Universidade Federal da Bahia
Universidade Salvador
Universidade Estadual de Feira de Santana

TESE DE DOUTORADO

Inappropriate Software Changes: Rejection and Rework

Rodrigo Rocha Gomes e Souza

Programa Multiinstitucional de
Pós-Graduação em Ciência da Computação – PMCC

Salvador
2015

PMCC-DSc-0020

RODRIGO ROCHA GOMES E SOUZA

INAPPROPRIATE SOFTWARE CHANGES: REJECTION AND REWORK

Esta Tese de Doutorado foi apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientadora: Christina von Flach Garcia Chavez
Co-orientador: Roberto Almeida Bittencourt

Salvador
2015

Souza, Rodrigo

Inappropriate Software Changes: Rejection and Rework / Rodrigo Rocha Gomes e Souza. – 2015.

111p.: il.

Inclui apêndices.
Orientadora: Profª. Drª. Christina von Flach Garcia Chavez.
Co-orientador: Prof. Dr. Roberto Almeida Bittencourt.
Tese (doutorado) – Universidade Federal da Bahia, Instituto de Matemática, Universidade Salvador, Universidade Estadual de Feira de Santana, 2014.

1. Engenharia de Software. 2. Evolução de Software. 3. Qualidade de Software. 4. Engenharia de Releases.
I. Chavez, Christina. II. Bittencourt, Roberto A. III. Universidade Federal da Bahia, Instituto de Matemática. IV. Universidade Salvador. V. Universidade Estadual de Feira de Santana. VI. Título.

CDD – 005.1
CDU – 004.41

"INAPPROPRIATE SOFTWARE CHANGES: REJECTION AND REWORK"

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multi-institucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS.

Salvador, 17 de julho de 2015

Profª. Drª Christina von Flach Garcia Chavez
Universidade Federal da Bahia

Prof. Dr. Cláudio Nogueira Sant'Anna
Universidade Federal da Bahia

Prof. Dr.Roberto Almeida Bittencourt
Universidade Estadual de Feira de Santana

Prof.Dr. Dalton Dario Serey Guerrero
Universidade Federal de Campina Grande

Prof. Dr. Marco Aurélio Gerosa
Universidade de São Paulo

ABSTRACT

Background: Writing source code changes to fix bugs or implement new features is an important software development task, as it contributes to evolve a software system. Not all changes are accepted in the first attempt, though. Inappropriate changes can be rejected because of problems found during code review, automated testing, or manual testing, possibly resulting in rework.

Our objective is to better understand the statistical association between different types of rejection—negative code reviews, supplementary commits, reverts, and issue reopening—to characterize their impacts within a project, and to understand how they are affected by certain process changes. To this end, this thesis presents an analysis of three large open source projects developed by the Mozilla Foundation, which underwent significant changes in their process, such as the adoption of rapid releases.

Methods: To pursue our objective, we analyzed issues and source code commits from over four years of the projects' history. We computed metrics on the occurrence of multiple types of change rejection and measured the time it takes both to submit a change and to reject inappropriate changes. Furthermore, we validated our findings by discussing them with Mozilla developers.

Results: We found that techniques used in previous studies to detect inappropriate changes are imprecise; because of that, we proposed an alternative technique. We determined that inappropriate changes are a relevant, daily problem, that affects about 18% of all issues in a project. We also discovered that, under rapid releases, although the proportion of reverted commits at Mozilla increased, the reverts were performed earlier in the process.

Keywords: software engineering; software evolution; mining software repositories; software quality; release engineering.

RESUMO

Introdução: A escrita de mudanças no código-fonte para corrigir defeitos ou implementar novas funcionalidades é uma tarefa importante no desenvolvimento de software, uma vez que contribui para evoluir um sistema de software. Nem todas as mudanças, no entanto, são aceitas na primeira tentativa. Mudanças inadequadas podem ser rejeitadas por causa de problemas encontrados durante a revisão de código, durante o teste automatizado, ou durante o teste manual, possivelmente resultando em retrabalho.

Nosso objetivo é entender melhor a associação estatística entre diferentes tipos de rejeição — revisões de código negativas, *commits* suplementares, reversão de *commits* e reabertura de tíquetes —, caracterizar seus impactos em um projeto e entender como elas são afetadas por certas mudanças de processo. Para este fim, esta tese apresenta uma análise de três grandes projetos de software livre desenvolvidos pela Mozilla Foundation, os quais sofreram mudanças significativas no seu processo, como a adoção de lançamentos frequentes.

Métodos: Para perseguir nosso objetivo, nos baseamos em tíquetes e *commits* de um período de mais de quatro anos do histórico dos projetos. Computamos métricas sobre a ocorrência de diversos tipos de rejeição de mudanças e medimos o tempo que leva tanto para submeter uma mudança quanto para rejeitar mudanças inapropriadas. Além disso, validamos nossos resultados com desenvolvedores da Mozilla.

Resultados: Descobrimos que técnicas usadas em estudos anteriores para detectar mudanças inadequadas são imprecisas; por isso, propusemos uma técnica alternativa. Determinamos que mudanças inadequadas são um problema relevante e diário, que afeta cerca de 18% de todos os tíquetes em um projeto. Também descobrimos que, quando a Mozilla adotou lançamentos frequentes, embora a proporção de *commits* revertidos tenha aumentado, as reversões foram realizadas mais cedo no processo.

Palavras-chave: engenharia de software; evolução de software; mineração de repositórios de software; qualidade de software; engenharia de releases.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Various software systems that are used daily need to be constantly changed to remain useful (LEHMAN et al., 1997). Changes are intended to fix problems, add features, or improve some other aspect of the software, such as performance or ease-of-use.

Every so often, however, some changes are considered inappropriate. For instance, a change may not fix entirely a bug it was intended to resolve, or it may cause new bugs, or it may even harm the system's maintainability. In any case, inappropriate changes cannot be completely avoided.

In order to detect inappropriate changes, many projects resort to quality control activities, such as code reviews, automated testing, and manual testing. An inappropriate change can be rejected at any of these steps. Change rejection triggers rework, since it requires developers to create a new, improved change, which will undergo further testing.

According to Boehm and Basili (2001), software projects spend 40–50% of their effort on rework. This estimate takes into account not only rework triggered by inappropriate changes, but also rework triggered by misunderstood requirements as well. Nonetheless, it shows that preventing rework can significantly improve productivity.

From the perspective of a researcher or a project manager, measuring rework within a project can provide insight on the quality of its software development process. A high rework rate is an indicator of a low-quality process, that could ultimately cause schedule overruns.

Researchers have relied on data from projects' issue tracking systems and source code repositories in order to detect change rejection and measure the resulting rework (SHIHAB et al., 2010; JONGYINDEE et al., 2011; PARK et al., 2012). They used this information to try to predict which changes would be rejected and measure the time spent on rework.

A recent study (AN; KHOMH; ADAMS, 2014) has found that current approaches to detect change rejection are imprecise, i.e., they often detect that a change was rejected when it was not. For instance, Park et al. (2012) assume that, when a developer submits another change to resolve an issue that was already resolved by a change, it means that

he is rejecting the first change. However, Le An et al. (2014) found that this is often not the case; it is often the result of the developer splitting his change in multiple smaller chunks. The imprecision in detecting change rejection makes it difficult to confidently estimate the amount of rework in a project.

1.1 OBJECTIVES

The main objective of this thesis is to better understand change rejection and the resulting rework in software projects. To this end we propose an improved technique to detect change rejection, based on the identification of reverted commits, and then use it to measure inappropriate changes and rework in projects maintained by the Mozilla Foundation. We compare metrics before and after Mozilla's adoption of rapid releases and integration repositories, two software development practices that altered Mozilla's process, in order to assess the impacts of those process changes. These practices are explained in Chapter 2.

We chose to study Mozilla's projects for three reasons. First, they have mature, documented software development processes, which makes them easier to understand. Second, they make software process data available, such as issue reports and source code, which is required for our research approach. And third, the projects have large user bases, and are therefore relevant. In particular, Mozilla's web browser, Firefox, has an estimated user base of half a billion users around the world.

1.2 RESEARCH GOALS

The research goals pursued by this thesis are detailed below:

Research Goal 1: Propose and compare techniques to detect change rejection. There is evidence that current techniques do not precisely detect change rejection (AN; KHOMH; ADAMS, 2014). We propose a technique based on reverts (see Chapter 2) and compare it to previously proposed techniques.

Research Goal 2: Quantify rework triggered by inappropriate changes. We aim to measure rework in terms of its occurrence and in terms of the time they account for in a project. Those measurements, together with developer feedback, give an idea of how harmful inappropriate changes are.

Research Goal 3: Empirically validate hypotheses about rework. Since we are already measuring time intervals, those measurements can also be used to evaluate some hypotheses. For instance, is it true that rework performed later in the process takes more effort?

Research Goal 4: Assess the impacts of process changes on rejections. In particular we study the impact of adopting rapid releases and integration repositories, process changes introduced at Mozilla, as explained in the next chapter.

1.3 DATA AND METHODS

In order to pursue the previously described goals, we rely on data from three Mozilla projects: Core, Firefox, and Thunderbird. Those projects were chosen for having the

highest number of issue reports since 2009, which is a proxy for development activity.

Specifically, we look at issue reports and commit logs, which contain descriptions of changes to the products' source code. Together, those two data sources provide information about a change's lifecycle, i.e., the stages through which a change goes through, from its creation to its integration into a released product, passing through code review, testing, and, possibly, its rejection and subsequent improvement.

By detecting important events, we can estimate, for each issue (i.e., for each work item), metrics such as the time needed to submit a source code change to resolve the issue and whether the change was eventually rejected, triggering rework. In case the change was rejected, we also determine which activity triggered rejection (i.e., code review, automated testing, or manual testing), and the time needed to submit an improved change.

These and other metrics provide the basis for investigating questions related to research goals 1, 2, 3, and 4. To interpret metrics, we resort to statistical tests and interviews with Mozilla engineers.

1.4 MAIN RESULTS

These are the main results of this thesis:

- we found that previous techniques to detect change rejection from historical data are imprecise, and proposed a new technique together with evidence that suggests that it is more precise;
- we found that inappropriate changes are frequent and contribute to longer issue lifetimes;
- we found that, under Mozilla's process, inappropriate changes are unlikely to reach end users;
- we found that, after Mozilla's process changes, there was a shift towards earlier rejection.

1.5 OUTLINE

The remaining of this thesis is organized as follows. In Chapter 2, we present software development concepts related to this thesis. In Chapter 3, we present related work on mining software repositories, change rejection, and other topics. In Chapter 4, we detail our research goals by presenting specific research questions, as well as presenting the data and methods used to answer them. In Chapter 5, we present the quantitative results of our study. In Chapter 6, we discuss the possible interpretations and implications of the results. Finally, in Chapter 7, we summarize our contributions and outline directions for future work.

The thesis also contains two appendices. Appendix A contains emails exchanged with Mozilla engineers about preliminary results of our research. Appendix B contains our previous publications directly related to this thesis.

Chapter 2

BACKGROUND

In this chapter, we explain software development tools and practices that are relevant to understand the development process of many software projects, including Mozilla's. In particular, we introduce issue tracking systems and version control systems, and explain concepts about continuous integration and release models. After that, we portray specifics of the process followed by projects developed by the Mozilla Foundation and describe the changes in this process that were introduced in 2011. Finally, we define some concepts that are used throughout this thesis, and show how they can be observed in Mozilla's process.

2.1 ISSUE TRACKING SYSTEMS

In software engineering, an issue is a "unit of work to accomplish an improvement in a system" (ANH et al., 2011a)[1]. Issues are most often bugs or features, but they can also be performance improvements, system restructuring, documentation, changes in the infrastructure, and generally any work unit that needs to be tracked.

Issue tracking systems are tools that allow users, developers, and other stakeholders to report and keep track of issues within a project. Each issue report, also known as ticket, contains fields with information about the issue, such as title, description, priority, the name of the person who reported it, a progress status, and so on (the exact field set varies from one issue tracking system to another). Issue reports also contain a history of all comments written by stakeholders about the issue and the changes in the issue status and other fields.

[1]This definition of "issue" is more common in the software development context, and is the one used throughout this thesis. In the context of IT service management, it has a different meaning, being synonymous with incident, i.e., an interruption or degradation of a service, which is different from the meaning intended in this thesis.

2.1.1 Terminology

Because issue tracking systems are generic enough to help keep track of any work, there is often intersection with bug tracking systems and project management systems. In practice, many tools implement aspects of issue, bug, and project management, so it is hard to classify them in one single category.

For instance, Bugzilla, originally a bug tracking system, now describes itself as "server software designed to help you manage software development." Nonetheless, in Bugzilla, a unit of work is still called a bug report, which can be confusing (ANTONIOL et al., 2008). A previous study on five open source projects showed that about 33% of all bug reports do not describe bugs; they refer, instead, to feature requests, requests for performance improvements, and other maintenance tasks (HERZIG; JUST; ZELLER, 2013).

For example, Mozilla Firefox's bug report #864250 is a request by a QA engineer for the creation of a test case; bug report #1096140 is for cleaning up unused source code; bug report #1081322 is for adding a field to a form. Although Bugzilla allows classifying a bug report as a request for enhancement, none of these three bugs was classified as such. In fact, in the data set we analyzed, less than 3% of all bug reports were classified as enhancements, which shows that the Firefox team does not systematically differentiate between corrective maintenance and other types of tasks in their bug reports.

Although in this thesis we analyze bug reports from Bugzilla, we chose to call them "issue reports," a neutral term, to avoid confusion about their contents.

2.1.2 Issue Reports

In Bugzilla, people can report issues and then update issue reports with information regarding the issue and the process of resolving the issue, either by uploading files (such as screenshots, trace logs, and patches) or by commenting and updating issue report fields. Each issue report has a status field, which can take the values `UNCONFIRMED`, `NEW`, `ASSIGNED`, `RESOLVED`, `VERIFIED`, and `REOPENED`.

Figure 2.1 shows typical issue status and their transitions. An issue starts with status `UNCONFIRMED`, if reported by a regular user, or `NEW`, if reported by a trusted user. After that, the issue may be `ASSIGNED` to a developer, and then `RESOLVED`. After successful manual testing, the status is changed to `VERIFIED`. Any issue marked `RESOLVED` or `VERIFIED` is considered closed and may be `REOPENED` if the initial solution was deemed inappropriate, so it can be `RESOLVED` correctly. It should be noted that projects outside Mozilla that also use Bugzilla may give different interpretations to each status (SOUZA; CHAVEZ, 2012).

For `RESOLVED` issues, a resolution must be chosen among `FIXED`, `INVALID`, `WONTFIX`, `DUPLICATE`, `WORKFORME`, or `INCOMPLETE`. Although the actual interpretation of each resolution is project-dependent, Bugzilla's documentation (BUGZILLA, 2015) proposes the following interpretations, which are followed by Mozilla's projects:

- `FIXED`: a solution for the issue was committed to a source code repository and tested;

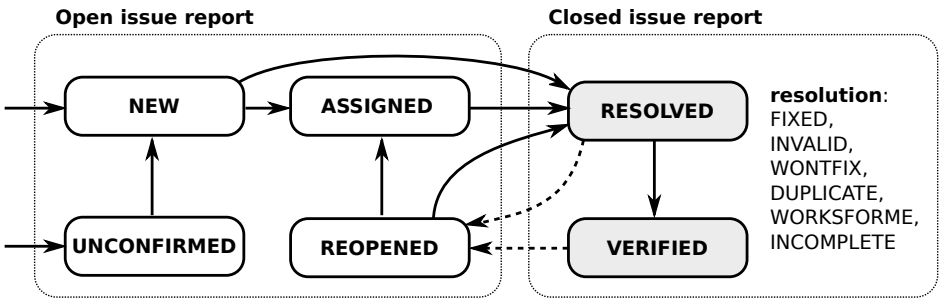

Figure 2.1: Bugzilla's status and transitions.

- `INVALID`: the report intends to describe a bug, but the team considered it not to be a bug;
- `WONTFIX`: the issue is a bug which developers have no intention of resolving;
- `DUPLICATE`: the report is a duplicate of another report;
- `WORKSFORME`: the issue is a bug that developers could not reproduce;
- `INCOMPLETE`: the report provides insufficient information about a problem.

2.1.3 Summary

In the context of software development, issues represent units of work, such as fixing bugs and implementing new features. Issue tracking systems allow people in a project to discuss issues and track updates to their status. Some systems refer to issues as bugs.

2.2 VERSION CONTROL SYSTEMS

Version control systems "record changes to a file or set of files over time" (CHACON, 2009). They are often used in software development to allow multiple developers to evolve a code base.

Version control systems have evolved from local to centralized, and then to distributed. A local version control system, such as RCS, manages multiples versions of files in a local file system. Centralized version control systems, such as CVS and Subversion, keep files in a central versioned repository accessed by developers, who should check out a copy of a specific version of the files to their computers before modifying them. In a distributed version control system, such as Git and Mercurial, each developer has a full copy of the repository and can share changes from one repository to the other; although not strictly needed, there is usually a central repository off which developers base their work.

A complete explanation of concepts and mechanisms of version control systems is beyond the scope of this thesis. For this reason, we focus on key concepts from distributed version control systems.

2.2.1 Revisions

A repository is a list of revisions of a set of files. Each revision (also called change set or version) records the contents of the files in a specific instant in time. Revisions are uniquely identified by a hash number and are explicitly created when a user performs a *commit* operation after changing the repository's files (for this reason, people also refer to revisions as commits). When performing a commit, the user must write a message that describes the changes, and this message becomes associated with the revision.

2.2.2 Remote Repositories

When collaborating on a project, a user should first clone its remote repository, an operation that creates a copy of the repository in their local file system. The commit operation creates revisions in the local repository; to send them to the remote repository, the user should perform a *push*. Because other users may also have pushed changes to the remote repository, the user should *pull* those changes to their local repository before performing a push.

2.2.3 Branches

Often a project needs to evolve independently two or more copies of its files. For instance, after a team releases a version of its software, it can start to work on the next version, but may need to keep fixing bugs in the old version. In this case, the repository is split into two *branches*, or *trees*, so that changes in one branch are isolated from changes in the other one. Eventually the changes made to one branch may need to be applied to the other one; in this case, we say that one branch is *merged* into the other one.

In the context of distributed version control systems, creating a branch is conceptually similar to cloning a repository, as in both cases the code base can be evolved in parallel, and in both cases the changes can be merged. The decision to create a branch or a clone is driven by technical details and user preference.

2.2.4 Summary

Version control systems record changes to a set of files in a project that are explicitly committed to a repository. Commits are associated with messages that explain the changes. A project's files can be in multiple repositories, and commits can be merged from one to another.

2.3 CONTINUOUS INTEGRATION

In a large software project, developers coordinate their work using issue tracking systems and evolve together a code base in a central repository (also known as *mainline*, *head*, or *master*) with the help of a version control system (FOWLER; FOEMMEL, 2005). Because all developers have a local copy of the repository, they can work on an issue in isolation, by making changes to their local repository before pushing them to the central repository. This isolation is useful to avoid sharing incomplete changes among developers.

If developers take a long time to integrate their changes, though, the individual repositories may have significantly diverged, to the point that there are conflicting changes. Developers must resolve conflicts manually before integrating the changes, a time-consuming and error-prone task. Longer periods between integrations also mean that, when something goes wrong, developers have to inspect many changes to try to find the cause.

Continuous integration (CI) is the practice of integrating developers' changes to the central repository multiple times a day. The objective is to reduce conflicts and get feedback on integration problems earlier, when the cost to solve them is lower (FOWLER; FOEMMEL, 2005).

Integrating daily, however, also means that there is no time to comprehensively test a change before integrating it into the central repository. As a result, the repository may occasionally receive inappropriate changes that can slow down other developers.

2.3.1 Automated Builds

In order to detect inappropriate changes early in the process, continuous integration advocates automated building of the code in the central repository, which consists of compiling the code (possibly for multiple platforms) and running automated tests. The automated build is performed by a continuous integration server (CI server) upon every change in the central repository, if possible, or as frequently as feasible given the available computing resources.

The CI server reports each build as successful or failing, depending on the outcome of the compilation and of the automated tests. We say that the central repository is *broken* when the build fails, or *stable* when the build succeeds. The central repository should accommodate frequent changes, but should also be stable enough to allow developers to base their work off it.

2.3.2 Broken Builds

To prevent breaking the repository, developers can run *private builds*, i.e., automated builds of their working copy, before integrating their changes. Because a complete build may take a long time to complete, developers can choose to build only on a subset of the platforms, or to run only a subset of the automated tests, although this practice increases the risk of problems not being detected until a full build is performed.

Even if developers always run private builds, their changes may break the central repository, be it because of differences in the developers' machines and the CI server, or because of interactions between changes. The highest priority, in this case, is to make the repository stable again as soon as possible.

If the cause of the breakage is simple, one can quickly write a fix, push it to the repository, and wait for the results of the new build, which should now be successful. This is called a *roll forward* or a *follow-up* change. However, often the quickest way to fix a broken repository is to simply *revert* (also called *back out* or *roll back*) the changes introduced since the last stable build. The problem can then be investigated in a local copy of the repository, while the central repository is stable and prepared to receive changes from other developers.

2.3.3 Integration Repository

If the central repository of a project breaks often, causing trouble to developers, the project can adopt an *integration repository* (also called *pending head* or *staging repository*). The integration repository is either a clone or a branch of the central repository (the distinction between a clone and a branch is not relevant in this case).

Within this model, developers still pull changes from the central repository, but they now push their changes to the integration repository. Periodically, the CI server builds off the integration repository and merges it to the central repository only if the build is successful. Breaking the integration repository is less of a problem, since developers still base their changes off the central repository, which is kept more stable.

2.3.4 Summary

In continuous integration, developers pull changes from and push changes to a central repository daily. A continuous integration server automatically builds every revision of the central repository, which includes compiling the code and running automated tests. A central repository that is often broken (i.e., that results in failed builds) harms developers' productivity. To avoid frequent breakage, developers can push changes to a separate integration repository, and have these changes merged into the central repository only after a successful build.

2.4 RELEASE CYCLES

The release cycle includes all stages from the initial development of a new version of a product until its release to users, including the release of minor versions containing bug fixes. The release cycle comprises multiple continuous integration cycles (change, integrate, build) and involves the coordination of multiple teams within a project.

A release model is a simplified description of a release process, i.e., of the activities performed in a release cycle. In this section, we compare two release models, traditional releases and rapid releases, by explaining in which aspects they differ. The main differences are highlighted in Table 2.1.

First, the two release models differ in the instant of release, which can be *feature-based* or *time-based*. Traditional releases are feature-based: there is a set of planned features that must be implemented for the software to be released, and the time it takes for that is variable. Under rapid releases, which are time-based, new versions of the software are released in fixed time intervals, with whatever features are ready when the release date comes.

Under traditional releases, a version with new features is released about one year or more after the previous version. Under rapid releases, this time is cut to a few weeks or months, usually with the help of continuous integration, since it makes code integration easier.

Another aspect to consider in a release model is the maintenance of old releases. Under traditional releases, new features are implemented for the next release only, but bug fixes are back ported to older releases. This approach guarantees that users of old version will

Table 2.1: Comparison between traditional and rapid releases.

	Traditional Releases	**Rapid Releases**
Release criteria	feature-based	time-based
Time between releases	a year or more	weeks or few months
New features	next major release	next rapid release
Supported versions	many	one or two
Bug fixes	bug fix release	next rapid release
Feature toggles	no	yes

get bug fixes in a timely manner, but requires the organization to support two or more versions at the same time. Under rapid releases, since releases are more frequent and the update process is smoother, it is usual to support only the latest version. Sometimes, there is also an extend support release, which is a version that receives only bug fixes and security updates for an extended period of time, usually intended for corporate environments which require stable software.

In both release models there is a *stabilization* period, which occurs just before the release. Usually during this period there is a *feature freeze*, during which no new features are admitted in the branch or repository that will be used for the release.

In a time-based release process, such as rapid releases, incomplete features need to be disabled during stabilization. To make that process easier, developers can implement a conditional rendering logic on user interface elements that allow users to access a feature (FOWLER, 2010). Those elements would be rendered or not depending on the value of a configuration variable. This technique is called *feature toggle*, *feature flag*, or *feature switch*. During the stabilization period, should the feature be disabled, it is just a matter of changing a configuration variable.

The rapid release model allows the organization to deliver new features to users earlier. Analogously to continuous integration, that reduces the risk of code integration by integrating often, rapid releases reduce the risk of release by releasing often. Traditional releases, on the other hand, may be appropriate when stability is more important than new features.

2.4.1 Summary

The traditional release model consists of feature-based releases separated by several months (usually 12 months or more), with bug-fix releases in-between. Older versions are updated with bug fixes. The rapid release model, on the other hand, consists of time-based releases separated by weeks (typically 6 weeks to 3 months). Usually only the latest version is supported, or sometimes the latest and an extended support release.

2.5 MOZILLA'S CODE INTEGRATION AND RELEASE PROCESS

Netscape was a company founded in 1994 that created a suite of applications for the Internet, such as the Netscape Navigator web browser. In 1998, it released an open source version of its applications and created a project, Mozilla, to coordinate their development (BAKER, 2008). Netscape was eventually disbanded, while Mozilla continued existing as a foundation, with many Netscape's ex-employees.

Back in 1997, Netscape already had a continuous integration server, running the software Tinderbox, developed in-house (O'DUINN, 2014). The Mozilla Foundation inherited the software and the process, and therefore has been using continuous integration from the start.

Mozilla also hosts the so-called Try server, a CI server that developers can use to run private builds of their local changes[2]. This way, developers can "try" their changes on a subset of platforms and test suites before pushing the changes to the central repository.

As of 2009, a build of Mozilla-Central, the central repository for Firefox and core libraries, took about 4 hours to complete: 1.5 hours to compile the code for all platforms, plus 2.5 hours to run all unit tests. Developers were expected to be available for the next 4 hours after pushing changes, in order to watch the results of the build and to take the necessary actions if the build failed, for instance, reverting their own commit. Build times and developer responsibilities are available at Mozilla's wiki (MOZILLA, 2015).

Before pushing a change to the central repository, a developer must submit the change to be reviewed by the module owner, which is the person responsible for the piece of code that was changed[3]. The module owner, in the role of reviewer, analyzes the code's design and coding style, checks if it resolves the problem, and comments on overall improvements that can be made[4].

The review can be positive or negative. A negative review should explain how the patch could be improved to get a positive review. Only after a patch gets a positive review can it be pushed to the code repository.

In 2011, Mozilla projects underwent significant changes in both its continuous integration and its release process. Those changes are described in the next sections.

2.5.1 Rapid Releases

Before March 2011, Firefox had been developed according to a traditional release schedule: features for the upcoming version were developed along with bug fixes and minor updates for the current stable release. The release was feature-based: major features would only be delivered to users with the release of a major version, which occurred when the planned features were implemented and tested. In practice, a new major version used to take 12 to 18 months to be released[5].

In 2008, Google entered the web browser market, when it launched Google Chrome, Firefox's direct competitor. In July, 2010, Google announced that it would release a

[2]See ⟨https://wiki.mozilla.org/ReleaseEngineering/TryServer⟩.
[3]See ⟨https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/How_to_Submit_a_Patch⟩.
[4]See ⟨https://developer.mozilla.org/en/docs/Code_Review_FAQ⟩.
[5]The list of release dates can be found at ⟨https://wiki.mozilla.org/Releases⟩.

major version every six weeks, effectively getting features to its users earlier and making the release schedule more predictable (LAFORGE, 2010).

Partly because of competition with Google Chrome (JONO, 2012), Mozilla decided to move to a rapid release model in which new versions would be released every six weeks. The first rapid release cycle, Firefox 5, started in March 22, 2011.

2.5.2 Release Channels

Together with the move to rapid releases, Mozilla started using four channels to distribute versions of Firefox and related products: central (or nightly), aurora, beta, and release. Each channel contains a version of Firefox that is more stable than the version in the former channel, and thus reaches a wider user base. Central is aimed at Mozilla contributors; aurora and beta are aimed at web developers and early adopters; release contains a version that was tested in aurora and beta and is ready to be installed by all users.

Each release channel has a corresponding source code repository. The repositories are called Mozilla-Central (or m-c), Mozilla-Aurora (m-a), Mozilla-Beta (m-b), and Mozilla-Release (m-r). As a general rule, developers push their patches with new features to m-c. Every six weeks, the central repository is merged into the aurora repository. No new features are admitted in aurora; during the next six weeks, the strings are localized to multiple languages and the code is stabilized by means of bug fixes and feature toggling. Then, aurora is merged into beta, where it is stabilized for six more weeks, this time with a larger user base. After that period, the code is merged into the release repository and becomes available as a stable version. Hence, a new feature can be delivered to users in 12 to 18 weeks, as changes flow through nightly, aurora, and beta channels.

In order to release a new version every 6 weeks, while a version is been stabilized, the next one is already being developed. Every 6 weeks there is the merge day, when changes in central are merged into aurora, changes in aurora are merged into beta, and changes in beta become part of the next release.

Figure 2.2 shows a simplified schedule for the release of versions 8, 9, 10, and 11. It can be seen that, in the fourth 6-week cycle, Firefox 8 is released, Firefox 9 and 10 are being stabilized in beta and aurora, respectively, and Firefox 11 is being developed in central. Firefox 9 is released in the next release cycle, when Firefox 10 becomes beta and Firefox 11 starts being stabilized in aurora.

2.5.3 Sheriff-Managed Integration Repositories

In June 8, 2011, a little more than two months after the adoption of rapid releases, Mozilla introduced integration repositories in the process. With this transformation, developers stopped committing directly to central, and started to commit to integration repositories instead, such as Mozilla-Inbound. The code on integration repositories is built and run against automated tests before being merged into central.

The merging of Mozilla-Inbound to Mozilla-Central is not performed by the developers who commit the patches; instead, it is performed by designated developers called *build sheriffs*, or *tree sheriffs*. Sheriffs watch the build and are responsible for reverting commits

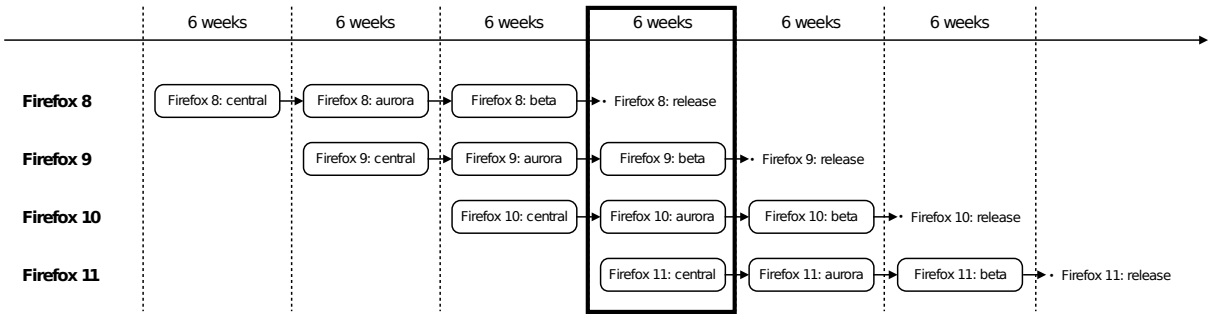

Figure 2.2: Release schedule for Firefox channels.

that break the build in order to stabilize Mozilla-Inbound before merging its commits into Mozilla-Central.

With integration repositories, only code that passes build and automated tests is merged into Mozilla-Central, keeping it more stable. Furthermore, because of sheriffs, developers do not need to watch builds after they push changes, and neither need to revert commits themselves.

2.5.4 Summary

Mozilla follows a continuous integration process with a code review step before automated builds. In 2011, it adopted rapid releases and integration repositories managed by sheriffs. In order to stabilize a version before releasing it officially, Mozilla publishes pre-release versions called aurora and beta intended for smaller audiences.

2.6 CHANGE LIFECYCLE AT MOZILLA

In this section, we explain the lifecycle of a change by showing how Mozilla's engineers interact with each other, with issue tracking systems, and with version control systems, and which data items are logged as a result. This explanation helps identify how to detect important events that can be used to compute metrics related to change rejection and rework.

The lifecycle is outlined in Figure 2.3. It shows that all changes that become part of a release undergo peer review, automated testing, and manual testing. In the following paragraphs, each step in the process is illustrated with an actual issue report from a Mozilla project. For privacy reasons, developers' names are replaced by their roles in the issue (e.g., Developer, Reviewer).

When a developer resolves an issue, he attaches the source code patch to the corresponding issue report in Bugzilla and asks a specific colleague to review it (setting the `review?` flag). Figure 2.4 shows an example from issue #787078, took from Firefox for Android. In this this issue, Developer attaches his patch to Bugzilla and asks his colleague Reviewer to review it. Figure 2.5 shows Reviewer's response, altering the `review?` flag to `review+`, signaling his acceptance of Developer's change. Had the change been rejected, Reviewer would alter the flag to `review-` and Developer would have to attach

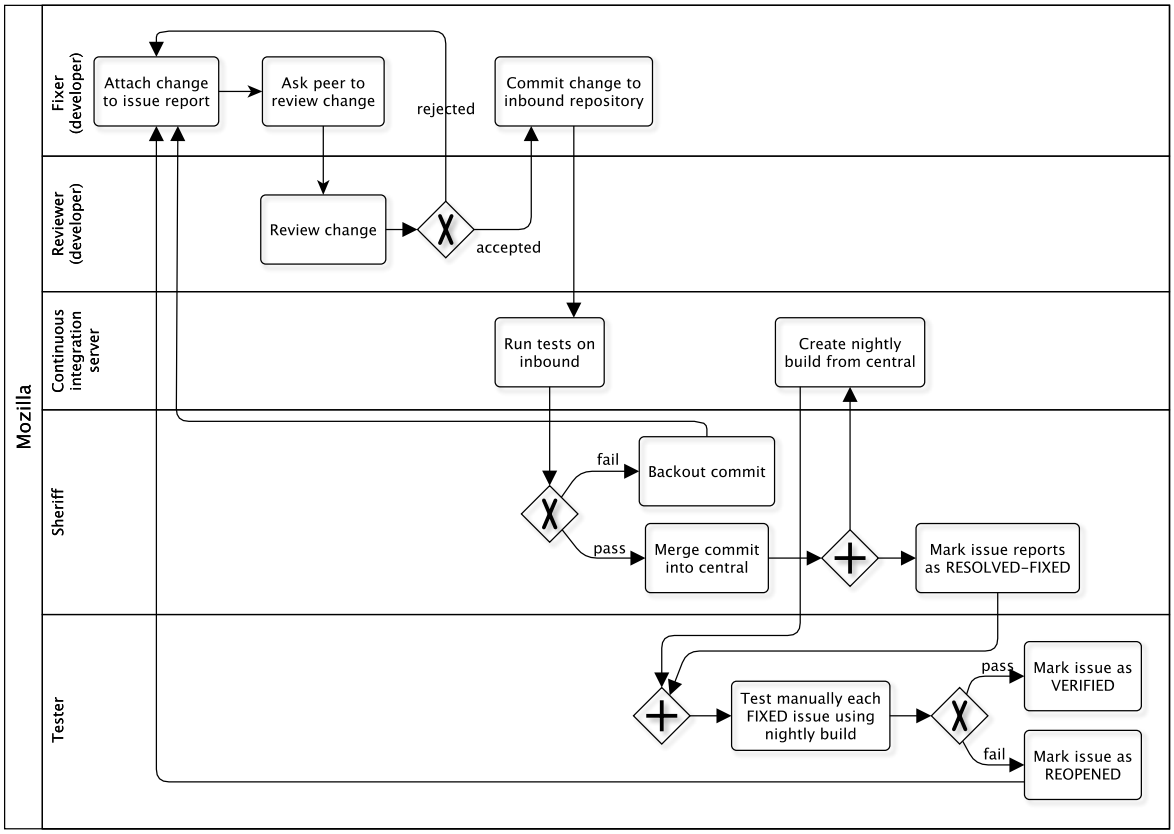

Figure 2.3: Change's lifecycle at Mozilla.

a new patch to be reviewed.

Because of the positive review, Developer pushes the change to the Mozilla-Inbound repository. The change can be viewed in Mozilla-Inbound's commit log, as shown in Figure 2.6. The commit message references the issue it resolves ("Bug 787078") and the user who reviewed the change ("r=reviewer").

Changes in Mozilla-Inbound undergo automated testing. If all tests pass, the build sheriff merges the patch into Mozilla-Central and changes the issue status to `RESOLVED` and the issue resolution to `FIXED`. This is in fact what happened with issue 787078, as shown in Figure 2.7. Had the change broken the build, the sheriff would have reverted the change and notified the developer.

In the case of issue 787078, the developer himself discovers a problem with his change. As a result, he reverts his commit, as shown in Figure 2.8, and restarts the process: he writes an improved change, Reviewer reviews it, he pushes the change to Mozilla-Inbound, where it passes the tests, and finally the sheriff merges the change into central.

Eventually, a tester takes a nightly build (built from Mozilla-Central) and verifies, by manually testing the build, whether the issue was indeed resolved. If that is the case, he changes the issue's status to `VERIFIED`, as shown in Figure 2.9. Otherwise, he changes the issue's status to `REOPENED` and the process starts from the beginning.

Figure 2.4: Developer asking for review of his change.

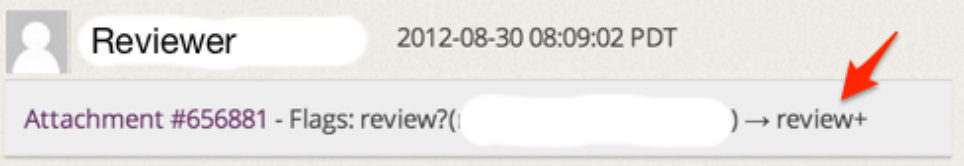

Figure 2.5: Reviewer accepting a change.

Commit hooks. Mozilla's developers follow conventions when writing commit messages. As illustrated in previous examples, commits that resolve an issue are associated with a message that starts with the word "bug" followed by the issue identifier. Besides commits that resolve an issue, there are also commits that revert inappropriate commits and commits that merge changes from one repository to the other.

Since November 2011, Mozilla enforces the commit message conventions using a Mercurial hook implemented in Mozilla-Central and in all repositories that merge into it. The hook prevents developers from pushing any commits that do not follow the conventions. Mozilla's wiki explains the rules for allowed messages (MOZILLA, 2015):

- Commit messages containing "bug" or "b=" followed by a bug number
- Commit messages containing "no bug" (please use this sparingly)
- Commit message indicating backout of a given 12+ digit changeset ID, starting with

```
changeset:  43dd8252f52d
user:       Developer
date:       Thu Aug 30 16:28:39 2012 +0100
summary:    Bug 787078 - Load Reader UI on pageshow instead of
            DOMContentLoaded (r=reviewer)
```

Figure 2.6: Developer committing a change.

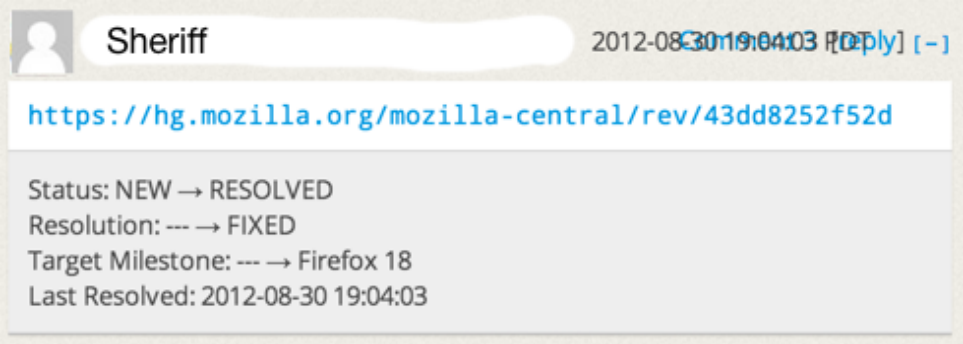

Figure 2.7: Sheriff merging a change that passed automated testing.

```
changeset:    0a93ae68184e
user:         Developer
date:         Fri Aug 31 15:18:19 2012 +0100
summary:      Bug 787078 - Backout 43dd8252f52d
```

Figure 2.8: Developer reverting a commit.

(back out—backing out—backed out—backout)(of)? (rev—changeset—cset)s? [0-9a-f]12

- Commit messages that start with "merge" or "merging" and are actually for a merge changeset.

As expected, the commits shown in Figure 2.6 and in Figure 2.8 conform to the rules.

Thunderbird. The process described before is valid for Core, Firefox, and other projects. The process for Thunderbird is basically the same, with a few differences. Although Thunderbird is released under the same schedule of other Mozilla products, it did not adopt sheriff-managed integration repositories; thus, commits are directly pushed to

Figure 2.9: Tester accepting a change.

the central repository and reverts are performed by all developers. Also, it is developed under a different repository, Comm-Central, which does not implement the same commit hooks used to enforce commit message conventions. Therefore, although Thunderbird developers tend to use the same conventions when writing commit messages, those conventions are not enforced.

2.7 CONCEPTS

After explaining basic concepts and tools, we can define concepts such as change, submission, rejection and rework, which are central to this thesis. We define *change* as any modification to the source code of a product that was submitted by its author as a solution to an issue.

Change submission and evaluation. We consider as *change submission* the submission of a change to be *evaluated*, i.e., to be code reviewed, to be automatically built (compiled and tested), or to be manually tested. At Mozilla, those submissions are performed by, respectively, requesting a code review on Bugzilla, pushing a commit to a remote repository, or closing the issue.

Change rejection. *Change rejection* is the manifestation by a person that a change is inappropriate. People indicate that a change is inappropriate by giving it a *negative review*, by *reverting* it from a code repository, or by *reopening* an issue report that was closed because of the change. Negative reviews, reverts, and reopenings are called *rejection types*. At Mozilla, the review process occurs before the change is committed; thus, negative reviews occur only before the corresponding commits. Reverts, on the other hand, can be performed only after a change is committed. We classify reverts as *early reverts*—when they occur before the corresponding issue is closed—or *late reverts*—when they occur after the issue is closed. Issue reopening, of course, can only be performed after an issue is closed.

Original and supplementary submissions. We define *original submission* as the first submission of a specific type for a specific issue. For instance, at Mozilla, every closed issue that required a source code change has an original review request, an original commit, and an original issue closing. Any subsequent submissions are called *supplementary submissions*. Supplementary submission may come after a rejection or not.

Rework. Change rejection triggers *rework*, which is the act of improving a change that was rejected or the act of evaluating the improved change after having evaluated the corresponding original change. In other words, rework means to perform additional work on an issue that was believed to be already resolved. For instance, when a change submitted to code review receives a negative review, there is rework for both the developer, who will work one more time on the change to resolve the issue, and for the reviewer, who will have to perform an additional code review.

Inappropriate change. It should be noted that, in this work, we do not attempt to theoretically define the concept of *inappropriate change*. Instead, we adopt an operational definition: a change is inappropriate if it was explicitly rejected (through negative reviews, reverts, or reopenings). The reason for a rejection can be diverse: maybe the change

introduced a bug, or failed to cover corner cases, or did not follow coding conventions. In some cases, it is hard to determine if a change was rejected: for instance, an issue may be reopened because of an inappropriate change or because of problems in the process (see Chapter 3).

Reverts as change rejection. Based on the observation of Mozilla's process data, we propose a new technique for detecting change rejection, based on reverted commits. As far as we know, this is the first study to use reverts to detect change rejection.

Commit reverts are easy to detect at Mozilla for two reasons. First, Mozilla adopts continuous integration, which encourages reverting commits that break the build instead of fixing the breakage by follow-up commits. This became even truer after build sheriffs started to watch the build, since they revert commits right away, without trying to fix the problem with follow-up commits.

Second, developers follow conventions when writing commit messages; in particular, they identify reverts by writing the word "backout" (or one of its variations, such as "back out" or "backing out"). Currently, a commit hook enforces those conventions in some projects, so we can expect a high conformance from developers.

Therefore, at Mozilla, reverts accurately represent the concept of change rejection for changes that were already committed. In other projects and organizations, reverting commits may not be a common practice.

Chapter 3

RELATED WORK

In this thesis, we analyze data from open source projects in order to measure rework and ultimately determine its relationship with certain software development practices adopted by the Mozilla Foundation. In this chapter, we review related work about mining software repositories, code reviews, change rejection, issue lifetime, and release engineering.

3.1 MINING SOFTWARE REPOSITORIES

Software repositories are collections of artifacts that are produced and archived during the evolution of a software system (KAGDI; COLLARD; MALETIC, 2007). We have already presented two types of software repositories, issue tracking systems and version control systems. Other software repositories include mailing lists, forums, and chat rooms.

The research field of mining software repositories leverages the historical record of software repositories in order to extract useful information for the future development of a specific software system (HASSAN, 2008) or even to validate hypotheses about software development in general.

Having access to software repositories is critical to research in this field, which traditionally relied on cooperation with large companies. Currently, with the popularization of open source systems, any researcher can have instant access to software repositories of large projects, developed by distributed teams, and with a large user base.

The availability of software repositories contributed to popularize mining software repositories as an approach for software engineering research. The Mining Software Repositories (MSR) conference is held annually since 2004.

The following are some research areas that rely on mining software repositories:

- **Defect prediction**. The objective of defect prediction is to predict the occurrence or the number of defects that will be found in each component of a system (D'AMBROS; LANZA; ROBBES, 2010). Techniques involve the analysis of the source code of a version of a software system (BASILI; BRIAND; MELO, 1996;

MARCUS; POSHYVANYK; FERENC, 2008) or the analysis of historical data from version control systems (MOSER; PEDRYCZ; SUCCI, 2008; NAGAPPAN et al., 2010) and issue tracking systems (KIM et al., 2007).

- **Mapping between defects and source code**. To validate defect prediction techniques it is necessary to have a reference system for which the number of defects in each component is known. To discover this information, it is necessary to map each defect reported on a bug tracking system to a source code component. The state-of-the-art of automatic mapping consists of finding commits that resolve a specific bug (usually, these commits refer to the bug report identifier in their messages). Then, one can conclude that the components modified by the commit contained the defect being resolved (FISCHER; PINZGER; GALL, 2003; ŚLIWERSKI; ZIMMERMANN; ZELLER, 2005; EADDY et al., 2008; D'AMBROS; LANZA; ROBBES, 2010).

- **Bias in mappings between defects and source code**. The effectiveness of automatic mappings depends on the programmers' conforming to the convention of referencing bug report identifiers in their commit messages. In practice, only a sample of all bug reports can be mapped (AYARI et al., 2007), and researchers have shown that this sample is biased (BIRD et al., 2009; NGUYEN; ADAMS; HASSAN, 2010). For instance, severe bugs are more prevalent in the sample than in the population of all bug reports. This bias diminishes the effectiveness of defect prediction algorithms and any other analysis that depends on the mapping. There are three suggestions to mitigate the problem (BIRD et al., 2009): seek more reliable data sets, map the defects manually, and train models in a sample of data whose distribution of features resembles closely that of the population.

- **Challenges in mining issue tracking systems**. Not all the coordination between developers related to resolving an issue is recorded precisely in the issue tracking system; the assignee of an issue report does not always participates in the resolution of defect (ARANDA; VENOLIA, 2009). For example, it may happen that an issue has been resolved before the creation of its report, or that the developer marks the issue as resolved long after its resolution. Extra care should also be taken with noise in the data and bulk edits (ARANDA; VENOLIA, 2009; SOUZA; CHAVEZ; BITTENCOURT, 2013a, 2013b). Another problem is that not all bug reports correspond to corrective maintenance; some correspond to perfective or preventive maintenance, discussions on architecture and other improvements (ANTONIOL et al., 2008; HERZIG; JUST; ZELLER, 2013).

3.2 CODE REVIEWS

Bird et al. (2007) developed an algorithm to detect patches in email messages and determine when such patches were applied to a code base. In their analysis of three open source projects—Apache, Python, and PostgreSQL—they found that between 25% and 49% of all submitted patches were applied without modifications to the code base. The

remaining patches were either rejected or applied with modifications (their algorithm could not tell the two cases apart).

Rigby et al. (2011, 2014) studied the peer review process of several open source projects. They identified two styles of review processes: review-then-commit (the most common, adopted by Mozilla), and commit-then-review. They characterized reviews according to their frequency, the participation of reviewers, among other dimensions.

Nurolahzade et al. (2009) inspected a random sample of 112 bug reports from Firefox. They identified recurrent patterns in the behavior of contributors and reviewers. For example, some contributors submit work-in-progress patches, and some reviewers avoid explicitly rejecting patches (what Jeong et al. calls "gentle rejection"). They also make a distinction between reviews conducted by module owners (i.e., developers responsible for specific modules in the software), who are concerned with long-term maintainability, and those conducted by other peers, who are usually more interested in functionality and usability.

3.3 ISSUE LIFETIME

A relevant information that can be extracted from issue tracking systems is each issue's lifetime, measured as the time between its creation and its resolution. A long lifetime is an indicator that the system is difficult to maintain (KIM; WHITEHEAD JR., 2006) and may even delay the release of new versions.

Several approaches have been employed to predict issue lifetime. One approach involves the use of data mining algorithms (PANJER, 2007) and regressions (ANBALAGAN; VOUK, 2009) applied to attributes from issue reports. Another approach is to use similarity metrics to predict the lifetime of an issue based on similar issues that were already resolved. The similarity can be measured using textual features in issues' titles and descriptions (WEISS et al., 2007) or grouping tickets with similar attributes (such as severity and component) through a neural network (ZENG; RINE, 2004).

Some conclusions found in studies about issue lifetime:

- more severe bugs are resolved faster (PANJER, 2007; HOOIMEIJER; WEIMER, 2007; BOUGIE et al., 2010; ZHANG et al., 2012);
- issue reports with easier to read descriptions are resolved faster (HOOIMEIJER; WEIMER, 2007);
- issue reports with many comments take more time to be resolved (PANJER, 2007; HOOIMEIJER; WEIMER, 2007; ANH et al., 2011b);
- the higher the number of participants in an issue report, the longer the time-to-resolve (ANBALAGAN; VOUK, 2009; ANH et al., 2011b);
- the past performance of a developer is a good predictor of the time he/she will take to resolve an issue (ANH et al., 2011b);
- long-lived bugs are characterized by specific source code constructs (CANFORA et al., 2011);

- issues that take more time to be assigned to a developer are resolved faster (after they are assigned) (OHIRA et al., 2012);

- when an issue is reported by a person, triaged by a different person, and resolved by a third person, the time-to-resolve is twice as long (OHIRA et al., 2012);

- other attributes that help predicting the time-to-resolve of an issue include its category (BOUGIE et al., 2010), the month when the issue was reported, the assigned developer (GIGER; PINZGER; GALL, 2010), and the reporter (GIGER; PINZGER; GALL, 2010) (although existing studies diverge on the relevance of the latter (BHATTACHARYA; NEAMTIU, 2011)).

3.4 CHANGE REJECTION

To the best of our knowledge, no other author has studied the problem of change rejection as broadly as defined in this thesis. However, researchers have studied sub-problems and related problems, described in the literature as rejection of patches during code review, supplementary changes, and issue reopening. These problems have been studied with many goals, such as eliciting the cause of bug reopening, predicting rejection, and assessing the costs associated with rejections.

3.4.1 Overview of Approaches to Change Rejection

Rejection of Patches During Code Review. Jeong et al. (2009) studied the acceptance of bug fixes submitted to peer code review in Mozilla Firefox and Mozilla Core. They found that only 50% of all review requests were explicitly accepted or rejected. The other half remained in an "open state", which may indicate a "gentle rejection", a lack of interest from reviewers, or that the initial bug fix was superseded by an improved version even before the reviewer had the opportunity to look at the first version.

Supplementary Changes. Supplementary changes are changes intended to resolve an issue that was already linked to a change (PARK et al., 2012). Their existence may indicate that the original change was inappropriate. We say that an issue has supplementary changes if it is associated with two or more commits.

While multiple commits may represent multiple attempts to resolve an issue, sometimes they are the result of a developer splitting a change into multiple small changes in order to facilitate peer review.

Issue Reopening. Issue reopening is the act of changing the status of an issue report from a closed state to an open state. In Bugzilla, this transition is represented by the `REOPENED` status.

The literature uses the term "bug reopening" to refer to this phenomenon. We use "issue reopening" instead because many projects analyzed in the literature do not make a distinction between bugs, feature requests, and other types of maintenance.

To better understand issue reopening, the reader should remember that an issue can be closed for many reasons. In the case of Bugzilla, the reason is documented in the resolution field of the issue report (see Figure 2.1). The resolutions can be separated in

two groups:

- `INVALID`, `WONTFIX`, `DUPLICATE`, `WORKSFORME`, `INCOMPLETE`: there is a problem with the issue report found during issue triaging;
- `FIXED`: there is a problem with the product, and it was solved by a source code change.

The first group is related to issue triaging; the second is related to changing the source code. Reopening an issue report that was closed during triaging means that the triaging was ineffective or that new information made the team reconsider their first evaluation. Reopening a `FIXED` issue report, on the other hand, usually means that the source code change was considered inappropriate, requiring the developer to write an improved change. Only the second group fits our definition of change rejection.

Some authors investigate issue reopening in a broad sense, mixing together triage-related and change-related reopening. Other authors investigate only fix-related reopening by studying a subset of issues with resolution `FIXED`.

Comparison Between Supplementary Changes and Issue Reopening. Le An et al. (2014) analyzed the relationship between supplementary changes and issue reopening on WebKit and projects from Eclipse Foundation and Mozilla Foundation, partially replicating the studies by Park et al. (2012) and by Shihab et al. (2010). They found that between 21% and 34% of all issues with supplementary changes were eventually reopened, and that only a little more than 50% of all reopened issues were associated with supplementary changes. They conclude that issue reopening is not always related to inappropriate changes.

In the next subsections, we describe in more detail the results found in the literature about bug reopening, supplementary bug fixes, and rejection during code review.

3.4.2 Why are Changes Rejected?

Zimmermann et al. (2012) conducted a research at Microsoft Research asking its employees why a bug report would be reopened multiple times before being fixed. The responses led to the following common causes (in no particular order):

(a) developers could not reproduce the bug and closed it as `WORKSFORME` (or the equivalent in Microsoft's issue tracking system);

(b) a bug initially marked as `WONTFIX` had its priority increased after new information came in;

(c) the test team evaluated a software release in which the fix had not yet been applied, and therefore reopened the issue report;

(d) the root cause of the bug was not properly identified, so developers just fixed a different bug;

(e) the developer missed a special case in a fix that was discovered during testing;

(f) the test team initially evaluated the fix as appropriate, but then found that it was incomplete.

Causes (a) and (b) are triage-related. Cause (c) is related to problems neither in the issue report nor in the change, but in the code integration process. Causes (d), (e), and (f) are change-related.

Park et al. (2012) analyzed changes containing programming errors that led to the need to submit one or more supplementary changes for the same problem. They analyzed a sample of 100 issue reports that were referenced at least twice in commits from Eclipse and Mozilla. They discovered the following types of errors of omission:

- the first change did not cover all platforms on which the product runs;
- the first change introduced an incorrect conditional statement;
- the first change modified a unit of code, but did not modify related units as required;
- the first change created an API that was later deemed inappropriate;
- the first change included an incomplete refactoring;

The Code Review FAQ page at the Mozilla Developer Network[1] provides a list of reasons why a change would be rejected during code review:

- the code did not resolve the right problem;
- the code modified an API and the resulting design was deemed inappropriate;
- the code was unreadable or ill-documented;
- the code did not follow the coding style guide;
- the code introduced potential security flaws (e.g., by not sanitizing the input);
- the code did not integrate well with other modules;
- the code did not include the required unit tests;
- the code did not follow the licensing rules.

[1] See ⟨https://developer.mozilla.org/en/docs/Code_Review_FAQ⟩.

3.4.3 What Characterizes Rejected Changes?

It is important to identify the causes of change rejection in order to avoid the mistakes that lead to it. No less important is to find out in which aspects rejected changes differ from the rest; these findings can lead to models to predict which changes are likely to be rejected and which issue reports are likely to contain rejected changes. Such prediction models can be used to focus early verification efforts on those issue reports and on changes that are more likely to cause trouble.

Shihab et al. (2010, 2012) developed a decision tree model to predict the reopening of issue reports from Eclipse 3.0, based on data available in the issue tracking system and the version control system. The model had a precision of 62.9% and a recall of 84.5%. Their model considered 22 factors that could influence the reopening of issue reports, including attributes of the issue report itself, attributes of the change, and human factors. The following factors were considered the most important in predicting reopening:

- *issue description and comments*: terms such as "debugging", "breakpoint", and "platforms" are associated with reopening;
- *time developers took to submit the first change to the issue*: the longer it takes, the greater the likelihood of reopening;
- *issue component*: certain components of a system are more prone to issue reopening.

Park et al. (2012) investigated the characteristics of commits that required supplementary commits for the same issue report. They found that such commits tend to be larger and more spread out in the code base.

Zimmermann et al. (2012) developed a logistic regression model to explain the reopening of issues in Windows Vista and Windows 7. They reached the following conclusions:

- bugs found by code review or by code analysis tools are less likely to be reopened, supposedly because they are easier to triage and resolve;
- on the other hand, bugs found by users or system testing are more likely to be reopened, because they are more complex and difficult to reproduce;
- bugs with greater initial severity are more likely to be reopened;
- when the person who reported a bug and the person who is initially assigned to resolve the bug are from different teams, the bug report is more likely to be reopened.

It should be noted that the authors did not make a distinction between triage-related and change-related reopening.

Caglayan et al. (2012) studied a corporate system to identify factors that characterize the reopening of issues. After training logistic regression models for the data of an issue tracking system, they determined that the following factors are relevant:

- *developer activity*: issues assigned to developers who resolved issues recently or who changed many source code units are more likely to be reopened;

- *issue centrality*: issues that require changes in source code units that are associated with many issues are more likely to be reopened;

- *geographical location*: when the issue reporter and its assignee are geographically apart, the issue is more likely to be reopened.

Again, authors did not make a distinction between triage-related and change-related reopening. Moreover, they interviewed developers from the company's quality assurance team in order to better understand the results. The developers suggested part of the results could be explained by two factors that favor reopening: *developer workload* and *issue complexity*.

Jongyindee et al. (2011) investigated issues from Eclipse that were reopened after receiving a commit. They found that issues resolved by developers who perform more commits are less likely to be reopened.

Almossawi (2012) investigated issues from the GNOME project that were reopened after being closed with resolution `FIXED`. They found that such issues tend to be associated with source code with high cyclomatic complexity.

3.4.4 What is the Cost of Rejected Changes?

The costs associated with change rejection have been characterized by its occurrence and by time costs. The occurrence is measured by the issue reopening rate (proportion of all issues that are reopened), the supplementary change rate, and the patch rejection rate (during code review). Time costs are measured by comparing the lifetime (i.e., time between the creation and the final closing of an issue report) of reopened and non-reopened issues.

The following costs have been found in the literature:

- *Issue reopening rate.* Analyzing issues with resolution `FIXED`, the reopening rate is 2.15% on the GNOME project, 1.35% on Evolution, and 3.88% on GTK+ (ALMOSSAWI, 2012), and oscillates between 9.0% and 9.5% on Eclipse and NetBeans (WANG; BAIK; DEVANBU, 2011). Analyzing only issue reports that can be linked to commits in Eclipse, the reopening rate raises to 11.7% (JONGYINDEE et al., 2011).

- *Supplementary changes rate.* Park et al. (2012) found that 22% to 33% of all closed issue reports were associated with supplementary commits (analyzing projects from Mozilla and Eclipse).

- *Change rejection rate (code review).* Jeong et al. (2009) found that about 7% of the changes submitted to code review are rejected.

- *Time costs.* Reopened issue reports have a longer lifetime. For Shihab et al. (2010), the lifetime of reopened issues (not making a distinction between triage-related and change-related reopening) is twice as long of that of non-reopened issues. Jongyindee et al. (2011) noticed that issues reopened after being resolved with a

commit also have a longer lifetime than non-reopened issues, although they did not quantify the difference. Park et al. (2012) concluded that issues associated with supplementary changes have a lifetime 56% to 91% longer than those associated with a single change.

3.5 RAPID RELEASES

Mozilla's adoption of rapid releases attracted the attention of researchers, who measured the impact of this change. They analyzed the impact of rapid releases on Mozilla under multiple perspectives, such as quality, reputation, and security.

Baskerville and Pries-Heje (2004) interviewed employees from companies that adopted rapid release cycles in 2000. According to an interviewee, rapid release cycles limit reuse and systematic thinking. Also, the authors found that, in the companies they studied, process quality and product quality were sacrificed in the name of meeting customers' vague requirements.

Khomh et al. (2012) studied the effect of rapid release cycles on the quality of Mozilla Firefox, assessed by three metrics: number of post-release bugs created per day, crash rate, and uptime (i.e., the time between a user starting up Firefox and experiencing a failure). Comparing data from traditional and rapid releases, they found no significant difference in the number of post-release bugs and crash rate. The uptime, on the other hand, was significantly lower in releases developed in rapid cycles, meaning that the software crashed earlier during its usage.

Mäntylä et al. (2013) studied the effects of rapid releases on test case executions at Mozilla. They found that, under rapid releases, testing was more focused: the number of test executions increased, but the scope was reduced. Instead of running the whole test suite, engineers narrowed the scope to high-risk features and regressions. The authors also noted that, to keep up with testing needs, more specialized testers were hired.

Plewnia et al. (2014) studied the impact of release cycle length on the market share and reputation of three web browsers, including Firefox. They observed that Firefox's market share dropped after the adoption of rapid releases, and its reputation among users reduced. They argue that Firefox users feared that rapid releases would break their stable environments (e.g., corporate web-based information systems) and would become a burden for system administrators who would have to install new versions of Firefox every six weeks. In the first versions developed under rapid releases, updates prompted a confirmation dialog and required administrative access. The reputation was gradually recovered with the introduction of silent updates and Mozilla's introduction of extended support releases aimed at corporate environments.

Clark et al. (2014) found that rapidly-released versions of Firefox do not contain more security vulnerabilities than the extended support releases. They also note that rapid releases are a "moving target" for attackers. Most vulnerabilities in a rapid release version are found only when the version is not longer the newest one.

3.6 SUMMARY

In this chapter, we presented related work on mining software repositories, code reviews, issue lifetime, change rejection, and rapid releases. These are some important points to remember:

- Mining software repositories, such as issue tracking systems and version control systems, allows researchers to validate hypotheses related to software development.
- Change rejection is not a concept found in the literature, but it is related to issue reopening, supplementary changes and to rejection during code reviews.
- Issue reopening can be triage-related or change-related. The latter is a form of change rejection, while the former is not.
- Reopened issues have a longer lifetime.
- At Mozilla, the introduction of rapid releases resulted in less comprehensive testing, but it did not cause an increase in post-release bugs or security vulnerabilities. At first, rapid releases harmed Mozilla's reputation, due to implementation and communication issues, but the reputation has since been recovered, partly due to the adoption of silent updates.

Chapter 4

DATA AND METHODS

In this chapter, we present the data used in this work and describe in detail how we pursued the research goals enumerated in Chapter 1. To this end, we further divide each research goal in research questions and explain how to answer them by computing metrics from issue tracking systems and source code repositories.

4.1 GOALS AND QUESTIONS

Each research goal is fulfilled by answering specific research questions (RQ), which are detailed below.

4.1.1 Research Goal 1

RG1: Propose and compare techniques to detect change rejection.

RQ1.1: How do supplementary commits and reverts compare? *Rationale*: various studies use techniques based on those rejection types to detect inappropriate changes (JEONG et al., 2009; SHIHAB et al., 2010; PARK et al., 2012). By answering this question, we determine whether those techniques are equivalent or not.

RQ1.2: How do reopening, late reverts, and late supplementary commits compare? *Rationale*: those rejection types occur after a change is committed and, if they are always associated with inappropriate commits, they should be equivalent. By counting the number of issues associated with each combination of those rejection types we can determine if they actually agree on the set of inappropriate commits.

RQ1.3: What is the performance (precision and recall) of existing techniques? *Rationale*: because those techniques are heuristics to detect inappropriate changes, it is worthwhile to understand how accurate they are so to better understand potential threats to validity when using them.

RQ1.4: How do negative reviews and reverts compare? *Rationale*: negative reviews and reverts are two rejection types that occur in distinct periods of a change's lifecycle.

Nonetheless, determining how often they overlap is useful to understand the relationship between code reviews and automated tests.

4.1.2 Research Goal 2

RG2: Quantify rework triggered by inappropriate changes.

RQ2.1: What proportion of issues involves rework (rejection rate)? *Rationale*: the rejection rate helps characterize the number of issues that require rework relative to the total number of issues.

RQ2.2: How often are issues rejected (rejections per day)? *Rationale*: are issues rejected every day? Every week? The number of rejections per day helps characterize rework in a project's schedule.

RQ2.3: What is the impact of inappropriate changes on issues' lifetimes (additional time)? *Rationale*: even if the rejection rate is low, the impact of inappropriate changes can be high if they contribute to a much longer issue lifetime.

4.1.3 Research Goal 3

RG3: Empirically validate hypotheses about rework.

RQ3.1: Do appropriate changes take longer to submit? *Rationale*: intuition suggests that developers should spend more time carefully writing and testing a change so it is more likely to be appropriate. We investigate whether time to submission is correlated with change appropriateness.

RQ3.2: Are inappropriate changes likely to be released? *Rationale*: while in RG2 we investigate the impact of inappropriate changes on the development team, in this question we investigate their impact on product quality as perceived by end users.

RQ3.3: Is time to post-rejection submission correlated with latent time? *Rationale*: intuition suggests that the longer a problem goes unnoticed (*latent time* or *time to rejection*), the longer it takes to fix it, due to the developers' fading memory about the context of the original issue. In this question we test this intuition.

RQ3.4: Is time to post-rejection submission correlated with time to original submission? *Rationale*: we investigate whether an issue that takes more time to resolve also take more time to re-resolve.

4.1.4 Research Goal 4

RG4: Assess the impacts of process changes on rejections.

RQ4.1: How has the developer workload changed under the new process? *Rationale*: developer workload is a factor that may affect other metrics considered in this study, such as rejection rate, so it is important to understand how it changed over time.

RQ4.2: How has the rejection rate changed under the new process? *Rationale*: measuring the rejection rate is a simple way of verifying whether the process was an improvement or a deterioration regarding the creation of inappropriate changes.

RQ4.3: How has the early revert rate changed under the new process? *Rationale*: separating early and late reverts helps put reverts in perspective, since late reverts are

riskier with respect to the release of inappropriate changes.

RQ4.4: How has the total additional time caused by inappropriate changes varied under the new process? *Rationale*: measuring the variation of the total additional time also helps verifying whether a new process was an improvement with respect to its predecessor but, differently from RQ4.2, it takes the time dimension into account.

4.2 RESEARCH METHODS

To answer the research questions, we rely on quantitative analysis of data from Mozilla's issue tracking system and version control repositories, as well as feedback from Mozilla engineers.

Research goals 1 and 2 are descriptive. To pursue these goals, we describe aspects of the data using Venn diagrams, tables, and box plots. Research goals 3 and 4 require determining whether there are significant differences between two subsets: appropriate vs. inappropriate changes, early vs. late rework, new process vs. old process, among others. To answer the corresponding research questions, we additionally perform statistical tests. In particular, for the questions related to rejection rate, we use Fisher's exact test for count data; for the questions related to time intervals or productivity, we perform Mann-Whitney's U test for ordinal data. Correlations are estimated using Kendall's tau statistic. All statistical tests used are non-parametric, suitable for non-normal distributions, which are common in our analyses.

In order to better understand the process and to help interpret the observed differences, we also talked to Mozilla engineers through the `firefox-dev` mailing list. We reported our findings using numbers and graphs and asked engineers if the results were expected and how they could be explained. The messages exchanged with engineers are available in Appendix A.

4.3 DATA EXTRACTION

In order to answer the research questions, we analyzed data from two sources: Mozilla's issue tracking system (Bugzilla), and version control repositories. In the following sections we describe both data sources.

4.3.1 Issue Tracking System

For the first data source, a Mozilla engineer provided a SQL database dump of Mozilla's issue tracking system. The dump contains everything that is stored on Mozilla's installation of Bugzilla—including issue reports and the history of all their modifications,—except for user data, left out for security and privacy reasons, and attachment contents, which would increase significantly the data set size. The data set contains almost 880,000 issue reports created for 97 projects from September 1994 to November 2013.

In this thesis, we focus on two database tables contained in the dump: `bugs` and `bugs_activity`. The table `bugs` contains one record for each issue report. Each record contains the issue's numeric identifier, the person who created it, the creation date, and the latest value for fields such as title, description, and status. The table `bugs_activity`

```
changeset:  ec12c4e4bcd3
user:       Developer 1
date:       Mon Nov 03 16:46:01 2014 -0500
summary:    Bug 1043699 - Backout of changeset 6921bd616ff1. DONTBUILD.

changeset:  b217ba1685f4
user:       Developer 2
date:       Tue Nov 04 06:35:12 2014 +0900
summary:    Bug 1092813 - Update the SDK path to 8.1. r=mshal

changeset:  66cdb18f36da
user:       Sheriff
date:       Wed Nov 12 15:32:16 2014 -0500
summary:    Merge b2g-inbound to m-c. a=merge
```

Figure 4.1: Data extracted from three commits.

contains all modifications users made to issue reports over time, including changes in status, resolution, review flags, and any other field in an issue report. Each record contains the new value of a field, the user who altered the field, and the time of the modification.

The database dump used to be publicly available, but was withdrawn in August 2014. As of November 2014, Mozilla engineers are working on improving the data sanitization process in order to release database dumps again. For more information see issue reports 1013953 and 1054795 at ⟨https://bugzilla.mozilla.org/⟩.

4.3.2 Version Control Repositories

We also cloned two version control repositories, Mozilla-Central and Comm-Central, publicly available at ⟨https://hg.mozilla.org/⟩. From those repositories we extracted the commit log, containing, for each commit, its identifier, author (user), date, and summary message.

Figure 4.1 illustrates the data extracted from three commits using the `hg log` command. The field `changeset` is the commit's hexadecimal identifier, and `summary` is the commit message.

4.4 DATA FILTERING

The available Bugzilla and source code repositories span multiple projects and a large time period. In this study, we select a subset of the projects and a smaller time span to analyze, as described next.

4.4.1 Projects

In our Bugzilla data set, there are 97 projects registered. They include popular software products, such as Firefox, but also less known products used internally and even projects not focused on software development, such as websites.

We looked at the most active software development projects since 2009, two years before the changes in Mozilla's process, measured by the number of issue reports with status `FIXED`. The rationale is that our analyses rely on issue reports and commits under two periods, before and after the process changes. Using these criteria, we selected three projects:

- **Core**. Shared components used by Firefox and other Mozilla software, including handling of Web content; Gecko, HTML, CSS, layout, DOM, scripts, images, networking, etc. Issues with web page layout probably go here, while Firefox user interface issues belong in the Firefox product.
- **Firefox**. For bugs in Firefox Desktop, the Mozilla Foundation's web browser. For Firefox user interface issues in menus, developer tools, bookmarks, location bar, and preferences. Many Firefox bugs will either be filed here or in the Core product.
- **Thunderbird**. Email client originally developed by the Mozilla Foundation and now maintained by its community.

Then, by browsing the projects' web pages, we identified and cloned the projects' version control repositories:

- **Mozilla-Central**: Mercurial repository containing code for Core, Firefox, and other projects.
- **Comm-Central**: Mercurial repository containing code for Thunderbird.

We selected only issues that are closed with resolution `FIXED`, and that are associated with at least one commit. This subset corresponds to the issues that were resolved by a source code change.

4.4.2 Time Span

Table 4.1 shows, for each data source, the dates of the earliest and latest issues and commits. Based on the table and on the release dates[1] for Mozilla's products, we selected the period from June 20, 2011 to September 16, 2013, i.e., from the release of Firefox 5 up to the release of Firefox 24. Under this period, Mozilla's projects adopted rapid, 6-week releases, and some projects, such as Core and Firefox, also adopted sheriff-managed integration repositories.

Specifically for the research goal 4, which requires comparing two periods of Mozilla's history—traditional releases and rapid releases,—we additionally use data from June 29, 2009 to March 21, 2011, as shown in Figure 4.2. During this period, Mozilla's engineers developed versions 3.6 and 4.0 of Firefox according to a traditional release process.

[1] Available at ⟨https://wiki.mozilla.org/RapidRelease/Calendar⟩.

Table 4.1: Minimum and maximum dates of issue creation and commits in the available data.

Repository	Min. Date	Max. Date
Core's issues	July 20, 1999	November 28, 2013
Firefox's issues	May 22, 2001	November 28, 2013
Thunderbird's issues	October 26, 2000	November 26, 2013
Mozilla-Central's commits	March 22, 2007	October 18, 2014
Comm-Central's commits	July 17, 2007	November 28, 2013

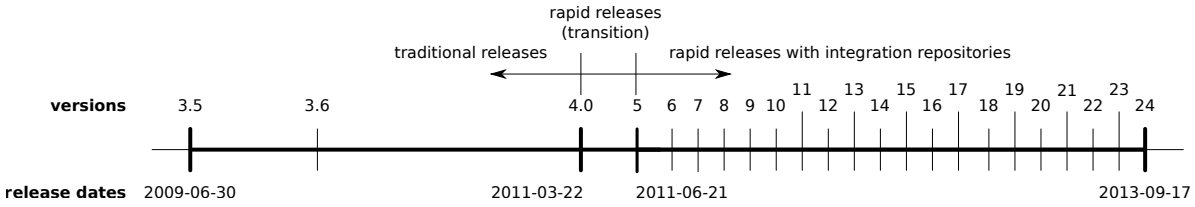

Figure 4.2: Periods under analysis.

4.5 EVENTS AND METRICS

In order to measure rework in multiple points in an issue's lifecycle, we first identify important events in this lifecycle and then define metrics around those events.

4.5.1 Events

Figure 4.3 shows important events associated with an issue report. We define event as the interaction of a user with an issue tracking system or a version control repository in a specific point in time with the goal of recording a task he or she performed. Events are related to an issue's creation and to a change's submission and rejection. Each row in Figure 4.3 corresponds to events related to a specific subprocess: (a) code review, (b) commit and automated testing, and (c) closing and manual testing.

A *change submission* event signals that a change is ready to be evaluated. There are three types of change submission:

- *review request*: the change is ready to be reviewed;
- *commit*: the reviewed change is ready to be automatically tested;
- *closing* (with resolution `FIXED`): the automatically tested change is ready to be manually tested.

A change submission can be either original or post-rejection. A change submission is original if it is the first submission of its type for its issue. Otherwise, it is post-rejection if it is a supplementary change that occurs after a rejection of its type (e.g., if it is a commit performed after a revert).

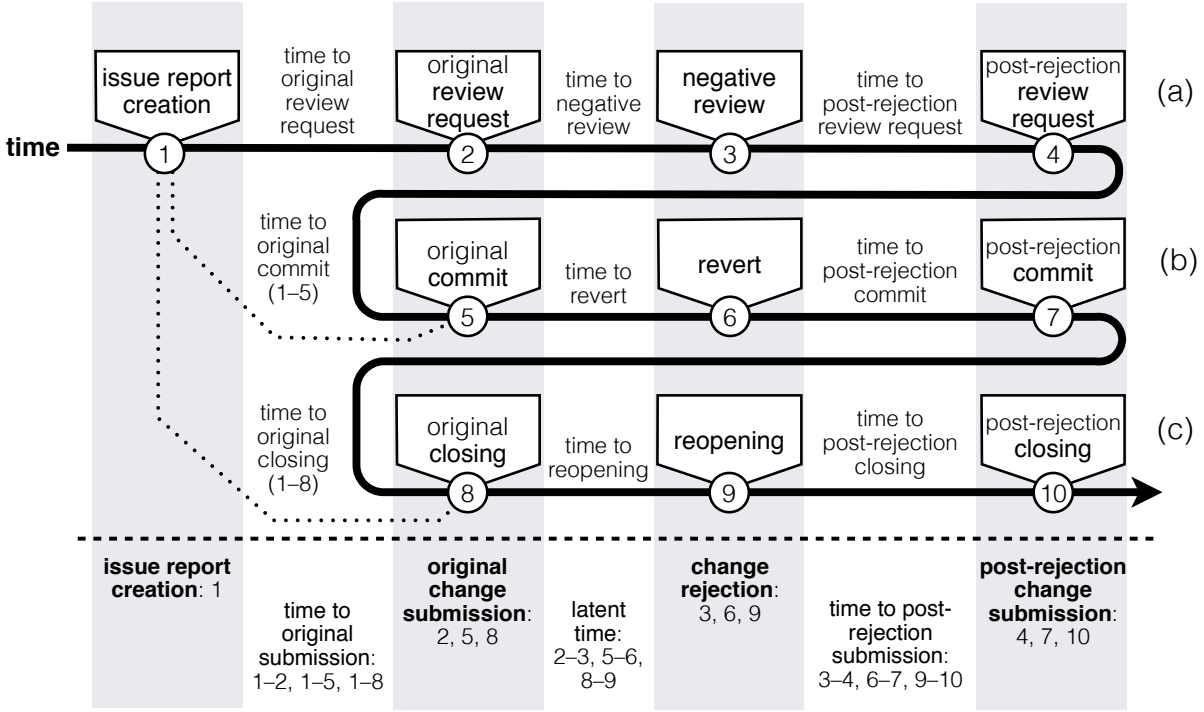

Figure 4.3: Time-interval metrics.

A *change rejection* event signals that the change was considered inappropriate. Analogously, there are three types of rejection:

- *negative review* (related to a review request): a problem was found during code review;
- *revert* (related to a commit): a problem was found after a commit, either before or after a successful build (*early revert* or *late revert*, respectively);
- *reopening* (related to an issue closing): a problem was found for an issue closed after a successful build; usually, that means that the change should also be reverted.

4.5.2 Metrics

From these events, we compute metrics in two categories: counting metrics and time-interval metrics. Counting metrics are computed by counting the number of issues that contain a particular event; those metrics are reported either as absolute values (e.g., number of issues with negative reviews), or relative to another event (e.g., proportion of issues with negative reviews among issues with review requests). Time-interval metrics are computed by measuring the time interval between two relevant events (e.g., time from creation to first review request).

For each project we compute the following absolute counting metrics:

- number of issues with a negative review;

- number of issues with a revert;
  - number of issues with an early revert;
  - number of issues with a late revert;
- number of issues with a reopening.

We also compute the following relative counting metrics:

- *negative review rate*: proportion of issues with a negative review (among issues with a review request);
- *revert rate*: proportion of issues with a revert (among issues with a commit);
  - *early revert rate*: proportion of issues with a revert before the issue is closed (among issues with a commit);
  - *late revert rate*: proportion of issues with a revert after the issue is closed (among issues with a commit);
- *reopening rate*: proportion of issues with a reopening (among all issues with a closing).

Figure 4.3 also shows time-interval metrics between events. Each time-interval metric is the duration of the time interval between two consecutive events. For instance, time to revert is the duration of the interval between an original commit and the subsequent revert. Time-interval metrics are grouped in three categories:

- *time to original submission*: time between an issue report creation and the original submission of a given type;
- *latent time* or *time to rejection*: time between the original submission and the subsequent rejection;
- *time to post-rejection submission*: time between a rejection and the subsequent submission.

4.5.3 Event Detection

Most events can be trivially detected from the data logged to issue reports, while others require performing pattern matching on textual data. The following events can be easily detected from the data available in issue reports:

- *issue report creation*: all issues have a creation event, and the creation date is stored in the issue's `reported` field;
- *review request*: occurs when the `review?` flag is added to an attachment in the issue;

- *negative review*: occurs when the `review-` flag is added to an attachment in the issue;
- *closing*: occurs when the issue's status is changed to `RESOLVED` and its resolution is changed to `FIXED`;
- *reopening*: occurs when the issue's status is changed to `REOPENED` when its resolution is set to `FIXED`.

The *commit* and *revert* events require analyzing the unstructured text of commit messages. The challenge lies in discovering if a commit represents a *commit* event (meaning that it is a change intended to resolve an issue) or a *revert* event (intended to revert an inappropriate commit), and then determining what issue it intends to resolve or revert.

Commit. We determine that a commit intends to resolve a specific issue report in a project if its message starts with the word "bug", followed by a 6-digit number, and that number corresponds to an issue report in the project. By querying the Bugzilla database, we determined that all bugs created between 2009 and 2013 had identifiers with exactly 6 digits. It should be noted that not all commits reference issue identifiers, since some commits are merges or "bumps" (commits intended to update version numbers); those commits are not classified as *commit events* in this study.

Revert. Commits whose message matches the following regular expression are detected as reverts:

```
back(ing|ed|s)?( it )?( |-)?out|backout
```

This regular expression was successively refined through manual inspections of commit messages and it matches the following expressions found in commit messages: "backout", "backed out", "back out", "backing out", "backs out", "backedout", "back-out", "backed-out", "backing-out", "back it out", and "backing it out". Some of these variations of the term "backout" are forbidden in Mozilla-Central's commit hooks, but they can nonetheless be found in older commits (created before the hook) and in Thunderbird's commits.

All 6-digit numbers following the regular expression are interpreted as the identifier of the issues being reverted. All 6- to 12-digit hexadecimal numbers containing at least one letter from A to F (i.e., letters that represent hexadecimal digits) are interpreted as identifiers for commits being reverted. When a revert makes reference to a commit identifier, we link the revert commit to the issue fixed by the referred commit. For instance, if the message for commit `f96e3c57e1d8` says "Backs out revision 0f76f410b03b", and the message for commit `0f76f410b03b` is "Bug 123456 – fix window size", then we infer that commit `f96e3c57e1d8` is a revert for the issue report 123456.

After reading a sample of revert commit messages, we observed that numbers following certain expressions do not represent the issue that was reverted. For instance, in the message "Backout bug 555133 to fix bug 555950", the number 555133 is the issue being reverted, however the number 555950 is the issue being resolved. After some analysis, we decided to ignore 6-digit numbers following the expressions listed below:

- *resolve*, *fix*: e.g., "Backout bug 555133 to fix bug 555950";
- *causing*, *cause*, *because*: e.g., "Backed out changeset 705ef05105e8 for causing bug 503718 on OS X";
- *due to*: e.g., "Backed out changeset 58fd8a926bf5 (bug 366203) due to it causing bug 524293";
- *suspicion*: e.g., "Backout revisions (...) on suspicion of causing (...) bug 536382".

4.6 THREATS TO VALIDITY

As in any empirical study, the validity of the results presented in this paper is subject to threats. The most relevant threats to construct, internal, and external validity are described below.

4.6.1 Construct validity

Construct validity means to what extent the study measures what it intends to measure. The main threats to construct validity in this study are related to time-interval metrics and to the techniques used to detect change rejection.

The original and post-rejection submission time-interval metrics can be thought as a proxy for the time spent developing either an original change or an improved change. However, with the available data, we cannot directly measure the time a developer spent actively writing a change to a specific issue, since the available time intervals include the time when the developer was either not working or working on other issues. Therefore, submission metrics should not be directly interpreted as amount of work or rework; instead, they should be regarded as metrics related to potential process delays.

The other threat refers to the techniques used to detect change rejection. None of them is 100% accurate. Nonetheless, we believe some of them are appropriate for the purpose of this study. In particular, we believe that negative reviews and commit reverts detect change rejections—respectively during code review and after code review—with high precision. In other words, almost all of the events they detect are indeed change rejections, although there are change rejections that they do not detect. For instance, when a patch has minor problems, a reviewer may choose to verbalize the problems without explicitly setting the `review-` flag. Also, an inappropriate commit may be fixed by a follow-up commit instead of being reverted. However, with recent changes in Mozilla's process, especially the assignment of sheriffs that watch the build, follow-up commits are uncommon.

4.6.2 Internal Validity

Internal validity means to what extent conclusions can be made from what was measured. For research goal 4, the observed differences under traditional and rapid releases could be attributed to factors other than the release length and the introduction of sheriff-managed integration repositories, since the development process may have changed over time in

many different ways. For this reason, in Chapter 6 we report other factors that could contribute to the observed changes, based on discussions with Mozilla engineers.

4.6.3 External Validity

External validity means to what extent results can be generalized. All the conclusions resulting from this study are based on data from one single software organization. The results are not expected to be generalizable to other projects; instead, the purpose of this study is to provide insights on the questions being studied. Furthermore, it is not trivial to extend the study to other projects, since detecting important events and interpreting the results require a high level of understanding about the process, in a detailed perspective, as well as about changes in the process over time.

4.7 SUMMARY

In this chapter, we presented the research questions related to previously presented research goals, described the available data and methods used to answer the questions, and discussed potential threats to validity. In the next chapter, we report the quantitative results of this study.

Chapter 5

RESULTS

This chapter presents quantitative results for the research questions enumerated in the previous chapter. In the next chapter, we discuss possible interpretations of the results.

It should be noted that, as stated in Section 2.6, Thunderbird's process differ from Core's and Firefox's because it does not use sheriff-managed integration repositories and does not enforce commit message conventions via commit hooks. Because of that, follow-up commits are more common in Thunderbird and, as a result, metrics based on reverts are less reliable as indicators of inappropriate changes in the project.

5.1 RG1: PROPOSE AND COMPARE TECHNIQUES TO DETECT CHANGE REJECTION

In the previous chapter, we presented a new technique to detect change rejection by looking for terms such as "backout" and "backing out", which represent reverts, in commit messages. In this section, we compare this technique with techniques proposed by other authors, namely, issue reopening, supplementary commits, and negative code reviews.

First, we compare two rejection types applicable to changes that were already committed: supplementary commits and reverts. Then, we compare reopening with late reverts and late supplementary commits, which are rejection types applicable to closed issues. After that, we evaluate the precision and recall of rejection types, assuming reverts as a reference. Finally, we compare negative reviews, which are pre-commit rejections, with reverts, which occur after a commit.

The comparisons are performed for all three projects, Core, Firefox, and Thunderbird. The period under analysis is the rapid release period, from June 20, 2011 to September 16, 2013.

5.1.1 RQ1.1: How do supplementary commits and reverts compare?

Figure 5.1 presents Venn diagrams that show the proportion of all issues that received either supplementary commits or reverts, or both. The percentages outside the circles

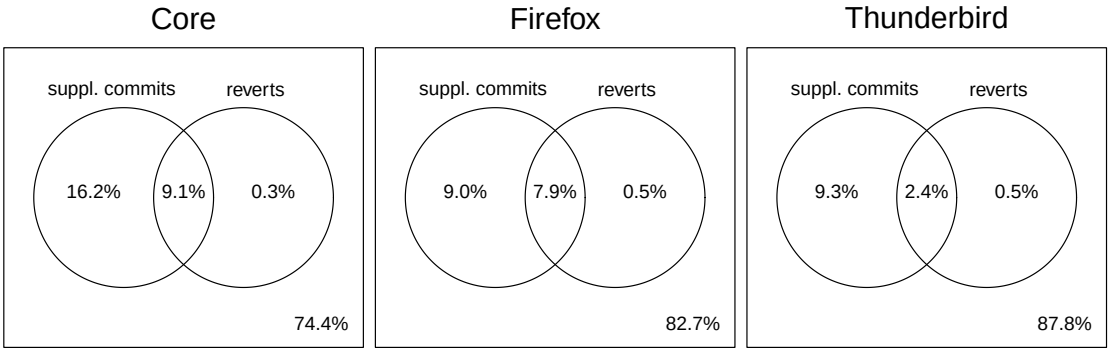

Figure 5.1: Venn diagram comparing supplementary commits and reverts.

are the proportion of issues in our data set that did not receive either supplementary commits or reverts. The sum of the proportions equals 100% (minus rounding errors).

The diagrams for Core and Firefox show a large intersection between reverts and supplementary commits. In fact, very few reverts are not associated with supplementary commits. For Thunderbird, there are comparatively fewer reverts, which suggests that Thunderbird developers either do not systematically revert inappropriate changes, or they fail to communicate which of their commits are reverts. The numbers for Thunderbird reverts are consistent with the observation that its source code repository, unlike Mozilla-Central, does not implement hooks to enforce commit message conventions.

On the other hand, most issues with supplementary commits are not associated with reverts. This observation suggests that supplementary commits account for situations that are not covered by reverts.

> Most reverts are associated with supplementary commits, but most supplementary commits are *not* associated with reverts.

5.1.2 RQ1.2: How do reopening, late reverts, and late supplementary commits compare?

Reopening is different from supplementary changes and reverts because an issue can only be reopened after it is closed, i.e., after the change is committed and passes automated testing. To better compare reopening to other rejection types, we define that an issue has late supplementary commits when it is associated with at least two commits, with at least one commit after the issue is closed. Analogously, we say that an issue has a late revert when it is associated with a revert performed when the issue was closed.

Figure 5.2 presents Venn diagrams comparing reopening with late reverts and late supplementary commits. The diagrams show that most reopenings are not associated with the other two late rejection types. These are what Le An et al. (2014) call *premature* reopenings, and are probably unrelated to inappropriate changes.

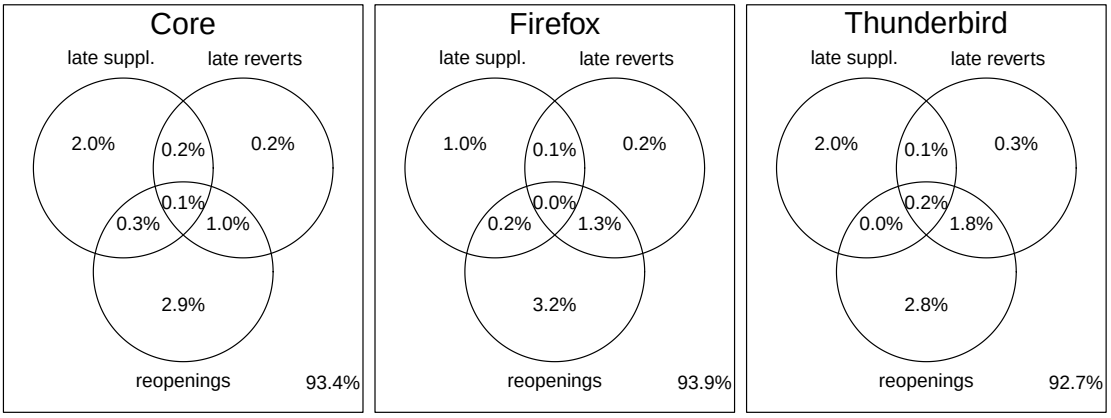

Figure 5.2: Venn diagram comparing reopening, late commits, and late supplementary changes.

Most reopenings are premature: they are not associated with either late reverts or late supplementary commits.

5.1.3 RQ1.3: What is the performance (precision and recall) of existing techniques?

We do not have an *oracle* that correctly classifies each issue as associated or not with inappropriate changes. Building such an oracle requires examining the contents of each change and reading each comment in the corresponding reports. Even so, determining whether a change is inappropriate or not is subjective, besides being a time-intensive and error-prone process.

Although we cannot confidently compute the precision and recall of existing techniques without such an oracle, we can provide rough estimates under the assumption that the oracle can be approximated by one of the rejection types. In this study, we choose reverts as the oracle to evaluate supplementary commits, because we believe that they best represent the act of rejecting an inappropriate change that was committed (the reasons for such belief are discussed in Chapter 6). For reopenings, we use late reverts as the oracle, because late reverts and reopenings are intended to detect the same subset of inappropriate changes, those rejected after the corresponding issue is closed.

Based on this assumption, we determine the performance of reopenings and supplementary commits as techniques to detect inappropriate changes, measured by their recall and precision. Recall is the proportion of all issues with inappropriate changes that the techniques correctly detect; precision is the proportion of all issues detected as having inappropriate changes that actually have inappropriate changes.

Table 5.1 presents precision and recall for both reopening and supplementary commits. Results for Thunderbird are shown for completeness, but they should be ignored because our previous results suggest that reverts underrepresent inappropriate changes

Table 5.1: Precision and recall of reopening and supplementary commits.

	Core		Firefox		Thunderbird	
Rejection Type	precision	recall	precision	recall	precision	recall
Supplementary commits	35.9%	96.9%	46.8%	94.5%	20.2%	81.5%
Reopening (vs. late reverts)	25.3%	75.0%	28.5%	79.7%	42.2%	82.6%

in Thunderbird.

The results in the first row of Table 5.1 suggest that supplementary commits have a very high recall at the expense of a low precision. That means that almost all issues with inappropriate changes are associated with supplementary commits, although most supplementary commits are false positives regarding inappropriate changes.

As shown in the second row of Table 5.1, using reopening to detect inappropriate commits that passed automated testing results in a moderately high recall, although lower than that of supplementary commits. However, the precision is low, meaning that most reopenings are not associated with inappropriate commits.

> Taking reverts as a reference, existing techniques can find most issues with inappropriate commits, but their precision is low.

5.1.4 RQ1.4: How do negative reviews and reverts compare?

Negative reviews and reverts are rejections that occur in distinct periods. While negative reviews represent the rejection of a change that was submitted to code review, reverts represent the rejection of a change that already passed code review and was committed after that. Nonetheless, an issue can be associated with both rejection types. A negatively reviewed change eventually results in an improved change that is positively reviewed and committed, but the improved change can be rejected after that.

Figure 5.3 shows the intersection between negative reviews and reverts. Although most issues with negative reviews are not associated with reverts and vice-versa, there is still a significant number of issues associated with both rejection types.

In order to better understand the relationship between negative reviews and reverts, we plotted mosaic plots, a generalization of bar plots, shown in Figure 5.4. The mosaic plots divide issues in four categories, according to the presence of negative reviews and reverts. The area of each tile is proportional to the number of observations within a category; for instance, the largest tile represent the number of issues with no negative reviews and no reverts. Highlighted tiles represent categories whose number of issues is significantly higher or lower than expected if negative reviews were independent from reverts. The mosaic plots show that there is a statistically significant positive association between negative reviews and reverts.

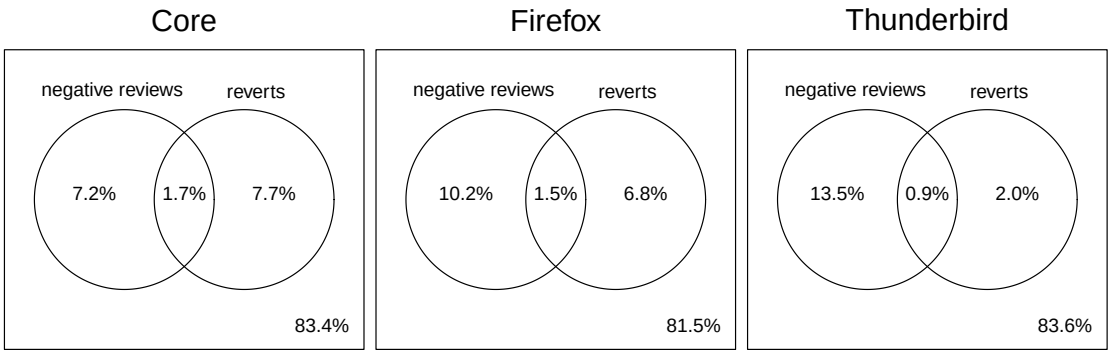

Figure 5.3: Venn diagram of rejection types.

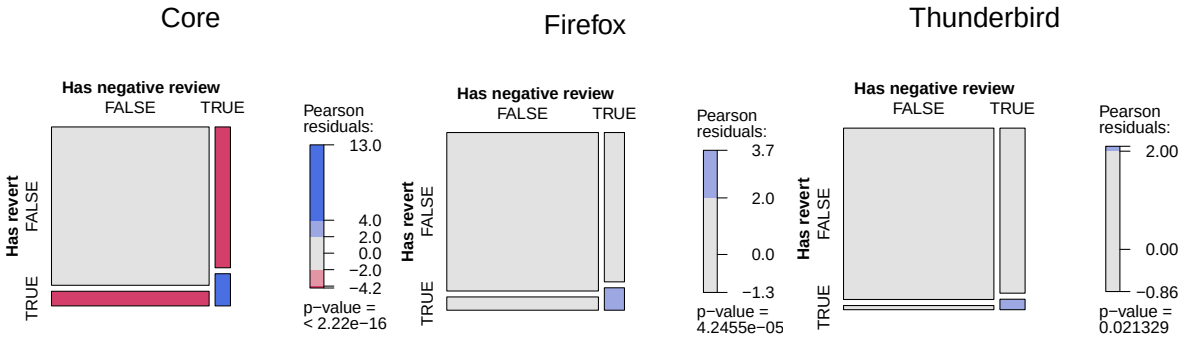

Figure 5.4: Mosaic plot showing association between negative reviews and reverts.

Issues with negative reviews are more likely to receive a revert afterwards.

5.2 RG2: QUANTIFY REWORK TRIGGERED BY INAPPROPRIATE CHANGES

In this research goal, we characterize rework by quantifying the proportion of all issues that contain rejected changes (rejection rate), which cause rework, and the proportion of an issue's lifetime that is spent on post-rejection submissions. Given the imprecision of reopening and supplementary commits when used to detect inappropriate changes (see Chapter 6), we analyze only negative reviews and reverts.

5.2.1 RQ2.1: What proportion of issues involves rework (rejection rate)?

Table 5.2 shows specific rejection rates in the period, i.e., what proportion of all resolved issues had at least one negative review or revert. Rejection rate is effectively the proportion of issues which eventually received inappropriate changes. The overall rejection rate is smaller than the sum of negative review and revert, since it is possible for a single issue to be associated with both rejection types.

The numbers show that inappropriate changes (i.e., changes that are eventually rejected) introduce a significant overhead in the process. Between 16.4% and 18.5% of all

Table 5.2: Rejection rate for multiple projects and rejection types.

Metric	Core	Firefox	Thunderbird
Negative review rate	8.9%	11.7%	14.3%
Revert rate	9.4%	8.4%	2.9%
Early revert rate	8%	6.7%	0.3%
Late revert rate	1.4%	1.7%	2.5%
Overall rejection rate*	16.6%	18.5%	16.4%

* Proportion of all issues that are associated with either a negative review or a revert, or both.

Table 5.3: Number of issues with rejections and average time between rejections in the rapid release period.

Rejection Type	Core	Firefox	Thunderbird
Negative review	1903 issues (0.4 days)	513 issues (1.6 days)	134 issues (6.1 days)
Revert	2007 issues (0.4 days)	366 issues (2.2 days)	27 issues (30.4 days)
Early revert	1706 issues (0.5 days)	292 issues (2.8 days)	3 issues (273.3 days)
Late revert	308 issues (2.7 days)	74 issues (11.1 days)	23 issues (35.7 days)
Any rejection	3556 issues (0.2 days)	812 issues (1 days)	153 issues (5.4 days)

changes are rejected either by negative review or revert (overall rejection rate), inducing overhead on developers, who have to write additional changes. Also, between 8.9% and 14.3% are rejected during code review, which also induces overhead on reviewers, who have to review another change, besides having to explain why the first one was rejected. Furthermore, except for Thunderbird, which is an outlier regarding reverts, more than 8% of all resolved issues are reverted, burdening sheriffs.

About 18% of all issues are associated with rejected changes.

5.2.2 RQ2.2: How often are issues rejected (rejections per day)?

Table 5.3 shows the number of issues with rejections and the average time between rejections in the rapid release period. For two projects, Core and Firefox, there is a rejection every day, on average, even multiple rejections a day. For Thunderbird, there is a rejection every 5 days, on average. In any case, the numbers show that rejection is an event that occurs frequently within the projects, triggering rework, and thus is worth studying and controlling.

Change rejections are a common event, occurring every day in some projects, on average.

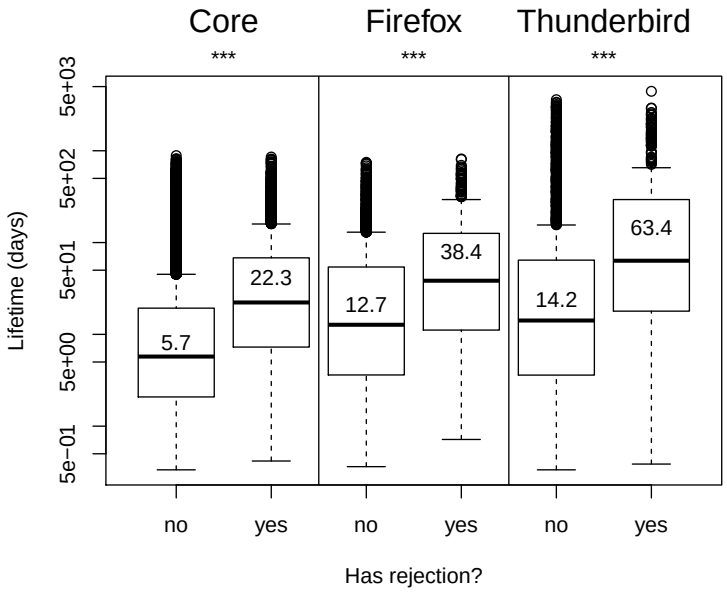

Figure 5.5: Distribution of issues' lifetimes. Asterisks represent p-values: *** $\Rightarrow p < 0.001$; ** $\Rightarrow p < 0.01$; * $\Rightarrow p < 0.05$.

Table 5.4: Average individual (per-issue) additional time attributed to inappropriate changes.

Individual additional time	**Core**	**Firefox**	**Thunderbird**
Negative review	145.9%	85.8%	128.2%
Revert	85.0%	44.2%	42.5%
Any rejection	130.0%	76.6%	119.8%

5.2.3 RQ2.3: What is the impact of inappropriate changes on issues' lifetimes (additional time)?

We determine the percentage increase of issues' lifetimes caused by inappropriate changes by measuring the lifetime of two categories of issues: issues associated with rejections and issues not associated with rejections. The lifetime is measured from an issue's creation to the last time its status is changed to `FIXED`.

Figure 5.5 shows the distribution of issues' lifetimes, grouped by whether they contain a rejection or not. As expected, issues associated with rejections have significantly longer lifetimes.

Table 5.4 shows the *individual additional time* (or per-issue additional time) associated with different rejection types. It measures how much longer, on average, the lifetime of an issue associated with a specific rejection type is when compared to an issue without any rejection. For instance, the first row compares issues whose only rejection was a negative review to issues without rejections.

The average individual additional time ranges from 76.6% to 130%. Negative reviews are associated with larger additional times when compared to reverts.

Table 5.5: Total additional time caused by inappropriate changes.

Total additional time	Core	Firefox	Thunderbird
Negative review	11.6%	9.5%	18.9%
Revert	7.2%	3.4%	0.9%
Any rejection	21.6%	14.2%	20.2%

Table 5.5 shows the *total additional time* of inappropriate changes, which measures how much additional time is spent on a project because of inappropriate changes. We estimate the total additional time by dividing the *actual total lifetime*, i.e., the sum of all issues' lifetimes, by the *expected total lifetime*, i.e., the estimated sum of all issues' lifetimes if there was no rejection. The expected total lifetime is computed as the product between the number of issues and the mean lifetime of issues without rejections.

The numbers show that negative reviews contribute the most to the total additional time, followed by reverts. This result is intuitive, since negative reviews are the most common rejection type and they are associated with a large individual additional time. The total additional time of issues associated with any rejection type varies between 14.2% and 21.6%, partly due to the fact that issues can be associated with multiple rejections.

> Inappropriate changes cause as much as a 21.6% increase to the sum of the lifetimes of a project's issues.

5.3 RG3: EMPIRICALLY VALIDATE HYPOTHESES ABOUT REWORK

In the previous sections of this chapter, we characterized rework using exploratory data analysis. In this section, we report the results of statistical tests performed to evaluate certain hypotheses about rework in light of the available data.

5.3.1 RQ3.1: Do appropriate changes take longer to submit?

To determine whether appropriate changes (i.e., changes that were not rejected) take longer to submit, we measure the time to submission, that corresponds to the time between an issue's creation and the original submission of a change for review. The metric is computed separately for appropriate and inappropriate changes. A change is considered inappropriate if it is ever negatively reviewed or reverted.

The rationale for this question is the belief that time pressure leads to inappropriate changes. Figure 5.6 tells a different story. It shows that inappropriate changes in fact take longer to submit (the difference is significant at the 0.05 level for all projects).

This result suggests that, instead, the time to submit may be correlated with issue difficulty: difficult issues require more development time, and are also more likely to be incorrectly resolved. However, we could not test this new hypothesis, since we do not have data on issue difficulty (in a future work, we may test the hypothesis using change

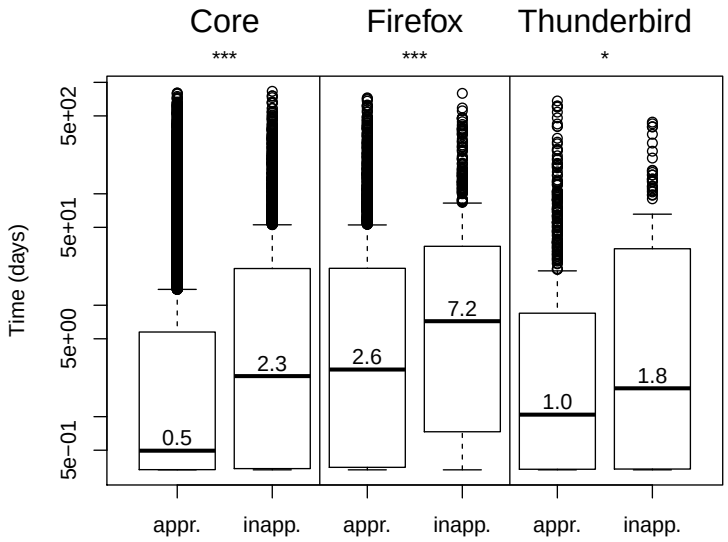

Figure 5.6: Time to submit inappropriate and appropriate changes. Asterisks represent p-values: *** $\Rightarrow p < 0.001$; ** $\Rightarrow p < 0.01$; * $\Rightarrow p < 0.05$.

Table 5.6: Mean time to submit appropriate and inappropriate changes.

Metric	**Core**	**Firefox**	**Thunderbird**
Mean time to review request (appropriate)	17.0 days	28.2 days	21.7 days
Mean time to review request (inappropriate)	29.6 days	38.7 days	39.9 days
Ratio (inappropriate / appropriate)	1.7x	1.4x	1.8x

size as a proxy for issue difficulty).

Table 5.6 shows the mean time to submit both appropriate and inappropriate changes, and the ratio between those mean times. In all projects, inappropriate changes take about 1.5x to submit, on average, compared to appropriate changes.

Changes that are eventually rejected take longer to submit; our hypothesis is that difficult issues are more likely to result in inappropriate changes.

5.3.2 RQ3.2: Are inappropriate changes likely to be released?

Latent time is the time between a rejection and the subsequent change submission. Latent time represents either the time between a code review request and a negative review, or the time between a commit and its revert. High latent times raise the risk that inappropriate changes are released to end users.

Table 5.7 shows the proportion of rejected issues that were latent for at most 12 hours, 24 hours, 1 week, and 12 weeks, before they were rejected by either a negative review or a revert. The minimum time interval before a change is released at Mozilla is 12 weeks, i.e., two 6-weeks release cycles, during which the change is stabilized in Mozilla-Aurora and Mozilla-Beta (Comm-Aurora and Comm-Beta for Thunderbird).

Table 5.7: Latent time: proportion of inappropriate issues rejected within 12 hours, 24 hours, 1 week, and 12 weeks.

Latent Time	Core		Firefox		Thunderbird	
	review-	revert	review-	revert	review-	revert
12 hours	54.5%	65.0%	55.0%	53.3%	38.1%	29.6%
24 hours	67.7%	70.1%	66.5%	61.7%	47.8%	44.4%
1 week	92.3%	85.1%	93.2%	80.6%	88.1%	66.7%
12 weeks	99.7%	97.5%	99.6%	98.4%	100.0%	92.6%

Table 5.8: Correlation between latent time and post-rejection submission metrics

Rejection Type	Core	Firefox	Thunderbird
Negative review	*** 0.22	*** 0.24	** 0.20
Revert	*** 0.15	** 0.14	-0.27

The first row of the table shows that latent times are usually low: for Core and Firefox, about half of all issues that are eventually rejected remain latent for 12 hours or less (24 hours or less for Thunderbird). Low latent times contribute to fast feedback cycles, which are valued in agile methodologies (COCKBURN; WILLIAMS, 2003). Looking at the last row, it can be seen that very few inappropriate changes take more than 12 weeks, or two release cycles, to be rejected. Thus, inappropriate changes are unlikely to be released to end users.

> At Mozilla, inappropriate changes are unlikely to be released to end users.

5.3.3 RQ3.3: Is time to post-rejection submission correlated with latent time?

During discussions in the `firefox-dev` mailing list, Mozilla engineers suggested that the greater the time between the submission of an inappropriate first change and its rejection (i.e., latent time), the longer it would take to submit the subsequent change. After all, in the end of a long latent time, the developer's memory about the issue context may not be as fresh as when he wrote the original change.

To evaluate this hypothesis, we measured the correlation between the latent time and the time to submit the subsequent post-rejection change. We used Kendall's tau statistic, which is suited for non-parametric data containing repeated values.

The estimated correlations are presented in Table 5.8. It can be observed that there is a weak positive correlation between the time to a negative review and the time to the subsequent review request, for all projects. The same trend can be observed for reverts in Core and Firefox. For Thunderbird, revert metrics are meaningless, since its developers do not systematically revert commits. Thus, longer latent time appear to be associated with more effort during rework, although the association is weak.

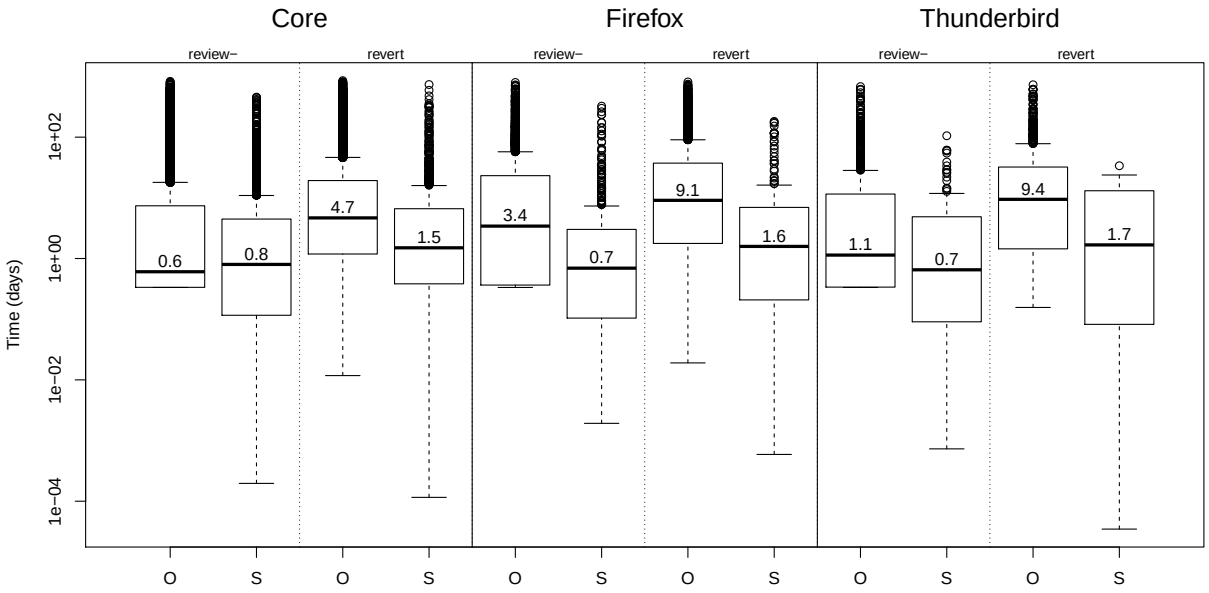

Figure 5.7: Distribution of original and post-rejection submission metrics (O = original, S = supplementary).

Table 5.9: Correlation between original and post-rejection submission metrics.

Submission Type	Core	Firefox	Thunderbird
Review request	*** 0.08	*** 0.13	0.07
Commit	*** 0.09	0.02	-0.10

The time needed to submit a post-rejection change is weakly correlated with latent time.

5.3.4 RQ3.4: Is time to post-rejection submission correlated with time to original submission?

Figure 5.7 characterizes the distribution of original change submission times and supplementary submission times. The overall trend is that supplementary submissions take less time than the respective original submissions. Furthermore, time to supplementary submission has larger dispersion.

Table 5.9 shows the correlation between the original and post-rejection submission of changes that were rejected, analyzing separately review requests and commits. The correlation is very weak at most, with the maximum statistically significant correlation being 0.13.

The time needed to submit an inappropriate change has little influence on the time needed to submit the subsequent post-rejection change.

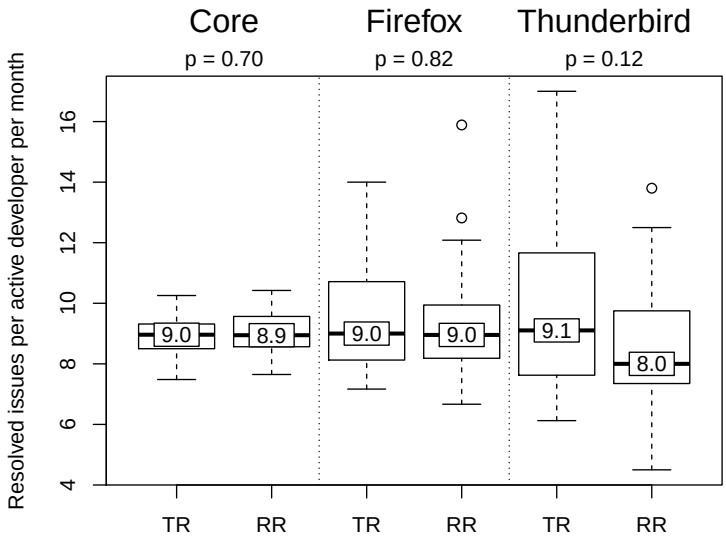

Figure 5.8: Developer workload for traditional and rapid releases.

5.4 RG4: ASSESS THE IMPACTS OF PROCESS CHANGES ON REJECTIONS

In order to better understand the impacts of Mozilla's process changes, we measure rejection-related metrics under two periods, from June 29, 2009 to March 21, 2011 and from June 20, 2011 to September 16, 2013. These periods are referred to as traditional releases (TR) and rapid releases (RR), respectively.

5.4.1 RQ4.1: How has the developer workload changed under the new process?

Workload is a confounding factor that can influence the rework rate. Supposedly, developers subject to a heavier workload are more likely to write inappropriate changes that ultimately lead to rework.

We measure developer workload as the average number of issues resolved with a commit by each active developer per unit of time. We consider a developer active in a specific month if he or she contributed with commits for at least three issues. We experimented other numbers of commits, with similar results regarding the variation of the average workload.

Figure 5.8 shows the distribution of monthly workloads for each project, under two periods: traditional releases (TR) and rapid releases (RR). There is no statistically significant difference in workload between the two periods (at the 0.05 level). Presented with the results, a Mozilla engineer stated that "a developer can only do so much work; growth is mostly adding developers nowadays, not the individual doing more".

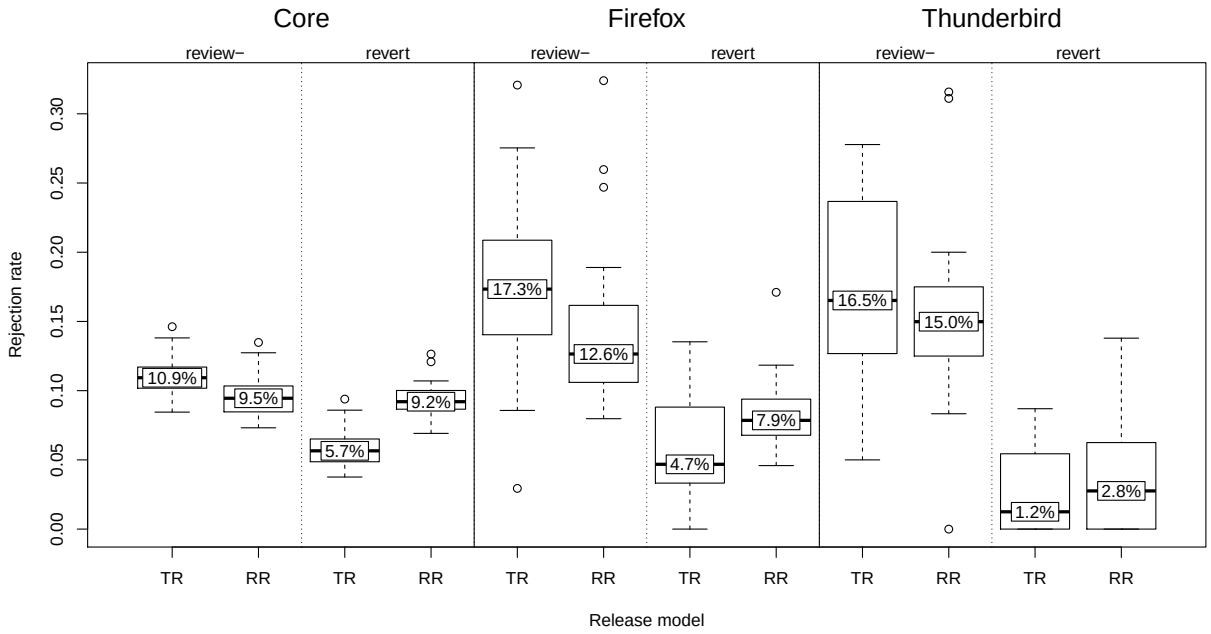

Figure 5.9: Distribution of monthly rejection rates for each project, rejection type, and period.

Table 5.10: Ratio between rejection rate for rapid and traditional releases.

Rejection Type	Core	Firefox	Thunderbird
Negative review	*** 0.86	*** 0.61	0.99
Revert	*** 1.62	** 1.35	1.80
Any rejection	** 1.10	*** 0.78	1.06

The workload did not significantly change under rapid releases.

5.4.2 RQ4.2: How has the rejection rate changed under the new process?

Figure 5.9 shows the monthly distribution of the rejection rate, for both negative reviews and reverts, under both traditional and rapid releases, for each project. Table 5.10 shows the ratio between rejection rates, i.e., the rejection rate under rapid releases divided by the corresponding rejection rate under traditional releases, for each rejection type.

There is no significant difference between traditional and rapid releases for Thunderbird. For Core and Firefox, however, there are two trends: the negative review rate became lower under rapid releases, while the revert rate became higher. The overall rejection rate raised in Core, dropped in Firefox, and remained stable in Thunderbird.

Under rapid releases, reverts increased and negative reviews decreased.

Table 5.11: Ratio between rejection rate for rapid releases and rejection rate for traditional releases.

Rejection Type	Core	Firefox	Thunderbird
Early revert	*** 2.72	*** 3.30	0.61
Late revert	*** 0.53	*** 0.37	* 2.60

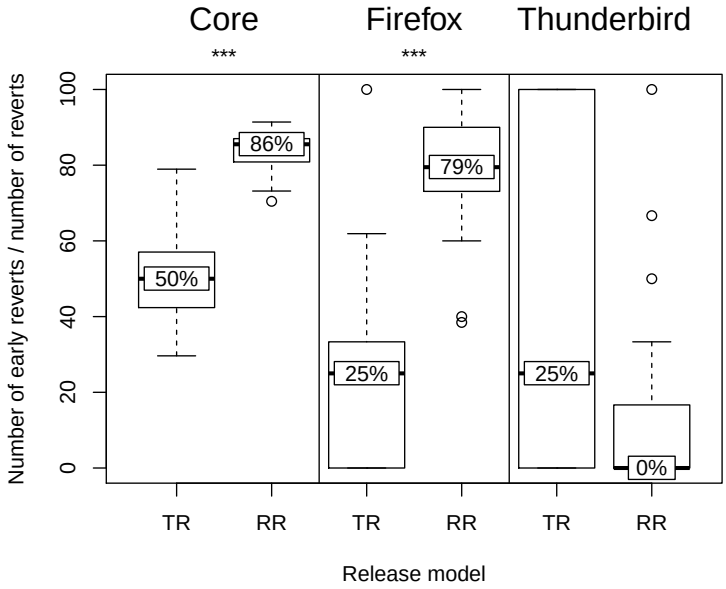

Figure 5.10: Proportion of early reverts under traditional releases (TR) and rapid releases (RR). Asterisks represent p-values: *** $\Rightarrow p < 0.001$; ** $\Rightarrow p < 0.01$; * $\Rightarrow p < 0.05$.

5.4.3 RQ4.3: How has the early revert rate changed under the new process?

In the previous section, we showed that the revert rate increased under rapid releases for Core and Firefox. In this section, we analyze separately early and late reverts.

Table 5.11 shows, for both early and late reverts, the ratio between the revert rate for rapid releases and the revert rate for traditional releases. It can be seen that the early revert rate had approximately a three-fold increase for both Core and Firefox, while late revert dropped to half or less. The numbers for Thunderbird are not reliable, since its developers do not systematically report commit reverts.

Figure 5.10 shows that the proportion of all reverts that are early reverts raised from 50% or less to about 80% for all projects (except Thunderbird). Therefore there is a trend towards earlier problem detection.

Under rapid releases, early reverts increased and late reverts reduced.

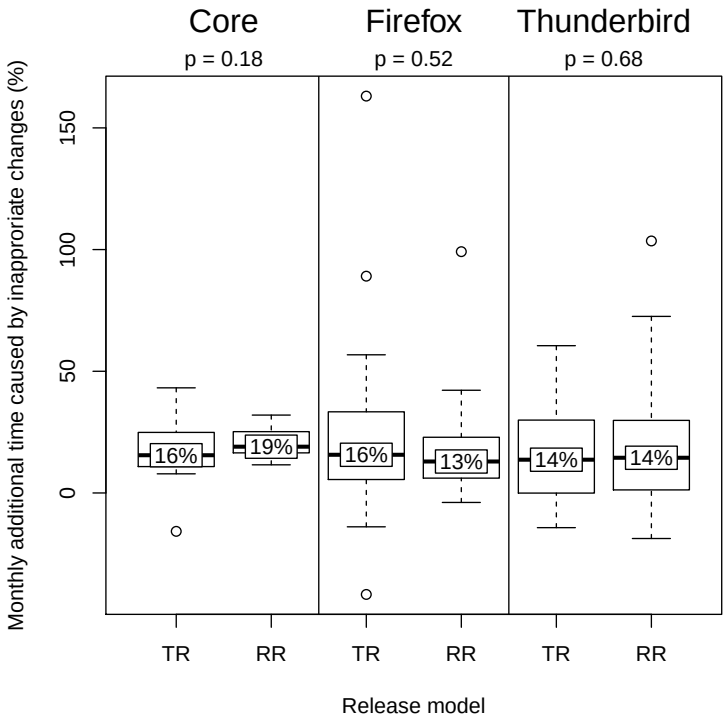

Figure 5.11: Total additional time under both traditional releases (TR) and rapid releases (RR).

5.4.4 RQ4.4: How has the total additional time caused by inappropriate changes varied under the new process?

In the previous research questions we investigated whether, under rapid releases, developers are writing more inappropriate changes. In this question, we investigate whether, under rapid releases, inappropriate changes cause more additional time. Additional time refers to the difference between the sum of the lifetimes of issues with and without inappropriate changes.

Figure 5.11 compares the distribution of the total additional time caused by inappropriate changes, computed for each month under traditional and rapid releases. There is no statistically significant difference between both periods, therefore no evidence that the additional time either increased or decreased.

> The additional time caused by inappropriate changes has not significantly changed under rapid releases.

Chapter 6

DISCUSSION

In the previous chapter, we presented quantitative results related to the proposed research questions. In this chapter, we interpret the results and point to their potential impacts.

6.1 RG1: PROPOSE AND COMPARE TECHNIQUES TO DETECT CHANGE REJECTION

Previous studies used supplementary commits (PARK et al., 2012) and issue reopening (SHIHAB et al., 2010) to detect inappropriate changes. A recent study compared these two strategies and found that they detect two distinct sets of issues with an intersection of less than 50% (AN; KHOMH; ADAMS, 2014).

We argued that looking for supplementary commits or issue reopening are imprecise strategies to detect inappropriate changes, and proposed a new strategy, based on the detection of commits that revert other commits. We found that most issues containing reverts also contain supplementary commits, but the opposite is not true: most issues that contain supplementary commits do not contain a revert.

Two interpretations are possible: either supplementary commits detect inappropriate changes not detected by reverts, or they find many false positives. To investigate this question, we read a sample of commits from Core for issues containing supplementary commits but no reverts.

Figure 6.1 presents the two commits associated with issue 665858. The two commits are a few seconds apart, and are annotated with the expressions "Part 1" and "Part 2". Clearly, this is not a case of inappropriate change; instead, it can be inferred that the developer chose to break the change in two commits. In fact, a quick analysis of commits for issues with supplementary commits but no reverts showed that 42.1% of those commits contain the word "part", suggesting that they are multi-commit changes.

About 3.7% of supplementary commits contain the word "follow-up", suggesting that they fix some aspect of the original commit. This is the case of issue 671029, shown in Figure 6.2. The supplementary commit however, performs only some code cleaning to

```
changeset:  58655e365c91
user:       ehsan
date:       2011-06-27 12:58:43
summary:    Bug 665858 - Part 1: Optimize the conversion of native ...

changeset:  d7ed8936e9f8
user:       ehsan
date:       2011-06-27 12:59:01
summary:    Bug 665858 - Part 2: Optimize nsContentEventHandler::...
```

Figure 6.1: Two-part commit.

```
changeset:  102481f5e2b9
user:       pbiggar
date:       2011-07-18 21:14:33
summary:    Bug 671029: Ignore Byte-Order-Mark in UTF-8 files ...

changeset:  52e36db1e8c7
user:       pbiggar
date:       2011-07-18 21:32:47
summary:    Bug 671029 (followup): Remove unused size parameter and
            uninitialized var warning (rs=jwalden)
```

Figure 6.2: Minor follow-up commit.

remove compiler warnings, suggesting that the original commit was not actually inappropriate, just in need of a small maintainability improvement.

The examples show that the occurrence of supplementary commits for an issue often do not imply that the original commit was inappropriate, unless preceded by a revert. Given the low precision of the technique, we recommend using it only when followed by manual inspection of commits.

When a developer reopens an issue, he is signaling that he thinks there is a problem with its solution. Using reopening to detect inappropriate changes, however, has two problems. First, this strategy yields a low recall, since it fails to detect inappropriate changes found before the issue report is closed. For Mozilla projects, an issue is only closed after it passes automated testing; therefore, inappropriate changes that are found during or before automated testing do not trigger reopening, since at this point the issue is not closed yet.

Second, the strategy is also imprecise, since many reopenings are not followed by

supplementary commits, as shown by Le An et al. (2014). Our results reinforce their observation, showing that most reopenings not followed by supplementary commits are also not followed by reverts. Such reopenings are said premature: often, they signal that a developer thought that the change was inappropriate, but after some discussion the team decided to keep the change.

There are many reasons to disagree with a reopening. For instance, in issue 665964, developers determined that the problem that led to reopening was not directly related to the issue, and because of that they created a new issue report. In issue 670003, a developer closed the issue before his change was merged into Mozilla-Central; because of that, another developer reopened the issue and waited for Mozilla-Central's build results before closing it again. In issue 670072, a user reopened the issue because his problem was not resolved in Firefox 6.0; a developer closed it the same day, though, because the change was scheduled for Firefox 8.0. In issue 675976, it appears that a developer reopened the issue by mistake. In summary, many reopenings are not related to inappropriate changes.

Reverts seem to better capture the notion of change rejection. Differently from reopening, a revert signals that there is a problem with a specific change. Differently from supplementary commits, that can be part of a multi-commit change, reverts show the intent of undoing the original change.

There are two problems with reverts, though. First, not all projects follow the practice of reverting inappropriate commits: some projects fix problems with follow-up commits. Second, it is hard to detect reverts unless revert messages explicitly state that they are reverting a specific issue or commit. Even within the Mozilla Foundation, which recommends reverting inappropriate commits and proposes a convention for writing commit messages, there are projects for which the proportion of reverts is unusually low, e.g., Thunderbird.

A few reverts are not associated with supplementary commits, and those reverts may or may not signal an inappropriate change. For instance, in issue 669236, the change was reverted because the issue was resolved in a broader way in the context of another issue. In issue 685258, the reverted change resolved a problem in a module that was later replaced by an alternative module. In issue 698986, the change was partially reverted in order to resolve another issue. It would be hard to determine whether those reverts signal an inappropriate change without reading the comments in the related issue.

In projects that systematically revert inappropriate changes and that follow conventions to identify reverts, reverts are probably the most reliable indicator of inappropriate change. Thus, we recommend studying a project's adherence to those practices before using reverts to detect inappropriate changes.

Negative reviews are another indicator of inappropriate changes. They are complementary to reverts, because they occur before the change is even committed. Also, they capture a different notion of inappropriateness: often, a change is not negatively reviewed for containing defects or for not resolving the issue, but for not being maintainable enough, or for integrating poorly with other modules[1].

We found that a significant proportion of the changes that were negatively reviewed

[1]For more information, see ⟨https://developer.mozilla.org/en/docs/Code_Review_FAQ⟩

led to improved changes that were rejected during testing. This result suggests that even rigorous code reviews, which lead to a negative review, often fail to detect problems that are discovered later. This observation is consistent with recent findings showing that "code reviews often do not find functionality issues that should block a code submission" (CZERWONKA; GREILER; TILFORD, 2015).

6.2 RG2: QUANTIFY REWORK TRIGGERED BY INAPPROPRIATE CHANGES

There is no agreement in the literature about the proportion of issues that contain inappropriate changes. One reason for this discrepancy is that each study adopts its own definition of change rejection, be it a negative review, a supplementary change, or a reopening. Reported rejection rates range from as low as than 1.3% (ALMOSSAWI, 2012) up to 33% (PARK et al., 2012). In this study, we performed our own analyses to answer the question: are inappropriate changes rare, irrelevant events, or are they frequent enough to be worth studying? Also, what is their impact on issues' lifetimes?

Our results indicate that inappropriate changes are common during software development. They are present in about 18% of all issues. In some projects, changes are rejected multiple times a day, on average.

The literature reports that issues associated with rejections have a lifetime 50% (PARK et al., 2012) to 100% (SHIHAB et al., 2010) longer than those without rejections. Our results suggest lifetimes 76.6% to 130% longer. Again, the results cannot be directly compared because of the multiple definitions of rejection adopted across studies.

What if we could completely eliminate the creation of inappropriate changes? How much of development time, including programming, reviewing, and testing, could be saved in this scenario? Our results point to an additional time of up to 21.6% caused by inappropriate changes, which means that eliminating inappropriate changes could save no more than 17.8% of the development time[2].

It should be noted that only part of the additional time represents rework overhead, since only a fraction of the extra time associated with inappropriate changes is spent with actual work (creating, submitting, reviewing, and testing changes); the other part is idle time. Furthermore, the additional time is an upper bound on the amount of time that could be saved in a project if all changes were appropriate. This is because some of the extra time used in supplementary submissions may be due to new source code that should have been added in the first submission; therefore, the time needed to write the new code would have been spent anyway and thus could not be saved. In a future work, one may try to determine how much of a supplementary submission consists of new code and how much consists of modifications to the code written for the original submission.

Of course it is virtually impossible to completely eliminate inappropriate changes, therefore projects should deal with them. There are two approaches to deal with inappropriate changes: *prevention* and *appraisal* (SLAUGHTER; HARTER; KRISHNAN, 1998). Prevention consists of initiatives to prevent problems before changes are even written, thus avoiding rework. One such initiative is training developers. A direct in-

[2]To convert the percentage increase reported in the previous chapter to a percentage decrease, we used the function $f(x) = x/(1 + x)$, where x is the percentage increase, in decimal format.

terpretation of the numbers suggests that investing more than 17.8% of developers' time in training aimed at preventing inappropriate changes is not effective in the short term, since in this case the costs outweigh the potential benefits. Of course, training has long term benefits that should also be considered.

Appraisal, in contrast, consists of initiatives aimed at detecting problems after changes are written and before they reach end users. This approach includes code reviews and testing. The objective, in this case, is not to avoid rework, but to improve the quality of released products.

In this section, we discussed the significance of inappropriate changes in terms of their impacts on additional time, which includes rework. In the next section, we discuss other questions related to inappropriate changes, including whether appraisal initiatives were effective in preventing problems from reaching end users of Mozilla's products.

6.3 RG3: EMPIRICALLY VALIDATE HYPOTHESES ABOUT REWORK

Code reviewing and testing aim at detecting problems in a product before it reaches end users. In a rapid release schedule, the time available for such activities is restricted. At Mozilla, the time between a change submission and its release ranges from 12 weeks (when a change is submitted in the end of a release cycle) to 18 weeks (when it is submitted in the beginning of the cycle), unless a problem is detected and there is not enough time to fix it within a release cycle.

Although inappropriate changes are common, current practices do a good job detecting them before they reach end users. In Core and Firefox, less than 3% of all rejections occur after 12 weeks of the respective change submission. Therefore, although inappropriate changes harm a project's productivity, it seems that they have little impact on a product's quality as perceived by end users. This impression is consistent with a Mozilla engineer's view:

> I think our development process gives us a margin of safety to detect regressions well before the code actually reaches the hands of users. As the user base of each repository grows gradually, we have an effective way to detect unexpected problems well in advance.

The engineer points out that, during the 12 weeks in which the product is being stabilized, it is also being used by early adopters that opted-in to alpha and beta versions. Those users are more likely to experience problems resulting from inappropriate changes, but they also contribute to finding those problems before a final version is released.

Although testing hypotheses about causes for inappropriate changes is not the focus of this study, the metrics we computed for other research questions allows us to test one hypothesis. The hypothesis is that, when faced with tight deadlines, developers write changes more quickly and with less care, and those changes are more likely to be inappropriate and, consequently, to be rejected. In contrast, a good, carefully written change would take comparatively more time to be submitted.

Comparing the distribution of the time to submit both appropriate and inappropriate changes, however, we found surprising results. Inappropriate changes actually tend to

take more time to be submitted. It is counterintuitive to think that careless programming requires more time than careful programming within the context of a single issue.

Our explanation is that there is a confounding factor that contributes to longer change submission times and also raises the likelihood of inappropriate changes. That factor is *issue difficulty*. Difficult issues are harder to get right the first time, and are thus more likely to receive an inappropriate change. Difficult issues also require that developers spend more time figuring out a solution.

The issue difficulty hypothesis sounds plausible, but we could not verify it, since we cannot directly measure issue difficulty. Also, the hypothesis does not help preventing inappropriate changes, since developers have no direct control of a product's issues and their difficulty.

If inappropriate changes cannot be completely avoided, one can at least try to reduce their impacts on development time, i.e., decrease the time needed to submit a change after a rejection. A Mozilla engineer hypothesized that a long latent time induces overhead on the developer who wrote the original change, since it is harder for him to remember the issue context. As a result, it takes more time to write and submit the post-rejection change. If the hypothesis is true, it means that, if code reviewing and testing were performed earlier in the process, less time would be spent in post-rejection changes.

We indeed found a correlation between latent time and time to post-rejection submission, although a weak one. This result suggests that there are other factors that are more important to determine post-rejection submission time.

We hypothesized that the time spent on the original submission was one of such factors. However, we found a very weak correlation, which suggests that the original and post-rejection changes are different in nature, and the effort needed to write the former has little influence on the effort needed to write the latter.

6.4 RG4: ASSESS THE IMPACTS OF PROCESS CHANGES ON REJECTIONS

Mozilla shortened the release cycle of its products in order to deliver new features to its users at a faster pace and gain market share. Although the benefits of such approach are clear, the drawbacks are not so well known, especially regarding the impacts on the development process.

A debatable question is whether rapid releases lead to rushed changes, that ultimately lead to low quality. On the one hand, developers may feel pressured to finish a feature within a single release cycle, leaving less time for testing and polishing. On the other hand, developers may feel comfortable to spend the time they need to finish a feature, because if they miss a release date, the next release cycle is no more than six weeks away.

Either way, the effect of rapid releases can be observed by studying the variation of inappropriate changes before and after the adoption of rapid releases. That being said, it is not feasible to evaluate the impact of rapid releases in isolation, since their adoption came together with other changes in the process, such as sheriff-managed integration repositories and overall improvements. For this reason, we study, instead, the impact of Mozilla's process changes—including rapid releases and sheriff-managed integration repositories—on the rejection rate of its products.

It would not be fair to analyze the variation of rejection rate if the developer workload varied over time, since a higher workload could lead to more inappropriate changes and higher rejection rate. By analyzing commit logs, we determined that there was not a significant variation on developer workload, and thus this is not a factor that influences our results.

We found that the revert rate significantly increased under rapid releases. In part, this difference could be attributed to the absence of commit hooks before 2011, making it harder for us to detect reverts from commit messages in this period. To mitigate this threat, when detecting reverts from commit messages, we included variations of the term "backout" that would not be accepted by the commit hook but were found when we read a sample of the commit messages.

In a naive interpretation, the increasing revert rate supports the theory that rapid releases lead to hurry and inappropriate changes. To better understand the results, though, we showed the rejection rates under both traditional and rapid releases to Firefox engineers and asked them to explain the difference. Their explanations included Mozilla's growth, the improvement of testing tools, and the adoption of integration repositories, as detailed next.

A larger code base and more products. Some engineers explained the increase in the overall revert rate by suggesting that because the code base grew over time, code conflicts became more likely. The number of supported platforms also increased because Firefox must support both new platforms, such as Windows 8, and older ones, such as Windows XP. Also, new products emerged, such as Firefox for Android and Firefox OS, that share code with the desktop Web browser. As one engineer explained,

> We have a lot more stuff that can break, on more platforms, as well as more tests—these days we don't have everyone working on just Firefox. Code landing for B2G [the Firefox OS] can break Fennec [Firefox for Android], for example, and B2G devs don't build and test on Fennec locally. Those kinds of changes will be caught and backed out [reverted] when they hit the trees [code repositories], not found beforehand.

The evolution of testing tools. Another engineer explained that the emergence of better testing tools promoted earlier detection of problems and improved the detection of problems that would have otherwise gone unnoticed, such as hard-to-detect memory leaks:

> Our automated testing has improved considerably since [release] 3.5. A number of memory-leak finding tools have been integrated into our test environments that are improving our early catch rate.

Integration repositories and revert culture. According to Firefox engineers, the increasing revert rate was also due to the sheriff-managed integration repositories and their effect on how developers test their code. Before 2011, because developers pushed changes directly to Mozilla-Central, the changes had to be thoroughly tested to avoid breaking the builds or introducing bugs. Since 2011, developers started to commit to

integration repositories, and the sheriff reverted problematic changes before merging them to Mozilla-Central, thus keeping it stable. So, developers were encouraged to commit to Mozilla-Inbound after having performed less testing. As someone stated in Mozilla's wiki,

> But breaking it [Mozilla-Inbound] rarely is ok. (...) Never breaking the tree [code repository] means you're running too many tests before landing [committing to the repository].[3]

Two Mozilla engineers reinforced this view:

> In the "old days," you were expected to have built, tested, done a Try build, etc. before the patch landed.
>
> The backout [revert] aggressiveness was even explicitly mentioned when we switched.

We decided to further investigate the hypothesis that integration repositories and the revert culture caused the increase in rejection rate. To this end, we split reverts into two groups, early reverts and late reverts, and measured them separately. We discovered that the early revert rate increased, while the late revert rate decreased. This result suggests that problems that used to be detected during a Try build and, thus, before change submission, are now detected during Mozilla-Inbound's build, leading to early reverts. To further interpret this result, we analyze the impact of the changes in revert rates on both developers and users.

Impact on developers. Every revert induces rework by requiring development of a new, improved change. However, in Mozilla's case, the increase of early reverts did not seem to induce significant overhead. Instead, it reflected a cultural shift toward committing changes before testing them comprehensively, therefore reducing the effort required to test changes. Such change was possible only because broken changes no longer reached Mozilla-Central. Sheriffs also ensured that changes that break the build were reverted as soon as possible, reducing the time in which the repository must be closed:

> I'd say amount of time spent testing patches before landing [pushing changes] and amount of time wasted with trees [repositories] closed due to bustage [a broken build] were [both] reduced.

Although all reverts induce rework, late reverts are severer. Problems that are not resolved early are more likely to end up in a release. So, users might have to wait another release cycle to receive the definitive change. Also, with integration repositories, inappropriate commits that were not reverted early end up in Mozilla-Central, on which developers base their work. By the time the commit is reverted, many other commits might have depended on it. Therefore the shift toward earlier reverts suggests that the sheriff-managed integration branches reduced the effort required to integrate changes.

[3] "Tree Rules/Integration," available at ⟨https://wiki.mozilla.org/Tree_Rules/Integration⟩.

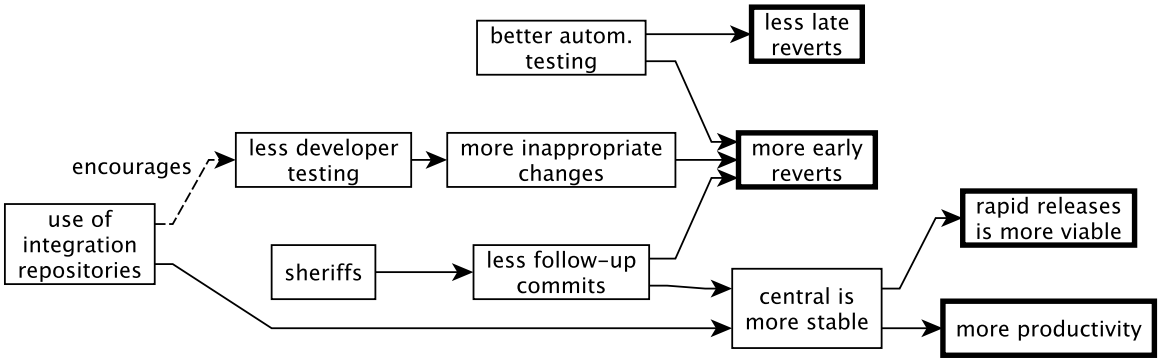

Figure 6.3: Forces contributing to variations in early and late revert rate.

Impact on users. Although Mozilla's move to rapid releases was a success from the release-engineering perspective, it upset users because of frequent update notifications and broken plug-in compatibility. As the then chair of Mozilla Foundation summarized on her blog post,

> We focused well on being able to deliver user and developer benefits on a much faster pace. But we didn't focus so effectively on making sure all aspects of the product and ecosystem were ready.[4]

However, nowadays inappropriate changes have almost no effect on users' perception of quality. This is because, after being committed to Mozilla-Central, all changes go through two other repositories, Mozilla-Aurora and Mozilla-Beta, where more tests occur during two release cycles before they are released to the general public. So, only very late reverts affect users, and these are rare under both traditional and rapid releases.

Figure 6.3 summarizes our understanding of the effects of the changes introduced by Mozilla in 2011 on revert rate and related aspects. Three factors explain the increase of early reverts. The adoption of *integration repositories* encourages developers to test less their changes, leading to more inappropriate changes and more early reverts, and also keeps the central repository from breaking often, which helps meeting the rapid release schedule. Also, *sheriffs* prevent follow-up commits by reverting commits right away, resulting in more early reverts and also keeping the central repository from breaking often. Finally, better *automated testing* tools anticipate problems that would otherwise be found during manual testing, reducing the late revert rate and increasing the early revert rate.

6.5 GENERAL CONSIDERATIONS

In this section we discuss two topics that are not directly related to a specific research goal. First, we discuss the relevance of this study to both open source and proprietary software, and then we discuss the role of rejection and rework within software projects.

[4]M. Baker, "Rapid Release Follow-Up," blog, 3 Oct. 2011; ⟨http://blog.lizardwrangler.com/?p=2996⟩.

6.5.1 Open Source and Proprietary Software

In this study, we analyze open source projects from a single organization, the Mozilla Foundation. Although the projects accept contributions from the community, most developers in those projects are Mozilla employees which work full-time on the projects.

We believe that this study is also relevant to closed-source, proprietary software projects, since those projects can also use practices such as code review and automated testing. Furthermore, proprietary projects are increasingly adopting practices typically associated with open source projects (KALLIAMVAKOU et al., 2015).

The key aspects to replicating this study to a project are (i) the project's adherence to a process that includes the evaluation of changes, and (ii) the availability of high quality data about the submission and rejection of changes. Those aspects are more important than whether a project is open source or not. In this sense, projects with less structured processes are less suitable for this study, even if they are open source.

6.5.2 What If All Rework Could Be Eliminated?

What would happen if all changes were appropriate since the first submission? At first, the productivity would raise, since the time otherwise wasted with rework could be used to implement new features. Also, code reviews and testing would become superfluous, since there would not be problems to be detected.

We argue that, even in this hypothetical perfect world, it is not desirable to get rid of code reviews and testing. Besides detecting problems, code reviews are a mechanism to share knowledge about the source code and best practices (BACCHELLI; BIRD, 2013). They increase the socialization within a project and allow continuous evaluation of the architecture and other code-related decisions. Furthermore, there is evidence that test-driven development contributes to improve software design quality (JANZEN; SAIEDIAN, 2008).

In summary, rejections are not entirely bad. While, on the one hand, a rejection reveals that there are problems in a source code change, on the other hand it shows that the project uses practices that raise the knowledge sharing, the socialization, and the reasoning about design within a project.

6.6 LESSONS FOR PRACTITIONERS

This study provides lessons for practitioners who aim to reduce rework or release more often, as described in the next two subsections.

6.6.1 Reduce Rework by Assessing Process Changes

Reducing rework in a project may require introducing practices or improving the process. While this study does not offer specific advice to reduce rework, it can support continuous improvement of software processes by providing new metrics to assess process quality. In order to get more reliable metrics, it may be necessary to enforce conventions on how to record important events in a change's lifecycle (e.g., using commit hooks).

We suggest that project managers continuously measure rejection rate and introduce practices and process improvements one at a time. After each introduction, they should assess whether the rejection rate varied. Significant increases in the rejection rate may signal problems in the process.

6.6.2 Moving Fast Without Breaking Things

In software-intensive markets, high competition pushes organizations to release new features at a faster pace. One main concern is to avoid that an initial gain in release speed results in process instability, contributing to poor quality and productivity loss in the long term. By computing metrics and talking to Mozilla engineers, we were able to identify two concrete measures taken by Mozilla that helped in keeping the process stable while allowing it to move faster:

- **Improve automated testing tools**. Automated testing is used to detect problems very early in the process. The earlier problems are detected, the faster it is to fix them and the earlier the fixes can be delivered to end users.
- **Use integration repositories**. It is ok to break the product more often, as long as it does not affect other developers' work. Integration repositories allow developers to focus on writing code, while sheriffs ensure that low-quality code is filtered out as soon as possible, before reaching the main source code repository.

While these two measures can be used to improve any project, the overhead involved in putting them into practice is more justifiable under rapid releases, since in this context it is important to keep the source code stable as often as possible. Having frequently stable code is also important when Mozilla has to deliver a "chemspill" release, i.e., a release that fixes critical security issues and, thus, should reach users as soon as possible.

Chapter 7

CONCLUSION

In this thesis, we sought to better understand change rejection and the resulting rework in software projects. We investigated the impacts of inappropriate changes and the variation in rework that resulted from a significant process change in projects developed by the Mozilla Foundation. To support our goals, we proposed and evaluated techniques to detect change rejection from issue reports and source code commits collected from a project's history.

We discovered that existing techniques to detect change rejection are imprecise and generally do not agree with each other. We proposed a new technique to detect change rejection in projects that adopt continuous integration, more specifically the practice of reverting inappropriate commits. Although further analysis is needed, this technique appears to be more precise than its alternatives.

We determined that inappropriate changes are common, appearing in almost one fifth of the issues that require source code changes. Because of the rework they induce, they also contribute to increasing the lifetime of the issues in a project.

While inappropriate changes and rework cannot be completely avoided, it is possible to prevent most of those changes from reaching end users by performing verification tasks. We showed that the process adopted by Mozilla, with code reviews, daily automated testing, manual testing, and alpha and beta releases, prevents about 97% of the inappropriate changes from reaching end users, even under short release cycles.

We observed that changes take more time to be submitted when they are inappropriate, suggesting an association between issue difficulty, time to submission, and inappropriateness. We also showed that the time to submit an inappropriate change and the time to reject it have little influence on the time required to write and submit the subsequent change.

Process changes potentially affect inappropriate changes, rejection, and rework, for better or for worse. We showed that, after Mozilla adopted rapid releases, a higher proportion of commits were reverted. A naive interpretation would be that the tighter schedule led developers to write more inappropriate changes. Upon further analysis, we

determined that, under rapid releases, most of the reverts occur early in the process. This shift to earlier rejection is likely the result of improvements in testing tools and of a recommendation for developers to spend less time testing with private builds, since, under the new process, a change is only pushed to the central source code repository after a successful build. Furthermore, we could not find evidence that inappropriate changes result in more additional time under rapid releases than under traditional releases.

7.1 MAIN CONTRIBUTIONS

We believe that this study contributes both to scientific literature and to software development practice in the following ways: (i) it proposes a new technique to detect inappropriate changes and compares it with existing techniques; (ii) it quantifies the impact of inappropriate changes on rework and product quality; and (iii) it shows how inappropriate changes were impacted by process changes, specifically, rapid releases and integration repositories.

Preventing and detecting inappropriate changes before they are integrated into a final release are important problems in software engineering. In the research approach known as software repository mining, software artifacts are analyzed to better understand causes and effects of inappropriate changes. To achieve this goal, however, it is necessary to detect inappropriate changes from historical data. Our results can be used to improve those studies and to better assess threats to validity of previous studies that rely on the detection of inappropriate changes.

While previous studies have measured the impacts of inappropriate changes using a single rejection type—either negative reviews, supplementary changes, or issue reopening—our study combines all rejection types in the same analysis. Also, we estimate the additional time needed to resolve issues in a project because of inappropriate changes, for each rejection type. Knowing in detail the impacts associated with inappropriate changes supports informed decisions on improving the software development process.

Finally, our study helps evaluate the impacts of rapid releases in projects of worldwide importance. The results should be useful for any organization that intends to adopt rapid releases.

7.2 FUTURE WORK

Although we believe reverts are the best currently available indicator of inappropriate changes (for projects in which reverting them is common practice), we did not gather enough quantitative evidence to support this belief. In a future work, in order to evaluate reverts, we may classify each issue in a sample according to whether it is associated with inappropriate changes. The classification would be inferred by reading issue reports and commit logs, and dubious cases would be discussed with developers and other researchers. This effort would lead to an oracle for inappropriate changes, which could be used to evaluate existing detection techniques and develop improved techniques, with higher precision and recall.

A better technique to detect inappropriate changes can lead to improved empirical

studies based on mining software repositories. For instance, one can compare the source code of inappropriate changes with that of appropriate changes to identify distinctive features. Another option is to measure the proportion of inappropriate changes within each module of a software system to discover which modules are more fragile, i.e., more likely to break the software when changed.

Some results in this thesis were not validated by developers. We may in the future ask developers to interpret those results, using qualitative methods such as semi-structured interviews and the Delphi method (DALKEY; HELMER, 1963). This approach would contribute to reduce biases in the interpretation of quantitative results.

While comparing rejection types, we observed that, compared to changes that are positively reviewed, changes that are negatively reviewed are more likely to lead to testing failures. We could not find a reasonable explanation for this counterintuitive result. One possible future work is to better understand the relationship between code reviews and automated testing, two methods for early problem detection.

In this study we evaluated the impact of rapid releases and integration repositories on change rejection. Measuring the variation of change rejection can be used to evaluate other design and process aspects of software development, such as the adoption of test driven development, distributed version control, or a microservices architecture.

For instance, one may hypothesize that modules that violate the prescribed architecture for a system are more fragile, so changes in those modules are more likely to break the software. One future work that we started to pursue is to test this hypothesis by comparing the proportion of inappropriate changes in modules that violate the architecture and modules that conform to it.

In brief, with a more precise technique to detect inappropriate changes, such as the one introduced in this work, researchers can better understand source code changes, a centerpiece of software development. They can identify software modules and source code features associated with rejection-prone changes, and also understand how process changes can affect the rate at which inappropriate changes are produced and how it affects rework and product quality.

Appendix

A

EMAILS EXCHANGED WITH MOZILLA ENGINEERS

This appendix contains emails exchanged with Mozilla engineers about preliminary results of our research. To protect their privacy, we replaced their names with numbers.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

Bug reopening and short release cycles

8 messages

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Mon, Oct 28, 2013 at 2:09 PM
To: firefox-dev@mozilla.org

I'm studying bug reopening as part of my PhD thesis at Federal University of Bahia, Brazil, and I'd like to get feedback from the community about some interesting results I've found so far using Firefox's data.
I've compared Firefox's bug reports before and after the project adopted 6-week release cycles. I've found, for example, that...

* the average time to resolve a bug after it has been reopened dropped from 77 days (before) to 32 days (after);
* the reopening rate (% of fixed bugs that are eventually reopened) dropped from 7.4% (before) to 4.5% (after).

Two 2-year periods were considered: before = 2009-03-22 up to 2011-03-22, and after = 2011-06-21 up to 2013-06-21.

The first result seems intuitive: faster releases imply bugs are fixed faster. However, I'm not sure about why the reopening rate would drop. Do you have any theories?

[]s
Rodrigo

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Mon, Oct 28, 2013 at 8:37 PM
To: dev-planning@lists.mozilla.org

[Quoted text hidden]
[Quoted text hidden]
The first result seems intuitive: faster releases imply bugs are fixed faster. However, I'm not sure about why the reopening rate would drop. Does anyone have any theories?

[]s
Rodrigo

Developer 1 Mon, Oct 28, 2013 at 8:44 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: dev-planning@lists.mozilla.org

On Monday 2013-10-28 21:37 -0200, Rodrigo Rocha Gomes e Souza wrote:
> The first result seems intuitive: faster releases imply bugs are fixed
> faster. However, I'm not sure about why the reopening rate would drop. Does
> anyone have any theories?

Bug reopening rates can vary substantially as a result of changes in culture: in particular, when it is considered correct to reopen an existing bug report vs. filing a new bug report. I'm skeptical of the value of any analysis of reopening rate that doesn't somehow account for this.

Developer 2 Mon, Oct 28, 2013 at 8:51 PM
To: dev-planning@lists.mozilla.org

On 10/28/13, 4:37 PM, Rodrigo Rocha Gomes e Souza wrote:
> The first result seems intuitive: faster releases imply bugs are fixed
> faster. However, I'm not sure about why the reopening rate would drop. Does
> anyone have any theories?

Reopening a bug report suggests that the bug was not fixed (entirely). A lower reopening rate might mean that users are just filing new bug reports instead of reopening existing bug reports. New users might not know about old bug reports.

Perhaps, Mozilla developers are now better at identifying and fixing the root cause for bugs. :)

You might also compare the bug states before and after reopening. Some users will reopen bugs that were closed

"RESOLVED WONTFIX" or "RESOLVED WORKSFORME" because they disagree with the resolution.

Developer 3 Mon, Oct 28, 2013 at 10:30 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: "dev. planning" <dev-planning@lists.mozilla.org>

The change in reopening rates is probably a result of us switching from checking stuff in to mozilla-central to checking it in on mozilla-inbound. When we landed stuff on central directly bugs would be closed as RESOLVED FIXED when the patches landed, but if they broke the build or tests the patches would get backed out and the bug would be REOPENED. Now we land stuff on mozilla-inbound, but still don't mark bugs as RESOLVED FIXED until mozilla-inbound gets merged to mozilla-central. And we only merge changesets that pass all the tests, so stuff now "bounces" without ever resolving (and hence reopening) the bug.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Tue, Oct 29, 2013 at 10:45 AM
To: Developer 3
Cc: "dev. planning" <dev-planning@lists.mozilla.org>

Thank you all! The results are preliminary and I'm going to refine the analyses with your feedback.

Regarding the bug status, I only counted bugs that were RESOLVED FIXED when they were reopened, so disagreements about the resolution are unlikely.
The understanding about "reopen an existing bug report vs. filing a new bug report" can be a source of disagreement, as illustrated by the discussion at https://groups.google.com/forum/#!topic/mozilla.dev.platform/UnxndrlUIL4. I'm not convinced, though, that the numbers could be explained by a cultural change.

On Mon, Oct 28, 2013 at 10:30 PM, Developer 3 wrote:

> The change in reopening rates is probably a result of us switching from checking stuff in to mozilla-central to checking it in on mozilla-inbound. [...]

Hmm, that's interesting... So, when did the switch occur? Together with the train model or after that?

> Now we land stuff on mozilla-inbound, but still don't mark bugs as RESOLVED FIXED until mozilla-inbound gets merged to mozilla-central. And we only merge changesets that pass all the tests, so stuff now "bounces" without ever resolving (and hence reopening) the bug.

In this case I can track comments that include a link to hg.mozilla.org/integration/mozilla-inbound/rev/... and treat them as if the resolution was changed to FIXED.

I'm not sure, however, how I would detect reopening in this case. What is recorded in Bugzilla when a patch doesn't pass the tests? Is there any flag for this situation? Or people just comment informally on the bug report?

[]s
Rodrigo

Developer 4 Tue, Oct 29, 2013 at 6:26 PM
To: dev-planning@lists.mozilla.org

>On Mon, Oct 28, 2013 at 10:30 PM, Developer 3 wrote:
>
>> The change in reopening rates is probably a result of us switching from
>> checking stuff in to mozilla-central to checking it in on mozilla-inbound.
>> [...]
>>
>
>Hmm, that's interesting... So, when did the switch occur? Together with the
>train model or after that?

inbound started around the same time as trains, though perhaps offset by
a few months.

>Now we land stuff on mozilla-inbound, but still don't mark bugs as RESOLVED
>> FIXED until mozilla-inbound gets merged to mozilla-central. And we only
>> merge changesets that pass all the tests, so stuff now "bounces" without
>> ever resolving (and hence reopening) the bug.
>>
>

>In this case I can track comments that include a link to
>hg.mozilla.org/integration/mozilla-inbound/rev/... and treat them as if the
>resolution was changed to FIXED.
>
>I'm not sure, however, how I would detect reopening in this case. What is
>recorded in Bugzilla when a patch doesn't pass the tests? Is there any flag
>for this situation? Or people just comment informally on the bug report?

usually you'll see a later "backout" rev, then another rev for checking
it back in. Not sure this is 100% accurate though, and complicated by
bugs with multiple patches that land at different times (and the
[leave-open] whiteboard tag which says "I checked in but don't resolve
it as fixed", often because there are more patches to land.

And then there are roc's 10-part patch landings.... ;-)

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Wed, Oct 30, 2013 at 9:24 PM
To: Developer 4
Cc: dev-planning@lists.mozilla.org

On Tue, Oct 29, 2013 at 6:26 PM, Developer 4 wrote:

> usually you'll see a later "backout" rev, then another rev for checking
> it back in. Not sure this is 100% accurate though, and complicated by
> bugs with multiple patches that land at different times (and the
> [leave-open] whiteboard tag which says "I checked in but don't resolve
> it as fixed", often because there are more patches to land.

Yeah, it seems complicated. Anyway, I redid the analysis considering as reopened not only bugs with REOPENED status, but also all bug reports containing comments that match the (case-insensitive) regexp "back.{0,5}out" (e.g., backout, backed out, backing out etc.).

With the new analysis, the reopening rates are pretty much the same before and after the train model.

Thank you again for the feedback and let me know if you have any other thoughts on bug reopening.

[]s
Rodrigo

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

Review and bug reopening

22 messages

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Mon, Feb 17, 2014 at 8:27 PM
To: firefox-dev@mozilla.org

I've been doing some statistical analyses on Firefox's bug reports as part of my PhD thesis, and the community feedback has been essential to help me interpret the results. I'd like to share some new results and ask for your feedback again:

--

Result 1: in Firefox and in Firefox for Android, bug fixes that receive at least one review- are 40-50% more likely to be reopened than those that receive only review+.

Is this an expected result for you? Why is that? I've thought some possible explanations, but I'd like to hear your hypotheses.

More info: http://rodrigorgs.github.io/blog/2014/02/07/the-curse-of-a-negative-review/

--

Result 2: in Firefox (Desktop), bug fixes that receive at least one review+ are 45% more likely to be reopened than those that don't receive any review.

This result is counterintuitive (reviews certainly don't cause reopening, right?), but probably can be explained by the criteria used to choose which bugs get reviewed. Maybe risky changes are more likely to be reviewed, and also more likely to cause bug reopening. What do you think?

Curiously, in Firefox for Android there's no significant difference between reviewed and non-reviewed bugs with respect to reopening.

More info: http://rodrigorgs.github.io/blog/2014/02/12/are-reviews-worth-the-effort/

--

I'll appreciate any feedback on these results or any thoughts on the relationship between code review and bug reopening.

[]s
Rodrigo

Developer 5 Mon, Feb 17, 2014 at 9:04 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

On 17/02/2014 23:27, Rodrigo Rocha Gomes e Souza wrote:

> Result 2: in Firefox (Desktop), bug fixes that receive at least one review+ are 45% more likely to be reopened than those that don't receive any review.
>
> This result is counterintuitive (reviews certainly don't cause reopening, right?), but probably can be explained by the criteria used to choose which bugs get reviewed. Maybe risky changes are more likely to be reviewed, and also more likely to cause bug reopening. What do you think?
>
> Curiously, in Firefox for Android there's no significant difference between reviewed and non-reviewed bugs with respect to reopening.
>
> More info: http://rodrigorgs.github.io/blog/2014/02/12/are-reviews-worth-the-effort/

What was the sample size for bugs that were marked 'fixed' and had no reviews (and what about the ones that did)? Did they have commits? In principle, any commit in fx desktop should have gone through a review; the ones that don't (usually comment fixes and such) would be a tiny minority.

I can't think of any bug that I've recently seen closed as 'fixed' that didn't have attachments with a review+, except those that were fixed by other bugs (and none of those were reopened). The latter would explain this result, but it

would also render it essentially meaningless as to how 'effective' reviews are, because it means your sampling didn't actually correlate reviews with the closing of the bugs they fixed correctly.

Developer 5 Mon, Feb 17, 2014 at 9:05 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Firefox Dev <firefox-dev@mozilla.org>

(second time to Rodrigo; apologies, I forgot to CC the list)

On 17/02/2014 23:27, Rodrigo Rocha Gomes e Souza wrote:

> Result 2: in Firefox (Desktop), bug fixes that receive at least one review+ are 45% more likely to be reopened than those that don't receive any review.
>
> This result is counterintuitive (reviews certainly don't cause reopening, right?), but probably can be explained by the criteria used to choose which bugs get reviewed. Maybe risky changes are more likely to be reviewed, and also more likely to cause bug reopening. What do you think?
>
> Curiously, in Firefox for Android there's no significant difference between reviewed and non-reviewed bugs with respect to reopening.
>
> More info:
> http://rodrigorgs.github.io/blog/2014/02/12/are-reviews-worth-the-effort/

[Quoted text hidden]

Developer 6 Mon, Feb 17, 2014 at 9:22 PM
To: firefox-dev@mozilla.org

On 2/17/14 3:27 PM, Rodrigo Rocha Gomes e Souza wrote:

> Result 1: in Firefox and in Firefox for Android, bug fixes that receive at least one review- are 40-50% more likely to be reopened than those that receive only review+.
>
> Is this an expected result for you? Why is that? I've thought some possible explanations, but I'd like to hear your hypotheses.

My first guess would be that this could be skewed by small/easy changes -- they'd be less likely to cause fallout, and more likely to pass review the first time. It might be interesting to see if reopening is correlated with the size (or number-of-lines changed) in the patch.

The other question is what reopening really indicates -- I'd assume that it usually happens very quickly (as a result of some unexpected test failure upon first landing), and comparatively rarely due to Nightly users finding a problem severe enough to result in backing out the patch.

It might be useful to compare reopening to the number of bugs added to the "depends on" field after landing. The usual Bugzilla pattern is to only reopen / backout a bug if it caused a major regression or fundamentally failed to fix the problem. Otherwise regressions are simply dealt with in the bugs reporting them (and marked as dependencies so we can track them).

> Result 2: in Firefox (Desktop), bug fixes that receive at least one review+ are 45% more likely to be reopened than those that don't receive any review.

This is an odd apples-to-oranges comparison, since the vast majority (essentially all) of code changes require a review. I'd suspect all you're seeing here are tons of user bugs being closed as WORKSFORME / INVALID (or FIXED by some other bug, etc), and reopening is somewhat infrequent for that (usually a result of an argumentative reporter ;).

Developer 7 Mon, Feb 17, 2014 at 9:22 PM
To: Developer 5

Cc: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>, Firefox Dev <firefox-dev@mozilla.org>

> >> This result is counterintuitive (reviews certainly don't cause
> >> reopening, right?), but probably can be explained by the criteria used
> >> to choose which bugs get reviewed. Maybe risky changes are more likely
> >> to be reviewed, and also more likely to cause bug reopening. What do
> >> you think?

I haven't worked in any component within Mozilla that doesn't require code review for all non-trivial changes.

There are some missing details about your methodology, but I think your conclusions are flawed for two reasons:

1. In some components, the majority of work is not conducted through Bugzilla attachments and review flags. That is: your use of Bugzilla review flags as an indicator for whether code has been reviewed is incorrect.

2. Some bugs that are marked RESOLVED FIXED do not have attachments. They might be procedural, analytical, fixed by some other issue but not a dupe, etc. That is, your use of the resolution of a bug as an indicator for code landing is incorrect.

You note that 84% of bugs in Firefox for Android are marked as RESOLVED with a review flag. I would contend that that means that ~16% of resolved bugs didn't involve landing code, not that 16% of our patches landed without review. (You don't say whether you look for bugs with patches but no review, how you handle bugs with reviewed patches that have been obsoleted by new patches, whether you look for a target milestone, etc.)

You might want to redo this analysis looking at commit logs, searching for r=me, r=trivial, r=none, a=borkage, a=test-only, etc. to find commits that landed without review.

I agree with part of your analysis, though: it's more likely that trivial commits land without a future reopening, because they're trivial.

Developer 7 Mon, Feb 17, 2014 at 9:28 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Firefox Dev <firefox-dev@mozilla.org>

> > You might want to redo this analysis looking at commit logs, searching for r=me, r=trivial, r=none, a=borkage, a=test-only, etc. to find commits that landed without review.

I spent five minutes doing this.

```
Commits considered:     13776
r=me:                   160
r=self:                 0
r=none:                 0
r=trivial:              20
Merges, backouts, etc:  ~2400
No r=:                  386
```

Conclusion: recent fx-team has a 4% non-review rate.

Developer 7 Mon, Feb 17, 2014 at 9:36 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Firefox Dev <firefox-dev@mozilla.org>

> > No r=:   386

Note that some of these are indeed trivial and/or follow-ups:

> No bug - Alphabetize robocop.ini. DONTBUILD
> Really fix whitespace change made during conflict resolution (no bug) :-)
> Fix whitespace changes made during conflict resolution (no bug) DONTBUILD
> Bug 935277 - Fix disabling so it disables the correct test

Some are kinda-automated:

> Bug 956129 - Uplift Addon SDK to Firefox
> Bug 942207: Update NSPR to NSPR_4_10_3_BETA2. Includes changes for

And many are procedural:

> Bug 908695 followup, touch CLOBBER because bug 928195
> no bug - touch the CLOBBER file to avoid "configure change" bustage.
> Merging in version bump NO BUG

so I'd guess the rate of landing code that actually ships (not testing or build harness) without review is ~1-2%, perhaps even lower.

As such, if you're looking to evaluate the impact of code review on fixes for bugs that actually didn't get fixed, and thus were reopened, you probably shouldn't be studying Firefox -- recent commits to the mozilla-central codebase appear to have about 99% review coverage.

Developer 8 Tue, Feb 18, 2014 at 5:58 AM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev <firefox-dev@mozilla.org>

> On Mon, Feb 17, 2014 at 3:27 PM, Rodrigo Rocha Gomes e Souza
> <rodrigorgs@gmail.com> wrote:
> >
> > Result 1: in Firefox and in Firefox for Android, bug fixes that receive at
> > least one review- are 40-50% more likely to be reopened than those that
> > receive only review+.

Another complicating factor is that many people are reluctant to give r- on patches because it feels too blunt. If they didn't like the patch, they might just clear the review flag, or alternatively give f+ instead.

Developer 9 Tue, Feb 18, 2014 at 1:47 PM
To: firefox-dev@mozilla.org

> [Quoted text hidden]

Count me as one of these. My wife is an academic and I was trying to explain to her that within my working set, r- is a very strong sign that your approach, rather than your implementation, is wrong; she's used to journal review, so it seemed rather quaint to her.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Tue, Feb 18, 2014 at 4:01 PM
To: firefox-dev@mozilla.org

Wow, thanks a lot for your feedback.

Just to clarify the methodology, I analysed RESOLVED FIXED bugs from 2001-04-07 to 2013-08-10 (Firefox), and from 2008-09-11 to 2013-06-03 (Firefox for Android).

However, I didn't look at Bugzilla attachments (my data set doesn't contain them) neither at the Mercurial repo. I assumed all RESOLVED FIXED bugs included or should have included patches, which now I see isn't true.

Also, I incorrectly assumed that all reviews were recorded in Bugzilla using review+/- flags.

I'm starting to look at the Mercurial repo to find more info about bugs. It's great to see that commit messages are quite consistent and informative.

I assume r=foo means that the patch was reviewed positively by foo before being landed. Correct?

Doubts

What does a=foo mean?

Besides a=..., r=..., sr=..., f=..., what other "commit flags" are consistently used in commit messages? Is this

documented anywhere?

Do you have any tool that picks a=... and r=... from the log (other than grep and wc) and do something useful with this information?

Is there any other place where I can find records of code reviews, such as mailing lists or a web code review tool (e.g., Gerrit)?

> The other question is what reopening really indicates -- I'd assume that it usually happens very quickly (as a result of some unexpected test failure upon first landing)

Aren't bugs marked FIXED only after the tests pass?

> Another complicating factor is that many people are reluctant to give
> r- on patches because it feels too blunt. If they didn't like the
> patch, they might just clear the review flag, or alternatively give f+
> instead.

What's f+? feedback+?
I'll rerun my analysis looking for f+ and clearing of the review flag.

Comments

> As such, if you're looking to evaluate the impact of code review on fixes for bugs that actually didn't get fixed, and thus were reopened, you probably shouldn't be studying Firefox -- recent commits to the mozilla-central codebase appear to have about 99% review coverage.

This is truly amazing!
But you're right, it makes Firefox a poor choice for this particular kind of analysis.

> [regarding the influence of review- on reopening:] My first guess would be that this could be skewed by small/easy changes -- they'd be less likely to cause fallout, and more likely to pass review the first time. It might be interesting to see if reopening is correlated with the size (or number-of-lines changed) in the patch.

Yeah, you're probably right. Since I don't have Bugzilla attachment data, I'll try to get this info from the Mercurial repo.

Thank you again for your feedback.

[]s
Rodrigo

Developer 10 Tue, Feb 18, 2014 at 5:47 PM
To: Developer 9
Cc: firefox-dev@mozilla.org

All of this boils down to the style of the reviewer. I've found it's really quite different based on the reviewer+reviewee pairing. I know the people who won't be bothered when I mark a r-, but if it's a new contributor I may shift it to a f+. Over time I've strayed away from clearing the review flag because it makes it harder to glance at a bug and see if a peer has looked at a patch (as compared to a patch that gets uploaded and no request for feedback/review is ever placed).

Sometimes if I have a lingering question about a patch I will reply with some written feedback but leave the review flag present as a tip that I may grant review if there is a reasonable response to the question.

Developer 11 Tue, Feb 18, 2014 at 6:03 PM
To: firefox-dev@mozilla.org

> On 02/17/2014 04:22 PM, Developer 6 wrote:
> >> Result 2: in Firefox (Desktop), bug fixes that receive at least one
> >> review+ are 45% more likely to be reopened than those that don't receive
> >> any review.
> >
> > This is an odd apples-to-oranges comparison, since the vast majority
> > (essentially all) of code changes require a review. I'd suspect all
> > you're seeing here are tons of user bugs being closed as WORKSFORME /

> INVALID (or FIXED by some other bug, etc), and reopening is somewhat
> infrequent for that (usually a result of an argumentative reporter ;)

Side note: I wonder if Rodrigo's script looks at obsolete attachments,
when checking for reviewed patches?

If it doesn't, that would account for some of these superficially
"unreviewed" bug fixes.

Background: a somewhat common workflow is:
a) Assignee posts patch v1, with review request
b) Reviewer marks r+, and asks for a few changes
c) Assignee uploads patch v2, marks patch v1 obsolete, and tags bug as
"checkin-needed"
d) Sheriff lands the patch and marks the bug FIXED.

This leaves the bug looking like it has been fixed without any reviewed
patches, *unless* you show obsolete attachments.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Tue, Feb 18, 2014 at 7:19 PM
To: Developer 11
Cc: firefox-dev@mozilla.org

On Tue, Feb 18, 2014 at 6:03 PM, Developer 11 wrote:
> Side note: I wonder if Rodrigo's script looks at obsolete attachments,
> when checking for reviewed patches?

In fact the script looks at the bug history page (e.g., https://bugzilla.mozilla.org/show_activity.cgi?id=167180), so it can see both reviews (for obsolete patch v1 and for patch v2).

[]s
Rodrigo

Developer 7 Wed, Feb 19, 2014 at 5:32 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev@mozilla.org, Developer 14

> I assume r=foo means that the patch was reviewed positively by foo before being landed. Correct?

Yes, modulo my earlier point about r=none, r=me, r=trivial.

> What does a=foo mean?

Approval. Either for landing on an upstream branch (Aurora, Beta, Release, ESR), on a closed tree or under another restriction (a=test-only, a=java-only, a=bustage), or to justify no review (a=doc-only).

> Do you have any tool that picks a=... and r=... from the log (other than grep and wc) and do something useful with this information?

You should read this.

http://gregoryszorc.com/blog/2013/11/08/using-mercurial-to-query-mozilla-metadata/

> Is there any other place where I can find records of code reviews, such as mailing lists or a web code review tool (e.g., Gerrit)?

Many or most reviews take place on Bugzilla.

There is also a tiny minority in ReviewBoard: http://reviewboard.allizom.org/

a bunch scattered across a few hundred GitHub repos, and some take place informally over IRC or over-the-shoulder in real life.

> The other question is what reopening really indicates -- I'd assume that it usually happens very quickly (as a result of some unexpected test failure upon first landing)

Aren't bugs marked FIXED only after the tests pass?

No. They're marked as FIXED when they're done and merged to mozilla-central. They're marked as VERIFIED after someone else has checked that things worked. They're backed out, often on an integration branch prior to being marked as FIXED, if they break tests.

But note that this use of the bug tracker varies over time, over components, under the use of 'twigs' (separate project branches — do you mark as RESOLVED when it hits the twig, or when the twig merges?), and has gradually converged on the current approach after mozilla-inbound was created.

What's f+? feedback+?

Yes.

Developer 5 Wed, Feb 19, 2014 at 5:34 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

On 18/02/2014 19:01, Rodrigo Rocha Gomes e Souza wrote:

Wow, thanks a lot for your feedback.

Just to clarify the methodology, I analysed RESOLVED FIXED bugs from 2001-04-07 to 2013-08-10 (Firefox), and from 2008-09-11 to 2013-06-03 (Firefox for Android).

However, I didn't look at Bugzilla attachments (my data set doesn't contain them) neither at the Mercurial repo. I assumed all RESOLVED FIXED bugs included or should have included patches, which now I see isn't true.

Also, I incorrectly assumed that all reviews were recorded in Bugzilla using review+/- flags.

I'm starting to look at the Mercurial repo to find more info about bugs. It's great to see that commit messages are quite consistent and informative.

I assume r=foo means that the patch was reviewed positively by foo before being landed. Correct?

Yes.

Doubts

What does a=foo mean?

approved for landing by foo. Usually only used on release/beta/aurora branches, however...

Besides a=..., r=..., sr=..., f=..., what other "commit flags" are consistently used in commit messages? Is this documented anywhere?

Nope. It's informal. There is not, in fact, even a commit hook that checks for these.

Do you have any tool that picks a=... and r=... from the log (other than grep and wc) and do something useful with this information?

Not to my knowledge.

> Is there any other place where I can find records of code reviews, such as mailing lists or a web code review tool (e.g., Gerrit)?

There's firefox-reviewers... don't know to what extent that is partitioned enough to work for your usecase.

https://mail.mozilla.org/pipermail/firefox-reviewers/2014-February/date.html

> > The other question is what reopening really indicates -- I'd assume that it usually happens very quickly (as a result of some unexpected test failure upon first landing)
>
> Aren't bugs marked FIXED only after the tests pass?

Bugs are normally marked FIXED when the relevant commits land on mozilla-central. Usually, but not always, that means they first landed on an integration branch such as mozilla-inbound/fx-team, and passed tests there. However, there have been cases where it's taken some time to realize that particular commits broke tests, usually because (a) those tests were failing randomly relatively frequently already, or (b) they only broke tests on very limited sets of build configurations. Then there's the fact that our tests aren't exhaustive. And then there's the fact that some bugs are just marked FIXED when they are fixed by some other commit and we know what fixed the issue.

> > Another complicating factor is that many people are reluctant to give r- on patches because it feels too blunt. If they didn't like the patch, they might just clear the review flag, or alternatively give f+ instead.
>
> What's f+? feedback+?

Yes.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Wed, Feb 19, 2014 at 6:32 PM
To: Developer 5

Thank you very much for the clarification, Developer 5.

Maybe you wanted to send your email to firefox-dev? (you sent it just to me)

[]s
Rodrigo

On Wed, Feb 19, 2014 at 5:34 PM, Developer 5 wrote:

> On 18/02/2014 19:01, Rodrigo Rocha Gomes e Souza wrote:
>
> > I assume r=foo means that the patch was reviewed positively by foo before being landed. Correct?
>
> Yes.

Developer 5 Wed, Feb 19, 2014 at 6:37 PM
To: Firefox Dev <firefox-dev@mozilla.org>

On 18/02/2014 19:01, Rodrigo Rocha Gomes e Souza wrote:

> Wow, thanks a lot for your feedback.
>
> Just to clarify the methodology, I analysed RESOLVED FIXED bugs from 2001-04-07 to 2013-08-10 (Firefox), and from 2008-09-11 to 2013-06-03 (Firefox for Android).
>
> However, I didn't look at Bugzilla attachments (my data set doesn't contain them) neither at the Mercurial repo. I assumed all RESOLVED FIXED bugs included or should have included patches, which now I see isn't true.
>
> Also, I incorrectly assumed that all reviews were recorded in Bugzilla using review+/- flags.

> I'm starting to look at the Mercurial repo to find more info about bugs. It's great to see that commit messages are quite consistent and informative.
>
> I assume r=foo means that the patch was reviewed positively by foo before being landed. Correct?

Yes.

> ## Doubts
>
> What does a=foo mean?

approved for landing by foo. Usually only used on release/beta/aurora branches, however...

> Besides a=..., r=..., sr=..., f=..., what other "commit flags" are consistently used in commit messages? Is this documented anywhere?

Nope. It's informal. There is not, in fact, even a commit hook that checks for these.

> Do you have any tool that picks a=... and r=... from the log (other than grep and wc) and do something useful with this information?

Not to my knowledge.

> Is there any other place where I can find records of code reviews, such as mailing lists or a web code review tool (e.g., Gerrit)?

There's firefox-reviewers... don't know to what extent that is partitioned enough to work for your usecase.

https://mail.mozilla.org/pipermail/firefox-reviewers/2014-February/date.html

> > The other question is what reopening really indicates -- I'd assume that it usually happens very quickly (as a result of some unexpected test failure upon first landing)
>
> Aren't bugs marked FIXED only after the tests pass?

Bugs are normally marked FIXED when the relevant commits land on mozilla-central. Usually, but not always, that means they first landed on an integration branch such as mozilla-inbound/fx-team, and passed tests there. However, there have been cases where it's taken some time to realize that particular commits broke tests, usually because (a) those tests were failing randomly relatively frequently already, or (b) they only broke tests on very limited sets of build configurations. Then there's the fact that our tests aren't exhaustive. And then there's the fact that some bugs are just marked FIXED when they are fixed by some other commit and we know what fixed the issue.

> > Another complicating factor is that many people are reluctant to give
> > r- on patches because it feels too blunt. If they didn't like the
> > patch, they might just clear the review flag, or alternatively give f+
> > instead.
>
> What's f+? feedback+?

Yes.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Wed, Feb 19, 2014 at 7:00 PM
To: Developer 7
Cc: firefox-dev@mozilla.org, Developer 14

On Wed, Feb 19, 2014 at 5:32 PM, Developer 7 wrote:

> > Do you have any tool that picks a=... and r=... from the log (other than grep and wc) and do something useful with this information?
>
> You should read this.

> http://gregoryszorc.com/blog/2013/11/08/using-mercurial-to-query-mozilla-metadata/

Thanks! There's a lot of inside knowledge in that source code. It'll certainly be very useful for me.

Well, thank you all for the precious feedback. I've been studying Mozilla's development process and Bugzilla usage, but there's always a lot to discover.

Now I have to digest all this new information to refine my current analyses and design new ones. I'll let you know when I have new results that may interest you.

[]s
Rodrigo

Developer 8 Thu, Feb 20, 2014 at 5:20 AM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev <firefox-dev@mozilla.org>

> On Tue, Feb 18, 2014 at 11:01 AM, Rodrigo Rocha Gomes e Souza
> <rodrigorgs@gmail.com> wrote:
> >
> > Besides a=..., r=..., sr=..., f=..., what other "commit flags" are
> > consistently used in commit messages? Is this documented anywhere?

You occasionally see rs=..., which is short for "rubber-stamp", which roughly means "I glanced at this but didn't really look at it closely". I think it's usually used for short patches, additions of tests, thing like that. Not an important case, but just in case you were wondering...

Developer 12 Thu, Feb 20, 2014 at 11:11 AM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

> Rodrigo Rocha Gomes e Souza wrote:
>
> > However, I didn't look at Bugzilla attachments (my data set doesn't contain them) neither at the Mercurial repo. I assumed all RESOLVED FIXED bugs included or should have included patches, which now I see isn't true.

do you know about https://bugzilla.mozilla.org/page.cgi?id=researchers.html ?

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Thu, Feb 20, 2014 at 11:38 AM
To: Developer 12

> On Thu, Feb 20, 2014 at 11:11 AM, Developer 12 wrote:
> > do you know about https://bugzilla.mozilla.org/page.cgi?id=researchers.html ?

No, I didn't. Thanks, I'll contact Mike.

[]s
Rodrigo

Developer 13 Thu, Feb 20, 2014 at 11:20 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev@mozilla.org

I've not seen anyone mention that Firefox for Android has a much more brittle testing infrastructure. It is somewhat common for a fix to cause intermittent oranges. Robocop is the main UI testing suite https://wiki.mozilla.org/Auto-tools/Projects/Robocop

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

Backouts due to test failures

7 messages

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Mon, Jun 2, 2014 at 7:34 PM
To: firefox-dev@mozilla.org

I'm trying to measure the proportion of backouts due to failing automated tests before and after the train model. I identify such backouts by looking for the words "backout" (or a variation of it) and "test". Example:
Backout bug 504524 due to test failure

Is it safe to assume that such commit messages always refer to automated tests? Is there any case in which a backout commit uses the word "test" to refer to manual test?

[]s
Rodrigo

Developer 5 Mon, Jun 2, 2014 at 7:43 PM
To: firefox-dev@mozilla.org

[Quoted text hidden]
That seems unlikely to me. However, I don't think backouts necessarily have the word "test" in them when they are enacted due to test failures. In other words, I suspect you will have false negatives rather than false positives if you rely on parsing log messages.

Developer 14 Mon, Jun 2, 2014 at 7:55 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>, firefox-dev@mozilla.org

[Quoted text hidden]
No. I'd look at file names. (But even that isn't perfect.)

# Commit messages with "backout" that also touched a file with "test"
$ hg log -r 'desc(backout) & file("**test**")'

See `hg help patterns`, `hg help revset`, and `hg help template` to construct a powerful query. Templating also contains conditional statements, so you can make Mercurial emit reports. See e.g. http://gregoryszorc.com/blog/2014/04/01/using-mercurial-for-status-reports/

Developer 15 Tue, Jun 3, 2014 at 12:32 AM
To: Developer 14
Cc: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>, firefox-dev@mozilla.org

----- Original Message -----
> On 6/2/14, 3:34 PM, Rodrigo Rocha Gomes e Souza wrote:
> > I'm trying to measure the proportion of backouts due to failing
> > automated tests before and after the train model. I identify such
> > backouts by looking for the words "backout" (or a variation of it) and
> > "test". Example:
> >
> > Backout bug 504524 due to test failure
> >
> > Is it safe to assume that such commit messages always refer to automated
> > tests? Is there any case in which a backout commit uses the word "test"
> > to refer to manual test?
>
> No. I'd look at file names. (But even that isn't perfect.)
>
> # Commit messages with "backout" that also touched a file with "test"
> $ hg log -r 'desc(backout) & file("**test**")'

The existence of a test in the patch doesn't seem strongly related to the whether it caused a failure in some test when it landed. Many times patches that touch no tests get backed out for test failures.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Tue, Jun 3, 2014 at 3:25 PM

To: Developer 5
Cc: Developer 14, Developer 15, firefox-dev@mozilla.org

> Thanks for your feedback. It appears there isn't much to do in this case, except reporting the limitations of the study.

[]s
Rodrigo

Developer 10 Tue, Jun 3, 2014 at 4:07 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Developer 5, firefox-dev@mozilla.org, Developer 14

> If you look at https://tbpl.mozilla.org, you can see the waterfall builds for each check-in. Subsequently, you will find that mozilla-inbound and fx-team are the two inbound branches that feed in to mozilla-central.
>
> You could write a script that looks for pushes that have a fair amount of "orange", followed by all green. A test suite will turn orange when an automated test fails, and will subsequently turn green when the test is fixed. You could then look for the word "backout" in the pushlog for the push that turned the tree back to green. This would show you that a patch was backed out due to an automated test breaking.
>
> Hope that helps,

Developer 12 Wed, Jun 4, 2014 at 2:25 AM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev@mozilla.org

> Rodrigo Rocha Gomes e Souza wrote:
>
> > Thanks for your feedback. It appears there isn't much to do in this case, except reporting the limitations of the study.
>
> http://futurama.theautomatedtester.co.uk/ may be useful.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

Is backout rate increasing?

23 messages

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Sun, Jul 27, 2014 at 12:12 PM
To: firefox-dev@mozilla.org

As part of my PhD, I've been keeping track of backouts from Firefox 3.5 up to Firefox 27, and I noticed that, since the adoption of 6-week releases, the backout rate (num. of backouts / num. of bugs fixed) has increased from 6.8% to 9.4% [1]. I wonder...

1. ... have you noticed the increase in backout rate or its effects?
2. ... why would the backout rate grow under rapid release cycles?

Would you help me with these questions? It's always good to get feedback from people who actually work on Firefox.

Some additional insights:

* I noticed that, after the adoption of rapid releases, the number of bugs fixed per day increased 74%, but also that the number of developers (computed as the num. of people that committed at least 10 bug fixes) increased 75%, so *apparently* developer workload haven't increased (please correct me if I'm wrong). Therefore, workload doesn't seem to explain the growth in backout rate.
* I also noticed that, under rapid releases, more than 85% of backouts ocurred during or before merging into central (i.e., before the bug status changed to FIXED), versus 55% under traditional releases. Seems like an improvement, since more problems are being discovered during automated testing and less manual tests need to be repeated due to backouts.

[]s
Rodrigo

[1]: http://rodrigorgs.github.io/images/firefox-backouts.png

Developer 16 Sun, Jul 27, 2014 at 4:10 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev@mozilla.org

Hi!

Interesting question.

If I had to guess, I'd say it's because our automated testing has improved considerably since 3.5. A number of memory leak finding tools[1] have been integrated into our test environments that are improving our early catch-rate. This is in addition to our ever-expanding set of unittests running on every checkin[2].

I think your second point ("more than 85% of backouts occurred during or before merging into central") supports the "better testing" theory. Proving this would be difficult as you'd need to mine all of the backout-related bugs to find the reason for the backout. Searching for keywords like "regression", "leak" or "fail" might be automatable.

1- https://developer.mozilla.org/en-US/docs/Mozilla/Debugging/Debugging_memory_leaks

2- https://developer.mozilla.org/en-US/docs/Mochitest

Developer 16 Sun, Jul 27, 2014 at 4:13 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev@mozilla.org

One other point occurs to me: We've hardened our landing processes during that timeframe. Where bugs might have had broken patches land and gotten fixed in-tree, our current process and tree sheriffs will backout obvious failures until the bugs get fixed before landing. It's much harder to land something that is known to be "broken" in our current process than it used to be.

cheers,

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Sun, Jul 27, 2014 at 6:06 PM
To: Developer 16
Cc: firefox-dev@mozilla.org

Thanks, Developer 16!

If you're curious, I've computed how often words like "regression", "leak", "fail", and "test" appear in backout messages, both under traditional and rapid releases:

```
| keyword \ cycle || traditional |   rapid    |
|-----------------||---------- ---|------------|
| regression      ||  0.09965338 | 0.02358804 |
| leak            ||  0.02599653 | 0.02923588 |
| fail            ||   0.1507799 |  0.2272425 |
| test            ||   0.1897747 |  0.2631229 |
```

(numbers are proportional to number of backouts; 0 = 0%, 1 = 100%)

You can't rely too much on word frequencies extracted from commit messages, but it appears that regressions have reduced and backouts due to test failures have increased.

So, summarizing what you said:

(a) bugs used to be fixed in-tree more often
(b) tricky problems, like memory leaks, used to be harder to detect

Regarding (b), I assume that it would contribute to a higher backout rate in the following cases:

(b.1) Small memory leaks, that would go unnoticed before, are now found, and the patch is backed out and fixed.
(b.2) Some memory leaks would be found much after the patch that created it was committed; as a result, a new bug report would be filled (instead of reopening the original bug report and backing out the patch)

Is it reasonable? Of course such hypotheses would be very difficult to prove...

[]s
Rodrigo
[Quoted text hidden]

Developer 17 Sun, Jul 27, 2014 at 6:55 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

Another possible reason for more backouts would be that the amount of code contributed to our codebase has substantially increased since Firefox 3.5 so code conflicts are far more likely... no matter what we do.

Developer 7 Sun, Jul 27, 2014 at 6:56 PM
To: Developer 16, Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: firefox-dev@mozilla.org

> We've hardened our landing processes during that timeframe. Where bugs might have had broken patches land and gotten fixed in-tree, our current process and tree sheriffs will backout obvious failures until the bugs get fixed before landing. It's much harder to land something that is known to be "broken" in our current process than it used to be.

Additionally:

• We now have various 'inbound' trees, rather than landing directly on m-c. These have full-time sheriffs, and you don't have to watch your push. For better or worse, these are often treated as an automated landing platform; if it breaks, it'll get backed out without the developer having to do much. In the "old days", you were expected to have built, tested, done a Try build, etc. before the patch landed.

That has perhaps led to more broken code being landed and backed out.

I remember feeling a great deal more pressure and worry prior to landing on m-c than I do on fx-team. I have landed code where I didn't do a full build after making changes requested in a review… and have broken the build as a result. And we routinely do so without a Try build. After all, if we're going to spend the money to do multiple builds and run tests, it might as well be on fx-team rather than on Try.

• We have a lot more stuff that can break, on more platforms, as well as more tests — these days we don't have everyone working on just Firefox. Code landing for B2G can break Fennec, for example, and B2G devs don't build and test on Fennec locally. Those kinds of changes will be caught and backed out when they hit the trees, not found beforehand.

I don't see anything in your analysis, Rodrigo, that indicates differentiation based on tree.

I think a much more interesting question is: what has the introduction of sheriffed integration branches done to the backout rate on mozilla-central? And relatedly, what's the backout rate on mozilla-beta as a result?

I bet Beta's backout rate is way lower than m-c's was. And I would expect that m-c's rate has dropped since 3.5, with mozilla-inbound and fx-team increasing (from zero).

Developer 18 Sun, Jul 27, 2014 at 7:10 PM
To: firefox-dev@mozilla.org

Rodrigo Rocha Gomes e Souza schrieb:
> 2. ... why would the backout rate grow under rapid release cycles?

Part of switching to Rapid Release was to make sure that Nightly was always stable to use and to not excuse brokenness there any more, which means being more aggressive when backing out patches. AFAIK, the backout aggressiveness was even explicitly mentioned when we switched.

Also, as others have pointed to, since we switched to landing patches almost exclusively on other integration branches, we both made it easier to land there and as people are not supposed to watch those trees for test failures, we do not end up in them fixing the issues with followup after followup fix but rather have them backed out right away.

> * I noticed that, after the adoption of rapid releases, the number of bugs
> fixed per day increased 74%, but also that the number of developers
> (computed as the num. of people that committed at least 10 bug fixes)
> increased 75%, so *apparently* developer workload haven't increased (please
> correct me if I'm wrong). Therefore, workload doesn't seem to explain the
> growth in backout rate.

Yes, a developer can only do so much work. Growth is mostly adding developers nowadays, not the individual doing more. And we end up re-landing fixed versions of the patches that were backed out, we're just not doing followup fixes after the initial commit as much.

At least that's my impression as someone watching but not actively developing himself.

Developer 19 Sun, Jul 27, 2014 at 7:40 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Firefox Dev <firefox-dev@mozilla.org>

This analysis doesn't seem to take into account the differences in
overall development volume over time. We have more developers, more
projects, more patches going in than we did in the Firefox 3.5 days.
Not having researched it, I suspect the differences in backout rates
are more likely tied to a) that overall increase in development
volume/breadth and b) different tree management practices and
different tools (increasing use of "integration branches" like
mozilla-inbound & different pre-landing Try practices).

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Sun, Jul 27, 2014 at 8:12 PM
To: Developer 7, Developer 18, Developer 19, Developer 17
Cc: firefox-dev@mozilla.org

Thank you all for your feedback, it all makes sense. Summarizing:

- with the increase of the code base and number of projects, conflicts and integration problems are more likely
- increased backout rate is not necessarily bad, it just means that developers are relying more on the process and infrastructure (tree sheriffs and inbound/fx-team trees)

> I don't see anything in your analysis, Rodrigo, that indicates differentiation based on tree.

While I have been analyzing only the m-c commit log, I've measured backout rate in distinct situations:

1. backouts ocurring before bug report status changes to RESOLVED-FIXED -- equivalent to backouts in inbound
2. backouts ocurring after RESOLVED-FIXED -- equivalent to backouts in m-c and later trees
3. backouts ocurring after VERIFIED -- equivalent to backouts in aurora/beta?

In fact, backouts after RESOLVED-FIXED decreased from 3.2% to 1.4%. Backouts after VERIFIED are rare in any case, so the difference is not significant.

> I bet Beta's backout rate is way lower than m-c's was. And I would expect that m-c's rate has dropped since 3.5, with mozilla-inbound and fx-team increasing (from zero).

How do I know, looking at commit logs only, if a backout ocurred in inbound, m-c, aurora, or beta? I mean, if someone commits a patch to inbound, then commits a backout, and then commits an improved version, the three commits are merged into m-c, right? Do I have to look for commits that are in m-c and aren't in inbound? If this is the case, is there a convenient Mercurial command to do it?

[]s
Rodrigo

Developer 20 Sun, Jul 27, 2014 at 9:59 PM
To: Developer 19, Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Firefox Dev <firefox-dev@mozilla.org>

As an observer of the dev process, that would jibe with what I'm seeing. The rules have been tightened enormously, the process moves much faster with less tolerance of errors, so increased backouts is a natural occurrence. Indeed, I see it as a very good thing that the backout rate has gone up. It means the system is working just the way it's meant to — if something isn't ready for prime time, it gets backed out and tried again on the next train.

On July 27, 2014 at 6:40:21 PM, Developer 19 wrote:

> This analysis doesn't seem to take into account the differences in overall development volume over time. We have more developers, more projects, more patches going in than we did in the Firefox 3.5 days. Not having researched it, I suspect the differences in backout rates are more likely tied to a) that overall increase in development volume/breadth and b) different tree management practices and different tools (increasing use of "integration branches" like mozilla-inbound & different pre-landing Try practices).

Developer 7 Mon, Jul 28, 2014 at 1:13 AM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Developer 18, Developer 19, Developer 17, "firefox-dev@mozilla.org Dev" <firefox-dev@mozilla.org>, Developer 14

> How do I know, looking at commit logs only, if a backout ocurred in inbound, m-c, aurora, or beta? I mean, if someone commits a patch to inbound, then commits a backout, and then commits an improved version, the three commits are merged into m-c, right? Do I have to look for commits that are in m-c and aren't in inbound? If this is the case, is there a convenient Mercurial command to do it?

Commits are merged between inbound/fx-team/m-c; at some instances in time these three trees contain the same commits, with the same hashes but different commit IDs. Via plain ol' `hg log` these are indistinguishable.

E.g., in fx-team:

changeset: 196163:1d6cb0c4b970
user: Carsten "Tomcat" Book <cbook@mozilla.com>
date: Fri Jul 25 15:59:52 2014 +0200
summary: Backed out changeset 3e869dd7e82a (bug 1016629)

in m-c:

changeset: 196151:1d6cb0c4b970
user: Carsten "Tomcat" Book <cbook@mozilla.com>
date: Fri Jul 25 15:59:52 2014 +0200

summary: Backed out changeset 3e869dd7e82a (bug 1016629)

Fortunately, the amazing Mr Szorc has tools that can help answer your questions:

http://moztree.gregoryszorc.com/api/changeset/1d6cb0c4b970

If you're trying to figure stuff out using Mercurial, you should talk to him.

Developer 14 Mon, Jul 28, 2014 at 1:12 PM
To: Developer 7, Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: Developer 18, Developer 19, Developer 17, "firefox-dev@mozilla.org Dev" <firefox-dev@mozilla.org>

[Quoted text hidden]
I would use my mozext extension as described at [1] for doing this kind
of analysis. e.g.

$ hg pull http://hg.stage.mozaws.net/gecko
$ hg pushlogsync
$ hg log --template '{rev}:{node|short} {firstpushtree}\n'

That will identify where each changeset landed first.

You should read the `hg help revset` and `hg help templates` output to
learn how Mercurial can be used to construct powerful queries and answer
the questions you seek.

[1]
http://gregoryszorc.com/blog/2013/11/08/using-mercurial-to-query-mozilla-metadata/

Developer 6 Mon, Jul 28, 2014 at 2:04 PM
To: firefox-dev@mozilla.org

> On 7/27/14 3:40 PM, Developer 19 wrote:
> > This analysis doesn't seem to take into account the differences in
> > overall development volume over time. We have more developers, more
> > projects, more patches going in than we did in the Firefox 3.5 days.

It's a percentage of landings, so it should be volume independent.

I think Developer 7 is on the right track, w.r.t. the changing standards of how careful people need to be when landing.

Another possible factor to consider: In the old cycle, the type of code being landed changed over the course of the cycle. As a release neared, there were extended periods of time when landings were restricted to fixes for critical/blocker issues only. I'd guess there was a lower backout rate then... OTOH, there was more attention focused on the tree at those times, so could actually be higher.

It would be interesting to see a graph of backout rate over time.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Mon, Jul 28, 2014 at 4:48 PM
To: Developer 142, Developer 6
Cc: Developer 7, Developer 18, Developer 19, Developer 17, "firefox-dev@mozilla.org Dev" <firefox-dev@mozilla.org>

> On Mon, Jul 28, 2014 at 1:12 PM, Developer 14 wrote:
> > I would use my mozext extension as described at [1] for doing this kind
> > of analysis. e.g.
> >
> > $ hg pull http://hg.stage.mozaws.net/gecko
> > $ hg pushlogsync
> > $ hg log --template '{rev}:{node|short} {firstpushtree}\n'

Thank you, it worked perfectly.

> On Mon, Jul 28, 2014 at 2:04 PM, Developer 6 wrote:
> > Another possible factor to consider: In the old cycle, the type of code being landed changed over the course of the cycle. As a release neared, there were extended periods of time when landings were restricted to fixes for critical/blocker issues only. I'd guess there was a lower backout rate then... OTOH, there was more attention focused on the tree at those times, so could actually be higher.
> >
> > It would be interesting to see a graph of backout rate over time.

There's a graph at http://rodrigorgs.github.io/images/firefox-backouts.png
The graph doesn't show the x-axis labels, so it's not easy to draw conclusions. However, we can see that the backout rate was the highest just before the 4.0 release (vertical dashed line).

[]s
Rodrigo

Developer 19 Mon, Jul 28, 2014 at 8:51 PM
To: Developer 6
Cc: Firefox Dev <firefox-dev@mozilla.org>

> On Mon, Jul 28, 2014 at 10:04 AM, Developer 6 wrote:
> > On 7/27/14 3:40 PM, Developer 19 wrote:
> >>
> >> This analysis doesn't seem to take into account the differences in
> >> overall development volume over time. We have more developers, more
> >> projects, more patches going in than we did in the Firefox 3.5 days.
> >
> >
> > It's a percentage of landings, so it should be volume independent.

That was kind of my point - I would not assume that the backout rate should be volume-independent. As a project grows I would expect the backout rate to also grow, all else being equal. The implied conclusion (rapid release caused a higher backout rate) seemed to depend on assuming otherwise.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Tue, Jul 29, 2014 at 7:50 AM
To: Developer 19
Cc: Developer 6, Firefox Dev <firefox-dev@mozilla.org>

> On Mon, Jul 28, 2014 at 8:51 PM, Developer 19 wrote:
> > That was kind of my point - I would not assume that the backout rate should be volume-independent. As a project grows I would expect the backout rate to also grow, all else being equal. The implied conclusion (rapid release caused a higher backout rate) seemed to depend on assuming otherwise.

Sure, it's a valid point. It's reasonable to assume that backout rate is somehow proportional to million lines of code (MLOC), but I think they're not linearly proportional. Let's assume that backout rate should be proportional to the square root of MLOC. In this case:

Before: 6.8% / sqrt(4.9) MLOC = 3.07
After: 9.4% / sqrt(8.1) MLOC = 3.30 (i.e., a 7% increase)

Some researchers propose using a metric like num. of backed out bugs / million lines of code (MLOC):

Before: 836 backouts / 4.9 MLOC = 170.6
After: 2305 backouts / 8.1 MLOC = 284.6 (i.e., a 66% increase)

I assumed volume could be one explanation to the increased backout rate, but given the high growth (from 6.8% to 9.4%), I thought it was worthwhile to look for additional explanations.

[]s
Rodrigo

Developer 21 Tue, Jul 29, 2014 at 5:24 PM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs-Re5JQEeQqe8AvxtiuMwx3w@public.gmane.org>, firefox-

dev@mozilla.org

> On 7/29/14, 6:50 AM, Rodrigo Rocha Gomes e Souza wrote:
> > It's reasonable to assume that backout rate is
> > somehow proportional to million lines of code (MLOC)

Backout rate depends not just on size of codebase but also on number of checkins.

Consider the lifetime of a typical checkin:

1) Pull code.
2) Make changes.
3) Test changes.
4) Rebase.
5) Push code.

You get a backout if sometime between step 1 and step 4 someone checked in something that makes the testing from step 3 invalid.

Backout rate also depends on number of supported configurations, because it complicates local testing. Note that we added a number of these since we started rapid release.

> > Let's assume that backout rate should
> > be proportional to the square root of MLOC.

Why not the square? Or the log? I mean, there's no reason to assume it's linear, but why square root?

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Tue, Jul 29, 2014 at 7:07 PM
To: Developer 21
Cc: Rodrigo Rocha Gomes e Souza <rodrigorgs-Re5JQEeQqe8AvxtiuMwx3w@public.gmane.org>, "firefox-dev@mozilla.org Dev" <firefox-dev@mozilla.org>

> On Tue, Jul 29, 2014 at 5:24 PM, Developer 21 wrote:
> > On 7/29/14, 6:50 AM, Rodrigo Rocha Gomes e Souza wrote:
> > > It's reasonable to assume that backout rate is
> > > somehow proportional to million lines of code (MLOC)
> > Backout rate depends not just on size of codebase but also on number of checkins.

That's a good point. If we define backout rate as backouts / checkins, then what you're saying is that, everything else equal, backouts should not be proportional to checkins, but to a function of the number of checkins (e.g., checkins^2), right?

> > > Let's assume that backout rate should
> > > be proportional to the square root of MLOC.
> > Why not the square? Or the log? I mean, there's no reason to assume it's linear, but why square root?

I actually just picked the first sublinear function I could think about. I couldn't find any empirically tested formula linking backout rate to MLOC, and it's not my goal right now to find this link (maybe in the future).

[]s
Rodrigo

Developer 22 Sun, Aug 3, 2014 at 5:42 PM
To: firefox-dev@mozilla.org

From my brief analysis of the numbers I have seen higher backout rates than you are reporting but they have been dropping over time.

For the last 7 days I have around 8% of pushes have a backout in them on Mozilla Inbound. (26 backouts and 321 pushes you can see the data at http://futurama.theautomatedtester.co.uk/)

The problem I found was that I could not rely on looking at hg log at all. This down to the interesting way that we handle integration trees and then do merges. Others have briefly mentioned this potential problems so I have been using the Pushes API from the Release Engineering team but that can only handle a certain timeline (I am waiting on treeherder API to be ready to get long term data since I would love 6 months worth of data).

I would love to see the code/papers that you are using to do this work since I have a special interest in this area since I manage the sheriffs.

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Mon, Aug 11, 2014 at 9:10 AM
To: Developer 22
Cc: "firefox-dev@mozilla.org Dev" <firefox-dev@mozilla.org>

> On Sun, Aug 3, 2014 at 5:42 PM, Developer 22 wrote:
> > From my brief analysis of the numbers I have seen higher backout rates than you are reporting but they have been dropping over time.

Sorry for the delay. Well, there are many ways to filter the data to compute backout rates, so the percentages may be different. I'll send you the details on how I compute statistics.

> For the last 7 days I have around 8% of pushes have a backout in them on Mozilla Inbound. (26 backouts and 321 pushes you can see the data at http://futurama.theautomatedtester.co.uk/)

That's interesting. Is the source code available?

[]s
Rodrigo

Developer 22 Mon, Aug 11, 2014 at 9:20 AM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>
Cc: "firefox-dev@mozilla.org Dev" <firefox-dev@mozilla.org>

The way that I calculate the backouts, and this might be the discrepancy, a push is counted a backout if there is a commit in there that has a backout. I don't go and check all the commits, I normally jump out of the loop when I find the first one.

Code can be found at https://github.com/AutomatedTester/futurama-data/blob/master/app/tree_controller.py#L46

Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com> Sun, Jan 18, 2015 at 7:18 AM
To: "firefox-dev@mozilla.org Dev" <firefox-dev@mozilla.org>

I'd like to thank all of you that helped me with my research on backouts in Firefox. You guys were really amazing. The feedback you gave me in this list was incredibly rich and essential for me to interpret the raw data from Bugzilla and Mercurial.

The result is a paper entitled "Rapid Releases and Patch Backouts: A Software Analytics Approach", that will be published on IEEE Software's special issue on release engineering. A draft of the paper is available at [2]. The final version will be published in March.

[1]: http://releng.polymtl.ca/RELENG2015/html/SI.html (pre-print)
[2]: http://rodrigorgs.github.io/publications/souza2015.pdf

[]s
Rodrigo

Developer 23 Tue, Jan 20, 2015 at 3:44 AM
To: Rodrigo Rocha Gomes e Souza <rodrigorgs@gmail.com>

Thanks for sharing your research results, Rodrigo. It is very interesting!

APPENDIX B

RELATED PAPERS BY THE AUTHOR

This appendix contains the first page of our previous publications directly related to this thesis (SOUZA; CHAVEZ, 2011; SOUZA; CHAVEZ; BITTENCOURT, 2014, 2015c, 2015a, 2015b).

Programa Multiinstitucional de Pós-graduação em Ciência da Computação PMCC-RT-01/2011

Impact of the Four Eyes Principle on Bug Reopening: the Case of Eclipse *

Rodrigo Rocha Christina Chavez

rodrigo@dcc.ufba.br , flach@dcc.ufba.br

Resumo. O princípio dos quatro olhos diz que, depois que um desenvolvedor soluciona um bug, um outro desenvolvedor deve verificar a solução. A justificativca é que quem soluciona um bug está muito próximo do código e pode se esquecer de cobrir casos excepcionais, fazendo com que o bug se manifeste mais tarde no ciclo de desenvolvimento, levando a baixa qualidade do software e retrabalho. Neste relatório, examinamos o sistema de rastreamento de bugs do Eclipse, um grande projeto de código aberto que recomenda o princípio dos quatro olhos, pra verificar se bugs resolvidos de acordo com o princípio têm maior probabilidade de terem sido definitivamente solucionados. Os resultados indicam que tal relação ocorre em alguns subprojetos do Eclipse, mas não em outros. Uma investigação mais aprofundada é necessária para que se descubram as razões por trás de tal assimetria.

Palavras-chave: práticas de desenvolvimento de software, relatórios de bugs, sistemas de código aberto, mineração de repositórios de software.

Abstract. The four eyes principle says that, after a developer fixes a bug, another developer should verify the fix. The rationale is that whoever fixes a bug is too close to the code and may forget to cover some corner cases, causing the bug to manifest itself later in the software lifecycle, which leads to poor software quality and rework. In this report, we examine the bug tracking system of Eclipse, a large open source project that recommends the four eyes principle, to see whether bugs that were handled accordingly to the four eyes principle are more likely to be definitely fixed. The results indicate that this relation occurs in some sub-projects but not in others. Further investigation is needed in order to discover the reasons behind such asymmetry.

Keywords: software development practices, bug reports, open source, software repository mining.

* Trabalho apoiado pela Fapesb. This work has been sponsored by Fapesb.

2014 Brazilian Symposium on Software Engineering

Do Rapid Releases Affect Bug Reopening? A Case Study of Firefox

Rodrigo Souza
Office of Information Technology
Federal University of Bahia, Brazil
Email: rodrigo@dcc.ufba.br

Christina Chavez
Computer Science Department
Federal University of Bahia, Brazil
Email: flach@dcc.ufba.br

Roberto A. Bittencourt
State University of Feira de Santana, Brazil
Email: roberto@uefs.br

***Abstract*—Large software organizations have been adopting rapid release cycles to deliver features and bug fixes earlier to their users. Because this approach reduces time for testing, it raises concerns about the effectiveness of quality assurance in this setting. In this paper, we study how the adoption of rapid release cycles impacts bug reopening rate, an indicator for the quality of the bug fixing process. To this end, we analyze thousands of bug reports from Mozilla Firefox, both before and after their adoption of rapid releases. Results suggest that the bug reopening rate of versions developed in rapid cycles was about 7% higher. Also, as a warning to the software analytics community, we report contradictory results from three attempts to answer our research question, performed with varying degrees of knowledge about the Firefox release process.**

I. Introduction

The internet has made easier to distribute software, effectively increasing the competition between software organizations. The so-called "internet time" [1] pushed organizations to release new features at a faster pace. As a result, in the last decade, projects such as Firefox and Unity3D started moving from a 12–18 month release cycle to a shorter, 1–3 month release cycle.

Although the main motivation for moving to rapid (or short) release cycles is time-to-market, some people argue that it also helps improve software quality. As developers are expected to always implement potentially shipable source code, testing efforts can be more frequent, so hopefully less bugs reach end users. On the other hand, short cycles may lead to tighter deadlines for testing, which may ultimately lead to less stable versions being released to the public.

Whether rapid releases actually reduce bugs is subject to discussion. Khohm et al. [2], for instance, found no significant difference in the number of bugs created each day when comparing traditional and short releases at Mozilla (although the median number of bugs increased a little under short releases).

In this paper we explore a related, but different question: how do rapid release cycles affect bug reopening? Bug reopening is what happens when, after a bug report is resolved, someone discovers that the resolution was inappropriate or incomplete.

Bug reopening is undesirable because it leads to rework. While it cannot be completely avoided, a high bug reopening rate may indicate an unstable development process [3].

Despite its importance, there is still little empirical evidence on how bug reopening is influenced by the software development process. In this paper, we investigate three research questions on the relationship between rapid releases and bug reopening:

RQ1: Do rapid releases impact the bug reopening rate? We expect that, with short release cycles, bug reopening would be more frequent. The rationale is that, in that case, bugs would be fixed faster, and might not cover corner cases. Also, there would be less time for testing.

RQ2: Do rapid releases impact the number of bugs reopened due to failing automated tests? If a bug is reopened due to a failing automated test, it means that the initial manual testing and core review were not successful. We expect that, because rapid releases leave less time for comprehensive manual testing, bug fixes would fail automated tests more often.

RQ3: Do rapid releases impact the time it takes for a bug to be reopened? The time between a bug fix and its reopening, also known as *latent time* [4], measures how fast faulty fixes are detected. Smaller latent times mean that end users are less likely to get software with bugs that were believed to be fixed. We expect that, when software is released more often, faulty bug fixes would be discovered faster by plugin developers and early users, leading to a shorter latent time.

To conduct this study, we performed statistical analyses on bug reports and commit logs from Mozilla Firefox, an open source web browser used by half a billion people worldwide[1]. Mozilla recently moved from a traditional, planned release cycle, to a rapid, 6-week release cycle.

The results suggest that (1) the bug reopening rate increased 7% under rapid releases; (2) the number of bugs reopened due to failing automated tests did not change significantly; and (3) long-lived faulty bug fixes tended to be discovered earlier under rapid releases. Although it is not possible to generalize from a single case, such results can provide insight for further studies.

The word "bug" is used in a broad sense, meaning any problem that was documented in a bug report. A significant proportion of bug reports do not refer to corrective maintenance tasks [5]; they may refer instead to performance im-

[1] https://blog.mozilla.org/press/ataglance/

978-1-4799-4223-7/14 $31.00 © 2014 IEEE
DOI 10.1109/SBES.2014.10

Rapid Releases and Patch Backouts

A Software Analytics Approach

Rodrigo Souza and Christina Chavez, Federal University of Bahia

Roberto A. Bittencourt, State University of Feira de Santana

// To investigate the results of Mozilla's adoption of rapid releases, researchers analyzed Firefox commits and bug reports and talked to Firefox's developers. The results show that developers are backing out broken patches earlier, rendering the release process more stable. //

RELEASE ENGINEERING deals with decisions that impact the daily lives of developers, testers, and users and thus contribute to a product's success. Although gut feeling is important in such decisions, it's increasingly important to leverage existing data, such as bug reports, source code changes, code reviews, and test results, both to support decisions and to help evaluate current practices. The exploration of software engineering data to obtain insightful information is called *software analytics*.[1]

In 2011, the Mozilla Foundation fundamentally changed its release process, moving from traditional 12- to 18-month releases to rapid, six-week releases. The motivation was the need to deliver new features earlier to users, keeping pace with the evolution of Web standards, the competition among Web browsers, and the emergence of mobile platforms.

Researchers have used software analytics to study the impact of Mozilla's adoption of rapid releases (see the sidebar). Those studies focused on changes from the viewpoint of users, plug-in developers, and quality engineers. Here, we focus on how rapid releases affect code integration, which is essential for the timely release of new versions.

In particular, we analyze how the *backout* rate evolved during Mozilla's process change. A backout reverts a patch that was committed to a source code repository, either because it broke the build or, generally, because some problem was found in the patch. Backout implies rework because it requires writing, reviewing, and testing a new patch. A high backout rate indicates an unstable process.

Code Integration at Mozilla

Over the last five years, development at Mozilla in general, and Firefox in particular, has intensely applied code review and automated testing at multiple levels, such as unit testing and user interface testing. This process has been supported by tools such as Bugzilla, a bug-tracking system, and Mercurial, a distributed version control system. Here we describe the process before 2011 and the changes that occurred after. Because we analyze only the period between 2009 and 2013, we ignore specifics of the process before 2009 and after 2013.

Before 2011: Traditional Releases

Before March 2011, Firefox development followed a traditional release schedule. Features for the upcoming version were developed along with bug fixes and minor updates for the

0740-7459/15/$31.00 © 2015 IEEE

On Integration Repositories, Build Sheriffs, and Patch Backouts

Rodrigo Souza
Federal University of Bahia

Christina Chavez
Federal University of Bahia

Roberto A. Bittencourt
State University of Feira de Santana

ABSTRACT

If a developer commits a patch that breaks the build (i.e., that does not compile or causes tests to fail), the given patch should be *backed out* in order to keep the repository stable while the developer writes a fix.

In 2011 Firefox adopted rapid release cycles and changed its code integration model. Previously, developers committed code to the central repository off which patches branched; in 2011, they started committing to a separate integration repository. Now, only patches that successfully build are merged by dedicated build sheriffs into the central repository. This liberates developers from needing to perform comprehensive testing prior to committing patches [3].

Khomh et al. [1] studied Firefox's process changes and concluded that bug fix patches were released quicker after 2011. Mäntylä et al. [2] showed that Firefox's rapid releases left less time for manual testing, which became focused on specific areas.

This study also observed Firefox's changes. Analyzing commits for 41,305 issues from 2009 to 2013, we determined that the proportion of issues with commits backed out because of broken builds increased from 3.5% (2009–2011) to 8.3% (2011–2013). This is a result of less comprehensive developer testing and sheriffs backing out broken patches.

This increase in backouts, however, is a non-issue. Under the new process, those backouts are performed in the integration repository prior to merging. The central repository became even more stable: in the same period, the proportion of issues with commits backed out later, during manual testing, dropped from 3.1% to 1.5%.

BODY

Integration repositories and build sheriffs allow developers to test less comprehensively while keeping the main repository stable.

REFERENCES

[1] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? An empirical case study of Mozilla Firefox. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 179–188, June 2012.

[2] M. Mantyla, F. Khomh, B. Adams, E. Engstrom, and K. Petersen. On rapid releases and software testing. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 20–29, Sept 2013.

[3] R. Souza, C. Chavez, and R. Bittencourt. Rapid releases and patch backouts: A software analytics approach. *Software, IEEE*, 32(2):89–96, Mar 2015.

Volume 3 of Tiny Transactions on Computer Science

Souza *et al. Journal of Software Engineering Research and Development* (2015) 3:9
DOI 10.1186/s40411-015-0024-z

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

Patch rejection in Firefox: negative reviews, backouts, and issue reopening

Rodrigo RG Souza[1*], Christina FG Chavez[1] and Roberto A Bittencourt[2]

*Correspondence:
rodrigo@dcc.ufba.br
[1]Department of Computer Science, Federal University of Bahia, Salvador, Brazil
Full list of author information is available at the end of the article

Abstract

Background: Writing patches to fix bugs or implement new features is an important software development task, as it contributes to raise the quality of a software system. Not all patches are accepted in the first attempt, though. Patches can be rejected because of problems found during code review, automated testing, or manual testing. A high rejection rate, specially later in the lifecycle, may indicate problems with the software development process.
Our objective is to better understand the relationship among different forms of patch rejection and to characterize their frequency within a project. This paper describes one step towards this objective, by presenting an analysis of a large open source project, Firefox.

Method: In order to characterize patch rejection, we relied on issues and source code commits from over four years of the project's history. We computed monthly metrics on the occurrence of three indicators of patch rejection—negative code reviews, commit backouts, and bug reopening—and measured the time it takes both to submit a patch and to reject inappropriate patches.

Results: In Firefox, 20 % of the issues contain rejected patches. Negative reviews, backouts, and issue reopening are relatively independent events; in particular, about 70 % of issue reopenings are premature; 75 % of all inappropriate changes are rejected within four days.

Conclusions: Patch rejection is a frequent event, occurring multiple times a day. Given the relative independence of rejection types, existing studies that focus on one single rejection type fail to detect many rejections. Although inappropriate changes cause rework, they have little effect on the quality of released versions of Firefox.

Keywords: Release engineering; Mining software repositories; Empirical software engineering; Patch rejection

Introduction

According to Lehman et al. (1997), many software systems need to be constantly changed to remain useful, and the quality of such systems will be perceived as declining unless they are rigorously maintained. Therefore, for a high quality product, that satisfies users' needs, it is important to keep track of issues with the product, the patches that resolve those issues, and all verification steps during the lifecycle of a software release.

A patch goes through multiple stages before it is integrated into a release, depending on the specific development process employed in a team. In Mozilla Firefox, for example,

BIBLIOGRAPHY

ALMOSSAWI, A. *Investigating the architectural drivers of defects in open-source software systems: an empirical study of defects and reopened defects in GNOME*. Dissertação (Mestrado) — Massachusetts Institute of Technology, 2012.

AN, L.; KHOMH, F.; ADAMS, B. Supplementary bug fixes vs. re-opened bugs. In: *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. [S.l.: s.n.], 2014. p. 205–214.

ANBALAGAN, P.; VOUK, M. On predicting the time taken to correct bug reports in open source projects. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. [S.l.: s.n.], 2009. p. 523 –526. ISSN 1063-6773.

ANH, N. D.; CRUZES, D.; AYALA, C. P.; CONRADI, R. Impact of stakeholder type and collaboration on issue resolution time in oss projects. In: HISSAM, S. A.; RUSSO, B.; NETO, M. G. de M.; KON, F. (Ed.). *OSS*. [S.l.]: Springer, 2011. (IFIP Advances in Information and Communication Technology, v. 365), p. 1–16. ISBN 978-3-642-24417-9.

ANH, N. D.; CRUZES, D. S.; CONRADI, R.; AYALA, C. Empirical validation of human factors in predicting issue lead time in open source projects. In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. New York, NY, USA: ACM, 2011. (Promise '11), p. 13:1–13:10. ISBN 978-1-4503-0709-3.

ANTONIOL, G.; AYARI, K.; PENTA, M. D.; KHOMH, F.; GUÉHÉNEUC, Y.-G. Is it a bug or an enhancement?: a text-based approach to classify change requests. In: *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. New York, NY, USA: ACM, 2008. (CASCON '08), p. 23:304–23:318.

ARANDA, J.; VENOLIA, G. The secret life of bugs: Going past the errors and omissions in software repositories. In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 298–308. ISBN 978-1-4244-3453-4.

AYARI, K.; MESHKINFAM, P.; ANTONIOL, G.; PENTA, M. D. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In: *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. New York, NY, USA: ACM, 2007. (CASCON '07), p. 215–228.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: IEEE PRESS. *Proceedings of the 2013 international conference on software engineering*. [S.l.], 2013. p. 712–721.

BAKER, M. *Celebrating 10 years of Mozilla*. 2008. Available at ⟨http://www-archive.mozilla.org/mozilla-ten-year.html⟩.

BASILI, V. R.; BRIAND, L. C.; MELO, W. L. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 22, n. 10, p. 751–761, out. 1996. ISSN 0098-5589.

BASKERVILLE, R.; PRIES-HEJE, J. Short cycle time systems development. *Inf. Syst. J.*, v. 14, n. 3, p. 237–264, 2004.

BHATTACHARYA, P.; NEAMTIU, I. Bug-fix time prediction models: Can we do better? In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2011. (MSR '11), p. 207–210. ISBN 978-1-4503-0574-7.

BIRD, C.; BACHMANN, A.; AUNE, E.; DUFFY, J.; BERNSTEIN, A.; FILKOV, V.; DEVANBU, P. Fair and balanced?: bias in bug-fix datasets. In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009. (ESEC/FSE '09), p. 121–130. ISBN 978-1-60558-001-2.

BIRD, C.; GOURLEY, A.; DEVANBU, P. Detecting patch submission and acceptance in oss projects. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007. (MSR '07), p. 26–. ISBN 0-7695-2950-X.

BOEHM, B.; BASILI, V. R. Software defect reduction top 10 list. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 34, n. 1, p. 135–137, jan. 2001. ISSN 0018-9162.

BOUGIE, G.; TREUDE, C.; GERMAN, D.; STOREY, M.-A. A comparative exploration of freebsd bug lifetimes. In: *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. [S.l.: s.n.], 2010. p. 106 –109.

BUGZILLA. *Bug Fields*. 2015. Available at ⟨https://bugzilla.mozilla.org/page.cgi?id=fields.html⟩.

CAGLAYAN, B.; MISIRLI, A. T.; MIRANSKYY, A.; TURHAN, B.; BENER, A. Factors characterizing reopened issues: a case study. In: *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*. New York, NY, USA: ACM, 2012. (PROMISE '12), p. 1–10. ISBN 978-1-4503-1241-7.

CANFORA, G.; CECCARELLI, M.; CERULO, L.; PENTA, M. D. How long does a bug survive? an empirical study. In: *Proceedings of the 2011 18th Working Conference*

on Reverse Engineering. Washington, DC, USA: IEEE Computer Society, 2011. (WCRE '11), p. 191–200. ISBN 978-0-7695-4582-0.

CHACON, S. *Pro Git*. 1st. ed. Berkely, CA, USA: Apress, 2009. ISBN 1430218339, 9781430218333.

CLARK, S.; COLLIS, M.; BLAZE, M.; SMITH, J. M. Moving targets: Security and rapid-release in firefox. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2014. (CCS '14), p. 1256–1266. ISBN 978-1-4503-2957-6.

COCKBURN, A.; WILLIAMS, L. Agile software development: It's about feedback and change. *Computer*, IEEE Computer Society, v. 36, n. 6, p. 0039–43, 2003.

CZERWONKA, J.; GREILER, M.; TILFORD, J. Code reviews do not find bugs. how the current code review best practice slows us down. In: *37th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2015.

DALKEY, N.; HELMER, O. An Experimental Application of the DELPHI Method to the Use of Experts. *Management Science*, v. 9, n. 3, p. 458–467, April 1963.

D'AMBROS, M.; LANZA, M.; ROBBES, R. An extensive comparison of bug prediction approaches. In: *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. [S.l.: s.n.], 2010. p. 31–41. ISBN 978-1-4244-6803-4.

EADDY, M.; ZIMMERMANN, T.; SHERWOOD, K. D.; GARG, V.; MURPHY, G. C.; NAGAPPAN, N.; AHO, A. V. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 34, n. 4, p. 497–515, jul. 2008. ISSN 0098-5589.

FISCHER, M.; PINZGER, M.; GALL, H. Populating a release history database from version control and bug tracking systems. In: *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003. (ICSM '03), p. 23–. ISBN 0-7695-1905-9.

FOWLER, M. *Feature Toggle*. 2010. Available at ⟨http://martinfowler.com/bliki/FeatureToggle.html⟩.

FOWLER, M.; FOEMMEL, M. *Continuous integration*. 2005. Available at ⟨http://www.martinfowler.com/articles/continuousIntegration.html⟩.

GIGER, E.; PINZGER, M.; GALL, H. Predicting the fix time of bugs. In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. New York, NY, USA: ACM, 2010. (RSSE '10), p. 52–56. ISBN 978-1-60558-974-9.

HASSAN, A. E. The road ahead for mining software repositories. *Frontiers of Software Maintenance*, 2008.

HERZIG, K.; JUST, S.; ZELLER, A. It's not a bug, it's a feature: How misclassification impacts bug prediction. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 392–401. ISBN 978-1-4673-3076-3.

HOOIMEIJER, P.; WEIMER, W. Modeling bug report quality. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007. (ASE '07), p. 34–43. ISBN 978-1-59593-882-4.

JANZEN, D. S.; SAIEDIAN, H. Does test-driven development really improve software design quality? *Software, IEEE*, IEEE, v. 25, n. 2, p. 77–84, 2008.

JEONG, G.; KIM, S.; ZIMMERMANN, T.; YI, K. *Improving code review by predicting reviewers and acceptance of patches*. [S.l.], 2009. Technical report.

JONGYINDEE, A.; OHIRA, M.; IHARA, A.; MATSUMOTO, K.-I. Good or bad committers? a case study of committers' cautiousness and the consequences on the bug fixing process in the Eclipse project. In: *Proc. of the 2011 Joint Conf. of the 21st International Workshop on Softw. Measurement and the 6th International Conf. on Softw. Process and Product Measurement*. Washington, DC, USA: IEEE Computer Society, 2011. p. 116–125. ISBN 978-0-7695-4565-3.

JONO. *Everybody hates Firefox updates*. 2012. Available at ⟨http://evilbrainjono.net/blog?permalink=1094⟩.

KAGDI, H.; COLLARD, M. L.; MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, John Wiley & Sons, Inc., New York, NY, USA, v. 19, p. 77–131, March 2007. ISSN 1532-060X.

KALLIAMVAKOU, E.; DAMIAN, D.; BLINCOE, K.; SINGER, L.; GERMAN, D. M. Open source-style collaborative development practices in commercial projects using github. In: *The 37th International Conference of Software Engineering (ICSE 2015)*. [S.l.: s.n.], 2015.

KHOMH, F.; DHALIWAL, T.; ZOU, Y.; ADAMS, B. Do faster releases improve software quality? an empirical case study of mozilla firefox. In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2012. p. 179–188. ISBN 978-1-4673-1761-0.

KIM, S.; WHITEHEAD JR., E. J. How long did it take to fix bugs? In: *Proceedings of the 2006 international workshop on Mining software repositories*. New York, NY, USA: ACM, 2006. (MSR '06), p. 173–174. ISBN 1-59593-397-2.

KIM, S.; ZIMMERMANN, T.; JR., E. J. W.; ZELLER, A. Predicting faults from cached history. In: *Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 489–498. ISBN 0-7695-2828-7.

LAFORGE, A. *Release Early, Release Often*. 2010. Available at ⟨http://blog.chromium.org/2010/07/release-early-release-often.html⟩.

LEHMAN, M. M.; RAMIL, J. F.; WERNICK, P. D.; PERRY, D. E.; TURSKI, W. M. Metrics and laws of software evolution-the nineties view. In: IEEE. *Software Metrics Symposium, 1997. Proceedings., Fourth International*. [S.l.], 1997. p. 20–32.

MANTYLA, M.; KHOMH, F.; ADAMS, B.; ENGSTROM, E.; PETERSEN, K. On rapid releases and software testing. In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.: s.n.], 2013. p. 20–29. ISSN 1063-6773.

MARCUS, A.; POSHYVANYK, D.; FERENC, R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 34, n. 2, p. 287–300, mar. 2008. ISSN 0098-5589.

MOSER, R.; PEDRYCZ, W.; SUCCI, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE '08), p. 181–190. ISBN 978-1-60558-079-1.

MOZILLA. *Committing rules and responsibilities*. 2015. Available at ⟨https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Committing_Rules_and_Responsibilities⟩.

NAGAPPAN, N.; ZELLER, A.; ZIMMERMANN, T.; HERZIG, K.; MURPHY, B. Change bursts as defect predictors. In: *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2010. (ISSRE '10), p. 309–318. ISBN 978-0-7695-4255-3.

NGUYEN, T. H. D.; ADAMS, B.; HASSAN, A. E. A case study of bias in bug-fix datasets. In: *Proceedings of the 2010 17th Working Conference on Reverse Engineering*. [S.l.: s.n.], 2010. p. 259–268. ISBN 978-0-7695-4123-5.

NUROLAHZADE, M.; NASEHI, S. M.; KH, S. H.; RAWAL, S. 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*. [S.l.: s.n.], 2009. p. 9–18.

O'DUINN, J. *Farewell to Tinderbox, the world's 1st? 2nd? Continuous Integration server*. 2014. Available at ⟨http://oduinn.com/blog/2014/06/04/farewell-to-tinderbox/⟩.

OHIRA, M.; HASSAN, A.; OSAWA, N.; MATSUMOTO, K. The impact of bug management patterns on bug fixing: A case study of eclipse projects. In: *Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM2012)*. [S.l.: s.n.], 2012.

PANJER, L. D. Predicting eclipse bug lifetimes. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007. (MSR '07), p. 29–. ISBN 0-7695-2950-X.

PARK, J.; KIM, M.; RAY, B.; BAE, D.-H. An empirical study of supplementary bug fixes. In: *9th IEEE Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2012. p. 40–49. ISSN 2160-1852.

PLEWNIA, C.; DYCK, A.; LICHTER, H. On the influence of release engineering on software reputation. In: *2nd International Workshop on Release Engineering*. [S.l.: s.n.], 2014.

RIGBY, P. C.; GERMAN, D. M.; COWEN, L.; STOREY, M.-A. Peer review on open source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology*, p. 34, 2014.

RIGBY, P. C.; STOREY, M.-A. Understanding broadcast based peer review on open source software projects. In: ACM. *Proceedings of the 33rd International Conference on Software Engineering*. [S.l.], 2011. p. 541–550.

SHIHAB, E.; IHARA, A.; KAMEI, Y.; IBRAHIM, W. M.; OHIRA, M.; ADAMS, B.; HASSAN, A. E.; MATSUMOTO, K.-I. Predicting re-opened bugs: A case study on the eclipse project. In: *Proceedings of the 2010 17th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2010. (WCRE '10), p. 249–258. ISBN 978-0-7695-4123-5.

SHIHAB, E.; IHARA, A.; KAMEI, Y.; IBRAHIM, W.; OHIRA, M.; ADAMS, B.; HASSAN, A.; MATSUMOTO, K.-I. Studying re-opened bugs in open source software. *Empirical Software Engineering*, Springer Netherlands, p. 1–38, 2012. ISSN 1382-3256. 10.1007/s10664-012-9228-6.

SLAUGHTER, S. A.; HARTER, D. E.; KRISHNAN, M. S. Evaluating the cost of software quality. *Commun. ACM*, ACM, New York, NY, USA, v. 41, n. 8, p. 67–73, ago. 1998. ISSN 0001-0782.

ŚLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? In: *Proceedings of the 2005 international workshop on Mining software repositories*. New York, NY, USA: ACM, 2005. (MSR '05), p. 1–5. ISBN 1-59593-123-6.

SOUZA, R.; CHAVEZ, C. *Impact of the Four Eyes Principle on Bug Reopening: the Case of Eclipse*. [S.l.], 2011. Technical report.

SOUZA, R.; CHAVEZ, C. Characterizing verification of bug fixes in two open source IDEs. In: *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*. [S.l.: s.n.], 2012. p. 70–73.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. Patterns for cleaning up bug data. In: *Proc. of the 1st Workshop on Data Analysis Patterns in Softw. Engineering*. [S.l.]: IEEE, 2013.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. Patterns for extracting high level information from bug reports. In: *Proc. of the 1st Workshop on Data Analysis Patterns in Softw. Engineering*. [S.l.]: IEEE, 2013.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. On integration repositories, build sheriffs, and patch backouts. *Tiny Transactions on Computer Science (TinyToCS)*, v. 3, Mar 2015.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. Patch rejection in firefox: Negative reviews, backouts, and issue reopening. *Journal of Software Engineering Research and Development*, 2015. To appear.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. Rapid releases and patch backouts: A software analytics approach. *Software, IEEE*, v. 32, n. 2, p. 89–96, Mar 2015. ISSN 0740-7459.

SOUZA, R.; CHAVEZ, C. von F. G.; BITTENCOURT, R. A. Do rapid releases affect bug reopening? A case study of firefox. In: *2014 Brazilian Symposium on Software Engineering, Maceió, Brazil, September 28 - October 3, 2014*. [S.l.: s.n.], 2014. p. 31–40.

WANG, X. O.; BAIK, E.; DEVANBU, P. T. Operating system compatibility analysis of eclipse and netbeans based on bug data. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2011. (MSR '11), p. 230–233. ISBN 978-1-4503-0574-7.

WEISS, C.; PREMRAJ, R.; ZIMMERMANN, T.; ZELLER, A. How long will it take to fix this bug? In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007. (MSR '07), p. 1–. ISBN 0-7695-2950-X.

ZENG, H.; RINE, D. Estimation of software defects fix effort using neural networks. In: *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*. Washington, DC, USA: IEEE Computer Society, 2004. (COMPSAC '04), p. 20–21. ISBN 0-7695-2209-2-2.

ZHANG, F.; KHOMH, F.; ZOU, Y.; HASSAN, A. An empirical study on factors impacting bug fixing time. In: *Proc. of the 19th Working Conference on Reverse Engineering (WCRE)*. [S.l.: s.n.], 2012.

ZIMMERMANN, T.; NAGAPPAN, N.; GUO, P. J.; MURPHY, B. Characterizing and predicting which bugs get reopened. In: *Proceedings of the 2012 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE 2012), p. 1074–1083. ISBN 978-1-4673-1067-3.