NAIST-IS-DD0961001

博士論文

OSS の不具合修正プロセス効率化のための コミッター推薦と不具合修正時期の予測

伊原 彰紀

2012 年 2 月 2 日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

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
博士 (工学) 授与の要件として提出した博士論文である。

伊原 彰紀

審査委員：

松本 健一 教授	（主指導教員）
鵜林 尚靖 教授	（九州大学）
関 浩之 教授	（副指導教員）
門田 暁人 准教授	（副指導教員）
大平 雅雄 助教	（副指導教員）

OSSの不具合修正プロセス効率化のための コミッター推薦と不具合修正時期の予測*

伊原 彰紀

内容梗概

本論文は，大規模オープンソースソフトウェア (OSS: Open Source Software) の不具合修正プロセスの効率化を目的として，不具合修正に貢献する有能なコミッターを見つけ出す方法，及び，不具合の修正時期を予測する方法を提案する．

(1) OSS プロジェクトのための有能なコミッターの予測

OSS プロジェクトでは，不具合が正しく修正されたか否かを検証する能力を持つ開発者（コミッター）が求められる．本論文では，不具合修正プロセスにおける開発者の活動量を用いて，多数の一般開発者の中からコミッターを予測するモデルを構築した．実験の結果，再現率が約 70%〜80%の精度でコミッターに昇格する開発者を予測できることが分かった．

(2) OSS 利用企業のための不具合修正時期の予測

OSS 利用企業は，修正を要する不具合が次期リリースまでに修正されるか否かを判断し，修正されないのであれば，自社で修正することを検討する必要がある．本論文では，不具合の特徴を示すメトリクス（ベース）に加えて，修正の進捗情報を示す 3 種類のプロセスメトリクス（状態，期間，参加者）を新たに提案し，OSS の不具合が次期リリースまでに修正されるか否かを予測するためのモデルを構築した．実験の結果，ベースメトリクスに加えて状態メトリクスを用いることで適合率が約 24%〜39%，再現率が約 14%向上することが分かった．

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DD0961001, 2012 年 2 月 2 日.

キーワード

オープンソースソフトウェア, コミッター予測, 不具合修正予測，不具合管理，リポジトリマイニング

Identifying Committers and Predicting the Bug Fix Resolution Period for Improving the Efficiency of the Bug Fixing Process of an Open Source Software Project*

Akinori Ihara

Abstract

The goal of this dissertation is to improve the efficiency of the bug fixing process of a large open source software (OSS) development, by (1) identifying a capable committer who contributes to fixing bugs and (2) predicting the period for resolving a bug by OSS developers.

(1) A capable committer prediction for OSS project

An OSS project requires some committers who have the ability to verify whether the OSS developers correctly fixed a bug or not. In this dissertation, we built a prediction model to identify a candidate committer using the developer activities in the OSS project. We found that our prediction model predicted the candidate committer with 70%-80% recall.

(2) A bug fix prediction for OSS users

When company developers detect a bug in OSS, they need to know whether the OSS developers will fix the bug. Even if the OSS developers will not fix the bug, the company developers need to consider fixing the bug themselves. In this dissertation, we built a bug fix prediction model based on the characteristics of

* Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0961001, February 2, 2012.

the bug (base metrics) and the progress of the bug fix (state, time, and developer metrics). As a result, our bug fixing model using both base metrics and state metrics showed better precision (24%-39%) and recall (14%) than a model using only base metrics.

Keywords:

Open Source Software, Committer Prediction, Bug Fix Prediction, Bug Management, Repository Mining

関連発表論文

学術論文誌

1. 伊原 彰紀, 大平 雅雄, 松本 健一．OSS 開発における不具合修正プロセスの現状と課題 ：不具合修正時間の短縮化へ向けた分析．情報社会学会誌, Vol.6, No.2, pp.1-12, 2011.（第 3 章に関連する）

2. 伊原 彰紀, 亀井 靖高, 大平 雅雄, 松本 健一, 鵜林 尚靖．OSS プロジェクトにおける開発者の活動量を用いたコミッター候補者予測．電子情報通信学会論文誌, Vol.J95-D, No.2, pp.237-249, 2012.（第 4 章に関連する）

国際会議発表

1. Akinori Ihara, Masao Ohira, and Kenichi Matsumoto. An analysis method for improving a bug modification process in open source development. In Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol'09). pp.135-143, 2009.（第 3 章に関連する）

2. Akinori Ihara, Masao Ohira, and Kenichi Matsumoto. Differences of time between modification and re-modification: an analysis of a bug tracking system. In Proceedings of the 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD'09), pp.17-22, 2009.（第 3 章に関連する）

査読付き国内研究集会発表

1. 伊原 彰紀, 大平 雅雄, 松本 健一. 不具合管理システム利用時の不具合修正プロセス改善のための滞留時間分析手法の提案. 情報処理学会 マルチメディア, 分散，協調とモバイル (DICOMO'09) シンポジウム論文集, pp.1221-1227, 2009.（第 3 章に関連する）

2. 伊原 彰紀，藤田 将司，大平 雅雄，松本 健一．OSS 開発におけるコミッター選出のための開発者の活動量に関する実証的分析. 門田暁人，上野秀剛（編），ソフトウェア工学の基礎 XVIII—日本ソフトウェア科学会 FOSE2011—, pp.81-90, 2011.（第 4 章に関連する）

その他の発表論文

学術論文誌

1. 亀井 靖高, 大平 雅雄, 伊原 彰紀, 小山 貴和子, 柗本 真佑, 松本 健一, 鵜林 尚靖．グローバル環境下における OSS 開発者の情報交換に対する時差の影響．情報社会学会誌, Vol.6, No.2, pp.13-30, 2011.

国際会議発表

1. Anakorn Jongyindee, Masao Ohira, Akinori Ihara, and Kenichi Matsumoto. Good or bad committers? a case study of committers' cautiousness and the consequences on the bug fixing process in the eclipse project. In Proceedings of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM/Mensura'11), pp.116-125, 2011.

2. Passakorn Phannachitta, Pijak Jirapiwong, Akinori Ihara, Masao Ohira, and Kenichi Matsumoto. An analysis of gradual patch application? a better explanation of patch acceptance. In Proceedings of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM/Mensura'11), pp.106-115, 2011.

3. Anakorn Jongyindee, Masao Ohira, Akinori Ihara, and Kenichi Matsumoto. A case study of the consequences from committers' activities on the bug fixing process in the eclipse project. In Proceedings of the 3rd International Workshop on Empirical Software Engineering in Practice (IWESEP'11), pp.9-14, 2011.

4. Passakorn Phannachitta, Pijak Jirapiwong, Akinori Ihara, Masao Ohira, and Kenichi Matsumoto. Understanding OSS openness through relation-

ship between patch acceptance and evolution pattern. In Proceedings of the 3rd International Workshop on Empirical Software Engineering in Practice (IWESEP '11), pp.37-42, 2011.

5. Soichiro Tani, Akinori Ihara, Masao Ohira, Hidetake Uwano, and Kenichi Matsumoto. A system for information integration between development support systems. In Proceedings of the 3rd International Workshop on Empirical Software Engineering in Practice (IWESEP'11), pp.33-34, 2011.

6. Shoji Fujita, Masao Ohira, Akinori Ihara, and Kenichi Matsumoto. An analysis of committers toward improving the patch review process in OSS development. In Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10), pp.369-374, 2010.

7. Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Kenichi Matsumoto. Predicting re-opened bugs: a case study on the eclipse project. In Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10), pp.249-258, 2010.

8. Masao Ohira, Kiwako Koyama, Akinori Ihara, Shinsuke Matsumoto, Yasutaka Kamei, and Kenichi Matsumoto. A time-lag analysis toward improving the efficiency of communications among OSS developers. In Proceedings of the 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD'09), pp.49-62, 2009.

査読付き国内研究集会発表

1. 藤田 将司, 伊原 彰紀, 大平 雅雄, 松本 健一. OSS 開発におけるパッチレビュープロセスの効率化に向けたコミッターの分類. 平成 22 年度 情報処理学会関西支部支部大会 講演論文集, Vol.2010, 2010.

2. 伊原 彰紀, 山本 瑞起, 大平 雅雄, 松本 健一. OSS 開発における保守対応の効率化のためのアウェアネス支援システム. 情報処理学会 マルチメディア, 分散, 協調とモバイル (DICOMO'10) シンポジウム, pp.1620-1629, 2010.

3. 小山 貴和子, 伊原 彰紀, 柗本 真佑, 亀井 靖高, 大平 雅雄, 松本 健一. OSS 開発における情報交換の効率改善へ向けたタイムラグ分析手法の提案. 情報処理学会シンポジウム　グループウェアとネットワークサービスワークショップ 2009 論文集, Vol.2009, No.8, pp.81-86, 2009.

4. 大平 雅雄, 柗本 真佑, 伊原 彰紀, 松本 健一. オープンメディアを活用した知識コミュニティのデザインに関する一考察. 情報社会学会 「知識共有コミュニティワークショップ ―インターネット上の知識検索サービス研究―」, pp.1-10, 2008.

5. 伊原 彰紀, 大平 雅雄, 柗本 真佑, 亀井 靖高, 松本 健一. 複数のサブコミュニティを有する OSS コミュニティを対象としたネットワーク分析. 情報処理学会 マルチメディア, 分散, 協調とモバイル (DICOMO'08) シンポジウム, pp.297-303, 2008.

国内研究集会発表

1. 伊原 彰紀, 大平 雅雄, 柗本 真佑, 松本 健一. OSS の開発状況理解支援のための可視化手法の提案. 情報処理学会シンポジウム　グループウェアとネットワークサービス・ワークショップ 2007 論文集, pp.63-64, 2009.

2. 伊原 彰紀, 大平 雅雄, 小山貴和子, 松本 健一. OSS の障害修正における開発者ネットワークの分析. 第 5 回ネットワーク生態学シンポジウム予稿集, pp.258-259, 2009.

3. 伊原 彰紀, 大平 雅雄, 松本 健一. 障害管理システム利用時の修正遅延要因の分析. ソフトウェア信頼性研究会 第 5 回ワークショップ論文集, pp.75-82, 2009.

4. 伊原 彰紀, 亀井 靖高, 大平 雅雄, 柗本 真佑, 松本 健一. OSS プロジェクトにおける障害に関する情報共有の分析. 情報処理学会関西支部　支部大会講演論文集, Vol.2008, pp.69-72, 2008.

5. 伊原 彰紀, 前島 弘敬, 柗本 真佑, 亀井 靖高, 大平 雅雄, 松本 健一. 複数のサブコミュニティを有する OSS コミュニティにおけるコーディネータの分析. 情報処理学会シンポジウム　グループウェアとネットワークサービス・ワークショップ 2007 論文集, Vol.2007, No.11, pp.13-18, 2007.

目 次

図 目 次

表 目 次

第1章　序章

1.1 本論文の背景

オープンソースソフトウェア (OSS: Open Source Software) は，誰でも自由に利用，改変，再頒布することができるため，国内外問わず多くのソフトウェア企業等で利用されている．OSS が社会基盤を支える高機能なソフトウェアに成長した大きな要因は，商用ソフトウェアと異なる開発形態「バザール方式」で開発されたことである [10][47]．Eric Raymond は著書「The Cathedral and the Bazaar（伽藍とバザール）」で，Linux プロジェクトを成功に導いた開発形態「バザール方式」について説明している [47]．バザール方式の特徴を以下に示す．

1. 開発者が自由に参加/離脱することが可能

   不特定多数の開発者が，世界中の参加者との協調作業の魅力，及び，開発技術の学習等を動機として OSS プロジェクトに参加している．開発者は，各自の意思で自由に参加/離脱することが可能である [28][65]．

2. 利用者参加型のプロジェクト

   OSS プロジェクトでは，開発者と利用者が直接コミュニケーションを行い，機能拡張や不具合に関するフィードバックを受けている．OSS の利用者が多いほど，OSS プロジェクトは数多くのフィードバックを受けることができ [12]，利用者は，プロジェクトにおいて重要な役割を果たしていることから，共同開発者と位置付けられている [65]．

3. 頻繁なリリース

   多くの OSS プロジェクトでは，メジャーリリース（新機能の追加，及び，ユーザインタフェースの大幅な変更）やマイナーリリース（不具合修正，及び，細かな変更）が頻繁に行われている．OSS プロジェクトはリリース回数を増やすことによって，利用者，及び，開発者から受けた機能拡張要求や不具合報告を反映したプロダクトを早期にリリースすることができる．このことは，利用者，及び，開発者のモチベーションの維持に効果がある．

Linux プロジェクトは,「バザール方式」で開発が進められ，全世界の 300 万人以上の利用者と 4 万人以上の開発者が携わる大規模プロジェクトへと成長した [49]．近年では，Apache，Mozilla，Eclipse 等，数多くの OSS がバザール方式による開発で，大規模なソフトウェアへと成長している．

1.2 本論文の動機と目的

OSS が大規模化するにつれて，多くの不具合が OSS プロジェクトに報告されている [7]．不具合の中には，長期間解決されていない不具合が多数存在し [6][29][45]，深刻度の高い不具合も存在している [63]．今後も不具合の増加が予想されるため，開発者には膨大な不具合を修正するための一連の作業（不具合修正プロセス）を見直す必要性が高まっている．

OSS を自社業務やソフトウェア開発に利用している企業は数多く存在し，早期に解決されない不具合は企業にとって大きな課題となっている [64]．OSS を利用する企業（以降，OSS 利用企業とする）は，OSS に発見した不具合を自社で修正することもあるが，修正コスト削減のために OSS プロジェクトで修正されることを望んでいる．

OSS の数多くの不具合を修正し，高品質な OSS を実現するためには，OSS プロジェクトによる不具合修正プロセスの改善だけでなく，プロジェクトで修正されない不具合を，OSS 利用企業も修正する必要性が高まっている．本論文では，大規模 OSS プロジェクト，及び，OSS 利用企業の各々の立場における不具合修正プロセスの効率化を目的とする．

OSS プロジェクトにおける不具合修正プロセスの効率化

OSS プロジェクトで修正されていない不具合が数多く存在している原因として，膨大な不具合を修正する開発者の不足が指摘されている [13]．通常，OSS プロジェクトによる不具合の修正は，図 1.1 において OSS プロジェクトの修正依頼者がプロジェクトの修正担当者に修正依頼を行うプロセスで修正されており，修正依頼者とレビューアの役割は，OSS プロジェクトのコミッター[1] と呼ばれる限られた開発者のみで担当している．コミッターは不具合修正プロセスにおいて重要な役割を担っているが，バザール方式の特徴から，コミッターは将来的に活動し続けるとは限らないため，既存コミッターは継続的に有能なコミッターを一般開発者の中から選出する必要がある．よって，大規模 OSS プロジェクトの不具合修正に貢献するコミッターを推薦するために，一般開発者の中から有能なコミッター候補者を見つけ出す方法を提案する．

OSS 利用企業における不具合修正プロセスの効率化

OSS プロジェクトで不具合が長期間修正されない場合，OSS 利用企業が OSS の不具合を自社で修正せざるを得ない．OSS 利用企業による不具合修正は，図 1.1 において OSS プロジェクトで不具合が長期間修正されない場合，OSS 利用企業の修正担当者が不具合を引き受けるプロセスで修正される．OSS 利用企業は，OSS プロジェクトで修正されるか否かを，一般に公開されている不具合の情報に基づいてプロジェクトの修正計画を予想することになるが，OSS プロジェクトは，進捗情報のみを提示し，今後の計画を示していない．よって，OSS 利用企業が修正する必要のない不具合を見極めるために，不具合の特徴，及び，修正の進捗情報に基づき，修正を要する不具合が OSS プロジェクトで次期リリースまでに修正されるか否かを予測する方法を提案する．

本論文では，大規模 OSS プロジェクトの不具合修正に貢献する有能なコミッターを選出するために，コミッター予測モデルを構築し，大規模 OSS プロジェクトを対象にモデルの有効性を確認した．図 1.1 の修正依頼と修正された不具合の

[1] OSS プロジェクトには，バージョン管理システムにコミットするための権限（コミット権限）を与えられている開発者（コミッター）と与えられていない開発者（一般開発者）が存在する．

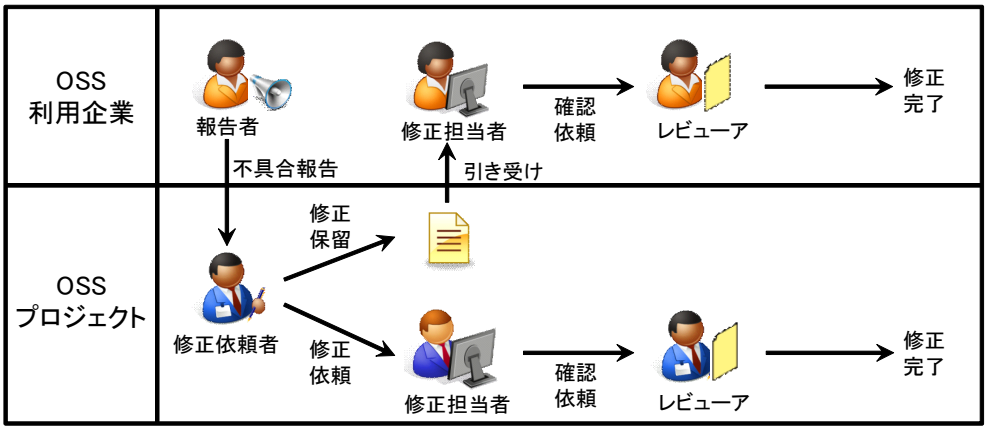


図 1.1 OSS の不具合修正プロセス

図 1.2 OSS の不具合修正に関わる参加者

確認の役割を務める有能なコミッターの推薦を容易にすることで，効率的に不具合の修正が実施され，不具合修正の効率的な実施が期待される．また，OSS プロジェクトにおいて，修正を要する不具合が次期リリースまでに修正されるか否かを予測するために不具合修正予測モデルを構築し，コミッター予測モデルと同様に不具合修正予測モデルの有効性を確認した．OSS 利用企業の利用者が OSS の不具合の修正時期を把握することにより，図 1.1 中の OSS 利用企業における修正担当者が，余分な修正コストをかけずに，不具合の修正を効率的に実施できると期待される．

1.3 論文構成

本論文は以降 5 つの章から構成される．続く第 2 章では，本論文が対象としている OSS プロジェクトの不具合修正について述べる．特に，不具合修正プロセスと不具合修正プロセス中の各作業に関わる開発者の役割について詳しく述べる．そして，不具合修正プロセスの効率化を目的とした関連研究を紹介する．

第 3 章では，続く第 4 章の「OSS プロジェクトのための有能なコミッター予測」，及び，第 5 章の「OSS 利用企業のための不具合修正時期の予測」を行う動機となった分析を述べる．大規模 OSS プロジェクトでは，膨大な不具合が報告され，修正が効率的に実施されていない．本章では，その原因を把握するために，大規模 OSS プロジェクトが管理する不具合管理システム（BTS: Bug Tracking System）に記録される不具合修正の進捗情報を用いて，過去の不具合修正に必要となった一連の作業内容や個々の作業時間の可視化と分析を行った．分析から，コミッターにかかる負担，及び，不具合修正プロセス中の各作業の効率化に向けた方策について述べる．

第 4 章では，大規模 OSS における膨大な不具合の修正を実施するために，プロジェクトに参加する一般開発者の中からコミッターに推薦されるべき有能な開発者を見つけ出すためのモデル構築方法について述べる．

第 5 章では，OSS 利用企業における不具合修正プロセスの効率化のために，修正を要する不具合が次期リリースまでに修正されるか否かを予測する方法を述べる．最後に第 6 章で，本研究を総括し，本論文をまとめる．

第2章　OSSプロジェクトにおける不具合修正

本章では，本論文で対象とするOSSの不具合修正，不具合がプロジェクトに報告されてから解決されるまでの一連の修正作業（不具合修正プロセス），そして，不具合修正に関わる開発者，について述べる．次に，OSSの不具合修正に関するOSSプロジェクトの課題とOSS利用企業の課題をそれぞれ述べ，最後に，不具合修正プロセスの効率化を目的とした関連研究について述べる．

2.1 不具合修正

本論文が対象とするOSSの不具合修正は，JIS規格 (JIS X 0161:2008)，及び，ISO規格 (ISO/IEC 14746:2006) で定義されているソフトウェア保守の一部である．それぞれの規格において，ソフトウェア保守は「ソフトウェアの完全性を維持しながら，既存のソフトウェア製品に対して修正を行うこと」と定義されており [66]，ソフトウェア運用以前の開発計画から，ソフトウェアを廃棄するまでを指している．本論文で対象とする不具合修正は，ソフトウェア保守を行う過程で，プロジェクトに不具合が報告されてから，当該不具合が解決するまでを対象とし，「修正計画」,「修正」,「検証」の3つの作業フェーズがある．

修正計画フェーズ：報告された不具合の再現性を確認し，修正すべき不具合か否かを確認する．修正すべき不具合であれば，可能な解決策を検討する．そして，適切な修正担当者に依頼する．

修正フェーズ：修正することによる影響範囲を分析し，問題がなければ修正を実施する．

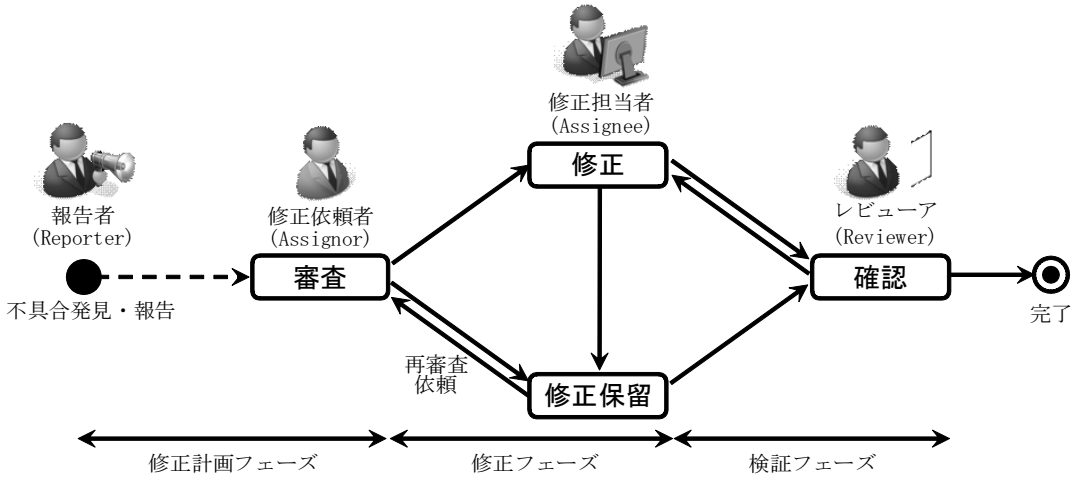


図 2.1 OSS プロジェクトにおける不具合修正プロセス

検証フェーズ：修正が正しく，且つ，問題なく完了しているか否かを確認し，修正されたソフトウェアの完全性が確認されれば，修正を承認し，プロダクトに加えた変更をバージョン管理システムにコミットする．

本節では，OSS プロジェクトに不具合が報告されてから解決されるまでの不具合修正プロセスとプロセスに関わる開発者について述べる．

2.1.1 不具合修正プロセス

不具合修正プロセスは，不具合がプロジェクトに報告されてから正しく修正されたことが確認されるまでの一連の作業からなる．図 2.1 は，OSS プロジェクトにおける一般的な不具合修正プロセスである.「修正計画フェーズ」は，不具合が報告されてから修正に向けた活動開始（担当者の決定，及び，修正すべき不具合の決定等），もしくは修正保留の決定までの期間を指す.「修正フェーズ」は，担当者が決定されてから修正が終了するまでの期間，もしくは修正が必要ないかを確認するまでの期間を指す.「確認フェーズ」は，修正内容，及び，修正が今後必要ないかを確認する期間を指す．以下に，不具合修正プロセス中の各作業について述べる．

不具合発見・報告 : OSS に不具合が発見されると，開発者や利用者が OSS プロジェクトの不具合管理システムに不具合の特徴をプロジェクトに報告する．

審査 : プロジェクトに不具合が報告された後，修正依頼者が当該不具合を修正すべきか否か判断する．

修正 : 修正依頼者が不具合を修正すべきと判断した後，修正担当者を決定し，修正担当者が不具合の修正を行う．

検証 : 修正担当者が修正を完了した後，レビューアが修正内容を確認する．または，修正保留と判断された不具合をレビューアが再度修正の必要がないかを確認する．

修正保留 : 報告内容が再現できない，実装するとパフォーマンスが下がる，修正依頼者や修正担当者からの問い合わせに対して報告者から応答がないような場合，修正依頼者が修正を実施しないと判断する，もしくは修正担当者が修正を中断する．

完了 : レビューアによって正しく修正されたと判断された場合，もしくは今後修正することがないと決定された場合，不具合の修正を終了する．

2.1.2 不具合修正に関わる開発者

OSS プロジェクトにおける不具合修正プロセス中の修正依頼，修正された不具合の確認はコミッターと呼ばれる開発者が行っており，OSS の不具合修正を行うために重要な役割を担っている．OSS プロジェクトにおいてコミッターは，プロダクトに加えた変更をバージョン管理システムにコミットするための権限（コミット権限）が与えられている開発者を指す [20]．コミット権限が与えられていない開発者を一般開発者と呼ぶ．多くのプロジェクトでは，プロダクトに不具合が混入した場合でもその原因の所在を突き止めやすくするために一部の開発者のみにコミット権限を与えている．OSS プロジェクトでは，高い技術力や一般開発者を取りまとめる能力を持っていると認められた一般開発者が，既存コミッターからの推薦と承認を経てコミッターに昇格する [31]．

2.2 OSS の不具合修正における課題

OSS の大規模化に伴い，OSS の膨大な不具合を開発者だけで対処することが困難になっている [27]．本節では，大規模 OSS の不具合修正を行う上での課題を OSS プロジェクトと OSS 利用企業の観点から述べる．

2.2.1 OSS プロジェクトにおける課題

OSS プロジェクトにおける多くの開発者はプロジェクトに長期間参加する責務が無いため，数カ月しか参加しないことが多い．また，OSS の不具合修正に積極的に取り組む開発者は少なく，OSS の不具合修正において重要な役割を担うコミッターに過度な負担がかかっている [13][51]．コミッターを中心とする少人数の開発者が不具合修正を行っているが，報告される不具合は日々増加しているため，解決されないまま長期間経過している不具合が数多く存在している [6][29][45]．その中には，OSS にとって深刻度が高い不具合も存在している [63]．将来，コミッターが継続的に活動するとは限らないため，OSS の不具合修正に貢献するコミッターを常に確保するための体制を整える必要性が高まっている．既存コミッターはプロジェクトに参加する一般開発者（大規模プロジェクトでは 1 万人を超えることも珍しくない）の中から，コミッターに推薦されるべき有能な開発者を見つけ出すことが重要な課題となっている [9][21][31]．

2.2.2 OSS 利用企業における課題

多くのソフトウェア開発企業が OSS を利用する理由として，低価格で高機能なソフトウェアが利用可能，開発スピードの向上等が挙げられている [64]．OSS を利用する企業が増加する一方で，OSS の不具合が早期解決していないことに不安を感じている企業も存在している．OSS 利用企業は，OSS の不具合を自社で修正することが可能であるが，修正には時間と労力がかかるため，OSS プロジェクトで修正されることを期待している．しかしながら，OSS プロジェクトで修正されない不具合が数多く存在する．例えば，一部の利用者の環境においてのみ発生する不具合や OSS に実装するとパフォーマンスが低下するような不具合は OSS プ

ロジェクトで修正されない．このような不具合は，OSS 利用企業が自社で修正することになるため，OSS プロジェクトで修正されるか否かを見極めることが必要である．

2.3 不具合修正プロセスの効率化を目的とした関連研究

OSS が普及するに伴い，OSS プロジェクトは数多くの不具合報告を利用者から得られるようになった．OSS プロジェクトへの不具合報告は，OSS の品質を高める上で重要な役割を担っているが [47]，不具合報告が増加するに伴い，報告に対処する開発者の負担は増加し，修正依頼者による修正担当者の決定や，レビューアによる修正内容の確認等に長い時間がかかっている [13][29]．このような背景から近年，不具合修正プロセスの効率化を目的とした研究が数多く行われている．

2.3.1 不具合修正プロセス中の作業状況の把握に関する研究

不具合修正プロセス中では，図 2.1 に示すように，審査，修正，確認の作業が行われており，これらの作業について過去の作業記録の分析や，現在の作業状況を把握するための研究が報告されている．

審査の作業に関する研究として，Bettenburg ら [7]，Just ら [34]，Rastkar ら [46] は，不具合が報告されてから修正に取り掛かるまでの作業を効率化するために，プロジェクトに報告される不具合票の質に関する分析を行っている．OSS プロジェクトへの不具合報告には，修正するための十分な情報が含まれていないことが多いため，修正に長い時間がかかっている．その原因の一つとして，Bettenburg ら [7] は，開発者と報告者を対象としたアンケート調査で，両者が不具合を修正するために有用であると考える記述に差が生じていることを挙げている．開発者は不具合を修正するために再現方法やテストケースの記述が有用であると考えているが，その一方で，報告者は不具合が発見されたプロダクトの種類や OS の種類が修正するために有用と考えている．

修正の作業に関する研究として，開発者が修正する不具合の偏在傾向や，修正の進捗状況を管理･把握支援するための可視化ツールが提案されている [17][18][38][50]．Sarma ら [50] は，OSS プロジェクトにおいて各モジュールを管理するコミッター

や不具合の修正担当者の把握を容易にするために各開発者が関与しているタスク，各タスクの進捗，モジュールの依存関係を可視化するツール Tesseract を提案している．

確認の作業に関する研究として，Nurolahzade ら [45] や Rigby ら [48] は，修正内容の検証にかかる時間や検証方法の有効性を分析している．OSS プロジェクトでは，一般開発者が不具合を修正した場合，修正内容をコミッターが確認する (RTC: Review-then-commit)．一方で，コミッターが不具合を修正する場合，修正内容を確認せずに OSS に反映され，不適切な内容であった場合のみ修正内容を確認する (CTR: Commit-then-review)．この RTC と CTR の効率性と有効性を Apache プロジェクトを対象に分析した結果，CTR は RTC より 2.2 倍早く確認を終了することができるが，確認による不具合の発見数に違いが見られなかった．

これまで，不具合修正プロセス中の各作業を効率的に実施することを課題とした研究が数多く行われているが，不具合修正プロセスを俯瞰的に把握し，各作業の状況を相対的に比較した研究はない．不具合修正プロセスを効率化するためには，長い時間かかっている作業を明らかにし，その作業を改善することが優先課題と言える．本論文では，不具合修正プロセスを俯瞰的に把握し，長い時間かかっている作業についての分析した結果を第 3 章で述べる．

2.3.2 不具合修正プロセスの効率化に関する研究

不具合修正プロセスの作業の一部を自動化することにより，不具合修正プロセスの効率化を実現する研究が報告されている．

審査，及び，修正の作業を効率化することを目的として，Anvic ら [2][3] や Davor ら [15] は，OSS プロジェクトに参加している開発者の中からタスクを適任の担当者に割り当てる労力を軽減するために，彼らは開発者が過去に担当した不具合に関するテキスト情報（不具合票に記載されたタイトルや詳細など）を基にモデルを構築し適当な修正担当者を予測する方法を提案している．また，Kim ら [37] は，プロジェクトのバージョン管理システムや不具合管理システムの記録から不具合を混入された時期と開発者を特定する手法を提案している．

また，確認の作業を効率化することを目的として Shihab ら [52] は，数多くの

修正された不具合の中から，修正記録に基づき，再修正を要する不具合の検出を自動化する方法を提案している．

OSS プロジェクトに報告される不具合が増加するに伴い，修正されない不具合が増加する傾向にあるため，不具合修正プロセスを見直すことが必要不可欠となっている．このような背景から，不具合修正プロセス中の一部の作業を自動化することによって，不具合修正プロセスの効率化を実現する方法が提案されている．また，大規模 OSS プロジェクトでは，修正されない不具合も数多く存在しているため [27]，OSS を利用するソフトウェア開発企業は，OSS プロジェクトによる不具合の修正を待つだけでなく，自社で修正せざるを得なくなっており，OSS プロジェクト，及び，OSS 利用企業の双方による不具合修正の必要性が高まっている．

第3章　OSS開発における不具合修正プロセスの現状分析

近年，OSS プロジェクトには膨大な不具合が報告され，修正に長期間かかっている．本章では，不具合修正プロセス中で修正に長期間かかっている作業を特定するために，OSS の不具合修正に必要となる一連の作業内容，及び，個々の作業時間を分析した結果について述べる．

3.1 はじめに

OSS プロジェクトへの不具合報告は，OSS の品質を高める上で重要な役割を担っており，本来歓迎されるべきものである [47]．しかしながら，大規模 OSS プロジェクトでは，報告された全ての不具合を短期間のうちに修正することは困難になりつつある [14]．

一般的な OSS 開発では，地理的に分散したボランティアの開発者同士が，メーリングリストや掲示板などの限られたコミュニケーションチャネルの中で議論を行いながら不具合の修正を行っている [1][11][67]．このような開発環境に加え，昨今の OSS 開発においては大量の不具合報告を処理する必要がある．そのため，各モジュールのコミッターが個々の不具合の内容を把握した上で適任の修正担当者を決定したり，不具合修正の進捗状況を鑑みて修正担当者毎に適宜指示やアドバイスを与えたりすることが難しくなっている．結果として大規模な OSS を開発する OSS プロジェクトで，不具合修正の長期化が問題となっている．OSS の社会的重要性は増す一方であり，不具合修正の長期化を解消するための方策が望まれている [6][32][34][50][57]．

本章では，大規模 OSS プロジェクトにおいて不具合修正が長期化する原因とな

る作業を明らかにし，不具合修正プロセスを効率化するための方法を検討する．ケーススタディとして，3 つの OSS プロジェクト（Apache HTTP Server，Eclipse platform，Mozilla Firefox）を対象とし，BTS に記録される不具合修正の進捗情報を用いて，過去の不具合修正に必要となった一連の作業内容や個々の作業時間の可視化・分析を試みる．

続く 3.2 節では，関連研究を紹介し，3.3 節で，OSS 開発における不具合管理とその問題点について述べる．3.4 節で，分析方法について詳述し，3.5 節では，ケーススタディの結果を示す．3.6 節では，分析結果から不具合修正プロセス中の各フェーズの効率化に向けた方策を検討する．最後に 3.7 節で本章のまとめを述べる．

3.2 関連研究

OSS の機能拡張が進むにつれ大規模かつ複雑になり，不具合報告も増加している．そのため，各モジュールのコミッターは効率的な不具合修正を実現するために，開発状況，不具合の修正状況，プロジェクトに参加する共同開発者などを把握することが容易ではなくなっており，近年，OSS 開発の現状を容易に把握するための研究が行われている [8][16][22][50]．

Macro ら [16] は，不具合修正プロセス中の修正フェーズで，開発者がソフトウェア中の不具合の偏在傾向を把握するために，ソースコードの変更と不具合の関係を可視化するシステムを提案している．具体的には，各ファイルのリビジョン履歴と当該ファイルに発見された不具合の修正状況の関係，各モジュールのコミット数の変化と不具合報告数の変化を可視化している．

また Wu ら [61] は，不具合修正プロセス中の検証フェーズで，コミッターが修正内容を確認すべき不具合を把握するために，バージョン管理システムにコミットされたソースコードと，コミットされた時期を時系列に可視化するシステムを提案している．

これまで各フェーズで修正状況を把握する方法が提案されてきた [7][17][34][48][50]．しかしながら，プロジェクトにとって最も効率化を要する作業を特定するためには，不具合修正プロセス中の全てのフェーズを俯瞰的に把握する必要がある．

3.3 OSS 開発における不具合管理

3.3.1 不具合の管理

一般的な OSS 開発では，世界中に点在する開発者同士が不具合に関する情報を共有するために，Bugzilla[2] ，Trac[3] ，RedMine[4] などの不具合管理システム (BTS: Bug Tracking System) が利用されている．BTS は，開発者あるいは利用者から報告された 1 つの不具合に対して 1 つの不具合票を作成し，個々の不具合修正の進捗を管理（追跡）するためのシステムである．また，Web ユーザインタフェースを介して誰でも不具合の報告と修正作業進捗の閲覧が行える点に特徴がある．

図 3.1 に BTS を用いた不具合修正の流れを示す．各不具合票には，不具合の基本情報（対象プロダクトやバージョン，重要度や優先度等），不具合の詳細情報（不具合ぶｇもの内容や不具合を再現するための手順），議論情報（不具合の内容や修正作業に関して行われる議論），状態遷移管理情報（不具合修正処理の状態管理を行うために記録される情報），の 4 種類の情報が記録される．

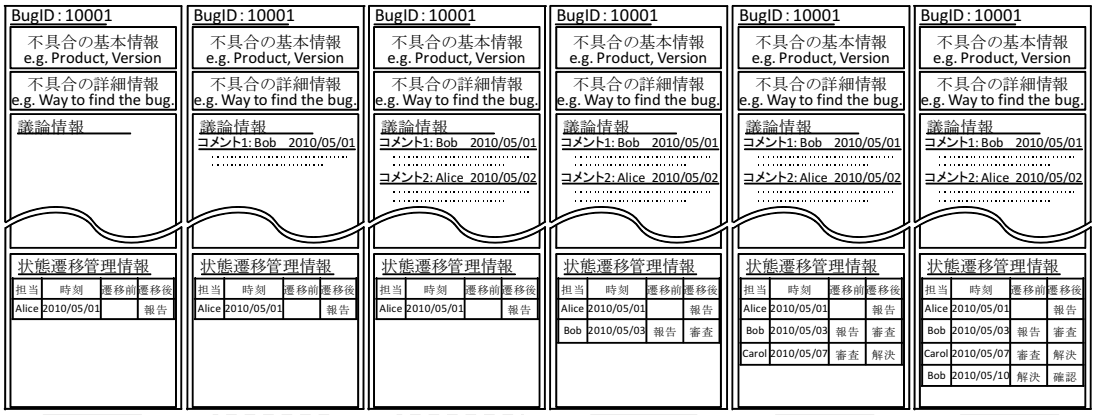
3.3.2 不具合修正プロセス

不具合修正プロセスは，不具合が報告されてから正しく修正されたことを確認するまでに必要とされる一連の作業からなる．図 3.2 は，Bugzilla プロジェクトが提示している修正の進捗情報に基づく不具合修正プロセスを示す．また，図 3.2 中の表の左列には，図 2.1 に示す OSS プロジェクトにおける不具合修正プロセスの状態との対応関係も示している．審査の状態は不具合が報告されてから修正に向けた取り組みが開始されるまでの期間（図 2.1 では,「議論開始」,「不具合承認」の作業が行われる期間）に該当する．修正の状態は修正が開始されてから修正作業が終了するまでの期間（図 2.1 では,「修正担当者決定」,「修正作業終了」,「再修正決定」の作業が行われる期間）に該当する．確認の状態は修正作業が終了後,

[2] Bugzilla: http://www.bugzilla.org/

[3] Trac: http://trac.edgewall.org/

[4] RedMine: http://www.redmine.org/



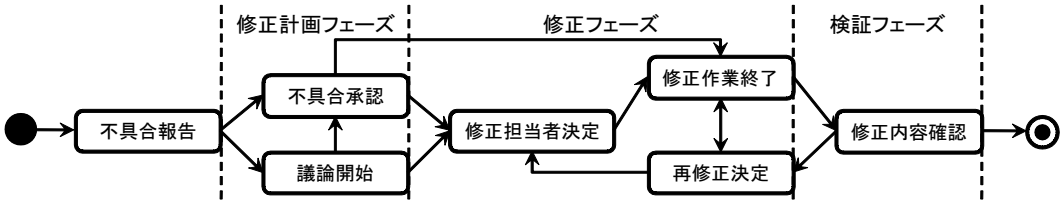
行動	役割	詳細
報告 (1)	報告者	開発者や利用者がOSSに不具合を発見した時，報告者は不具合票の「不具合の基本情報」と「不具合の詳細情報」に不具合に関する情報を記載する
コメント (2,3)	修正貢献者	開発者は報告者に不具合に関する追加情報の要求，類似・重複不具合票の提示，修正に関するアドバイスなどをする場合，コメントを書き込む.
審査 (4)	修正依頼者	報告された不具合を修正すべきか否かを審査する
解決 (5)	修正担当者	開発者が不具合の修正作業を行い，作業終了後に不具合が解決したことを記録する.
確認 (6)	レビューア	修正後，不具合が正しく修正されたか否かを確認する.

図 3.1 BTS を用いた不具合修正の流れ

修正作業の確認が完了するまでの期間（図 2.1 では,「修正内容確認」の作業が行われる期間）に該当する.

3.3.3 不具合管理の課題

OSS 開発においてコミッターは，不具合修正プロセスの 3 つの区間（修正計画フェーズ，修正フェーズ，検証フェーズ）それぞれで作業の進捗状況を把握する必要がある．修正計画フェーズでは，報告された不具合に関して議論が開始されているかどうか（不具合報告が誰の目にも触れられず放置されていないかどうか）を監視しなければならない．修正フェーズでは，修正担当者が適切に割り当てられているかどうかや，開発者が修正できずに思案していないかを把握する必要がある．検証フェーズでは，修正担当者によって修正された不具合の検証が滞



一般的な不具合の状態	Bugzillaプロジェクトが提示する不具合の状態	詳細
不具合発見・報告	不具合報告	開発者又は利用者によって発見された不具合が報告(BTSに登録)された状態.
審査	議論開始	BTSに登録された不具合情報に関して議論が開始された状態.
	不具合承認	議論の結果, 各モジュールのコミッターによって修正すべき不具合として承認された状態.
修正	修正担当者決定	各モジュールのコミッターによって修正担当者が決定された状態.
	修正作業終了	修正担当者によって修正作業が終了した状態.
	再修正決定	不具合の修正が不十分であり, 再度修正が要求された状態.
確認	修正内容確認	各モジュールのコミッターによって不具合が正しく修正されたことが確認された状態.

図 3.2 Bugzilla プロジェクトで提示している修正の進捗情報に基づく不具合修正プロセス

りなく進められているかどうかや，検証作業の結果，再修正が必要と判断された不具合が放置されていないかどうかを監視している．しかしながら，多くの大規模 OSS プロジェクトでは，数多くの不具合が報告されるため [14][32][58]，不具合を網羅的に監視することは困難である．

また，長期間修正にかかっている不具合の中には，深刻度が高い不具合（例えば，セキュリティホールなど，早急に解決すべき不具合）が含まれていることも多く [63]，各モジュールを担当しているコミッターは修正が滞りなく進んでいるかどうかを BTS の検索機能を用いて定期的に調査している．修正が滞る原因となる作業を把握するためには，各々の不具合票に記載されている情報から遅れの生じている作業を相対的に比較・検討する必要があり，不具合修正プロセス中の特に改善すべき修正作業（長時間かかっている作業）を特定することが課題である．

3.4 分析方法

3.4.1 概要

本節では，BTS に記録される不具合修正の進捗情報を用いて，不具合修正プロセスを状態遷移図で表す手順を示す．次に，過去の不具合修正で必要となった一連の作業内容や個々の作業時間を可視化・分析し，プロジェクトの不具合修正プロセスを効率化するための指針を検討する．分析では特に，プロジェクト特有の状態遷移や，時間を要する作業・フェーズを把握することに重点を置く．コミッターは，本章の分析方法を用いて自身のプロジェクトの不具合修正プロセスの問題点を発見できると期待される．

3.4.2 手順

3.4.2.1 状態遷移図の作成

状態遷移図の作成手順（図 3.3）について説明する．図中及び以下の説明では，具体例として Apache プロジェクトの BTS に実際に登録されている不具合の修正履歴情報を用いている．

1 状態遷移時刻表の作成

BTS に登録されている各不具合票の状態遷移管理情報から状態遷移と遷移時刻を取得する．また，議論情報から最初のコメントが投稿された時刻を取得し，状態遷移時刻表（図 3.3-(a)）を作成する．

2 状態遷移時間表の作成

各不具合票の各状態遷移にかかる時間を算出し，状態遷移時間表（図 3.3-(b)）を作成する．その時,「不具合報告」から次の状態に遷移するよりもコメントされる時刻の方が早い場合,「議論開始」を経由して，次の状態に遷移することとする．例えば状態遷移時刻表（図 3.3-(a)）では,「不具合報告」→「不具合承認」の順で状態遷移が記録されていても,「不具合承認」より

前に議論（コメント）が開始されていれば,「不具合報告」→「議論開始」→「不具合承認」の順で状態を遷移したものとする．

3 作業時間表の作成

すべての不具合を対象として遷移元と遷移先が同じ状態遷移対を抽出（例えば「不具合報告」→「議論開始」という状態遷移対をすべて抽出する）し，状態遷移に要した作業時間を算出する（作業時間表（図 3.3-(c)））．各作業時間の算出には，各遷移時間の中央値を用いる（理由は後述）．

4 状態遷移図の作成

作業時間表（図 3.3-(c)）に抽出された各状態と各遷移を状態遷移図として記述する．さらに，導出した各作業時間を遷移経路のエッジに記載し（括弧なし），その遷移経路を経由した不具合の数（遷移対の数）をエッジに記載する（括弧あり）．

なお，手順 3. において中央値を用いる理由は主に，OSS 開発での不具合修正における次の 3 つの特徴に依拠する．1 つ目は，プロジェクトへの自由な参加/離脱を認める OSS 開発では，不具合修正を行う開発者のスキルにばらつきが大きい，すなわち，開発者のスキルを統制することが難しいことである．2 つ目は，各開発者のプロジェクト参加理由の違い（業務の一部，余暇の時間を使った趣味，プログラミングやプロジェクトベース開発の学習・体験のためなど）により，開発者が OSS 開発に充てられる時間にばらつきが大きいことである．最後に，軽微な修正に留まる場合から，プロダクト全体に影響を及ぼすものまで，個々の不具合そのものの複雑度・難易度にばらつきが大きいという特徴がある．

これらの特徴から，OSS の不具合修正において必要とされる各作業時間は，一般的に正規分布に従わず偏った分布になる．1 つの不具合を修正するのに数年かかるようなケースも含まれ，外れ値によって平均値が直観的に解釈できないほど大きくなることが少なくないため，サンプルの代表性を示す際に平均値を用いるのは妥当ではない．そのため，十分な数の不具合票を保持する大規模プロジェクトを対象とした分析では，中央値を用いてサンプルの傾向を調べることが有効となる．

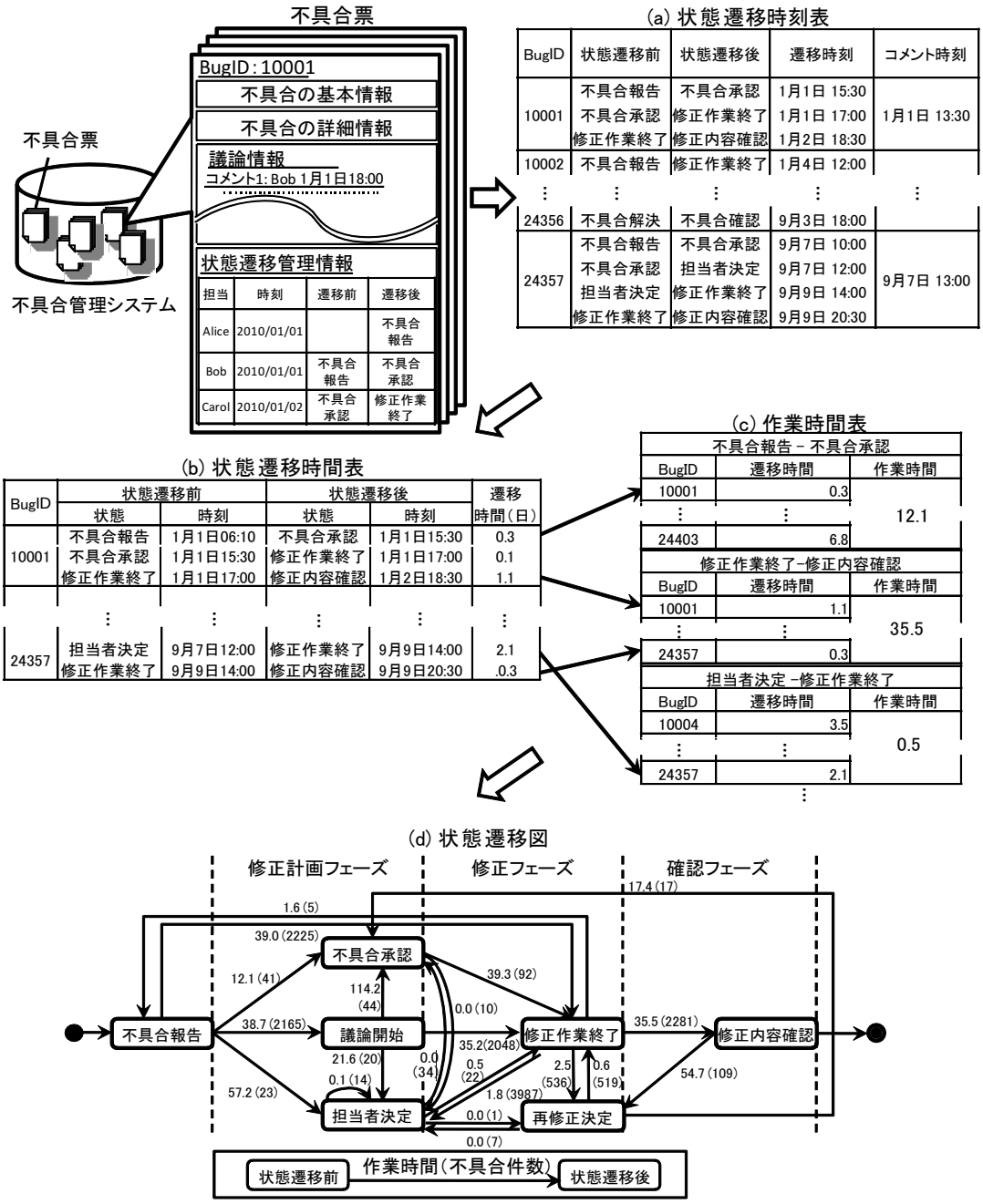


図 3.3 状態遷移図の作成手順

本手法で作成した状態遷移図は，図 3.2 に示す状態遷移図と異なる点がある．それは， Bugzilla プロジェクトが提示する不具合修正プロセスと異なるプロセスを経由することを OSS プロジェクトが可能にしているためである．例えば，OSS プロジェクトでは，コミッターが不具合報告を行う場合，不具合承認を経由せずに直接修正担当者を決定する場合がある．また，報告者自身が修正を行う場合,「不具合報告」から直接「修正作業終了」に遷移することがある．また，図 3.2 には「議論開始」という状態が定義されていない．しかし実際の OSS プロジェクトでは,「議論開始」を起点として不具合修正のための各種作業が開始されることが多いことから，本研究では BTS の議論情報に記録されている最初のコメント時刻を用いて「議論開始」状態を定義し，状態の 1 つとして用いている．

3.4.2.2 状態遷移図を用いた分析

状態遷移図を用いて，不具合修正を遅延させる原因となる作業やフェーズを特定する手順を述べる．

(1) 不具合修正プロセスの把握
状態遷移図を用いて，各フェーズに存在する状態遷移を把握する．例えば図 3.3 の Apache プロジェクトの状態遷移図には，図 3.2 の Bugzilla プロジェクトで提示されている不具合修正プロセスには存在しない状態遷移（「不具合報告」→「不具合解決」など）が存在することが見てとれる．

(2) 各フェーズ中で時間を要する作業の把握
状態遷移図を用いて，フェーズ毎に多く不具合で実施される作業を把握し，その作業に要する時間を把握する．一つの状態から複数の遷移がある場合，対象とする状態から多くの不具合が遷移している作業に着目する．例えば図 3 の状態遷移図の「不具合報告」の場合,「不具合報告」の次の状態は 4 つ（「不具合解決」,「不具合承認」,「議論開始」,「修正担当者決定」）のいずれかであり，合計 4,454 件 (2225+41+2165+23=4454 件）の不具合が関係する．その不具合の内，約 50% ($\frac{2225}{4454} \approx 0.50$）が「不具合解決」へ，約 49% ($\frac{2165}{4454} \approx 0.49$) が「議論開始」へ遷移することが分かる．また，状態遷移にそれぞれの 40 日，61 日かかっていることが分かる．このように，分析 2 で

は，多くの不具合が経由する状態遷移を見つけるために，各状態から次の状態に遷移する割合を用いて分析する．また，修正担当者の変更（「修正担当者決定」→「修正担当者決定」）や再修正（「不具合解決」→「再修正決定」）のように，修正が滞る直接的な原因となる作業も，各作業が実施される不具合件数と各作業に要する時間を用いて分析する．

(3) 最も時間を要するフェーズの把握
手順 (2) で挙げた各フェーズの時間を要する作業を比較し，その中で最も時間を要している作業を特定しそのフェーズを把握する．

3.5 ケーススタディ

本章では，BTS を利用している 3 つの大規模な OSS プロジェクトを対象にケーススタディを行った．本章ではケーススタディを行った結果を報告する．

3.5.1 分析対象データ

本章では，多くの不具合を管理している 3 つの OSS プロジェクト (Apache HTTP Server， Eclipse platform， Mozilla Firefox) を対象にケーススタディを行った．対象プロジェクトに関するソフトウェアの種類と不具合件数を表 3.1 に示す．これら 3 つの OSS プロジェクトは，（１）多くの開発者・利用者を有しており社会的に大きな影響を持つこと，（２）開発・運営スタイルが他の OSS プロジェクトの模範となっており対象プロジェクトから得られる知見の一般性を期待できること，（３）それぞれ利用されるドメインと利用者層が大きく異なるため比較により OSS プロジェクトの多面性や多様性をある程度捉える事ができること，（４）不具合管理のために Bugzilla を利用しており分析結果の比較が可能なこと，（５）プロジェクト立ち上げから相当な時間経過がありプロジェクト運営が安定していること（得られるデータの意味が時期によって大きく変わらないこと），（６）知見の一般性を高めるに足る十分なデータ（不具合報告数）が得られること，を主な理由として選定した．

表 3.1 ケーススタディの対象

プロジェクト	Apache HTTP Server	Eclipse platform	Mozilla Firefox
種類	Web サーバソフトウェア	統合ソフトウェア開発環境	Web ブラウザ
不具合件数	4,923	26,113	63,652
データ取得期間	2003/1〜2008/12	2001/10〜2009/11	2003/1〜2008/12

3.5.2 分析結果

各プロジェクトの分析結果を詳細に説明する．図 3.4，図 3.5，図 3.6 はそれぞれ，Apache HTTP Server，Eclipse platform，Mozilla Firefox プロジェクトの不具合修正プロセスを状態遷移図として可視化したものである．

3.5.2.1 Apache HTTP Server プロジェクト

修正計画フェーズでは,「不具合報告」から次の状態に遷移する不具合の約 99% ($\frac{2225+2165}{2225+41+2165+23} \approx 0.99$) が「修正作業終了」または「議論開始」に遷移している．また，修正フェーズでは，不具合の多くは「議論開始」後，直接「修正作業終了」に 35.2 日で遷移している．検証フェーズでは，約 81% ($\frac{2281}{2281+519+3+5} \approx 0.81$) の不具合が約 35 日で正しく修正されたと判断（「修正内容確認」）されており，約 19% ($\frac{519}{2281+519+3+5} \approx 0.19$) の不具合は約 1 日で再修正が必要と判断されている．しかしながら，正しいと判断された不具合のうち約 5% ($\frac{109}{2281} \approx 0.05$) は再修正が求められている．

以上のことから，Apache HTTP Server プロジェクトでは，多くの不具合は各フェーズに同時間（約 40 日）かかっていることを把握することができた．

3.5.2.2 Eclipse platform プロジェクト

修正計画フェーズでは,「不具合報告」から次の状態に遷移する不具合の約 82% ($\frac{10567+10472}{4756+2+10567+10472} \approx 0.82$) が「議論開始」または「修正担当者決定」に遷移しており，それぞれに遷移するまでに 6.0 日かかっている．修正フェーズでは，多くの不具合は「修正担当者決定」後,「修正作業終了」に至っている．しかしなが

ら，始めに割り当てられた担当者が適切でないため再度修正担当者を決定していることが多く（「修正担当者決定」からの自己ループ：10,105 回），修正担当者の再決定には「修正担当者決定」後の「修正作業終了」に要する時間（4.0 日）よりも多くの時間がかかっている（4.9 日）ことが見てとれる．検証フェーズでは，約 72%($\frac{2281}{2281+536+351} \approx 0.72$) の不具合は 35.5 日で正しく修正されたと判断（「修正内容確認」）されており，約 17%($\frac{536}{2281+536+351} \approx 0.17$) の不具合は 2.5 日で再修正が必要と判断されている．しかしながら，正しいと判断された不具合のうち約 13%($\frac{289}{2281+1} \approx 0.13$) は再修正が求められている．

以上のことから，Eclipse platform プロジェクトでは，検証フェーズの「修正内容確認」に遷移するまでの時間が最も長いことを把握することができた．

3.5.2.3 Mozilla Firefox プロジェクト

修正計画フェーズでは，不具合報告から次の状態に遷移する不具合の約 92%($\frac{18806+34843}{11+18806+1749+34843+2758} \approx 0.92$) が「修正作業終了」，または「議論開始」に遷移しており,「修正作業終了」に遷移する場合は約 5 日,「議論開始」に遷移する場合は約 11 日かかっている．修正フェーズでは，多くの不具合は「議論開始」後に直接「修正作業終了」に遷移しており，約 13 日かかっている.「修正担当者決定」の自己ループから，修正担当者の再決定は頻繁に行われていない (1,781 回) が，担当者が変更された場合は約 1 カ月要することが見てとれる．検証フェーズでは，修正解決後，次の状態に遷移する不具合のうち約 72% ($\frac{11326}{11326+1267+219+2951} \approx 0.72$) は正しく修正されたと判断（「修正内容確認」）されており，約 8% ($\frac{1267}{11326+1267+219+2951} \approx 0.08$) は再修正が必要と判断されている．Mozilla Firefox プロジェクトは不具合の検証に約 2 日かかっている．Mozilla Firefox プロジェクトでは，正しいと判断された不具合のうち，再修正が求められている不具合は約 1% ($\frac{101}{11326+11+37} \approx 0.01$) であった．

以上のことから，Mozilla Firefox プロジェクトでは，修正計画フェーズや修正フェーズに長い時間を要していることを把握することができた．

3.5.2.4 結果のまとめ

表 3.2 はケーススタディの結果をまとめたものである．表中の各セルには，多くの不具合に実施される作業 (1)，とその作業時間 (2) を示す．最下行には，不具

合修正プロセスの中で最も時間を要するフェーズを示す.

表 2 より，プロジェクトそれぞれで不具合解決までに必要な作業内容が大きく異なることが見てとれる．また，各フェーズを比較することで改善を要するフェーズも確認することができる（Apache の場合は，すべてのフェーズにおいて１つの不具合を解決するために１カ月以上要しているため，すべてのフェーズでプロセス改善が必要とされていると解釈できる）．ただし，不具合管理のルールやポリシーはプロジェクトにより異なるため，各作業で想定（あるいは許容）される作業時間は一様に定義することはできない.

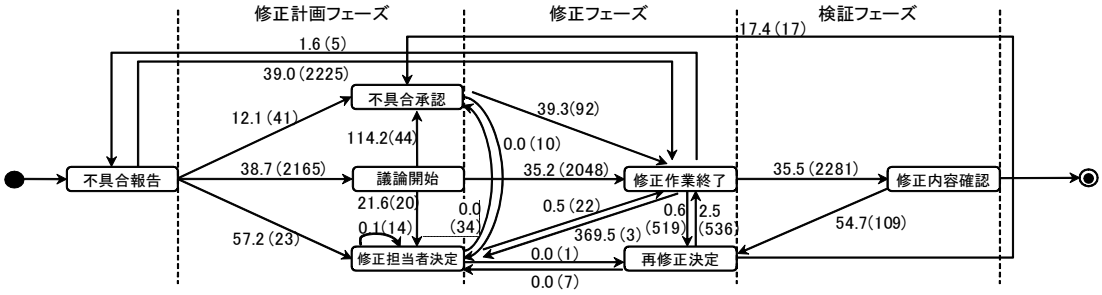


図 3.4 Apache HTTP Server プロジェクトの不具合修正プロセス

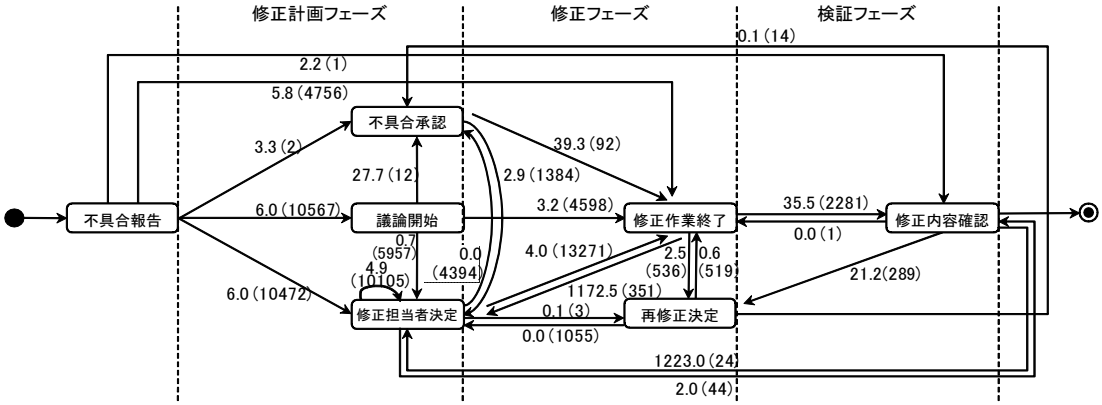


図 3.5 Eclipse platform の不具合修正プロセス

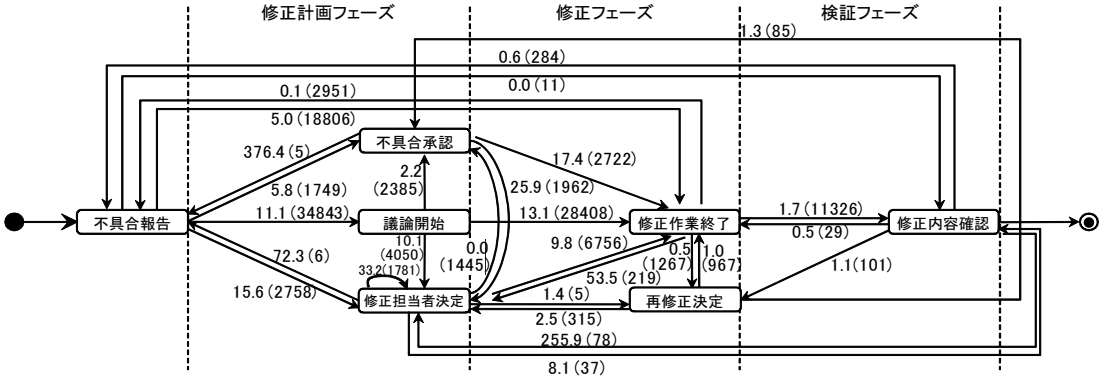


図 3.6 Mozilla Firefox の不具合修正プロセス

表 3.2 ケーススタディの結果

	Apache HTTP Server	Eclipse platform	Mozilla Firefox
修正計画フェーズ	(1) 不具合報告→修正作業終了, 不具合報告→議論開始 (2) 約 40 日	(1) 不具合報告→議論開始, 不具合報告→修正担当者決定 (2) 約 6 日	(1) 不具合報告→修正作業終了, 不具合報告→議論開始 (2) 約 11 日
修正フェーズ	(1) 議論開始→修正作業終了 (2) 約 40 日	(1) 修正担当者決定→修正担当者決定, 修正担当者決定→修正作業終了 (2) 約 4 日	(1) 議論開始→修正作業終了 (2) 約 13 日
検証フェーズ	(1) 修正作業終了→修正内容確認 (2) 約 35 日	(1) 修正作業終了→修正内容確認 (2) 約 35 日	(1) 修正作業終了→修正内容確認 (2) 約 2 日
最長時間フェーズ	判断できない	検証フェーズ	修正計画フェーズ，修正フェーズ

3.6 考察

3.6.1 不具合修正プロセス中の各フェーズの効率化に向けた方策

本節では，ケーススタディの結果を踏まえ，不具合修正プロセス中の各フェーズの効率化に向けた方策を検討する．

修正計画フェーズでは，3 つのプロジェクト全てにおいて，不具合報告後に議論開始状態に遷移する不具合票が数多く存在することを確認した．特に Mozilla Firefox プロジェクトでは，全体の約 60%の不具合が，報告後，議論開始に遷移していた．その原因の一つには，報告者が不具合票に記載する内容と，開発者が求める情報が異なっていることが多く [6][34]，修正依頼者や修正担当者が不具合に関する追加情報を求めるため議論開始に遷移していることが考えられる．Mozilla Firefox は Apache HTTP Sever や Eclipse platform に比べてソフトウェアの知識を持たない利用者からの不具合報告が多いため，開発者が不具合修正のために必要としている情報が不具合票に記載されていないことが考えられる．ソフトウェアの知識を持たない利用者の多いプロジェクトでは特に，修正計画フェーズにかかる時間の短縮に向けて，報告内容として必要な情報に漏れがないように，不具合票の報告フォーマット（不具合の再現方法やスクリーンショットを添付する仕組みなど）をあらかじめ用意しておく必要がある．

修正フェーズでは，繰り返し修正担当者が変更されるため時間を要することがある．それは各プロジェクトの修正依頼者がプロジェクト参加者の専門性を把握できていないため，適切な修正担当者を決定することが困難であることが大きな原因となっている [13][32]．本ケーススタディにおいても，各プロジェクトの状態遷移図における状態「修正担当者決定」の自己ループ回数，また作業時間を分析した結果，Eclipse platform プロジェクトでは繰り返し修正担当者が変更されており（10,105 回，4.9 日），Mozilla Firefox プロジェクトでは修正担当者の決定に時間を要している（33.2 日）ことを確認することができた．Eclipse Platform プロジェクトには，多くの IBM 社員が開発に携わっていることが知られており [62]，社内メンバ間で担当者の依頼を容易に行うことができるため，担当者の変更が繰り返し行われたと考えられる．一方，Mozilla Firefox プロジェクトの場合，世界中

に開発者が多く存在しているため，適切な担当者を見つけることが困難と指摘されており [32]，修正担当者の決定に時間を要すると考えられる．OSS 開発において適切な修正担当者を決定するための研究は数多く行われており [2][13][32]，目的に応じて利用できる．例えば，過去の修正担当者の変更履歴を用いて適切な修正担当者を自動的に決定する手法 [32] や，開発者がバージョン管理システムや BTS に記述した文章内容を基に開発者の専門知識を把握し，新しく報告された不具合の修正に適切な開発者を決定する方法 [13] などがある．

検証フェーズでは，不具合の再修正によって解決日が先延ばしになってしまうことがある．本ケーススタディの結果から，Apache HTTP Server プロジェクトでは不具合の修正確認後，再修正と判断されるまでに約 2ヶ月かかっていることが分かった．検証フェーズの長期化は，再修正が必要と判断されたとき，以前にどのような修正を行ったか思い出す必要がある．不具合解決後，時間の経過に伴い，以前の修正内容を思い出すことが困難となるため，再修正不具合発見の遅れは修正時間の長期化に繋がる．これまで，再修正される不具合を予め把握するために，不具合票に記載される情報を用いて再修正が必要となる不具合を予測するための予測モデルが提案されている [17]．再修正が必要となる可能性の高いモジュールをあらかじめ見積もっておくことで，コミッターは再修正が必要と予測される不具合を優先的に検証し，修正者担当候補者へ早急に再修正の依頼を行うことができる．

3.6.2 制約

本章で行った不具合情報に基づく解析では，不具合票の状態が正確に記録されていることが重要である．ケーススタディで対象とした OSS プロジェクトでは不具合報告や修正などの状態の変更に関する規則が各プロジェクトの Web ページに記載されているため，比較的正確に記載されているが，プロジェクトによっては，担当者が変更されているにも拘わらず，状態の変更が行われていない場合などがある．本章で分析の対象としたプロジェクト以外のコミッターが自身のプロジェクトで同様の分析を行う際には，状態の変更が正しく行われているか確認をしてから利用する必要がある．

また，本章で扱った各作業時間は，実際の作業に要した時間とは必ずしも一致しない場合がある．例えば，修正は終えているがコミットを翌日に行った場合，実質の作業を行った時間と不具合票に記録される作業時間に時間差が生じる．今後，OSS プロジェクトで Hackystat[33] のような開発者の行動履歴を取得するツールが利用されれば，BTS に記録される時間と実質の作業を行った時間との時間差は小さくなると考えられるが，現在はそのような正確な情報を取得することができないため，本章では，不具合票に作業の完了を記録した時を作業の終了時刻としている．

本章では，統計処理に足る不具合データを保持する大規模 OSS プロジェクトを対象としているため，不具合修正プロセスにおける各作業時間の算出には中央値を用いた．しかしながら，小規模 OSS プロジェクトのように十分な不具合票を保持していない場合，プロジェクトに関わる開発者も限られており，開発者のスキルなどに依存することが予想されるため中央値を用いることは適切でないと考えられる．小規模プロジェクトを対象とする場合は，開発者個々の生産性から不具合の修正作業に時間を要する原因を調査するなど他の方法を用いて分析する必要がある．

3.7 まとめ

本章では不具合の修正状況を俯瞰的に把握し，不具合修正が長期化する原因となる作業やフェーズを特定するために，BTS を利用している 3 つの大規模な OSS プロジェクトを対象に分析を行った．本章で用いた分析方法により，不具合修正プロセス中で最も時間を要するフェーズや作業，担当者の変更回数，再修正を要する不具合の件数などを網羅的かつ俯瞰的に把握できることが分かった．

3 つの OSS プロジェクトにおいて，各フェーズの分析から得られた知見は以下の通りである．

- 修正計画フェーズでは，報告後に議論を必要とする不具合が多く，修正依頼者は報告者や修正者との合意形成を円滑に行うことが重要である．
- 修正フェーズでは，修正担当者の度重なる変更，及び，修正担当者による

不具合内容の理解に時間がかかっているため，修正依頼者はプロジェクトの開発者の適性を理解して修正を依頼する必要がある．

- 検証フェーズでは，再修正が必要と判断されるまでに長い時間かかるため，レビューアは再修正にならない十分な検証が必要である．

各フェーズの分析結果より，不具合修正プロセスにおいて豊富な経験を必要とする作業に長時間かかっていることが分かった．特に，OSS の不具合修正に貢献するコミッターが行う作業に長期間かかっており，現在活動しているコミッターを中心とする開発者だけでは，修正に対処することが難しくなっていることが示唆される．開発者はプロジェクトへの参加/離脱が流動的であるため，継続して有能なコミッターを確保するために，コミッターの選出方法を確立することが必要と考える．

また，数多くの不具合の修正は約 1 カ月以上かかっていることが示された．OSS を商用ソフトウェアや社内システムに利用している場合，納期があるため，OSS プロジェクトによる修正に長期間かかる不具合は自社で修正せざるを得ない．その際，OSS 利用企業は，必要最小限の不具合の修正に取り組むために，当該不具合が OSS プロジェクトで修正されるか否かを見極める必要がある．本章の分析結果より各プロジェクトの修正作業は長期間かかる作業と短期間で終了する作業があるため，予測時点の不具合の状態や，進捗情報が不具合の修正時期を予測するために有用と考えられる．

第4章　OSSプロジェクトのための有能なコミッターの予測

OSS プロジェクトでは，膨大な不具合を対応するコミッターへの負担を軽減するために，継続して有能なコミッターを確保することが必要である．本章では，プロジェクトに参加する一般開発者の中からコミッターに推薦されるべき有能な開発者を見つけ出すためのコミッター予測モデルの構築方法について述べる．

4.1 はじめに

OSS プロジェクトでは，高い技術力や開発者を取りまとめる能力を持っていると認められた開発者にのみ，プロダクトに加えた変更をバージョン管理システムにコミットする権限が与えられる [20]．しかしながら，開発者から投稿される不具合修正パッチの検証作業を，少数のコミッターが中心となって負担するという作業負担上のデメリットを生む場合がある．OSS の社会的普及により，プロジェクトに報告される不具合の件数は年々増加しているため，特に大規模プロジェクトではコミッターにかかる負担は極めて大きい [13][51].

コミット権限付与方針のメリットを維持しつつコミッターの負担を軽減させるためには，コミッターを増員する方法が効果的である．しかし，プロジェクトに参加する一般開発者（大規模プロジェクトでは 1 万人を超えることも珍しくない）の中からコミッターに昇格すべき有能な開発者を見つけることは実際には容易ではない．一般開発者は，既存コミッターからの推薦と承認を経てコミッターに昇格する [31]．その際，既存コミッターは，プロジェクトに対する一般開発者の過去の活動内容，具体的には機能拡張や不具合修正に関連する活動を総合的に評価している．活動実績評価の必要性から，一般開発者がコミッター候補者として推

薦されるには，通常 1 年以上の継続的な活動が求められる．そのため，有能な一般開発者であっても途中で意欲を失いコミッターに推薦される前にプロジェクトを去ってしまうことが少なくない [9].

この問題を解決するためには，コミット権限を付与するに足る開発者を出来る限り早い段階で見つけ出しコミッター候補者として推薦する必要がある．本章では，コミッターに推薦されるべき有能な開発者を見つけ出すために，既存コミッターの過去の活動とその活動量を分析し，コミッター予測モデルを構築する．本章においてコミッター候補者は実験対象期間内にコミッターに昇格した者とし，それ以外の開発者は一般開発者とする．ただし，プロジェクト参加開始時点でコミット権限を持つ開発者，及び，分析開始時点で既にコミッターに昇格している開発者は実験対象外とする．そして，モデル構築には，コミッター候補者と一般開発者の不具合修正のための活動（パッチの投稿，パッチの検証，開発に伴う議論）の履歴とプロジェクトでの活動期間を用いる．

モデル構築にあたり本章では，プロジェクトが長期間に渡って運営され，多くのコミッターが参加している大規模 OSS (Eclipse platform, Mozilla Firefox) プロジェクトを対象に，以下の Research Question に取り組む.

RQ1: コミッター候補者を見つけ出すために有用な活動量は何か？

RQ1 では，コミッター候補者と一般開発者の活動量（パッチ投稿回数，パッチ検証回数，コメント回数），活動期間の違いを分析し，コミッター候補者の予測精度向上に寄与すると考えられる活動を把握する．コミッター候補者と一般開発者の活動量に大きな差がある活動はコミッター候補者を見つけ出すために有用と考えられる.

RQ2: 開発者の活動量からコミッター候補者をどの程度の精度で予測できるのか？

RQ2 では，開発者の活動量を用いてコミッター予測モデルを構築し，モデルを評価する.

以降 4.2 節では，関連研究について述べ本論文の立場を明らかにする．4.3 節では，パッチがプロダクトに反映されるまでの開発者の活動について詳述し，4.4 節では，本章の目的を達成するための Research Question と，実験手順について述

べる．4.5 節では，実験に用いるデータセットと，各 Research Question の実験結果について述べる．4.6 節で本章の考察を行い，最後に 4.7 節で本章のまとめを述べる．

4.2 関連研究

OSS プロジェクトは，新たなコミッターを選出するために客観的なデータではなく彼らの経験に基づいて開発者の活動量と活動期間を評価し，コミッター候補者を決定している．

Jensen ら [31] は，OSS プロジェクトの開発者にインタビューを行い，既存コミッターがコミッター候補者のどのような活動を評価しているのか調査している．Apache プロジェクトでは，開発者の技術的な活動（パッチの投稿など）を参考に，既存コミッターがプロジェクトに多く貢献している開発者をコミッター候補者として，プロジェクト管理委員 (PMC Member) に推薦する．その後，PMC Member がコミッター候補者の貢献を認めるとコミッターに昇格することができる．Mozilla プロジェクトも同様に，既存コミッターは開発者が行ってきた技術的な活動に加えて，開発者が行う社会的な活動（開発の指示など）を参考に，コミッター候補者を見つけている．

また，Bird ら [9] はコミッターに昇格する開発者の活動期間について分析している．PostgreSQL プロジェクトでは，約 1 年間の活動実績がある一般開発者はコミッターに昇格する開発者として適切であると結論付けている．開発者の活動期間が 1 年よりも短い場合，コミッターとしてふさわしいかどうかを判断するのは難しく，また，1 年より長い場合，開発者はモチベーションを失ってしまう危険性が高いことを挙げている．各プロジェクトで新たなコミッターを見つけ出すために，開発者の活動量や活動期間を参考にしているが，どの程度の活動を行っている開発者をコミッターとして推薦するのかを示す目安は提示されてない．

Fujita ら [21] は，コミッター候補者と一般開発者を機械的に判別できるか否かを確認するために，コミッター候補者と一般開発者のパッチの投稿・検証回数とパッチの編集・検証時間に関して分析している．その結果，パッチの投稿回数とパッチの検証回数については，コミッター候補者と一般開発者で違いがあること

が分かった．しかし，これらの活動量を用いてどの程度コミッター候補者を見つけ出すことができるかについては定かでない．本章では，既存コミッターがコミッター候補者を見つけ出すために参考にする活動の中で，最も注目すべき活動量を提示し，コミッター候補者をどの程度予測することができるかを分析する．

4.3 開発者の活動

本章で実験対象とするEclipseプロジェクト，Mozillaプロジェクトでは，コミッターへの昇格に関するガイドラインが定められている[5]．ガイドラインには，プロジェクトに対して貢献が認められた場合にコミット権限を付与すると記載されているが，貢献と認められる活動内容や活動量についての具体的な記述はない．本章では，既存コミッターがコミッター候補者を推薦する際に参考にしている開発者の技術的・社会的活動（パッチ投稿・検証，開発に伴う議論など）の履歴を用いて，コミッター候補者の活動量と一般開発者の活動量を比較する．開発者が機能拡張や不具合修正を行うプロセスを図4.1に示し，各活動の内容について説明する．

A: パッチの作成：開発者は機能拡張や不具合修正のためにパッチを作成し，その後，プロジェクトに投稿する．パッチは全ての開発者が投稿することができる．

B: パッチの検証：投稿されたパッチに関係する開発者（変更の適用対象となるモジュールの開発者など）がパッチの品質を検証する．その時，パッチを検証する開発者の中にバージョン管理システムにコミットする権限を持つコミッターが含まれている．修正内容が不十分であった場合，検証を行った開発者はパッチ投稿者に再修正を依頼する．修正内容が適切であった場合，検証を行ったコミッターがパッチをプロダクトに反映する．パッチの検証は，全ての開発者が行うことが可能である．しかし，パッチをプロダクト

[5] Eclipse: http://wiki.eclipse.org/Development_Resources/HOWTO/Nominating_and_Electing_a_New_Committer
Mozilla: http://www.mozilla.org/hacking/committer/

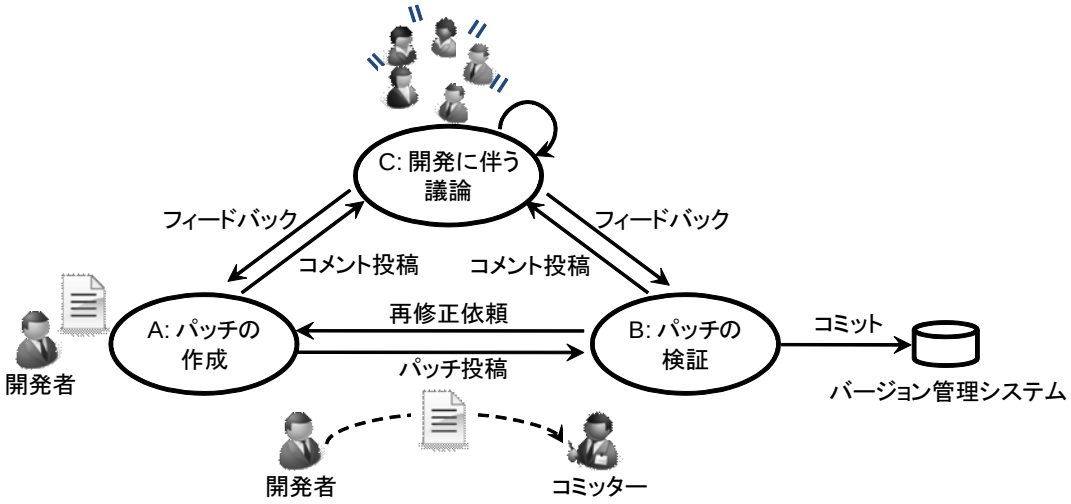


図 4.1 パッチがプロダクトに反映されるまでのプロセス

に反映できるのはコミッターのみであり，コミッターがパッチを承認しない限りパッチはプロダクトに反映されない．

C: 開発に伴う議論 ：パッチの投稿やパッチの検証と並行して，開発者間で議論を行う．議論の内容は，機能拡張や不具合修正の方針に関する相談や報告，類似不具合に関する情報提供，再修正依頼などが挙げられる．

本章では，これらの活動についてコミッター候補者の活動量と一般開発者の活動量を比較し，コミッター候補者を予測する．

4.4 Research Question

本節では，開発者の活動量を用いてコミッター候補者を予測するために取り組む Research Question と，各 Research Question の実験手順について述べる．

RQ1: コミッター候補者を見つけ出すために有用な活動量は何か？

動機：OSS の機能拡張や不具合修正のために，開発者らは図 4.1 に示すようなパッチ投稿，パッチ検証，開発に伴う議論を繰り返す．既存コミッターは，このような過去の活動内容を参照し，積極的に活動している開発者をコミッター候補者に推薦する．それゆえ，コミッター候補者と一般開発者では活動量に違いがあると考えられる．コミッター候補者と一般開発者の活動量に違いが認められれば，1 万人を超える開発者の中からコミッター候補者を容易に見つけ出すことができると考えられる．RQ1 では，コミッター候補者の活動量と一般開発者の活動量を比較し，コミッター候補者を予測するためにはどの活動が有用と考えられるかを分析する．

手順：RQ1 では，(1) コミッター候補者と一般開発者の活動量をそれぞれ抽出し，(2) コミッター候補者と一般開発者の活動量に差があるか否かを分析する．その後，(3) コミッター候補者と一般開発者の活動量の差がどの程度であるのかを分析し，他の活動と比べて，実質的に活動量の差が大きい活動を見つける．以下で各分析の詳細を説明する．

(1) コミッター候補者と一般開発者の活動量を抽出

本章で収集する開発者の活動量を表 4.1 に示す．OSS プロジェクトの開発者の中には，参加期間が短い者が数多く存在するが [43]，プロジェクトへの貢献が認められれば，コミッターに昇格する場合がある [20]．よって，一時期のみ積極的に活動した開発者と継続して活動している開発者を区別するために，各活動の総回数と各活動の月回数（中央値）を導出する．また，コミッター候補者の活動期間は，パッチ投稿，パッチ検証，開発に伴う議論のいずれかの活動を初めて行った日からコミッターに昇格するまでの期間とする．一般開発者の活動期間は，コミッ

表 4.1 開発者の活動

活動	変数名	内容
パッチ投稿	総パッチ投稿数	各開発者が全分析期間にパッチ形式のファイルを投稿した総回数.
	月パッチ投稿数	各開発者が活動期間中で 1ヶ月にパッチを投稿した回数の中央値.
パッチ検証	総パッチ検証数	パッチが投稿された直後に投稿された他の開発者からのコメントをレビューと定義したときの，開発者毎の全分析期間におけるレビュー数の総数.
	月パッチ検証数	各開発者が活動期間で 1ヶ月に検証したパッチ数の中央値.
開発に伴う議論	総コメント投稿数	パッチを含まない不具合票でのメッセージをコメントと定義し，各開発者毎の全分析期間におけるコメント総数.
	月コメント投稿数	各開発者が活動期間で 1ヶ月に投稿したコメント数の中央値.
活動期間	活動期間	各開発者の全分析期間において，パッチ投稿，パッチ検証，不具合修正に関する議論のいずれかの活動が実施された月数.

ター候補者と同様に上記のいずれかの活動を行った日から，それらの活動を最後に行った日までの期間とする．

(2) コミッター候補者と一般開発者の活動量に統計的有意差があるかを検定

表 4.1 に示す活動量に関して，コミッター候補者の活動量と一般開発者の活動量の分布を比較する．分布の比較にはウィルコクソンの順位和検定を用い，有意水準 5%で検定を行う．

(3) コミッター候補者と一般開発者の活動量の差がどの程度であるのかを分析

手順 (2) で行う検定では，活動間の比較は行えないため，コミッター候補者と一般開発者の活動量の差が実質的に大きい活動を効果量を用いて分析する．効果量は，2 つの変数の平均値の差を標準化すること

により，単位の異なる変数間で，効果の大きさを比較することができる指標である．効果量 (d) は以下の式で求めることができる [35][41].

$$d = \frac{\tilde{x}_a - \tilde{x}_b}{\sqrt{S_a^2 + S_b^2}}$$

ここで用いる変数 $\tilde{x}_a$，$\tilde{x}_b$，S_a，S_b は，それぞれ任意の活動におけるコミッターと一般開発者の活動量の集合（a，b）について，$\tilde{x}_a$ はコミッター候補者の活動量の平均値，$\tilde{x}_b$ は一般開発者の活動量の平均値，S_a はコミッター候補者の活動量の分散，S_b は一般開発者の活動量の分散を示す．効果量の値が大きいほどコミッター候補者と一般開発者の活動量の差が実質的に大きいことを意味する．

RQ2: 開発者の活動量からコミッター候補者をどの程度の精度で予測できるのか？

動機：既存コミッターはプロジェクトに参加する 1 万人を超える開発者の中からコミッター候補者を見つけ出し，推薦しなければならない．そこで，RQ2 では，コミッター候補者を容易に見つけ出すために，表 4.1 に示す開発者の活動量を用いて，開発者の中からどの程度の精度でコミッター候補者を予測することができるか評価実験を行う．

手順：表 4.1 に示す全ての活動量を用いてコミッター予測モデルを構築する．本章では，開発者の活動量のように正規分布に従っていないデータを対象とした多変量解析に用いられているロジスティック回帰分析 [5][40][55] でコミッター予測モデルを構築する．コミッターに昇格するか否かを目的変数とし，予測モデルの出力値（0 から 1 の連続値）が 0.5 以上のときにコミッターに昇格すると判別する．ただし，多重共線性を避けるために，本章で用いる変数から任意に選び出した 2 変数に 0.8 以上の相関があれば，片方を削除する．0.8 以上の相関を持つ組が多い変数は優先的に削除する．

モデルの評価には，適合率 (Precision)，再現率 (Recall)，F1 値 [24][36] と正確に予測できたコミッター候補者数を用いる．適合率は，コミッターに昇格すると判断された開発者のうち，実際にコミッターに昇格

した開発者の割合を示す．また再現率は，全コミッターの中で，コミッターに昇格すると判別されたコミッターの割合を示す．ただし，適合率と再現率はトレードオフの関係にあるため，両方の評価指標が高い時，性能の高い予測モデルが構築できたことになる．そこで本章では，適合率と再現率に加えて，適合率と再現率の調和平均で与えられるF1値を評価指標の一つとして用いる．F1値は以下の式で定義され，値が大きいほどモデルの判別精度が高いことを示す．

$$F1\text{-}value = \frac{2 \times Recall \times Precision}{Recall + Precision}$$

また，本章では既存コミッターがコミッター予測モデルを使用する場合，本モデルによって予測されるコミッター候補者の中から，実際に昇格させる者を検討するものと想定している． そのため，数多くのコミッター候補者を正確に予測することが予測モデルにとって重要であると考える．そこで，適合率，再現率，F1値に加え，正確に予測できたコミッター候補者数を評価指標の一つとして用いる．

4.5 実験

本節では，実際のOSSプロジェクトに参加する開発者の活動量を用いて，コミッター候補者を見つけ出すのに有用な開発者の活動量を分析する．また，開発者がコミッターに昇格するまでの活動量に基いてコミッターの予測モデルを構築し，モデルの評価実験を行う．

4.5.1 概要

本節では，コミッター候補者と一般開発者の活動がコミッター候補者を見つけ出すために有効であるかどうかを確認するために，大規模なOSSプロジェクトであるEclipse platformプロジェクト，Mozilla Firefoxプロジェクトを対象としてケーススタディを行う．本章で対象とする分析期間，また各プロジェクトにおける開発者数を表4.2に示す．

表 4.2 実験対象データの概要

	Eclipse platform	Mozilla Firefox
実験対象期間	2001/10-2010/12	2004/01-2008/12
全コミッター人数	89	147
コミッター候補者人数	55	51
一般開発者人数	8,964	12,287

実験対象期間の終了時点までに一度でもバージョン管理システムにコミットを行った開発者をコミッターと呼び，そのうち活動を開始してから初めてコミットを行うまでをコミッター候補者と呼ぶ．そして，実験対象期間の最終日までにコミッターに昇格していない開発者は,（たとえ実験対象期間後にコミッターに昇格している場合でも）一般開発者とする．一方で，開発者の中にはプロジェクトに携わり始めたときからコミッターである場合もある．本章では，実験対象期間の以前からコミッターである開発者については，実験対象期間中の活動がその開発者のコミッター昇格に直接関係していないと考えられるため除外している．Eclipse platform プロジェクトでは，89 人のコミッターを確認したが，34 人はパッチの投稿，パッチの検証，開発に伴う議論を行う前にバージョン管理システムにコミットしていたため，プロジェクトに参加した時点で既にコミット権限をもっていた，もしくは，実験開始日の以前にコミッターに昇格していたことが考えられる．そのため，本章では残りの 55 人をコミッター候補者として実験を行う．同様に，Mozilla Firefox では，147 人のコミッターを確認したが，96 人は実験対象期間中でパッチの投稿，パッチの検証，開発に伴う議論を行う前にバージョン管理システムにコミットしている記録があるため，残りの 51 人をコミッター候補者として実験を行う．

4.5.2 実験対象プロジェクトの選出条件

本章では，多くの不具合を管理している 2 つの OSS (Eclipse platform， Mozilla Firefox) プロジェクト を対象に実験を行った．2 つの OSS プロジェクトは (1) 不

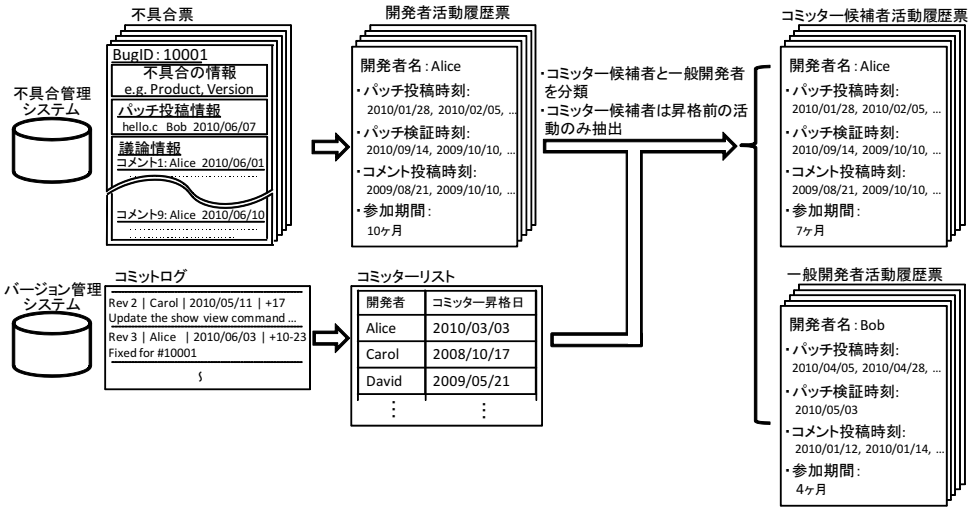


図 4.2 データ抽出方法

具合管理のために Bugzilla を利用しており実験結果の比較が可能なこと，(2) 定期的にリリースを行っており，プロジェクト運営が安定していること，(3) 知見の一般性を高めるに足る十分なデータ（コミット数，不具合報告数）が得られること，を主な理由として選定した．

4.5.3 実験データの収集と整形

本章では，パッチ投稿，パッチ検証，開発に伴う議論に関する情報を収集するために BTS[6] のデータを用いる．また，コミッター候補者と一般開発者を区別するためにバージョン管理システムのコミットログデータを用いる．BTS，バージョン管理システムを用いたデータの抽出手順を，図 4.2 に沿って説明する．

まず，各開発者の活動量を BTS から抽出する．各 OSS プロジェクトでは，不具合情報や機能拡張を希望する情報などを BTS の不具合票に記録している．各不具合票には，変更対象のモジュール名，修正を行ったパッチ，修正に伴う議論等が記録されている．本章では，BTS から不具合票を収集し，その後，各開発者が実施した図 4.1 に示す機能拡張や不具合修正のための活動（パッチの作成，パッチ

[6] Eclippse Bugzilla: https://bugs.eclipse.org/bugs/
Mozilla Bugzilla: https://bugzilla.mozilla.org/

検証，開発に伴う議論）量を収集するために，不具合票からパッチ投稿者，パッチ投稿時刻，パッチ検証者，パッチ検証時刻，コメント投稿者，コメント投稿時刻，そして，プロジェクトでの活動期間を抽出する．抽出方法は開発者の活動履歴を用いた従来研究と同様の方法 [11][21][60] を用いており，抽出の自動化が可能である．パッチの検証はパッチ投稿直後に投稿されたコメントを当該パッチに対する検証とする．パッチ投稿，パッチ検証，コメント投稿のいずれかの活動が初めて実施された月をプロジェクト参加開始月とし，いずれかの活動が最後に実施された月をプロジェクト参加最終月とする．活動期間はプロジェクト参加開始月からプロジェクト参加最終月までの期間とする．以上の記録を開発者ごとに開発者活動履歴票にまとめる．

次に，開発者がコミッターに昇格した日時をバージョン管理システムから抽出する．バージョン管理システムでは，ソースコードをコミットした開発者名とコミット時刻がソースコード別にコミットログとして記録されている．本章では，バージョン管理システムからコミットログを収集し，コミット経験のある開発者名，各開発者が初めてコミットした時刻を抽出する．

最後にコミッターリストを用いて開発者活動履歴票をコミッター候補者の活動履歴（コミッター候補者活動履歴票）と一般開発者の活動履歴（一般開発者活動履歴票）に分類する．本章では，開発者が初めてバージョン管理システムにコミットした日時をコミッター昇格日時と定義し，コミッターはコミッター昇格日時までの活動のみ抽出する．

モデル構築，及び，モデルを評価するためのデータの作成には，多くの研究で用いられている交差検証法を使用する [19][39][53]．交差検証法はデータセットを n 個のブロックに分割し，n-1 個のブロックでモデルを構築し，それ以外のブロックを検証用に用いる．通常 n は，評価ブロックにデータセットの偏りが出ないように 10〜20 とする．しかしながら，本章で用いるデータのように予測対象のコミッター候補者が極端に少ない場合，コミッター候補者を 10〜20 のブロックに分類すると 1 ブロックあたりのコミッター候補者が少なく（n を 10 とした場合，各ブロックのコミッター候補者は約 5 人），適合率を適切に計測できない．そのため，本章では，コミッター候補者活動履歴票と一般開発者活動履歴票を 2 等分

し，一方はモデルを構築するために，もう一方はモデルを評価するために用いる．コミッター候補者活動履歴票と一般開発者活動履歴票を 2 等分するとき，ランダムに活動履歴票を選ぶ．モデル構築の実験は 1000 回繰り返され，各実験で得られた 4 つの評価結果（適合率，再現率，F1 値，正確に予測できたコミッター候補者数）の中央値を比較する．

4.5.4 結果

RQ1: コミッター候補者を見つけ出すために有用な活動量は何か？

Eclipse platform におけるコミッター候補者と一般開発者の各活動の統計量（中央値，平均値，分散，p 値，効果量）を表 4.3 に，Mozilla Firefox プロジェクトにおけるコミッター候補者と一般開発者の各活動の統計量を表 4.4 に示す．Eclipse platform プロジェクトでは，表 4.1 に示す全ての活動においてコミッター候補者と一般開発者の活動量に統計的有意差があることが分かった．Mozilla Firefox プロジェクトでは，月コメント投稿数を除いて全ての活動量で統計的有意差が見られた．よって，ほとんどの活動は，コミッターに昇格する開発者を見つける際に有用であると考えられる．

次に，Eclipse platform プロジェクトにおける開発者の各活動について効果量の分析を行った結果，全ての活動の中で，コミッター候補者と一般開発者の月パッチ検証数の差が実質的に最も大きいことが分かった．次いで，月パッチ投稿数，月コメント投稿数，総パッチ検証数，総パッチ投稿数，活動期間，総コメント投稿数の順であった．Mozilla Firefox プロジェクトにおいても，コミッター候補者と一般開発者の月パッチ検証数の差が実質的に最も大きいことが分かった．次いで，月パッチ投稿数，活動期間，月コメント投稿数，総パッチ投稿数，総パッチ検証数，総コメント投稿数の順であった．総パッチ投稿数よりも月パッチ投稿数の方が，総パッチ検証数よりも月パッチ検証数の方が，総コメント投稿数よりも月コメント投稿数の方が効果量が大きいことから，パッチ投稿，パッチ検証，開発に伴う議論のそれぞれのひと月あたりの活動回数は同じ活動の総回数よりも効果量が大きいことが分かった．このことから，活動量の総数が同じであったと

表 4.3 開発者の活動の統計量（Eclipse platform）

		中央値	平均値	分散	p 値	効果量
総パッチ投稿数	コミッター(注 1)	11.50	31.39	2634.77	0.00	0.01
	一般開発者	0.00	0.69	66.63		
月パッチ投稿数	コミッター	1.75	4.29	38.58	0.00	0.11
	一般開発者	0.00	0.09	0.34		
総パッチ検証数	コミッター	0.00	2.39	22.25	0.00	0.06
	一般開発者	0.00	0.32	25.83		
月パッチ検証数	コミッター	0.00	0.57	0.72	0.00	0.72
	一般開発者	0.00	0.05	0.06		
総コメント投稿数	コミッター	42.50	77.54	17547.67	0.00	0.00
	一般開発者	2.00	15.03	48197.35		
月コメント投稿数	コミッター	3.00	8.96	228.48	0.00	0.03
	一般開発者	1.00	1.42	6.79		
活動期間 (月)	コミッター	6.47	13.94	197.59	0.00	0.01
	一般開発者	1.03	9.51	295.55		

表 4.4 開発者の活動の統計量（Mozilla Firefox）

		中央値	平均値	分散	p 値	効果量
総パッチ投稿数	コミッター(注 1)	1.00	38.83	16719.80	0.00	0.00
	一般開発者	0.00	0.45	44.01		
月パッチ投稿数	コミッター	0.00	1.33	6.04	0.00	0.21
	一般開発者	0.00	0.05	0.23		
総パッチ検証数	コミッター	1.50	31.08	13438.86	0.00	0.00
	一般開発者	0.00	0.27	12.29		
月パッチ検証数	コミッター	0.00	0.75	2.80	0.00	0.25
	一般開発者	0.00	0.04	0.05		
総コメント投稿数	コミッター	17.00	248.67	292007.70	0.00	0.00
	一般開発者	2.00	7.79	8854.90		
月コメント投稿数	コミッター	1.00	8.10	365.67	0.30	0.02
	一般開発者	1.00	1.33	5.10		
活動期間 (月)	コミッター	24.12	30.66	584.30	0.00	0.04
	一般開発者	1.00	6.23	157.78		

(注 1) 表 4.3，表 4.4，表 4.9，表 4.10 で，コミッターはコミッター候補者を意味する．

しても，長い年月をかけて活動を続けている開発者の方がコミッターに昇格していることが分かった．例えば，任意の 1 カ月に 100 回のパッチ投稿を行った開発者よりも 1ヶ月に 10 回のパッチ投稿を 10 カ月続けた開発者の方がコミッターに昇格しているということを意味する．

RQ2:開発者の活動量からコミッター候補者をどの程度の精度で予測できるのか？

本章で（全変数を用いて）構築したモデルにおける適合率，再現率，F1 値，正確に予測できたコミッター候補者数を表 4.5，表 4.6 に示す．また，本章のモデルの有意性を示すために，表 4.1 で提示する変数 1 つのみを用いて構築したモデルで予測した場合，及び，ランダムに予測した場合の結果を示す．そして，本章のモデルを用いて予測する場合と，ランダムに予測した場合と比べて，どの程度精度が向上したかを併記する．具体的なモデルの評価は，適合率や F1 値がランダムで予測した場合より高いモデルの中で，再現率が高い（且つ，正確に予測できたコミッター候補者数が多い）モデルを有意性の高いモデルとして判断する．

本章で構築したモデルで予測した場合と単一の変数を用いて構築したモデルを比較した結果，Eclpse platform プロジェクトでは，本章で構築したモデルで予測した場合の適合率，再現率，F1 値が最も高いことが分かった．Mozilla Firefox プロジェクトでは，再現率は本章で構築したモデルで予測した場合が最も高くなったが，適合率と F1 値は単一の変数（例えば，総パッチ投稿数）を用いて構築したモデルで予測した場合が最も高いことが分かった．

実験の結果，適合率は両プロジェクト共に 10%以下であるが，再現率は Eclipse platform プロジェクトで約 70%，Mozilla Firefox プロジェクトで約 83%であった．また，ランダムに予測する場合，適合率はプロジェクトによって本章で構築した予測モデルよりも低い場合があるが，本論文で構築した予測モデルの再現率は両プロジェクト共にランダムに予測するよりも高い精度で予測可能であることが分かった．

次に，各モデルで正確に予測できたコミッター候補者数は，Eclipse platform

表 4.5 コミッター候補者の予測結果 (Eclipse platform)

		適合率	再現率	F1 値	正確に予測できたコミッター候補者数
本実験モデル（全変数）を用いた予測		0.07	0.70	0.13	19
単一の変数を用いた予測	総コメント投稿数	0.06	0.70	0.11	19
	月コメント投稿数	0.01	0.33	0.03	9
	総パッチ投稿数	0.05	0.67	0.10	18
	月パッチ投稿数	0.05	0.48	0.09	13
	総パッチ検証数	0.06	0.56	0.10	15
	月パッチ検証数	0.05	0.41	0.10	11
	活動期間	0.00	0.33	0.00	9
ランダム予測		0.04	0.50	0.09	14
向上率(注 2)		1.75	1.40	1.44	1.36

表 4.6 コミッター候補者の予測結果 (Mozilla Firefox)

		適合率	再現率	F1 値	正確に予測できたコミッター候補者数
本実験モデル（全変数）を用いた予測		0.02	0.83	0.05	20
単一野変数を用いた予測	総コメント投稿数	0.05	0.67	0.09	16
	月コメント投稿数	0.00	0.54	0.01	13
	総パッチ投稿数	0.05	0.54	0.10	13
	月パッチ投稿数	0.06	0.46	0.11	11
	総パッチ検証数	0.05	0.67	0.10	16
	月パッチ検証数	0.04	0.38	0.07	9
	活動期間	0.02	0.79	0.04	19
ランダム予測		0.04	0.54	0.09	13
向上率(注 2)		2.00	1.54	0.56	1.54

(注 2) 本章のモデルを用いて予測する場合がランダムに予測した場合に比べて，どの程度精度が向上したかを意味する．

プロジェクトで最大で 19 人（再現率: 約 70%），Mozilla Firefox プロジェクトで最大 20 人（再現率: 約 83%）であり，両プロジェクト共に本論文で構築したモデルが最も多くのコミッター候補者数を正しく予測することができた．

本章で構築したモデルと単一の変数を用いて構築したモデルはランダムで予測する場合よりも適合率と F1 値が高く，さらに，本章で構築したモデルは再現率が高い（且つ，正確に予測できたコミッター候補者数が多い）ことが分かった．よって，総合的に判断すると，本章で構築したコミッター予測モデルが最も有意性が高いと考えられる．

本実験では適合率が低い値となったが，その原因は，コミッター候補者と一般開発者の人数比が大きく偏っている点にある．Menzies ら [42] は予測対象のサンプル数（本章ではコミッター候補者の人数）が極端に少ない場合，高い適合率を得ることは難しいと指摘している．また，再現率が高ければ，ランダムに予測するより適合率が高い場合に限り，そのモデルは有用であると述べている．

本実験では，ロジスティック回帰分析の出力値が 0.5 以上の場合にコミッター候補者として判定した．コミッターとして相応しい開発者を正確に抽出したい場合は閾値の値を大きく，多くのコミッター候補者を挙げたい場合は閾値を小さくすることが可能である．例えば，閾値を 0.8 に設定した場合，各プロジェクトの適合率は Eclipse platform プロジェクトで約 11%（再現率: 63%），Mozilla Firefox プロジェクトで約 3%（再現率: 71%）になる．ただし，閾値を大きくし過ぎると，コミッター候補者と判断される開発者数が減少するため適合率は向上する一方で再現率は低下する．反対に閾値を小さくし過ぎると，コミッター候補者と判断される開発者数が増加するため再現率が向上する一方で適合率は低下する点に注意する必要がある．RQ1 の実験結果で，月パッチ検証数の効果量は高いが，月パッチ検証数のみで構築したモデルで正確に予測できたコミッター昇格数は両プロジェクト共に多くはなかった．その理由は，月パッチ検証数はコミッター，及び，一般開発者の活動量は両プロジェクト共に中央値が 0.00 回であるため，月パッ

チ検証数のみで構築したモデルを用いて多くのコミッターを予測することは難しい．しかしながら，検証作業の経験を持つ開発者はコミッターに昇格する可能性が高いため，有能なコミッターを選出するために重要な変数と言える．

4.6 考察

4.6.1 コミッター昇格後の活動

本章で対象とした，Eclipse platform プロジェクト，Mozilla Firefox プロジェクトでは共に約 50 人の開発者がコミッターに昇格していた．しかしながら，コミッターに昇格した後も継続してプロジェクトに貢献しているかどうかを確認していない．既存コミッターは開発者の中から，昇格後も積極的に活動し続ける開発者をコミッター候補者として推薦することが望ましい．そこで，本章で対象としたコミッター候補者が，昇格後も継続して積極的に活動しているか否かを確認した．Eclipse platform プロジェクトにおけるコミッター候補者の昇格前後の活動量を表 4.7 に，Mozilla Firefox プロジェクトにおけるコミッター候補者の昇格前後の活動量を表 4.8 に示す．両プロジェクト共に，ほとんどの活動は，コミッター昇格後に活動量が減少していなかった．唯一パッチ投稿数が減少していたが，二つの理由が考えられる．一つ目は，自身でパッチをコミットすることができるため，他の開発者から検証を受ける義務がなくなった．二つ目は，パッチ検証，開発の指示，プロジェクト運営のための活動など，機能拡張や不具合修正以外の活動が増加するため，パッチ投稿数が減少した．このことはパッチ検証数やコメント投稿数が増加していることからも見てとれる．

コミッター昇格後，多くの開発者が積極的に活動する一方で，コミッター昇格後に活動を停止する開発者も存在していた．しかし，コミッター昇格後もいずれかの活動を一年以上継続する開発者は Eclipse platform プロジェクトで 55 人中 44 人（約 80%），Mozilla Firefox プロジェクトで 51 人中 30 人（約 59%）存在していた．また，本章で構築したモデルは，Eclipse platform プロジェクトでコミッ

表 4.7 コミッター昇格前後の活動 (Eclipse platform)

	総コメント投稿数		月コメント投稿数		総パッチ投稿数		月パッチ投稿数	
	昇格前	昇格後	昇格前	昇格後	昇格前	昇格後	昇格前	昇格後
中央値	50.00	442.00	2.00	5.00	12.00	20.00	1.50	1.00
平均値	78.11	1236.84	8.05	16.97	31.93	87.96	4.49	2.05
分散	12434.32	4064627.92	193.27	759.85	2437.66	19424.67	43.59	10.29

	総パッチ検証数		月パッチ検証数	
	昇格前	昇格後	昇格前	昇格後
中央値	1.00	20.00	0.50	1.00
平均値	17.27	73.02	2.49	1.48
分散	1540.46	17268.13	23.07	4.57

表 4.8 コミッター昇格前後の活動 (Mozilla Firefox)

	総コメント投稿数		月コメント投稿数		総パッチ投稿数		月パッチ投稿数	
	昇格前	昇格後	昇格前	昇格後	昇格前	昇格後	昇格前	昇格後
中央値	11.00	13.00	1.00	1.00	1.00	0.00	0.00	0.00
平均値	141.63	193.94	4.41	3.10	25.80	17.57	1.03	0.32
分散	156099.81	441047.58	184.38	52.04	8493.65	1724.27	3.58	0.27

	総パッチ検証数		月パッチ検証数	
	昇格前	昇格後	昇格前	昇格後
中央値	1.00	1.00	0.00	0.00
平均値	19.25	33.39	0.69	0.57
分散	6563.62	8789.65	1.65	1.27

ターと予測した 19 人中 16 人（約 84%），Mozilla Firefox プロジェクトでコミッターと予測した 20 人中 15 人（約 75%）はいずれかの活動を一年以上継続する有能な開発者を選出しているため有用性が高いと考えられる．

4.6.2 コミッターに昇格する開発者の活動量

本章では，実験対象期間中にコメント投稿，パッチ投稿，パッチ検証を行った開発者を対象とした．しかしながら，一般開発者の中にはコメント投稿のみ行っている者は Eclipse plarform プロジェクトで 4,481 人中 4,022 人（約 90%），Mozilla Firefox で 6,119 人中 5,772 人（約 94%）も存在していた．このような一般開発者がコミッターに昇格することは一般的に考えにくい．そこで，パッチ投稿とパッチ検証を一度も行っていない開発者を除いて RQ1 と RQ2 の追加実験を行い，RQ1 の結果を表 4.9 と表 4.10 に，RQ2 の結果を表 4.11 に示す．なお，RQ2 では，学習データに対してのみ当該開発者を除いており，テストデータに対しては行っていない．

Eclipse platform プロジェクトでは，パッチ投稿やパッチ検証の経験を持つ開発者のみを対象としても，コミッター候補者と一般開発者のパッチ投稿数やコメント投稿数に有意な差（有意水準 5%）があった．よって，Eclipse platform プロジェクトの既存コミッターはパッチ投稿やコメント投稿をより多く行う開発者を調査することで，コミッター候補者を見つけ出すことができると考えられる．その一方で，Mozilla Firefox プロジェクトでは，コミッター候補者のパッチ投稿数やコメント投稿数の中央値は一般開発者に比べて多いものの，有意な差はなかった．つまり，Mozilla Firefox プロジェクトでは，パッチ投稿やパッチ検証を行っているか否かでフィルタリングするのは，コミッター候補者に推薦されない開発者を選定する上では有効であるが，より多く行っているからといってコミッター候補者であるとは限らないことが分かった．コメント投稿数，パッチ投稿数，パッチ検証数の効果量は，両プロジェクト共に総回数よりも一カ月あたりの活動量の方が大きく，フィルタリング前と同様に，長い期間活動を続けている一般開発者の方がコミッターに昇格している．

また，パッチ投稿数とパッチ検証数が 0 件の開発者を削除して構築したモデル

表 4.9 パッチ投稿やパッチ検証の経験を持つ開発者の活動の統計量（Eclipse platform）

		中央値	平均値	分散	p 値	効果量
総パッチ投稿数	コミッター(注 1)	25.00	43.95	3163.21	0.00	0.01
	一般開発者	1.00	6.73	611.01		
月パッチ投稿数	コミッター	4.25	6.00	44.00	0.00	0.12
	一般開発者	0.00	0.91	2.59		
総パッチ検証数	コミッター	1.50	3.35	28.24	0.04	0.00
	一般開発者	1.00	3.16	243.68		
月パッチ検証数	コミッター	0.75	0.80	0.83	0.11	0.34
	一般開発者	0.00	0.48	0.41		
総コメント投稿数	コミッター	62.00	97.45	23187.94	0.00	0.00
	一般開発者	7.00	104.71	460840.25		
月コメント投稿数	コミッター	3.25	7.98	97.93	0.00	0.06
	一般開発者	0.00	1.37	43.17		
活動期間 (月)	コミッター	12.55	17.33	235.76	0.39	0.00
	一般開発者	19.70	27.65	684.23		

表 4.10 パッチ投稿やパッチ検証の経験を持つ開発者の活動の統計量（Mozilla Firefox）

		中央値	平均値	分散	p 値	効果量
総パッチ投稿数	コミッター(注 1)	3.00	15.78	988.07	0.05	0.01
	一般開発者	1.00	7.02	640.87		
月パッチ投稿数	コミッター	1.00	0.97	1.19	0.17	0.06
	一般開発者	0.00	0.79	2.99		
総パッチ検証数	コミッター	1.50	8.72	172.68	0.06	0.02
	一般開発者	1.00	4.16	175.67		
月パッチ検証数	コミッター	1.00	0.78	0.54	0.22	0.27
	一般開発者	1.00	0.58	0.49		
総コメント投稿数	コミッター	12.00	64.44	24207.56	0.62	0.00
	一般開発者	7.00	75.66	130965.30		
月コメント投稿数	コミッター	0.00	0.94	1.85	0.58	0.00
	一般開発者	0.00	1.13	22.70		
活動期間 (月)	コミッター	18.54	21.15	301.81	0.81	0.00
	一般開発者	18.67	25.27	497.15		

表 4.11 パッチ投稿やパッチ検証の経験を持つコミッター候補者の予測結果

		適合率	再現率	F1 値	正確に予測できたコミッター候補者数
Eclipse platform	本実験モデルを用いた予測	0.07	0.70	0.13	19
	パッチ投稿やパッチ検証の経験を持つ開発者を用いた予測	0.10	0.56	0.20	15
Mozilla Firefox	本実験モデルを用いた予測	0.02	0.83	0.05	20
	パッチ投稿やパッチ検証の経験を持つ開発者を用いた予測	0.00	0.63	0.01	15

でコミッター候補者を予測した結果，本論文で構築したモデルよりも再現率が低くなる（正確に予測できたコミッター候補者数が少ない）．その理由は，パッチ投稿やパッチ検証の経験はないがコメント投稿数の多いコミッター候補者が学習データに含まれていないため，モデルの表現力が下がり，主にコメント投稿を行っているコミッター候補者を予測できなかったからである．パッチ投稿やパッチ検証といった具体的な貢献がなくとも，プロジェクトを運営する力を持つ開発者がコミッターに昇格することがあり [20]，このことから，パッチ投稿やパッチ検証が 0 件の開発者をデータセットに含めることは，主にコメント投稿を行っている開発者を予測するために有効であると考えられる．

RQ1 の分析結果，及び，本節で行ったパッチ投稿とパッチ検証を一度も行っていない開発者を削除した実験の効果量に関する結果から，各活動量でコミッター候補者と一般開発者の活動量の実質的な差は，総回数よりも 1ヶ月あたりの回数の方が顕著に表れることが分かった．実際に Eclipse platform プロジェクトの月パッチ投稿数や月パッチ検証数の上位者を分析すると，月パッチ投稿数が上位 10 名（月パッチ投稿数 15 件以上）の開発者のうち 6 名はコミッター候補者，月パッチ検証数が上位 10 名（月パッチ検証数 3 件以上）の開発者のうち 3 名はコミッター候補者であった．月パッチ投稿数，月パッチ検証数の上位者の一般開発者の中には，今後コミッターに昇格する開発者が存在する可能性が高い．今後，これらの活動量がコミッターを推薦するための目安になることが期待できる．

一般開発者の中にはコミッター候補者と同等以上に活発に活動している開発者が存在しており，その中からコミッターに相応しい開発者を推薦するために，開発者の活動内容（コメント内容やパッチの品質等）を分析することも将来的に必要になると示唆される．

例えば，既存コミッターは開発者が建設的なコメントを多く投稿しているか否かを判断していることが考えられ，開発者のコメントがプロジェクトに良い貢献をもたらしているかを分析することが重要である．また，パッチの品質については，投稿されたパッチが承認されたか否か，また，その投稿されたパッチが不具合を含んでいるか否かを判断する必要がある．しかしながら，従来研究 [4][54] で，投稿されたパッチと構成管理システムへのコミットされたソースコードを関連付けることは困難である．その理由は，1) 多くのコミットは不具合と関連付けられていない，2) 仮に関連付けられていても，パッチを開発した者がコミットするとは限らず，不具合票やコミットログからパッチを開発した者を正確に特定することが困難であるからである．これらの問題を解決するために現在も研究が進められており，今後，それらの研究成果が報告されれば，その成果を用いることでコミッター予測モデルの構築に用いることができると考える．

4.6.3 制約

本章では，コミッター候補者の活動として，パッチ投稿からプロダクト反映までのプロセス中の活動に着目した．モデル構築に用いることのできる開発者の活動量には，コメント投稿，パッチ投稿，パッチ検証以外に，パッチの行数が挙げられる．従来研究 [60] で，投稿される大多数のパッチは 10 行以下であること（パッチの変更行数に差が無いこと）が示されており，開発者が投稿したパッチの総行数は投稿回数と非常に高い相関が出る（パッチ投稿回数の多いコミッター候補者はパッチの総行数が多く，パッチ投稿回数の少ない一般開発者はパッチの総行数が少ない）ことが分かっていたため，本論文ではパッチの総行数を用いなかった．一方で，前節で述べたコメントの内容やパッチの品質はコミッター候補者と一般開発者で技術的な違いがあると考えられる．しかしながら，コミッターによるコメントの質や，承認されるパッチを作った開発者を正確に自動抽出することは現

時点で難しい [4].

コミッターの中にはコミッター昇格前の活動が少ない（一般開発者とほぼ同等の活動量の）開発者が存在する．このような開発者はプロジェクト参加当初からコミッターである可能性が高い．本章では，コミッター昇格前（初めてリポジトリにコミットする以前）に活動していない開発者は除外した．しかし，初めてリポジトリにコミットするまでにパッチ投稿，パッチ検証，コメント投稿を行えば，当該開発者は一般開発者として活動したことになる．OSS プロジェクトの既存コミッターが本手法を実際に適用する場合には，データセットの準備時にコミッター候補者か否かを正確にラベル付けできるため，再現率の向上が見込まれる．

4.7 まとめ

本章では，一般開発者から有能なコミッター候補者を見つけ出すことを目的として，開発者の活動量を用いてコミッターに昇格する開発者をどの程度の精度で予測することができるかを実験的に評価した．Eclipse platform プロジェクト，Mozilla Firefox プロジェクトを対象に実験を行った結果，Eclipse platform プロジェクトでは再現率が約 70%，Mozilla Firefox プロジェクトでは約 80%の精度でコミッターに昇格する開発者を予測できることが分かった．また，継続的にパッチの投稿，パッチの検証を行う開発者がコミッターに昇格していることが分かった．

第5章　OSS利用企業のための不具合修正時期の予測

OSS プロジェクトに報告される不具合の中には，長期間修正されていない不具合や，開発者が積極的な理由で修正を行わない不具合が数多く存在する．このような不具合は，OSS 利用企業が自社で不具合修正を行う必要がある．本章では，OSS 利用企業における不具合修正プロセスの効率化のために，修正を要する不具合が次期リリースまでに修正されるか否かを予測する方法を述べる．

5.1 はじめに

近年，大規模な OSS の不具合を開発者が全て修正することは困難になっている [27]．例えば，Mozilla Firefox プロジェクトに報告された不具合の中で，修正されていない不具合は約 84.4%（29,339 件/34,778 件）も存在している[7] ．

OSS 利用企業にとって，修正を要する不具合が次期リリースまでに修正されるか否かを判断し，修正されない場合，自社で修正することを検討する必要があるため，OSS 利用者は次期リリースまでにどの不具合が修正されるのか見極めることが重要となる．

本章では，多くの不具合報告を受けている大規模 OSS (Eclipse platform, Mozilla Firefox) プロジェクトを題材とし，利用者に公開されている不具合の特徴，及び，修正の進捗情報をメトリクスとして用いて予測に寄与するメトリクスを明らかにし，不具合修正予測モデルを構築と，その予測精度を評価を行う．具体的には，次の 2 つの Research Question に取り組む．

[7] 2011 年 2 月時点.

RQ1:不具合が修正されるか否かを予測するのに寄与するメトリクスは何か？

RQ2:予測日から次のリリースまでに修正される不具合をどの程度の精度で予測できるのか？

従来から不具合の修正にかかる時間を予測する研究が行われている [23][26][27][59]. これらの研究では，各不具合の報告時に予測することを想定しており，不具合の特徴（不具合が発見されたコンポーネントや重要度等）を用いた予測を行っている．一方，本章では任意の時点で予測することを想定しており，不具合の特徴のみならず，修正の進捗情報（修正担当者，不具合内容の変更，報告からの経過時間等）を用いて予測モデルを構築する点が異なり，予測精度の向上が期待できる．

本章では，現リリースで発見された不具合が 3ヶ月後のリリースまでに修正されるか否かを予測する[8]．予測モデルの構築には，不具合の特徴を示すベースメトリクス（不具合が発見されたコンポーネント，不具合の重要度等）に加え，修正の進捗情報を示す 3 種類のメトリクス（状態，期間，参加者）を用いる．状態メトリクスは，修正履歴を基に予測日時点の修正状態，報告から予測日までの進捗状況を示す．期間メトリクスは，不具合報告時期，不具合票の変更からの経過時間を示す．参加者メトリクスは，報告者，修正依頼者，修正担当者が誰なのか，それらの各役割を担う参加者が同一人物か否かを示す．

続く 5.2 節では，関連研究について述べる．5.3 節では，OSS 開発で修正される不具合について述べる．5.4 節では，本章の Research Question にこたえるための実験方法について述べる．5.5 節では，不具合修正予測モデルを構築するために用いるメトリクスについて述べる．5.6 節では，実験に用いるデータセットと，各 Research Question の実験結果について述べる．5.7 節で本章の考察を行い，最後に 5.8 節で本章のまとめを述べる．

[8] 3ヶ月前であれば，OSS プロジェクトからリリース日が公開されており，且つ，リリース日の突然の変更が少ないと考えたからである．

5.2 関連研究

不具合の中には修正に長時間かかる不具合が存在し [7][13][29]，開発者はリリース日に向けて計画的に不具合の修正に取りかかる必要がある．これまで，ソフトウェア開発におけるテスト計画やリリース計画の立案を支援するために不具合の修正にかかる時間を予測する研究が数多く行われている [23][26][27][59].

Hewett ら [26] は，不具合報告時に記録されるコンポーネントや優先度等をメトリクスとして用いて，不具合の修正にかかる時間を予測するためのモデルを構築している．また，Cathrin ら [59] は，報告時に記録されるテキスト情報が類似する不具合の情報を用いて，修正にかかる時間を予測するためのモデルを構築している．ただし，Hewett らや Cathrin らは不具合の修正時間を予測するため，修正不具合のみを対象としており，修正されなかったを対象としていない（Mozilla Firefox の場合，全不具合の約 92%が扱われていない）．一方，本章は，利用者が直面した不具合が修正されるか否かを判断することを支援するため，修正される不具合と修正されない不具合，すなわち全不具合を対象としている．

また，Hooimeijer ら [27] は，開発者が不具合報告に対応する（不具合を修正する，または修正しないと判断する）必要があるか否かを予測するモデルを構築している．ただし，対応策を検討する不具合の中には修正しない不具合（非修正不具合）が存在する．利用者が発見した不具合がモデルによって非修正不具合と判断された場合，利用者は当該不具合を自身で修正せざるを得ない．よって，利用者にとっては報告された不具合に OSS プロジェクトの開発者が対応するか否かを予測するだけでは不十分である．また，Hooimeijer らの研究のように開発者が不具合に対応するか否かを予測する場合は，修正が開始されるまでに得られる不具合の特徴（主にベースメトリクス）を用いてモデルを構築するが，利用者は修正中の不具合を予測する場合がある．利用者が修正途中の不具合を予測する場合，修正の進捗情報を用いて不具合修正予測モデルを構築することにより高い予測精度が期待できる．よって，本章ではベースメトリクスに加え，進捗情報を示す 3 種類のメトリクス（状態，期間，参加者）を用いて不具合がリリースまでに修正されるか否かを予測するモデルを構築する．

従来研究は不具合の修正時間を予測する開発者を支援対象者としているのに対

して，本章は不具合に直面した利用者を支援対象者としていることが最も異なる点である．さらに，本章は報告直後の不具合，修正途中の不具合の両方を対象とし，修正の進捗情報をモデル構築に用いる点が従来研究と異なる．

5.3 不具合の解決

不具合の特徴と修正の進捗情報を示すメトリクス（状態，期間，参加者）を用いた予測を行うにあたって，修正される不具合（修正不具合）と修正されない不具合（未修正不具合）の違いについて説明する．OSS プロジェクトに報告された不具合は，修正不具合，非修正不具合，修正中不具合の 3 つに分類される．

修正不具合 : 修正後，修正内容の確認が行われ，修正が完了した不具合を修正不具合とする．図 2.1 では**修正**，**検証**の後，**完了**に遷移する不具合が該当する．

非修正不具合 : 報告された内容が不具合ではない，不具合の原因がつかめない，次のバージョンでは修正される見込みがないとレビューアによって判断された不具合を非修正不具合とする．図 2.1 では**修正保留**，**検証**の後，**完了**に遷移する不具合が該当する．

修正中不具合 : 修正中，もしくは修正が滞っている不具合を修正中不具合とする．図 2.1 では**完了**まで遷移していない不具合が該当する．

本章の実験で対象とした Eclipse platform プロジェクトと Mozilla Firefox プロジェクトの不具合を**修正不具合**，**非修正不具合**，**修正中不具合**に分類すると，各不具合数は表 5.1 のようになる．多くの不具合は修正中不具合であることが分かる．本章では，次のリリースまでに修正不具合となる不具合を予測するためのモデルを構築する．

5.4 Research Question

本節では，不具合の特徴，及び，修正の進捗情報を用いて不具合の修正時期を予測するために取り組む Research Question と，各 Research Question の実験手順について述べる．

表 5.1 修正される不具合数と修正されない不具合数

		Eclipse platform	Mozilla Firefox
修正不具合		258	443
未修正	非修正不具合	32	400
不具合	修正中不具合	1,986	9,623
全不具合数		2,276	10,466

RQ1: 不具合が修正されるか否かを予測するのに寄与するメトリクスは何か？

動機：プロジェクトに報告される不具合が，OSS 開発者によって修正されるとは限らず，修正されない場合には，OSS 利用企業は自社で当該不具合を修正せざるを得ない．しかしながら，OSS 利用企業が，不具合の進捗情報に基づき修正計画を判断し，修正することは容易ではない．RQ1 では，不具合が次期リリースまでに修正されるか否か見極めるために，不具合の特徴を示すベースメトリクス，及び，進捗情報を示す 3 種類のメトリクス（状態，期間，参加者）を用いて不具合修正予測モデルを構築し，それぞれのメトリクスを用いた時に，有用となるメトリクスを分析する．

手順：RQ1 では，予測モデルに寄与するメトリクスを把握するために，逸脱度を用いる．逸脱度は各変数をモデルに加えた時に変化する尤度（モデルが正しい結果を出力する確率）の変化量である．つまり，逸脱度が大きい変数はモデル構築に大きく寄与していると言える．本章では，各変数の逸脱度をメトリクスごとに総和し，対立仮説（メトリクスを加えたモデル）の逸脱度を帰無仮説（切片のみで構築したモデル）の逸脱度で割ることで，メトリクスがモデルの構築にどの程度寄与しているのかを分析する．本章ではメトリクスの中に名義尺度が含まれており，第 4 章で用いた各変数の平均値と分散から導出する効果量を使用することは適切でない．

RQ2: 予測日から次のリリースまでに修正される不具合をどの程度の精度で予測できるのか？

動機：OSS 利用企業は，OSS プロジェクトが公開している不具合票の情報を参照するが，どの情報を参照すべきか，どの程度の精度で予測できるか明らかでない．RQ2 では，不具合が次期リリースまでに修正されるか否か予測するために，ベースメトリクス，及び，進捗情報を示す 3 種類のメトリクス（状態，期間，参加者）を用いて，不具合修正予測モデルを構築し，修正される不具合がどの程度の精度で予測することができるのか評価実験を行う．

手順：RQ2 では，不具合修正予測モデルを構築し，その予測精度を評価する．本章では，従来研究で用いられていたベースメトリクスのみで構築したモデル，修正の進捗情報から計測した 3 種類のメトリクス（状態，期間，参加者）から構築したモデル，ベースメトリクスと 3 種類のメトリクスを組み合わせて構築したモデルを構築し，比較する．さらに，ベース，状態，期間，参加者からランダムにメトリクス選択し，予測した結果を比較する．

モデルの構築には，名義尺度を含むデータや，正規分布に従っていないデータを用いた多変量解析として用いられるロジスティック回帰分析 [5][40][55] を用いる．任意の期間内に不具合が修正されるか否かを目的変数とし，予測モデルの出力値（0 から 1 の連続値）が 0.5 以上のときに期間内に不具合が修正されると判別する．ただし，多重共線性を避けるために，41 種類の変数から任意に選び出した 2 変数に 0.8 以上の相関があれば，片方を削除する．0.8 以上の相関を持つ組が多い変数は優先的に削除する．

モデルの評価は第 4 章と同様，適合率，再現率，及び F1 値を用いる．適合率は，期間内に修正されると判断された不具合のうち，実際に期間内に修正された不具合の割合を示す．また再現率は，本章で対象とする不具合の中で，期間内に修正されると判別された不具合の割合を示す．

モデル構築に用いるデータ，及び，検証に用いるデータの作成には，交差検証法を用いる [19]．データセットを 10 個のブロックに分割し，9 個のブロックでモデルを構築し，残り 1 個のブロックを検証用に用いる．交差検証法を 1000 回試行し，各試行で得られた評価結果（適合率，再現率，F1 値）の中央値を比較する．

5.5 不具合修正予測モデル構築のためのメトリクス

本実験で用いる 41 種類の変数を表 5.2 に示す．従来研究では，報告時に記録される不具合の特徴を示す変数を中心としたベースメトリクスが用いられていた．本論文では，修正途中の不具合の予測精度向上のために，修正の進捗情報を示す 3 種類のメトリクス（状態，期間，参加者）をさらに用いる．

5.5.1 ベースメトリクスと進捗情報を示すメトリクス

ベースメトリクス：ベースメトリクスには，従来の不具合修正時間の予測に関する研究 [26][27] で用いられてきた一般的なメトリクスが含まれる．報告時に記録される不具合に関する情報（不具合が発見されたコンポーネント (Component) や，不具合が発生する環境（OS (Os)，ハードウェア (Hardware) 等），重要度 (Severity) 等），修正に向けたコメントの投稿数（NumComment），不具合の環境や修正アイデアを記載した添付ファイル数（NumAttachment）が挙げられる．

状態メトリクス：状態メトリクスは，修正履歴を基に予測日時点の修正状態，及び，報告から予測日までの修正状況から計測するメトリクスが含まれる．予測日時点の不具合の状態（CurrentStatus），修正担当者が決定されているかの有無（ChAssignee），各状態を過去に経由した回数（NumModifying, NumSuspending），予測日時点の不具合の状態に変更されてからの経過時間（PeriodOnStatus）が挙げられる．不具合の状態が**審査**，及び，**修正保留**である場合，修正担当者を決定，及び，不具合修正を伴うため修正に長い時間がかかる [2][46].

期間メトリクス：期間メトリクスには，報告年（ReportedYear），報告日（ReportedMonth），ベースメトリクスが変更されてからの経過日数を示すメトリクス等が含まれる．ベースメトリクスが変更されてからの経過日数は，コンポーネントや重要度が変更されてから予測日までの経過時間（LastComponentTime, LastServerityTime）を意味する．もし報告日から予測日までに変更が無ければ，報告日から予測日までの時間とする．報告者は間違った内容を不具合票に記録することがある．このような場合，修正依頼者や修正担当者は誤った内容を修正し，改めて修正担当者を決定するため，修正に長い時間がかかると考えられる．

参加者メトリクス：参加者メトリクスは，報告，修正依頼，修正に関与した者に関するメトリクスが含まれる．報告者（Reporter），修正依頼者（Assignee），修正者（Assignor），3 つの役割を担う開発者が同一人物か否か（Reporter_Assignor, Assignor_Assignee, Reporter_Assignee），修正者の変更回数（NumAssignee）が挙げられる．OSS 開発では報告者が自ら修正を行う場合がある．その場合，不具合の原因をあらかじめ分かっているため短い時間で修正が完了すると考えられる [30]．

5.5.2 名義尺度の扱い方

変数が名義尺度である場合，ダミー変数として展開する．しかし，Component や Reporter の変数をダミー変数として展開すると説明変数が過多になり，モデルが適切に構築されず，予測精度が低下する可能性がある [44][56]．そのため本章では，出現頻度が高い上位 3 つはダミー変数化し，それ以外は一つの変数（others）にまとめる．例えば，報告される不具合は，Component が com-A, com-B, com-C, com-D, com-E の順に多いとする．この場合，com-A, com-B, com-C はそのままダミー変数化し，com-D と com-E は others に置き換える．

表 5.2 メトリクスの概要

メトリクス	変数名	尺度	詳細
ベース	NumDescriptionWord	間隔	不具合票に記載される不具合の詳細情報のワード数
	NumAttachment	間隔	添付ファイル数
	Keywords	名義	キーワード記載の有無
	Component	名義	コンポーネント名
	Priority	名義	優先度の程度 (high, normal, low)
	Severity	名義	重要度の程度 (high, normal, low)
	Os	名義	オペレーシングシステム名
	Hardware	名義	ハードウェアのタイプ (x86, PowerPC 等)
	Milestone	名義	マイルストーン名
	NumComment	間隔	コメント数
	Cc	間隔	不具合票の更新情報を受け取っている人数
	Blocks	間隔	当該不具合の解決を妨げている不具合の数
	Dependson	間隔	当該不具合に依存している不具合数
状態	NumModifying	間隔	不具合の修正回数
	NumSuspending	間隔	修正保留に判断された回数
	ChAssignee	名義	修正担当者決定の有無
	CurrentStatus	名義	予測日時点の不具合の状態
	PeriodOnStatus	間隔	予測日時点の不具合の状態に変更されてからの経過時間（日）
期間	ReportedYear	名義	報告年
	ReportedMonth	名義	報告月
	LastAttachmentTime	間隔	予測日直前に添付ファイルが追加されてから予測日までの時間（日）
	LastKeywordsTime	間隔	予測日直前にキーワードが変更されてから予測日までの時間（日）
	LastComponentTime	間隔	予測日直前にコンポーネントが変更されてから予測日までの時間（日）
	LastPriorityTime	間隔	予測日直前に優先度が変更されてから予測日までの時間（日）
	LastSeverityTime	間隔	予測日直前に重要度が変更されてから予測日までの時間（日）
	LastOsTime	間隔	予測日直前に OS が変更されてから予測日までの時間（日）
	LastHardwareTime	間隔	予測日直前にハードウェアが変更されてから予測日までの時間（日）
	LastMilestoneTime	間隔	予測日直前にマイルストーンが変更されてから予測日までの時間（日）
	LastCommentTime	間隔	予測日直前にコメントが投稿されてから予測日までの時間（日）
	LastCcTime	間隔	予測日直前に CC が追加/削除されてから予測日までの時間（日）
	LastBlocksTime	間隔	予測日直前に不具合の修正を妨害する不具合を追加/削除されてから予測日までの時間（日）
	LastDependsonTime	間隔	予測日直前に不具合の修正を依存する不具合を追加/削除されてから予測日までの時間（日）
	LastAssigneeTime	間隔	予測日直前に担当者が追加/変更されてから予測日までの時間（日）
	RepNowTime	間隔	不具合報告されてから予測日までの時間（日）
参加者	Reporter	名義	報告者の email アドレス
	Assignee	名義	修正者の email アドレス
	Assignor	名義	修正依頼者の email アドレス
	Reporter_Assignor	名義	reporter（報告者）と assignor（修正依頼者）が同一人物か否か
	Assignor_Assignee	名義	assignor（修正依頼者）と assignee（修正者）が同一人物か否か
	Reporter_Assignee	名義	reporter（報告者）と assignee（修正者）が同一人物か否か
	NumAssignee	間隔	修正者の変更回数

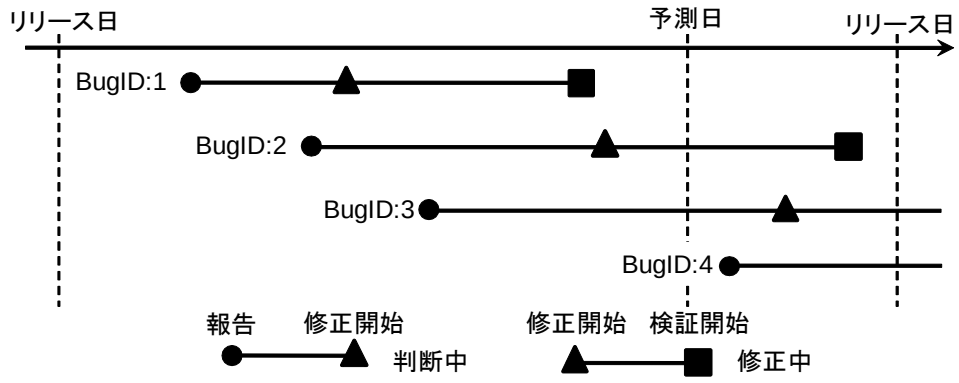


図 5.1 実験対象不具合

5.6 実験

本節では，OSS プロジェクトの開発者によって不具合が修正されるか否かを予測するための不具合修正予測モデルを構築し，実験を通して予測精度向上のために有用なメトリクスを明らかにする．

5.6.1 概要

一般的な OSS プロジェクトはリリースに向けて数多くの不具合を修正する．本章では，不具合が 3ヶ月後のリリースまでに修正されるか否かを予測する実験を行う．本実験で対象とする不具合を図 5.1 に示す．予測日の時点で修正中の不具合のみ（予測日以降に修正されるか否かは問わない）を実験対象とするため，BugID:2 と 3 が実験対象であり，それぞれ表 5.1 の修正不具合と未修正不具合に対応する．これらの不具合がリリース日までに修正されるか否かを予測する．図 5.1 の BugID:2 のように予測日からリリース日の 3ヶ月間に解決した不具合（図 2.1 で，**修正**後，**検証**に遷移する不具合）の予測精度を評価する．

5.6.2 実験対象プロジェクトの選出条件

第 4 章と同様に本章でも多くの不具合を管理している 2 つの OSS (Eclipse platform， Mozilla Firefox) プロジェクト を対象に実験を行った．対象プロジェクト

表 5.3 実験対象データの概要

	Eclipse platform	Mozilla Firefox
全不具合数	2,276	10,466
修正不具合数	183	230
未修正不具合数	2,093	10,236

に関する不具合件数を表 5.3 に示す．予測日からリリース日までに不具合の修正が解決した場合を修正不具合とし，それ以外を非修正不具合とする．本章で対象とする 2 つの OSS プロジェクトは (1) 不具合管理のために Bugzilla を利用しており実験結果の比較が可能なこと，(2) 定期的にリリースを行っており，プロジェクト運営が安定していること，(3) 知見の一般性を高めるに足る十分なデータ（不具合報告数）が得られること，を主な理由として選定した．

5.6.3 実験結果

RQ1: 不具合が修正されるか否かを予測するのに寄与するメトリクスは何か？

両プロジェクトにおいて，4 種類のメトリクスがモデル構築にどの程度寄与しているかを分析するために，逸脱度を調べた．メトリクスごとの逸脱度，及び，4 種類全てを用いた時の逸脱度を表 5.4 に示す．また，各変数の逸脱度を図 5.2 に示す．各プロジェクトで逸脱度の上位 10 変数のみ変数名を記載している．

実験の結果，モデル構築にベースメトリクスを用いた時の逸脱度が最も大きいため，ベースメトリクスがモデル構築に最も寄与していることが分かった．次に状態メトリクスがモデル構築に寄与していることが分かった．よって，修正されるか否かを予測するためには，ベースメトリクスを説明変数として用いることを前提として，次に状態メトリクスが有用であることが分かった．

表 5.4 各メトリクスの逸脱度

	Eclipse platform	Mozilla Firefox
ベース	27.19	28.82
状態	11.50	15.11
期間	2.04	1.01
参加者	3.27	1.49
全て	44.00	46.43

また，図 5.2 の各変数の逸脱度を分析した結果，両プロジェクトともに優先度が低いか否か（Priority_low），予測日時点に修正の状態である不具合が，当該状態に変更されてからの経過時間が 1 カ月以内であるか否か（PeriodOnStatus_Modifying_0-30），がモデル構築に寄与していることが分かった．優先度が低い不具合は修正時間が長いため [25][43]，リリース日までに修正されない不具合を判断するのに寄与していると考えられる．修正の状態に変更されてからの経過時間については，考察で議論する．

RQ2: 予測日から次のリリースまでに修正される不具合をどの程度の精度で予測できるのか？

不具合修正予測モデルによって予測する場合とランダムに予測した場合の適合率，再現率及び F1 値を表 5.5 示す．また，最も F1 値が高かった予測モデルが，ベースメトリクスのみで構築したモデル，またはランダムに予測した場合と比べて，どの程度精度が向上しているかを向上率として示す．実験の結果，各メトリクスを単体で用いるよりも，ベースメトリクスに他のメトリクスを加えた方が F1 値は高いことが分かった．そして，F1 値は Eclipse platform プロジェクト，Mozilla Firefox プロジェクト共にベースメトリクスと状態メトリクスを説明変数としてモデルを構築した時が最大となった．具体的な精度は，Eclipse platform プロジェクトの適合率が約 31%，Mozilla Firefox プロジェクトの適合率が約 12%であり，再現率がそれぞれのプロジェクトで約 71%と約 85%であった．ベースメトリクスと状態メトリクスを用いて構築した予測モデルはベースメトリクスのみで構築したモデ

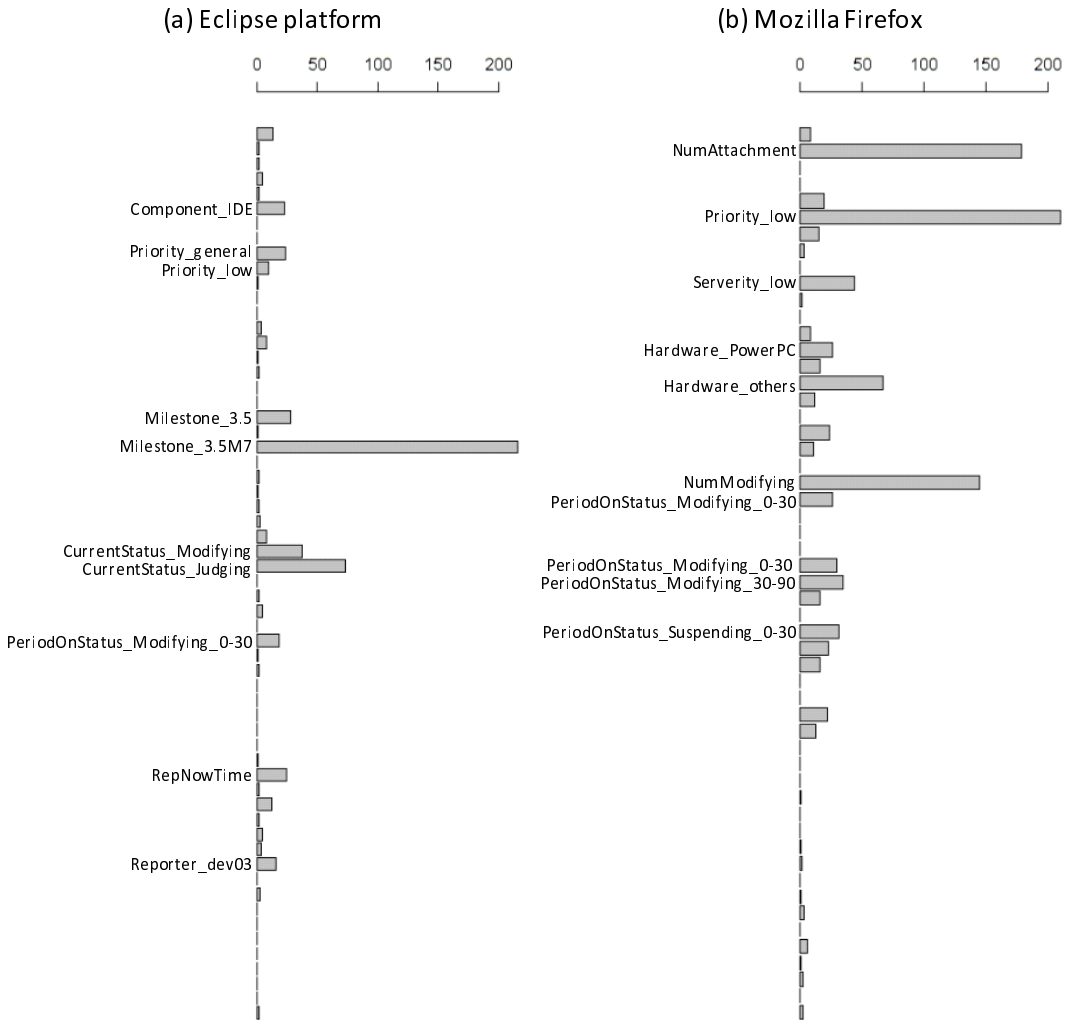

(a) Eclipse platform
0
50
100
150
200
Component_IDE
Priority_general
Priority_low
Milestone_3.5
Milestone_3.5M7
CurrentStatus_Modifying
CurrentStatus_Judging
PeriodOnStatus_Modifying_0-30
RepNowTime
Reporter_dev03
(b) Mozilla Firefox
0
50
100
150
200
NumAttachment
Priority_low
Serverity_low
Hardware_PowerPC
Hardware_others
NumModifying
PeriodOnStatus_Modifying_0-30
PeriodOnStatus_Modifying_0-30
PeriodOnStatus_Modifying_30-90
PeriodOnStatus_Suspending_0-30

図 5.2 各変数の逸脱度

表 5.5 修正不具合の予測結果

		Eclipse platform			Mozilla Firefox		
		適合率	再現率	F1 値	適合率	再現率	F1 値
本実験モデルで予測	ベース	0.23	0.62	0.33	0.09	0.71	0.16
	状態	0.24	0.66	0.35	0.08	0.82	0.14
	期間	0.18	0.52	0.27	0.06	0.71	0.11
	参加者	0.31	0.49	0.29	0.07	0.65	0.13
	ベース+状態	**0.31**	**0.71**	**0.43**	**0.12**	**0.85**	**0.20**
	ベース+期間	0.26	0.69	0.38	0.10	0.78	0.18
	ベース+参加者	0.24	0.66	0.35	0.10	0.79	0.18
	全て	0.25	0.78	0.38	0.11	0.82	0.19
ランダムで予測		0.24	0.57	0.33	0.08	0.71	0.14
予測モデル（ベースのみ）に対する向上率（%）[1]		1.39	1.14	1.31	1.24	1.14	1.23
ランダムで予測した場合に対する向上率（%）		1.29	1.25	1.30	1.50	1.20	1.43

（注 1）最も F1 値が高かった予測モデル（ベース+状態）が，ベースメトリクスのみで構築したモデルと比べて，どの程度精度が向上しているかを示す.

ルに比べて，適合率が約 24%〜39%，再現率が約 14%向上した．また，ベースメトリクスと状態メトリクスを用いて構築した予測モデルはランダムに予測するに比べて適合率が約 1.29〜1.50 倍，再現率が約 1.20〜1.25 倍の精度で予測できることが分かった.

本実験では適合率が低い値となったが（Eclipse platform プロジェクトで約 31%，Mozilla Firefox プロジェクトで約 12%），その原因は，予測日からリリースまでに修正される不具合と修正されない不具合の比が大きく偏っている点にある．本実験では，ロジスティック回帰分析の出力値（不具合修正の期待値）が 0.5 以上の場合に不具合は 3 カ月以内に修正されると判定した

が，この閾値を大きくすることで適合率を向上することができる．例えば，閾値を 0.8 に設定した場合，各プロジェクトの適合率は Eclipse platform プロジェクトで約 43%（再現率: 約 59%），Mozilla Firefox プロジェクトで約 20%（再現率: 約 67%）になる．ただし，閾値を大きくし過ぎると，修正されると判断される不具合数が減少するため，再現率が低下する点に注意する必要がある．

5.7 考察

5.7.1 修正される可能性が高い不具合

RQ2 の結果より，不具合の修正を予測するためには，ベースメトリクスに加え，状態メトリクスが有用であることが分かった．このことから，状態メトリクスが開発者によって修正されるか否かを予測するために影響していると言える．そこで，状態メトリクスとして用いている予測時の状態と予測時の状態に遷移してからの経過時間を示すメトリクスが，開発者によって不具合がリリースまでに修正されるか否かを利用者が判断する目安として用いることができるか否かをオッズ比を用いて分析する．オッズ比 (OR) はある事象の起こりやすさを 2 つの群で比較するための尺度であり，以下の式で求められる．

$$OR = \frac{p/(1-p)}{q/(1-q)}$$

本章では，予測日時点で不具合が任意の状態かつ任意の経過時間である時に，修正される不具合の割合を p，それ以外の状態，経過時間である時に，修正される不具合の割合を q とする．例えば，Eclipse platform プロジェクトで不具合の状態が**審査**で，且つ経過時間が 1ヶ月以内である場合，修正される不具合は 138 件，修正されない不具合は 1,690 件であるため，p は約 0.08 ($\frac{138}{138+1690} \approx 0.08$)，分子は約 0.08 となる．また，不具合の状態が**審査**，且つ経過時間が 1ヶ月以内でない場合，修正される不具合は 45 件，修正されない不具合は 403 件であるため，q は約 0.10 ($\frac{45}{45+403} \approx 0.11$)，分母は 0.11 となる．よって，**審査**で，且つ経過時間が 1ヶ月以内である時のオッズ比は 0.73 となる．

オッズ比の結果を，表5.6，表5.7に示す．3行目はp，4行目はq，5行目がオッズ比を示す．オッズ比が1以上であるとき，任意の状態で任意の経過時間である不具合は，当状態かつ経過時間でない不具合よりも，リリースまでに修正される可能性が高いことを意味する．例えば，Eclipse platformプロジェクトの不具合において，予測時に不具合の状態が**審査**であり，且つ**審査**に変更してからの経過時間が1ヶ月以内 (0-30) である場合，オッズ比は0.73であることから，この状態の不具合はリリースまでに修正される可能性が低いことが分かる．

表5.6，表5.7より，予測日（リリース3ヶ月前）に不具合の状態が**審査**であった場合，Eclipse platformプロジェクトでは経過時間が長いほどリリースまでに不具合が修正される可能性が高く，Mozilla Firefoxプロジェクトでは経過時間が1カ月以内である不具合がリリースまでに最も修正される可能性が高いことが分かる．つまりEclipse platformプロジェクトでは以前に報告された不具合から修正される傾向にあり，Mozilla Firefoxプロジェクトでは状態が変更されて1ヶ月以内の不具合が修正される可能性が高いことが分かる．また，予測時点で不具合の状態が**修正**である場合，たいていのオッズ比は1以上であるため，他の状態の不具合よりも修正される可能性が高いことを示している．特に**修正**に遷移してから1ヶ月以内の不具合は，RQ1でもモデルに寄与していることが分かっており，修正されやすい状態であると示唆される．その一方，予測時点で**修正保留**である場合，Eclipse platformプロジェクトでは**審査**や**修正**の状態である方がリリースまでに修正される可能性が高く，Mozilla Firefoxプロジェクトでは，**修正保留**に遷移して3ヶ月以内であればリリースまでに修正される可能性が高いが，半年以上経過すると修正される可能性が低いことが分かった．

以上より，予測日に不具合の状態が**審査**や**修正**の不具合はリリースまでに修正される可能性が高いが，**修正保留**の不具合は経過時間が長くなるにつれて修正される可能性が低くなる．結果の傾向はプロジェクトによって一部異なるが，予測日時点の状態と経過時間は，リリースまでに修正されるか否かを判断する目安になることが示唆される．

表 5.6 各不具合の状態に対する修正される不具合と修正されない不具合のオッズ比 (Eclipse platform)

状態	審査				修正				修正保留			
経過時間	0-30	30-90	90-180	180-	0-30	30-90	90-180	180-	0-30	30-90	90-180	180-
p	0.08	0.10	0.36	0.57	0.42	0.42	0.11	0.30	0.07	0.07	0.04	0.00
q	0.11	0.07	0.07	0.08	0.08	0.08	0.09	0.06	0.09	0.09	0.10	0.09
オッズ比	0.73	**1.49**	**5.13**	**6.92**	**5.09**	**5.18**	**1.31**	**4.85**	0.81	0.77	0.40	0.00

表 5.7 各不具合の状態に対する修正される不具合と修正されない不具合のオッズ比 (Mozilla Firefox)

状態	審査				修正				修正保留			
経過時間	0-30	30-90	90-180	180-	0-30	30-90	90-180	180-	0-30	30-90	90-180	180-
p	0.23	0.00	0.03	0.03	0.09	0.08	0.03	0.00	0.10	0.05	0.02	0.00
q	0.02	0.03	0.02	0.02	0.02	0.02	0.02	0.03	0.02	0.02	0.02	0.03
オッズ比	**11.50**	0.00	**1.50**	**1.50**	**4.50**	**4.00**	**1.50**	0.00	**5.00**	**2.50**	1.00	0.00

5.7.2 不具合修正予測モデルに有用でなかったメトリクス

RQ1 より期間メトリクスや参加者メトリクスはベースメトリクスや状態メトリクスに比べてモデル構築に寄与していないことが分かった．

期間メトリクスがモデル構築に寄与していなかったのは，多くの変数間に高い相関があり，モデル構築時に多重共線性を避けるため削除したことが原因と考えられる．

また，参加者メトリクスがモデル構築に寄与していなかったのは，上位 3 人のみ開発者として判断しており，4 番目以降の開発者はとして一つの変数 (others) にまとめたことが原因と考えられる．表 5.8 に，不具合報告，修正依頼，修正の回数の多い 10 名がそれぞれ携わっている不具合の中でリリースまでに修正された不具合数を示す．分析の結果，開発者によって，全く修正されない場合があることが分かった．Eclipse platform プロジェクトでは修正担当者に dev02，dev11，dev18，dev19 が割り当てられると不具合が修正されず，Mozilla Firefox プロジェクトでは，修正依頼者が dev10 以外であれば修正されないことが分かる．修正される割合が高い参加者と低い参加者が混在しているため，本論文で定義した参加者メトリクスでは，修正される不具合を予測するモデルの精度向上を期待できない．今後，不具合報告，修正依頼，修正を担当した開発者，またその開発者の役割（コミッターまたは一般開発者）を調査し，相関ルール分析を用いて開発者ごとの特徴を学習することで予測精度の向上が期待される．

5.7.3 制約

本章では，不具合が修正されるか否かを予測するために 4 種類のメトリクスを用いてモデルを構築した．従来までの研究ではベースメトリクスのみで不具合予測，修正時間予測が行われていたため，これまで利用されていなかった変数を用いたことは本研究の新規性の一つである．しかし，本章で用いた BTS から得られるメトリクスだけでなく，メーリングリストなどで行われる議論もリリースまでに修正されるか否かに影響する可能性がある．不具合修正予測モデルの精度を向上させるために，不具合修正に向けたその他の活動についても調査を続ける．

本実験で用いたメトリクスは多くの OSS プロジェクトで利用されている Bugzilla

を始めとして，Trac や Redmine からも同様に取得することができる．そのため，Bugzilla 以外の BTS を用いているプロジェクトであっても，同様の実験を再現することができる．また，日々数多くの不具合が報告され，それらを BTS で管理している Eclipse platform プロジェクトと Mozilla Firefox プロジェクトにおいて同様の結果（F1 値はベースメトリクスと状態メトリクスを用いたモデルが最も高く，予測するために有用であるメトリクスはベースメトリクスと状態メトリクス）であることから，その他の大規模な OSS プロジェクトを対象としたとしても同様の結果が得られると考えられる．しかしながら，プロジェクトの規模や，リリースが不定期に行われている場合は結果が異なると考えられる．不具合修正予測モデルの一般性を確かめるために，プロジェクト規模やリリースが不定期に行われているプロジェクトに対して調査を続ける．

本章で用いた名義尺度のメトリクスは，出現頻度が高い上位 3 つの変数をそのまま用い，4 番目以降の変数を一つの変数 (others) にまとめた．しかしながら，変数によっては出現頻度が 4 番目以降の変数もそのまま用いることで，より予測精度の高いモデルを作ることができる可能性がある．例えば，不具合報告者の報告回数が少なくても，開発経験を持つ開発者であれば，報告時に修正するための有益な情報を提供するため当該不具合が修正される可能性が高いと考えられる．よって，変数の出現頻度が低くても予測精度の向上に寄与する場合があると考えられる．

また本章の実験では，予測日をリリースの 3ヶ月前に設定した．これは多くの OSS プロジェクトが 1 年に 1 回程度のメジャーリリースを行っていることを考慮して決定した．予測日からリリースまでの期間が長過ぎる場合，リリース日が変更される可能性があるため，リリース日が変更された時点でモデルを再構築する必要がある．また，予測日からリリースまでの期間が短い場合，修正される不具合が少なすぎてモデルの評価が難しいと考えられる．

表 5.8 開発者別の修正数

Eclipse platform

	報告者				修正依頼者				修正担当者			
	名前	変換後	投稿数	修正数	名前	変換後	投稿数	修正数	名前	変換後	投稿数	修正数
1	dev01	dev01	43	14	—	other	740	46	—	other	740	46
2	dev02	dev02	39	1	dev11	dev11	219	0	dev06	dev06	126	23
3	dev03	dev03	38	12	dev12	dev12	215	8	dev02	dev02	119	0
4	dev04	other	34	2	dev06	dev06	194	17	dev16	dev16	108	2
5	dev05	other	33	8	dev13	other	184	5	dev17	other	95	3
6	dev06	other	31	5	dev14	other	122	2	dev18	other	84	0
7	dev07	other	31	8	dev04	other	60	3	dev12	other	69	16
8	dev08	other	26	1	dev03	other	59	0	dev19	other	61	0
9	dev09	other	24	6	dev15	other	57	4	dev13	other	61	13
10	dev10	other	22	4	dev16	other	50	5	dev11	other	57	0

Mozilla Firefox

	報告者				修正依頼者				修正担当者			
	名前	変換後	投稿数	修正数	名前	変換後	投稿数	修正数	名前	変換後	投稿数	修正数
1	—	others	0	0	—	others	10,043	151	—	others	10,043	151
2	dev01	dev01	85	3	nobody	others	111	0	dev17	dev17	41	15
3	dev02	dev02	70	5	dev02	dev02	29	0	dev02	dev02	40	11
4	dev03	dev03	59	3	dev10	dev10	23	5	dev10	dev10	25	7
5	dev04	others	50	16	dev11	dev11	18	0	dev18	others	22	1
6	dev05	others	49	3	dev12	others	15	0	dev19	others	19	6
7	dev06	others	38	1	dev13	others	15	0	dev13	others	14	6
8	dev07	others	35	1	dev14	others	14	0	dev20	others	13	2
9	dev08	others	35	3	dev15	others	13	0	dev21	others	12	0
10	dev09	others	32	2	dev16	others	10	0	dev22	others	12	0

5.8 まとめ

本章では，OSS プロジェクトによって不具合が修正されるか否かを判断する利用者を支援するために，不具合の特徴，修正の進捗状況に関するメトリクスに基づき，リリースまでに修正される不具合をどの程度の精度で予測することができるかを実験的に評価した．Eclipse platform プロジェクト，Mozilla Firefox プロジェクトを対象に実験を行った結果，ベースメトリクスに状態メトリクスを加えることで適合率が約 24%〜39%，再現率が 14%向上した．

第6章　結論

本論文は，大規模OSSプロジェクト，及び，OSS利用企業の各々の立場における不具合修正プロセスの効率化に向けて，OSSの不具合修正に貢献する有能なコミッターを見つけ出す方法，及び，不具合の修正時期を予測する方法を提案した．

第3章では，大規模OSSプロジェクトにおいて不具合の修正が効率的に実施されていない原因となる作業を特定するために，BTSに記録される不具合修正の進捗情報を用いて，過去の不具合修正で必要となった一連の作業内容や個々の作業時間を可視化し，分析を行った．大規模OSS（Apache HTTP Server，Eclipse platform，Mozilla Firefox）プロジェクトを対象にケーススタディを行った結果，プロジェクトの不具合修正プロセスの一連の作業を可視化することで，プロジェクトにおいて時間を要するフェーズや作業を網羅的かつ俯瞰的に把握することができた．

本論文で対象としたOSSプロジェクトは，社会に広く流通しているプロジェクトの一部に過ぎないが，その他のプロジェクトも同様に，広く流通したが故に報告される大量の不具合と向き合わざるを得ない事態となっている．このような現状を多くの利用者は認識しておらず，近年開発に携わる利用者が減少し，コミッターへの負荷は過大になる一方である．誰でも自由にプロジェクトに参加できるというOSSの開発形態であるバザール方式は，プロジェクト発足当初に開発者を確保するために有効であるが，プロジェクトの大規模化に伴い膨大な不具合報告数と，修正に足る十分な開発者（特に高い技術力やプロジェクトを管理する能力を持つコミッター）数を確保する必要性が示唆される．

また，大規模OSSプロジェクトの不具合の修正には約1ヶ月以上かかっていることも多く，OSS利用企業がOSSに不具合を発見した場合，早期の対応が期待できないと考えられる．よって，OSS利用企業による修正を前提として，OSSを

利用することが必要である．

次に，第 4 章では，大規模 OSS プロジェクトにおける膨大な不具合の修正を効率的に実施することを目的として，コミッターに昇格した開発者の過去の活動を分析し，一般開発者の活動量の中からコミッターに昇格する開発者をどの程度の精度で予測することができるかを実験的に評価した．Eclipse platform プロジェクトや Mozilla Firefox プロジェクトを対象に実験を行った結果，再現率が約 70%，Mozilla Firefox プロジェクトでは約 80%の精度でコミッター候補者を予測することができた．また，継続的にパッチの投稿，パッチの検証を行う一般開発者がコミッターに昇格していることが分かった．

大規模な OSS プロジェクトにおいて，膨大な不具合に対応するコミッターの過大な負担を軽減するためには，有能なコミッターに早い段階でコミット権限を与え，既存コミッターへの負担を軽減させることが重要である．しかしながら，安易にコミッターに適していない一般開発者にコミット権限を与えると，プロジェクトに悪影響（不具合の混入，開発者間の争い）を及ぼすおそれがある．今後，プロジェクトの大規模化に伴い不具合数の増加は大きな課題と言え，膨大な不具合の対応に向けてコミッターをどの程度増員させるかは検討する必要がある．

最後に，第 5 章では，OSS 利用企業における不具合修正プロセス効率化のために，利用者に公開されている不具合の特徴，及び，修正の進捗情報をメトリクスとして用いて，修正を要する不具合が OSS の次期リリースまでに修正されるか否かの予測を試みた．そして，その予測精度を評価するとともに，予測に寄与するメトリクスを分析した．Eclipse platform プロジェクトと Mozilla Firefox プロジェクトを対象として実験を行った結果，不具合の特徴を示すベースメトリクスに加え，不具合の状態に関するメトリクスを用いてモデル構築を行うことで，適合率は約 12%〜31%，適合率が適合率が約 71%〜85%となり，ベースメトリクスのみで予測した時に比べ適合率は約 24%〜39%，再現率が 14%向上することが分かった．

OSS 利用企業の多くの修正担当者は，OSS プロジェクトに参加していないた

め，OSS プロジェクトの開発状況を把握することが難しく，OSS に発見した不具合の修正計画，及び修正コストの計画を立てることは容易ではない．本論文で構築した不具合修正予測モデルを用いることで，修正されるか判断できず，全ての不具合を修正する場合と比較すると，修正コストを削減することができる．より正確にプロジェクトの修正計画を把握するためには，利用者がプロジェクトに積極的に参加し，OSS 開発に携わることが求められる．

現在，OSS は個人利用だけでなく，法人利用も増加しているため，OSS の社会的重要性が高まっている．OSS が広く流通したが故に，開発者らは膨大な不具合と対峙せざるを得ない．低コストで，高機能な OSS を利用するためには，利用者自身も開発者の一人という認識を持ち，積極的に OSS 開発へ関与する姿勢が求められる．本論文の成果は，大規模 OSS プロジェクト，及び，OSS 利用企業において不具合修正プロセスが効率化されるとともに，開発者と利用者の双方による OSS の開発が促されることが見込まれる．これによって，今後数多くの不具合が修正され，高品質な OSS の実現に結びつくことを切に願っている．

謝辞

本研究を進めるに当たり多くの方々に，御指導，御協力，御支援を賜りました．ここに御世話になった方々へ感謝の意と御名前を記させて頂きます．

はじめに，主指導教員であり本論文の審査委員を務めて頂いた，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，厚く御礼を申し上げます．大学院入学以来，研究者としての心構えや在り方等，数多くの御指導を賜りました．そして，研究活動だけでなく，社会活動や教育活動を経験する場を提供して頂きました．特に海外の著名な研究者と交流する機会を数多く頂き，本論文に関するフィードバックを頂くことができました．私が本研究を成し遂げることができたのは，研究に対する深い御理解と温かい御助言に励まされたおかげです．先生の御尽力に敬意を表し，心より感謝致します．

副指導教員であり本論文の審査委員を務めて頂いた，九州大学 大学院システム情報科学研究院 情報知能工学部門 鵜林 尚靖 教授に対し，厚く御礼を申し上げます．先生の御好意により，2ヶ月間，九州大学での研究活動の受け入れを快諾していただき，数多くの建設的なアドバイス，及び，集中して研究を行う環境を賜りました．私が本研究について探究できたのは，先生の御協力のおかげです．先生の御尽力に敬意を表し，心より感謝致します．

副指導教員であり本論文の審査委員を務めて頂いた，奈良先端科学技術大学院大学 情報科学研究科 関 浩之 教授に対し，厚く申し上げます．本研究を進めるに当たり，非常に細やかな御指導を賜りました．また，学内の発表では多数の御意見と御指導を頂きました．先生の御尽力に敬意を表し，心より感謝致します．

副指導教員であり本論文の審査委員を務めて頂いた，奈良先端科学技術大学院大学 情報科学研究科 門田 暁人 准教授に対し，厚く御礼申し上げます．本研究に対して多くの建設的なアドバイス，研究の伝え方等，数多くの御指導を賜りま

した．研究活動だけでなく，社会活動の場においても数々の御助言を賜り，数多くのことを学ぶことができました．先生の御尽力に敬意を表し，心より感謝致します．

副指導教員であり本論文の審査委員を務めて頂いた，奈良先端科学技術大学院大学 情報科学研究科 大平 雅雄 助教に対し，厚く御礼申し上げます．大学院入学以来，研究，生活，進路等について数多くの熱意ある御指導を賜りました．本論文をまとめるに至るまで，私を対等の研究者として見ていただき，幾度となく議論をさせて頂きました．議論を通して，研究の興味深さ，研究に対する姿勢等，数多くのことを学ぶことができました．また，普段から大変親しく接していただき，研究活動を離れても楽しい時間を共有させて頂きました．先生の御尽力に敬意を表し，心より感謝致します

九州大学 大学院システム情報科学研究院 情報知能工学部門 亀井 靖高 助教に対し，厚く御礼申し上げます．大学院入学以来，研究，生活，進路等，様々な面で数多くの御意見を賜りました．本論文を進めるにあたり，数多くの議論の機会を設けて頂きました．学術論文誌執筆のため，九州大学で 2ヶ月間滞在した時は，御宅に居候させて頂き，日常生活においても心優しく接して頂きました．心より感謝致します．

カナダ Queen's 大学 Ahmed E. Hassan 助教に対し，厚く御礼申し上げます．本論文を進めるにあたり，私の拙い英語を理解していただき，数多くの御指導を賜りました．また，世界レベルの研究者としての心がけを学ばせて頂きました．研究に限らず，私自身の将来についても数多くのアドバイスを頂きました．心より感謝致します．

カナダ Queen's 大学 Emad Shihab 氏に対し，厚く御礼申し上げます．本論文を進めるにあたり，数多くの御意見を賜りました．国際会議で初めて出会い，同じ博士後期課程の学生でありながら高いモチベーションを持つ氏から，数々の刺激を受けました．心より感謝致します．

OpenOffice.org 日本語プロジェクトリード 中田 真秀 氏に対し，厚く御礼申し上げます．私が OSS の開発記録に基づき分析した知見とは必ずしも一致しない開発者の心情について御聞きし，本研究の必要性を痛切に感じました．そして，本

論文を勧めるにあたり，開発者の視点から数多くの建設的な御指摘，御意見を賜りました．また，研究に限らず，日常生活においても心優しく接して頂きました．心より感謝致します．

静岡大学 情報学部 情報社会学科 森崎 修司 助教に対し，厚く御礼申し上げます．本論文について一歩離れたところから有益な御指摘，御意見を頂きました．また，社会における研究者の位置づけ，研究者としての心構えを御指導して頂きました．心より感謝致します．

神戸大学大学院 システム情報学研究科 柗本 真佑 特命助教に対し，厚く御礼申し上げます．論文執筆に対するアドバイス，プレゼンテーションの作法等，適切かつ建設的なアドバイスを数多く賜りました．氏の研究に対する真摯な姿勢に心打たれ，研究意欲を駆り立てられました．また，研究に限らず，日常生活においても心優しく接して頂きました．心より感謝致します．

奈良先端科学技術大学院大学 情報科学研究科 Mike Barker 氏に対し，厚く御礼申し上げます．国際会議論文執筆，発表資料作成にあたり，英語の知識が未熟な私に懇切丁寧に手助けを賜りました．心より感謝致します．

奈良先端科学技術大学院大学 ソフトウェア工学研究室 乾 純子 氏に対し，厚く御礼申し上げます．研究の遂行に必要な事務処理など多岐に渡り手助けを賜りました．心より感謝致します．

奈良先端科学技術大学院大学 情報科学研究科 角田 雅照 特任助教，戸田 航史 研究員に対し，厚く御礼申し上げます．本論文について一歩離れたところから有益な御指摘，御意見を賜りました．研究だけでなく，日常生活においても心優しく接して頂きました．心より感謝致します．

奈良先端科学技術大学院大学 ソフトウェア工学研究室ならびにソフトウェア設計学研究室の皆様には日頃より多大な御協力と御助言を頂きました．特に同期である，木浦 幹雄 氏（キャノン株式会社），左藤 裕紀 氏（株式会社日立製作所），田村 晃一 氏（株式会社野村総合研究所），松田 侑子 氏（シャープ株式会社），石田 響子 氏（株式会社日立製作所），高田 純 氏（日本マイクロソフト株式会社），山科 隆伸 氏（日本ユニシス株式会社）には研究や日々の生活で多大なご支援を頂き，数々の楽しい時間を共有させて頂きました．また，後輩である小山

貴和子 氏（株式会社東芝），山本 瑞起 氏（株式会社トプコン），大澤 直哉 氏（株式会社リクルート），藤田 将司 氏（株式会社日立製作所），正木 仁 氏（奈良先端科学技術大学院大学），Papon Yongpisanpop（奈良先端科学技術大学院大学），には，論文の体裁チェック，研究発表の練習，研究補助に数多く付き合って頂きました．研究と大学院の生活全般において，皆様からの数々の御支援，励ましが私の大きな支えとなり，充実した 5 年間を過ごすことができました．心より感謝致します．

その他にも，本論文を遂行するにあたり数多くの方々から有益な御意見，励ましの御言葉を頂きました．

TIS 株式会社 橋本 則之 氏には，博士後期課程への進学に対して親身になって相談に乗っていただき，数々の激励の御言葉を賜りました．心より感謝致します．

ECC 外語学院の先生，スタッフ，生徒の皆様には，日ごろから数々の激励を賜り，皆様のお言葉ひとつひとつが私の励みとなりました．心より感謝致します．

最後に，日頃より私を温かい言葉で励まし，支えてくれた家族に対し，心より深く感謝致します．

参考文献

[1] Roberto Abreu and Rahul Premraj. How developer communication frequency relates to bug introducing changes. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol'09)*, pp. 153–158, 2009.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pp. 361–370, 2006.

[3] John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. In *ACM Transaction Software Engineering Methodology*, Vol. 20, pp. 1–35, 2011.

[4] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE'10)*, pp. 97–106, 2010.

[5] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. In *IEEE Transactions on Software Engineering*, Vol. 22, pp. 751–761, 1996.

[6] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE'08)*, pp. 308–318, 2008.

[7] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *Proceedings of the 24th International Conference on Software Maintenance (ICSM'08)*, pp. 337–345, 2008.

[8] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pp. 121–130, 2009.

[9] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*, pp. 6–13, 2007.

[10] Christian Bird, David Pattison, Raissa D. Souza, Vladimir Filkov, and Premkumar Devanbu. Chapels in the bazaar? latent social structure in OSS. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE'10)*, pp. 24–35, 2008.

[11] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'10)*, pp. 301–310, 2010.

[12] Frederick P. Brooks. *The mythical man-month : essays on software engineering*. Addison Wesley, Boston, MA, 1975.

[13] Gerardo Canfora and Luigi Cerulo. Supporting change request assignment in open source development. In *Proceedings of the 21st Symposium on Applied Computing (SAC'06)*, pp. 1767–1772, 2006.

[14] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*, pp. 73–88, 2001.

[15] Davor Cubranic and Gail C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pp. 92–97, 2004.

[16] Marco D'Ambros and Michele Lanza. Software bugs and evolution: a visual approach to uncover their relationship. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pp. 229–238, 2006.

[17] Marco D'Ambros, Michele Lanza, and Martin Pinzger. "a bug's life" visualizing a bug database. In *Proceedings of the 4th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'07)*, pp. 113–120, 2007.

[18] Cleidson R. de Souza, Stephen Quirk, Erik Trainer, and David F. Redmiles. Supporting collaborative software development through the visualization of socio-technical dependencies. In *Proceedings of the International Conference on Supporting Group Work (GROUP'07)*, pp. 147–156, 2007.

[19] Bradley Efron. Estimating the error rate of a precision rule: improvements on cross-validation. In *Journal of the American Statical Association*, Vol. 78, pp. 316–331, 1983.

[20] Karl Fogel. *Producing open source software: how to run sucessful free software project*. O'Reilly Media, Sebastopol, CA, 2005.

[21] Shoji Fujita, Akinori Ihara, Masao Ohira, and Kenichi Matsumoto. An analysis of committers toward improving the patch review process in OSS devel-

opment. In *Supplementary Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, pp. 369–374, 2010.

[22] Daniel M. German, Abram Hindle, and Norman Jordan. Visualizing the evolution of software using softChange. In *Proceedings of the 16th Internation Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pp. 336–341, 2004.

[23] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering (RSSE'10)*, pp. 52–56, 2010.

[24] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. In *ACM Transaction on Information Systems*, Vol. 22, pp. 5–53, 2004.

[25] Israel Herraiz, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR'08)*, pp. 145–148, 2008.

[26] Rattikorn Hewett and Phongphun Kijsanayothin. On modeling software defect repair time. In *Empirical Software Engineering*, Vol. 14, pp. 165–186, 2009.

[27] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE'07)*, pp. 34–43, 2007.

[28] James Howison, Keisuke Inoue, and Kevin Crowston. Social dynamics of free and open source team communications. In *Proceedings of the 2nd International Conference on Open Source Systems (OSS'06)*, pp. 319–330, 2006.

[29] Akinori Ihara, Masao Ohira, and Kenichi Matsumoto. An analysis method for improving a bug modification process in open source software development. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol'09)*, pp. 135–144, 2009.

[30] Akinori Ihara, Masao Ohira, and Kenichi Matsumoto. Differences of time between modification and re-modification: an analysis of a bug tracking system. In *Proceedings of the 3rd International Workshop on Knowledge Collaboration in Software Development (KCSD'09)*, pp. 17–22, 2009.

[31] Chris Jensen and Walt Scacchi. Role migration and advancement processes in OSSD projects: a comparative case study. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 364–374, 2007.

[32] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pp. 111–120, 2009.

[33] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: lessons learned from Hackystat-UH. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'04)*, pp. 136–144, 2004.

[34] Sascha Just, Rahul Premraj, and Thomas Zimmermann. Towards the next generation of bug tracking systems. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'08)*, pp. 82–85, 2008.

[35] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. Systematic review: A systematic review of effect size in software engineering experiments. In *Information and Software Technology*, Vol. 49, pp. 1073–1086, 2007.

[36] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? In *IEEE Transaction on Software Engineering*, Vol. 34, pp. 181–196, 2008.

[37] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE'06)*, pp. 81–90, 2006.

[38] Michele Lanza and Martin Pinzger. "A bug's life": visualizing a bug database. In *Proceedings of the 4th International Workshop on Visualizing Software for Analysis and Understanding (VISSOFT'07)*, pp. 113–120, 2007.

[39] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, pp. 311–321, 2011.

[40] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE'10)*, pp. 93–102, 2010.

[41] Mark W. Lipsey and David B. Wilson. *Practical meta-analysis*. Sage Publications, Thousand Oaks, CA, 2001.

[42] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. In *Automated Software Engineering*, Vol. 17, pp. 375–407, 2010.

[43] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. In *ACM Transactions on Software Engineering and Methodology*, Vol. 11, pp. 309–346, 2002.

[44] Ingunn Myrtveit, Erik Stensrud, and Ulf Olsson. Analyzing data sets with missing data: an empirical evaluation of imputation methods and likelihood-based methods. In *IEEE Transactions on Software Engineering*, Vol. 27, pp. 999–1013, 2001.

[45] Mehrdad Nurolahzade, Seyed M. Nasehi, Shahedul H. Khandkar, and Shreya Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol'09)*, pp. 9–18, 2009.

[46] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, pp. 505–514, 2010.

[47] Eric S. Raymond. *The cathedral and the bazzar: musings on linux and open source by an accidental revolutionary.* O'Reilly and Associates, Sebastopol, CA, 1999.

[48] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pp. 541–550, 2008.

[49] Jan Sandred. *Managing open source projects.* John Wiley & Sons, Hoboken, NJ, 2001.

[50] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pp. 23–33, 2009.

[51] Bianca Shibuya and Tetsuo Tamai. Understanding the process of participating in open source communities. In *Proceedings of the ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS'09)*, pp. 1–6, 2009.

[52] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Kenichi Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pp. 249 – 258, 2010.

[53] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, pp. 300–310.

[54] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*, pp. 1–5, 2005.

[55] Yasunari Takagi, Osamu Mizuno, and Tohru Kikuno. An empirical approach to characterizing risky software projects based on logistic regression analysis. In *Empirical Software Engineering*, Vol. 10, pp. 495–515, 2005.

[56] Hee Beng Kuan Tan, Yuan Zhao, and Hongyu Zhang. Conceptual data model-based software size estimation for information systems. In *ACM Transaction on Software Engineering Methodology*, Vol. 19, pp. 1–37, 2009.

[57] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pp. 461–470, 2008.

[58] Yi Wang, Defeng Guo, and Huihui Shi. Measuring the evolution of open source software systems with their communities. In *SIGSOFT Software Engineering Notes*, Vol. 32, pp. 1–10, 2007.

[59] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*, pp. 1–8, 2007.

[60] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the 5th International Working Conference on Mining Software Repositories (MSR'08)*, pp. 67–76, 2008.

[61] Jingwei Wu, Richard C. Holt, and Ahmed E. Hassan. Exploring software evolution using spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pp. 80–89, 2004.

[62] 小山貴和子. OSS 開発における時差の分析 : OSS 開発者の情報交換への影響. 奈良先端科学技術大学院大学 修士論文, NAIST-IS-MT0851047, 2010.

[63] 伊原彰紀. 障害管理システム利用時の修正遅延要因の分析. 奈良先端科学技術大学院大学, 修士論文, NAIST-IS-MT0751012, 2009.

[64] IPA（独立行政法人情報処理推進機構）. 第 3 回オープンソースソフトウエア活用ビジネス実態調査（2009 年度調査）. 2009.

[65] 柗本真佑, 亀井靖高, 大平雅雄, 松本健一. OSS コミュニティにおけるオープンコラボレーションの理解. 情報社会学会誌, Vol. 3, No. 2, pp. 29–42, 2009.

[66] 日本工業標準調査会審議. ソフトウェア技術 -ソフトウェアライフサイクルプロセス- 保守 JIS X 0161：2008 (ISO/IEC 14764：2006). 日本規格協会, 2008.

[67] 飯尾淳, 清水浩之, 谷田部智之, 比屋根一雄. 東アジア IT 市場のオープンソースソフトウェアによる活性化. 三菱総合研究所所報, No. 46, pp. 52–65, 2006.