手法や，テストドキュメントの有無を調査している．その結果，67% のプロジェクトではテストのドキュメントが存在しておらず，テストのドキュメントが存在していても，情報が更新されていなかったり，十分な情報が掲載されていないことがわかった．さらに，Takasawa らは [22]，テストコードを公開している OSS プロジェクト 791 件を対象に，テストを実行した．その結果，ビルドが成功したプロジェクトは 452 件 (57.1%) であり，その 452 件を対象にテストの実行した結果，全てのテストケースが成功したプロジェクトは 117

件 (14.8%) であった．OSS 開発におけるテスト実施状況に着目しているが，本論文では，欠陥数，見逃し欠陥数に基づく，CI 実施によるテストの有効性を分析しているため，目的が異なる．

6.2 継続的インテグレーション（CI）に関する研究

CI はアジャイル開発などで実施されており，欠陥の早期発見や品質の維持に寄与する手法であることから，近年，OSS 開発においても CI を実施するプロジェクトが増加している．CI の実態調査や CI を支援するツールの開発に関する研究が盛んに行われている [5]．Michael ら [15] は，34000 件の OSS プロジェクトを対象に，CI の実施状況を調査した．調査の結果，約 4 割のプロジェクトで CI が実施されていることを明らかにした．HIlton[11] らは，CI を実施ている OSS プロジェクトを対象に，CI を実施しているプロジェクトの，特徴，実施の動機，コスト，利点について調査した．Beller ら [3] は，CI におけるテストフェーズの研究が不足していることから，TravisTorrent のデータセットを公開し，TravisTorrent を分析対象に準備実験を行った．CI を実施しているプロジェクトと CI を実施してないプロジェクトを対象にテストの実行結果を調査した．CI を実施しているプロジェクトの方が CI を実施していないプロジェクトよりテストが失敗している回数が多いことから，CI を実施しているプロジェクトの方がより多くの欠陥を検出することが可能であることを示した．先行研究では，OSS における CI の調査という本論文と同じ動機に対して，CI の実施状況や CI 実施の動機など実態調査という観点で調査が行われている．

6.3 ソフトウェアの品質評価に関する研究

ソフトウェアの品質を評価するメトリクスは様々である [6][9]．コードカバレッジは，ソフトウェアの品質やテストの妥当性を評価する上で利用されるメトリクスの一つである．ミューテーションスコアは，プロダクトコードに人工的に作成した欠陥を埋め込み，テストを実行することによって，テストによる欠陥の検出数をスコア化したものである．Kochhar ら [15] と InozemtsevaI ら [13] は，コードカバ

レッジと欠陥検出率の相関関係を実際の欠陥やミュータントバグを利用して分析している．また，品質を評価するメトリクスとして，Hossain ら [12] は，OSS の品質をプロジェクト単位で修正された欠陥の数で評価を行っている．我々は，CI で実施されるテストの有効性を欠陥数，見逃し欠陥数とカバレッジに基づき分析を行った．

第 7 章　制約

外的妥当性: 本研究では，28 件の OSS プロジェクトを対象としてケーススタディを行ったが，結果の一般性を示すために，今後はプロジェクトの件数を増やして同様の分析を行う必要がある．

内的妥当性:　本研究では，Sliwerski ら [21] らが提案した SZZ アルゴリズムをもとに，プロダクトコードに混入した欠陥を特定したが，本手法には制約がある．開発者の習慣により，バグ ID が記載されていないコミットメッセージが Git には存在するため，ソフトウェアに混入している全ての欠陥を特定することは不可能である．また，GitHub の issue tracker に保存されている欠陥報告票には，'bug' のタグが付与されていないプロジェクトが多数存在する．この場合，欠陥報告票が作成された日時を特定できないため，SZZ アルゴリズムを適用できず，プロダクトコードに混入している欠陥を特定することができない．さらに，見逃し欠陥の特定について，テストの実行結果であるカバレッジレポートを利用する．カバレッジレポートはテスト自動化ツール Jmockit を実行することで，生成される．プロダクトコード，またはテストコードが削除されており，テストが実行できないプロジェクトが存在するために，欠陥は特定可能だが，見逃し欠陥は特定できないプロジェクトが存在した．

構成概念妥当性: SZZ アルゴリズムで特定した欠陥は，行レベルでは特定不可能であり，a 行目から b 行目といったような複数行の中に混入する欠陥を特定可能である．そのため，欠陥特定の粒度を高くするために，SZZ アルゴリズムを改良し，欠陥を特定する必要がある．

第 8 章　おわりに

近年，企業などでの OSS の導入に伴い，OSS の品質に注目が集まっている．OSS の品質向上を目的に CI を実施するプロジェクトが増加している中で，従来研究では CI の有効性をテストが失敗した回数に基づき検証した．本論文では，28 件の OSS プロジェクトを対象に，欠陥数，見逃し欠陥数，カバレッジを調査した．欠陥数，見逃し欠陥数とカバレッジの関係を分析することで，CI のテストの有効性を明らかにした．

プロジェクトのテストカバレッジが高くなるほど, 混入している欠陥数は少ないことがわかった．また，テストカバレッジが低いプロダクトコードにおいても, テストで網羅されている箇所には欠陥が混入していないことが明らかとなった．CI はテスト対象箇所の欠陥を検出するうえで有効であることがわかった．本研究を適用することで，より多くの欠陥を検出することが可能となり，OSS の品質向上につながると考えられる．

謝辞

本研究を進めるにあたり，多くの方々にご指導，ご協力，ご支援を賜りました．ここに謝意を添えてお名前を記させていただきます．

主指導教員であり，本論文の審査委員を務めていただいた奈良先端科学技術大学院大学　情報科学研究科　松本健一教授に対し，厚く御礼申し上げます．研究に対する直接的な指導に限らず，研究に取り組む上での心構えから研究者としての在り方まで，いろいろな面で熱心なご指導を賜りました．先生のご尽力に心より感謝申し上げます．

副指導教官であり，本論文の審査委員を務めていただいた奈良先端科学技術大学院大学　情報科学研究科　安本慶一教授には，学内発表において研究の質を高める上で非常に貴重なご意見を頂きました．心より感謝申し上げます．

奈良先端科学技術大学院大学　情報科学研究科　伊原彰紀助教には，研究の進め方から，論文執筆，プレゼンテーション作法まで何度も明敏なご指導を頂きました．伊原先生から頂いた数々の指導は私の将来の財産になると確信しております．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学研究室の皆様とは, 日頃より苦楽を共にさせて頂きました. お陰様で非常に楽しく充実した二年間を過ごすことができました．本当にありがとうございました．

参考文献

[1] Beller, M., Gousios, G., Panichella, A. and Zaidman, A.: When, How, and Why Developers (Do Not) Test in Their IDEs, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 179–190 (2015).

[2] Beller, M., Gousios, G. and Zaidman, A.: How (Much) Do Developers Test?, *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15 (2015).

[3] Beller, M., Gousios, G. and Zaidman, A.: Oops, my tests broke the build: An analysis of Travis CI builds with GitHub, *PeerJ PrePrints*, Vol. 4, p. e1984 (2016).

[4] Beller, M., Gousios, G. and Zaidman, A.: TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration, *Proceedings of the 14th working conference on mining software repositories* (2017).

[5] Brandtner, M., Giger, E. and Gall, H.: Supporting continuous integration by mashing-up software quality information, *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)* (2014).

[6] del Bianco, V., Lavazza, L., Morasca, S., Taibi, D. and Tosi, D.: The QualiSPo Approach to OSS Product Quality Evaluation, *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '10 (2010).

[7] Dimov, A., Chandran, S. K. and Punnekkat, S.: How Do We Collect Data for Software Reliability Estimation?, *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, CompSysTech '10 (2010).

[8] Ezeala, A., Kim, H. and Moore, L. A.: Open Source Software Development:

Expectations and Experience from a Small Development Project, *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46 (2008).

[9] Gopinath, R., Jensen, C. and Groce, A.: Code Coverage for Suite Evaluation by Developers, *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 72–82 (2014).

[10] Gousios, G.: The GHTorent Dataset and Tool Suite, *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13 (2013).

[11] Hilton, M., Tunnell, T., Huang, K., Marinov, D. and Dig, D.: Usage, Costs, and Benefits of Continuous Integration in Open-source Projects, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pp. 426–437 (2016).

[12] Hossain, L. and Zhou, D.: Measuring OSS Quality Trough Centrality, *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '08 (2008).

[13] Inozemtseva, L. and Holmes, R.: Coverage is Not Strongly Correlated with Test Suite Effectiveness, *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 435–445 (2014).

[14] Kochhar, P. S., Bissyandé, T. F., Lo, D. and Jiang, L.: Adoption of Software Testing in Open Source Projects–A Preliminary Study on 50,000 Projects, *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 353–356 (2013).

[15] Kochhar, P. S., Thung, F. and Lo, D.: Code coverage and test suite effectiveness: Empirical study with real bugs in large systems, *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015).

[16] Kochhar, P. S., Thung, F., Lo, D. and Lawall, J.: An Empirical Study on the Adequacy of Testing in Open Source Projects, *21st Asia-Pacific Software Engineering Conference*, pp. 215–222 (2014).

[17] Orso, A. and Rothermel, G.: Software Testing: A Research Travelogue

(2000–2014), *Proceedings of the on Future of Software Engineering*, FOSE 2014 (2014).

[18] Pavneet Singh KOCHHAR, Tegawende F. Bissyande, D. L. and JIANG, L.: An Empirical Study of Adoption of Software Testing in Open Source Projects (2013).

[19] Romo, B. A. and Capiluppi, A.: Towards an Automation of the Traceability of Bugs from Development Logs: A Study Based on Open Source Software, *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15 (2015).

[20] Salvaneschi, P.: System Testing of Repository-style Software: An Experience Report, *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16 (2016).

[21] Śliwerski, J., Zimmermann, T. and Zeller, A.: When Do Changes Induce Fixes?, *SIGSOFT Softw. Eng. Notes*, Vol. 30, No. 4, pp. 1–5 (2005).

[22] Takasawa.R, Sakamoto.K, Ihara.A, Washizaki.H and Fukuzawa.Y: Do open source software projects conduct tests enough?, *Proc. of the 15th International Conference of Product Focused Software Development and Process Improvement (PROFES'14)*, pp. 322–325 (2014).

[23] Tosi, D. and Tahir, A.: *A Survey on How Well-Known Open Source Software Projects Are Tested* (2013).

[24] Vasilescu, B., Yu, Y., Wang, H., Devanbu, P. and Filkov, V.: Quality and Productivity Outcomes Relating to Continuous Integration in GitHub, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 805–816 (2015).

発表リスト

[1] 南智孝，伊原彰紀，坂口英司，松本健一, “プロダクトコード変更に伴い共進化するテストコード特定手法の提案” ウィンターワークショップ 2016・イン・逗子 論文集 volume 2016, pp.51-52, 2016 年 1 月.
[2] 南智孝，坂口英司，伊原彰紀，松本健一, “継続的インテグレーションを導入している OSS のテスト結果の信頼性の検証” ウィンターワークショップ 2017・イン・飛騨高山 論文集 volume 2017, pp.39-40, 2017 年 1 月.