Supporting Development Decisions with Software Analytics

by

Olga Baysal

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation, these include:

- Oleksii Kononenko,
- Dr. Michael W. Godfrey,
- Dr. Reid Holmes, and
- Dr. Ian Davis.

Abstract

Software practitioners make technical and business decisions based on the understanding they have of their software systems. This understanding is grounded in their own experiences, but can be augmented by studying various kinds of development artifacts, including source code, bug reports, version control meta-data, test cases, usage logs, etc. Unfortunately, the information contained in these artifacts is typically not organized in the way that is immediately useful to developers' everyday decision making needs. To handle the large volumes of data, many practitioners and researchers have turned to *analytics* — that is, the use of analysis, data, and systematic reasoning for making decisions.

The thesis of this dissertation is that by employing software analytics to various development tasks and activities, we can provide software practitioners better insights into their processes, systems, products, and users, to help them make more informed data-driven decisions. While quantitative analytics can help project managers understand the big picture of their systems, plan for its future, and monitor trends, qualitative analytics can enable developers to perform their daily tasks and activities more quickly by helping them better manage high volumes of information.

To support this thesis, we provide three different examples of employing software analytics. First, we show how analysis of real-world usage data can be used to assess user dynamic behaviour and adoption trends of a software system by revealing valuable information on how software systems are used in practice.

Second, we have created a lifecycle model that synthesizes knowledge from software development artifacts, such as reported issues, source code, discussions, community contributions, etc. Lifecycle models capture the dynamic nature of how various development artifacts change over time in an annotated graphical form that can be easily understood and communicated. We demonstrate how lifecycle models can be generated and present industrial case studies where we apply these models to assess the code review process of three different projects.

Third, we present a developer-centric approach to issue tracking that aims to reduce information overload and improve developers' situational awareness. Our approach is motivated by a grounded theory study of developer interviews, which suggests that customized views of a project's repositories that are tailored to developer-specific tasks can help developers better track their progress and understand the surrounding technical context of their working environments. We have created a model of the kinds of information elements that developers feel are essential in completing their daily tasks, and from this model we have developed a prototype tool organized around developer-specific customized dashboards.

The results of these three studies show that software analytics can inform evidence-based decisions related to user adoption of a software project, code review processes, and improved developers' awareness on their daily tasks and activities.

Acknowledgements

First, I would like to recognize my co-authors and collaborators on much of this work:

- Michael W. Godfrey (PhD Supervisor),
- Reid Holmes (PhD Supervisor),
- Ian Davis (SWAG member and researcher),
- Oleksii Kononenko (peer, friend and collaborator), and
- Robert Bowdidge (researcher at Google).

I am very grateful for meeting many wonderful people who encouraged, inspired, taught, criticized, or advised me to become a researcher I am today. I thank all of you for being very influential in my PhD life and beyond. Mike, your input into my career plans and research directions were most influential; you passed many of your values and skills to me that have shaped me as a researcher, these include: the importance of seeing the big picture; being humbled; taking a high road when dealing with human egos, failures and rejections; being honest yet constructive with feedback; being a good parent and knowing when to say "no" to always demanding work tasks. I am thankful for all your advice on many aspects of my everyday life - buying a house, having and raising a child, vacationing and travelling, and much more. Reid, thank you for stepping in and guiding my research during Mike's sabbatical leave; your fresh mind boosted my research progress and your everyday lab visits cured my procrastination problem. My PhD committee — Michele Lanza, Derek Rayside, and Frank Tip — for their valuable input.

I would also like to thank the entire community of researchers, as well as students and staff at the University of Waterloo. In particular, students, peers and friends – Sarah Nadi, Oleksii Kononenko, Hadi Hosseini, Georgia Kastidou, Karim Hamdan, John Champaign, Carol Fung, Lijie Zou – for sharing our PhD journeys (with both successes, failures, and challenges), conference travel, and personal experiences. Ian Davis for his remarkable technical expertise, big heart and willingness to help others. Margaret Towell and Wendy Rush for their tremendous support, assistance and help with administration. Suzanne Safayeni for her guidance and discussions about everyday life and career choices. Martin Best, Mike Hoye, and Kyle Lahnakoski from the Mozilla Corporation for their help in recruiting developers and providing environment for conducting interviews. Mozilla developers, especially Mike Conley, who participated in our study for their time and feedback on the developer dashboards. Rick Byers and Robert Kroeger from Google Waterloo for their valuable input to the WebKit study. Greg Wilson and Jim Cordy, for sharing your own stories of making career choices and dealing with fears and challenges in life, and for being so inspirational personalities. Massimiliano Di Penta and Giuliano Antoniol for planting

seeds of knowledge of statistics that I will continue to harvest, Andy Marcus for advancing my skills in information retrieval techniques, Bram Adams for introducing machine learning, Andrew Begel for teaching how to do qualitative research the right way and for giving advice on conducting research studies. Rocco Oliveto, for providing valuable advice on my research work and career plans. Michele Lanza, for having interesting conversations about academic life with all its charms and challenges, making me face my demons and shaking my beliefs.

I would also like to acknowledge those who awarded me with scholarships and provided financial support through my studies:

- IBM for funding the Minervan project,
- NSERC for awarding me with a NSERC PGS-D scholarship,
- University of Waterloo for honouring me with the Graduate Entrance scholarship and the President's scholarship, and
- David Cheriton and the David R. Cheriton School of Computer Science for the Cheriton scholarship I received.

And most important, I'd like to thank my family for their support and love. My loving husband Timuçin, this journey would not have been possible without your love, your sense of humour, your support and your talent of keeping my spirits high. My beautiful daughter Dilara, for just being around and making me forget work stress with your smiles, giggles, hugs and tens of awesome drawings. My mom and dad, sister and brother, thank you for teaching me to be a better person every day. Anne, I am very grateful to have you as my mother-in-law and thankful for all your help around the house when I was travelling or working on my thesis. Baba, thank you for sending angels to watch over and guide me.

> "A bird doesn't sing because it has an answer, it sings because it has a song."
>
> — Maya Angelou

To my family with love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chapter Organization Section 1.1 describes the role of decisions in software development. Section 1.2 provides our definition of software analytics. Section 1.3 describes the field of mining software repositories and how software analytics fits in this area. Section 1.4 illustrates three key problems we are addressing in this dissertation. Section 1.5 highlights main contributions of this dissertation. Section 1.6 describes the structure and organization of this dissertation.

1.1 Software Development and Decision Making

Software development projects generate impressive amounts of data. For example, source code, check-ins, bug reports, work items and test executions are recorded in software repositories such as version control systems (Git, Subversion, Mercurial, CVS) and issue-tracking systems (Bugzilla, JIRA, Trac), and the information about user experiences of interacting with software is typically stored in log files.

Successful software projects require that a variety of stakeholders continuously answer a breadth of questions. For example, managers are trying to meet their company's business goals, marketing and sales teams to increase company's revenue, release engineers to meet product deadlines, and developers to build error-free systems. Software practitioners make decisions based on the understanding they have of their software systems; this understanding is often grounded in their own experiences and intuition. Decisions based on previous experiences or intuition might work well on some occasions but often carry risk [160].

While vast quantities of information are generated during development, very little is organized, stored and presented in a way that is immediately useful to developers and managers to support their decision-making. But at the same time, the information contained in various development

artifacts including source code, bug report data, commit history, test suits, documentation, etc. could provide valuable insights about software project. This thesis investigates how the data from these artifacts can be used to support development decisions of various stakeholders. To handle the large volumes of data repositories, many practitioners and researchers have turned to *analytics* — that is, the use of analysis, data, and systematic reasoning for making decisions [177].

1.2 Software Analytics

Many prominent tech companies including IBM, Microsoft, and Google have embraced an analytics-driven culture to help improve their decision making. Embedding analytics into an organization's culture can enhance competitive advantage [110]. Analytics are typically used in marketing to better reach and understand customers. The application of analytics to software data is becoming more popular. A recent issue of *IEEE Software* [119] is entirely dedicated to the topic of software analytics, which is defined as "analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions" [31].

Analytics is the science of gathering, preprocessing, transforming and modelling raw data with the purpose of highlighting useful information and drawing conclusions from it. Software analytics [31, 82, 177, 178] provide means to empower software development stakeholders to gain and share insight from their data to make better decisions.

Software analytics can be broken down into four areas (as shown in Figure 1.1): data mining, statistical analysis, interpretation and leveraging. *Data mining* is the process of collecting and extracting information from a data source and pre-processing the data (data cleanup, normalization, transformation, feature extraction and selection, etc.) for further use. *Statistics*, in this context, mean the process of exploring the data and performing analysis. *Interpretation* is the process of drawing conclusions from the data including presentation and visualization of findings via charts and graphs. Finally, the *Leveraging* phase focuses on implications of the data such as building applications or predictive models.

Analytics are used to leverage large volumes of data from multiple sources to help practitioners make *informed* decisions about their projects. Software analytics provide practitioners with the fact-based views about their projects so they can answer questions specific to its development [82]. For example, by mining various software repositories, we can analyze and measure development activity, explore the state of the project, estimate and predict efforts needed to resolve defects, infer knowledge about a developer's expertise, or predict future software qualities.

In this thesis, we demonstrate the use of different kinds of analytics. Different kinds of analytics might require choosing different methods and techniques. For example, employing

Figure 1.1: Software Analytics

analytics to live versus historical data can become challenging because access to the data might be available only to the internal teams due to security concerns.

Analytic approaches strive to provide actionable real-time insights; these insights are often presented as quantitative multi-dimensional reports. However, analytic approaches can be both quantitative and qualitative in nature. A common myth with qualitative methods is that they are less useful than quantitative methods, in this thesis we demonstrate that this is not the case [14]. While quantitative analytics can highlight high-level trends of the data, qualitative analytics enable real-time decision making for tasks that are lower level and more frequent.

Most analytics approaches focus on quantitative historical analysis, often using chart-like dashboards (see examples in Figure 1.2). These dashboards are often geared towards helping project managers monitor and measure performance, for example, "to provide module owners with a reliable tool with which to more effective manage their part of the community" [61].

David Eaves, a member of the Mozilla Metrics team who worked on community contribution dashboards (a quantitative analytic tool), states: "we wanted to create a dashboard that would allow us to identify some broader trends in the Mozilla Community, as well as provide tangible, useful data to Module Owners particularly around identifying contributors who may be participating less frequently." Figure 1.2 demonstrates quantitatively-oriented dashboards for

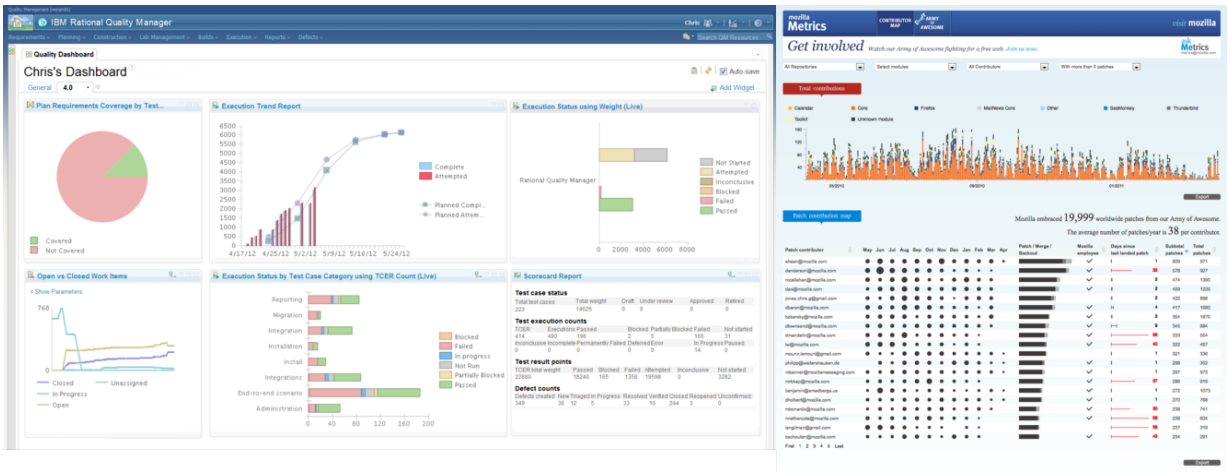


Figure 1.2: Quantitative dashboards: IBM Rational Quality Manager (left) and Mozilla Metrics (right).

the IBM Jazz development environment[1] (left) and the Mozilla Metrics project[2] (right). The Jazz quality dashboard provides high-level charts describing various project statistics, e.g., how long issues have been open and closed (bottom left), and how various test cases execute (bottom middle). The Mozilla dashboard provides a quantitative description of various community contributions, including a graph at the top showing where these contributions were made and a table that describes per-developer statistics below. While both of these dashboards effectively convey information about specific aspects of the project, the information would likely be more applicable to managers than to individual developers as they perform their daily tasks.

Our own study (described in Chapter 5.2) suggests that current dashboards poorly support developers' day-to-day development tasks. Developers require a different kind of dashboard to improve their situational awareness of their tasks. Ehsan Akhgari, a Mozilla developer, says, "What I really want is a dashboard to show the important stuff I need to see — review status on the dashboard, assigned bug with a new comment or change since the last time I looked at it, bugs that have a certain status could go on the dashboard, etc." [21].

By definition, a qualitative property is one that is described rather than measured. While quantitative dashboards provide statistical summaries of various development metrics, qualitative dashboards emphasize the attributes and relationships of a set of artifacts of interest to a developer. Thus, qualitative dashboards provide developers with the means to define, organize, and monitor their personal tasks and activities. Unlike quantitative dashboards that address developer questions such as "How many bugs are pending on me?" or "How well did I perform last

[1] https://jazz.net/products/rational-quality-manager/
[2] https://metrics.mozilla.com/

quarter?", qualitative dashboards provide insights into the specific items developers are working on: "You look at the bug and you think, who has the ball? What do we do next?" and "What has changed since the last time I have looked at [this bug]?". Developers want to be able to get information about their issues based on what has changed since the last time they looked at them. Being able to keep abreast of the volume of changes taking place on active development teams can be challenging; by helping developers focus on the evolution of their issues, and those they are interested in, they can better prioritize their own tasks.

1.3 Relation to Mining Software Repositories

Mining software repositories (MSR) research [102] aims to extract information from the various artifacts (e.g., source code, bug reports, commit history, test suits, documentation) produced during the evolution of a software system and inferring the relationships between them. MSR researchers examine these artifacts with the goal of uncovering meaningful relationships and trends among them. Software analytics is a sub-field of MSR concerning the discovery of the patterns in the data. Data collection, analysis and transformation into knowledge can be used to make improvements and changes to the current software processes or practices.

While software analytics provide a means to monitor and reason about a project, tools and techniques for mining software repositories serve as mechanics for extracting the data from various sources and identifying useful and interesting patterns for analytics to happen. Most analytics approaches focus on quantitative historical analysis and thus are often geared towards project managers. Such analytics poorly support developers' day-to-day development tasks such as resolving issues, submitting a patch for a review, conducting code reviews, etc. Our goal is to support developers with task-specific decision making and improved awareness of their working context.

1.4 Research Problems

In this dissertation, we provide three examples of software analytics:

1. **Understanding user adoption of software systems**
   The amount of usage data produced across the globe has been increasing exponentially and is likely to continue growing at an accelerating rate for the foreseeable future [148]. At organizations across all industries, servers are overflowing with usage logs, message streams, transaction records, sensor data, etc. Analysis of these huge collections of data can create significant value for the world economy by enhancing productivity, increasing efficiency and delivering more value to consumers [51]. Usage logs, an artifact that has yet received little

attention, stores valuable information about users and their behaviour related to adoption and use of software systems. In this work, we explore how usage data extracted from these logs can be unified with product release history to study questions that concern both users' detailed dynamic behaviour as well as broad adoption trends across different deployment environments.

2. **Managing community contributions**
   Improving community contribution and engagement is one of the challenges that many open source projects face. Code review is the practice of inspecting code modifications (submitted as patches) before they are committed to the project's version control repository. Code review is particularly important for open source software development since contributions — in the form of bug fixes, new features, documentation, etc. — may come not only from core developers but also from members of the greater user community [7, 134, 141]. Open source projects evolve through community participation and engagement. However, the review process can promote or discourage these contributions. In this work, we investigate whether contributions from various developers are being reviewed equally regardless of the developers' participation in a project (frequent vs. occasional). We also examine what factors might affect the outcome and the time it takes for a review to happen.

3. **Personalizing issue-tracking systems**
   Issue-tracking systems are used by developers, managers, end-users to submit bug reports and feature requests, and are also used for other tasks such as collaborative bug fixing, project management, communication and discussion, code reviews, and history tracking. Most issue trackers are designed around the central metaphor of the "issue" (bug, defect, ticket, feature, etc.), and thus, lack support of providing developers with a global view on the tasks they are working on or the activities of others. As a result, developers often find themselves trying to identify the status of a bug, as well as trying to track their own tasks. In this work, we explore how to improve issue trackers by making them more developer-centric. Developers can be provided with a personalized landing page that can have public and private information and serve as a heads up display showing them their own tailored view of the issue-tracking system.

By utilizing software analytics based on MSR-like techniques and methods, we aim to provide evidence-based answers on some of the questions stakeholders ask about their projects including topics related to:

1. **User adoption and experience**
   How are our software systems used? Who are our users? How do development characteristics affect deployment, adoption, and usage?

2. **Contributor experience and engagement**
   Who contributes to our project? Does our practice of assessing community contributions encourage future participation? How can we improve community engagement on the project?

3. **Developer productivity and awareness of their working context**
   Do developers have a good global understanding of the issues they are working on? How can we improve our tools to help developers gain and maintain ongoing awareness on the project, as well as track progress of their own activities?

1.5 Thesis Statement and Contributions

The thesis of this dissertation is that by employing both quantitative and qualitative analytics to software development projects, we can support practitioners by offering data that can provide better insights into processes, systems, products, and users and support more informed decision making. Quantitative analytics can help project managers understand the big picture, plan for future, and monitor trends by performing data exploration and analysis. Qualitative analytics can enable developers to perform their daily tasks and activities more quickly and to help them manage high volumes of information.

The main contributions of this dissertation include:

- a model of extracting usage data from the log files to measure user adoption [13];
- a demonstration of how analysis of real-world usage data can provide insight into user dynamic behaviour and adoption trends across various deployment environment [13];
- a data analysis pattern called *artifact lifecycle model* for synthesizing knowledge from software development artifacts [18];
- evidence of the effectiveness of patch lifecycle models in providing actionable insights into the contribution management practice [17, 19];
- evidence that organizational and personal factors influence review timeliness and outcome on the projects featuring both competition and collaboration [19];
- a categorization of the key issue-tracking design aspects to consider when implementing next-generation tools [12];
- evidence that qualitative analytics are much needed for software developers [14];
- the developer dash tool that allows developers to explore their daily task items and keep abreast of the changes [15, 107]; and,

- evidence that our personalized dashboard-like approach to issue tracking can reduce the flood of irrelevant bug email by over 99% [16], increase developer's awareness on the issues they are working on and also improve support for their individual tasks [15].

1.6 Organization

This dissertation has been written to maximize the readability of the main text by deferring many of the less important details in the appendices (see Appendix A through Appendix C). The main text of the dissertation contains references to the relevant appendix that includes detailed supporting materials.

Chapter 1.1 starts with example of decisions various stakeholders make and our explanation of why and how software analytics can help them in guiding their daily decision-making. We also state the thesis of this dissertation and describe three problems used to validate it.

Chapter 2 describes existing research in the area of mining software repositories and data analysis that have inspired much of the work within this dissertation.

Chapter 3 presents our work on studying users and how analysis of real-world usage data can reveal valuable information on how software systems are used in practice.

Chapter 4 proposes artifact lifecycle models to synthesize the knowledge from the development artifacts and shows how these models can be used to evaluate code review of three industrial projects. In this chapter, we also investigate the influence of various factors on code review time and outcome by presenting two empirical studies.

Chapter 5 presents our case of applying qualitative analytics to help developers overcome challenges related to information overload. Our prototype of the developer dash was based on the results of the qualitative study of how developers interact and use current issue-tracking systems (Section 5.2). Based on the results of this study we derived a model (Section 5.3) that addresses the key challenges of issue-tracking systems by making them more developer centric; the model was then implemented as a prototype tool. Our tool for supporting developers' daily activities and tasks was iteratively developed and evaluated both qualitatively and quantitatively. The evaluations are described along with the prototype and tool in Section 5.4.1.

Chapter 6 includes a discussion and an overview of future work.

Chapter 7 concludes the dissertation by summarizing its main contributions.

Chapter 2

Related Work

Our research focuses on methods, tools, and techniques to capture, process, analyze, and leverage the information from the various kinds of software artifacts. Research efforts relevant to this dissertation have been categorized into two primary categories: *mining software repositories*, the field and the associated methods and tools that are focused on extracting information from the development repositories and artifacts (Section 2.1), and *data analysis*, the approaches and techniques to gather, pre-process and analyze the data from these artifacts (Section 2.2). This chapter outlines relevant related work and differentiates the research in this dissertation from previous research efforts.

Chapter Organization Section 2.1 describes existing research in the field of mining software repositories that relates to the problems we address in this dissertation. Section 2.2 presents the main approaches and techniques we have used in this dissertation including statistics, machine learning, natural language processing, information retrieval, and grounded theory. Section 2.3 summarizes the related research.

2.1 Mining Software Repositories

This research heavily relies on the field of mining software repositories (MSR) [76, 79, 80, 81, 82, 102, 174, 175]. Mining software repositories research aims at extracting information from various artifacts (e.g., source code, bug reports, commit history, test suits, documentation) that are produced during the evolution of a software system. The knowledge and the behaviours are discovered by studying these artifacts and uncovering meaningful relationships and trends among them.

While software analytics provide means to monitor and reason about a project, tools, and techniques for mining software repositories serve as mechanics for extracting the data from various sources and identifying useful and interesting patterns for analytics to happen. For example, by inferring links between various artifacts such as version control systems, bug repositories, electronic communication and online documentation, a developer is provided with a recommendation of an artifact that may be relevant to the task being performed [164]. To help with a bug fixing task, the developer is presented with the code modification that was made to resolve a similar bug in the past.

A number of research efforts have demonstrated the value of software analytics in mobile application development and analysis [78, 123]. Harman et al. have introduced the "App store MSR" approach — app store mining and analysis — for evaluating apps' technical, customer and business aspects. Minelli and Lanza have studied the usefulness of software analytics in [123] and developed a web-based software analytics platform for mobile applications [122]. Analytics has been also successfully applied to study video game development [94, 132].

Some studies have already showed how software analytics are applied in practice [48, 177, 178]. Microsoft has developed CODEMINE — a software development data analytics platform for collecting and analyzing engineering process data, its constraints, and organizational and technical choices [48].

2.1.1 Usage Mining

The problem of understanding user adoption of a system is related to the web usage mining research. Web usage mining applies data mining techniques to discover usage patterns of web data. Web usage mining research provides a number of taxonomies summarizing existing research efforts in the areas, as well as various commercial offerings [40, 153].

Mobasher et al. [124] discussed web usage mining including sources and types of data such as web server application logs. They propose four primary groups of data sources: usage, content, structure, and user data. The key elements of web usage data pre-processing are data cleaning, pageview identification, user identification, session identification, episode identification, and the integration of clickstream data with other data sources such as content or semantic information.

Empirical software engineering research has focused on mining software development data (source code, electronic communication, defect data, requirements documentation, etc.). Relatively little work has been done on mining usage data. El-Ramly and Stroulia [62] mined software usage data to support re-engineering and program comprehension. They studied system-user interaction data that contained temporal sequences of user-generated events, and they developed a process for mining interaction patterns and applied it to legacy and web-based systems. The discovered patterns were used for user interface re-engineering and personalization. While also

mining web logs, they examined only user navigation activities on a web site to provide recommendations on other potential places to visit, "a user who visited link A also visited link B". In our work, we explored a number of questions related to users' detailed dynamic behaviour and adoption trends across various deployment environments. Li et al. [112] investigated how usage characteristics relate to field quality and how usage characteristics differ between beta and post-releases. They analyzed anonymous failure and usage data from millions of pre-release and post-release Windows machines.

In our previous work [11], we examined development artifacts — release histories, bug reporting and fixing data, as well as usage data — of the Firefox and Chrome web browsers. In that study, two distinct profiles emerged: Firefox, as the older and established system, and Chrome, as the new and fast evolving system. When analyzing the usage data, we focused on only the difference in adoption trends and whether the volume of defects affects popularity of a browser. Figure 2.1 depicts observed trends in user adoption of the two browsers. In this dissertation, we plan to take a more detailed look at the usage data by studying characteristics of the user populations of the browsers.

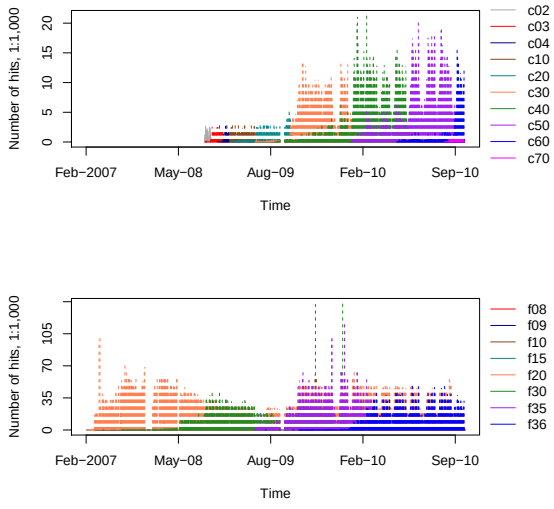


Figure 2.1: User Adoption Trends for Chrome (top) and Firefox (bottom).

Google Research performed a study on comparing update mechanisms of web browsers [59]. Their work investigates the effectiveness of web browser update mechanisms in securing end-users from various vulnerabilities. They performed a global scale measurement of update effectiveness comparing update strategies of five different web browsers: Google Chrome, Mozilla Firefox,

Opera, Apple Safari, and MS Internet Explorer. By tracking the usage shares over three weeks after a new release, they determined how fast users update to the latest version and compared the update performance between different releases of the same and other browsers. They applied a similar approach of parsing user-agent strings to determine the browser's name and version number. They evaluated the approach on the data obtained from Google web servers distributed all over the world. Unlike the Google study that investigates updates within the same major version of various web browsers, we studied major releases of the web browsers. We realize that our data is several orders of magnitude smaller than the Google data. However, we address different research questions related to the characteristics of user populations and looked at broader analyses than just update speed.

We are interested in studying user characteristics across an entire product release history and their relation to the software adoption and actual usage. In particular, we plan to mine server log files for the purpose of extracting usage information and exploring how usage data can be unified with product release history to study questions that concern both users' detailed dynamic behaviour, as well as broad adoption trends across different deployment environments.

Summary While usage data collected by organizations continues to grow, little of this data is analyzed to support development decisions such as "Should we provide support for the Linux platform?", "Should we invest in internationalization and localization features?". We seek to mine usage logs and extract insights on the user data related to the topics of user adoption and deployment of software systems.

2.1.2 Community Engagement, Contribution and Collaboration Management

Community engagement and contribution management are one of the main challenges open source projects face. Poor management can limit the size and potential of the community. On the contrary, making it easier for people to cooperate, collaborate, and experiment increases the community's capacity and chances for success.

Contribution management practices including code review, a process of inspecting contributions from the community members, have been previously studied by a number of researchers. Mockus et al. [125] were one of the first researchers who studied OSS development. By studying the development process of the Apache project, they identified the key characteristics of the OSS development, including heavy reliance on volunteers, self-assignment of tasks, lack of formal requirement or deliverables, etc. Rigby and German [140] presented a first study that investigated the code review processes in open source projects. They compared the code review processes of four open source projects: GCC, Linux, Mozilla, and Apache. They discovered a number of review patterns and performed a quantitative analysis of the review process of the Apache project.

Later, Rigby and Storey [142] studied mechanisms and behaviours that facilitate peer review in an open source setting.

Much of the existing research proposes various models of contribution management process in OSS projects [7, 22, 149], provides recommendations on managing contribution process [22, 149], measures parameters related to the patch evolution process [134], studies developer and reviewer behaviour [134] or studies the process itself [167].

Weissgerber et al. [167] performed data mining on email archives of two open source projects to study patch contributions. They found that the probability of a patch being accepted is about 40% and that smaller patches have higher chance of being accepted than larger ones. They also reported that if patches are accepted, they are normally accepted quickly (61% of patches are accepted within three days).

Asundi and Jayant [7] looked at the process and methodologies of free/libre/open source software (FLOSS) development. They examined the process of patch review as a proxy for the extent of code review process in FLOSS projects. They presented a quantitative analysis of the email archives of five FLOSS projects to characterize the patch review process. Their findings suggested that while the patch review process varies from project to project, the core developers play a vital role in ensuring the quality of the patches that are ultimately accepted. They defined core-group members as those who are listed on the project's development pages. We observed that for the Mozilla project, some official employees submit very few patches, while developers outside of the core development team provide frequent contributions. Therefore, we employed a different definition of core and casual contributors. In spite of the differences in definitions, our study confirms some of the results reported by Asundi and Jayant [7]. In particular, we also found that core contributors account for a greater proportion of submitted patches and that patches from casual contributors are more likely to be left unreviewed.

Sethanandha et al. [149] proposed a conceptual model of OSS contribution process. They presented the key practices for patch creation, publication, discovery, review, and application and also offered some recommendations on managing contribution process. Bettenburg et al. [22] performed an empirical study on the contribution management processes of two open source software ecosystems, Android and Linux. They developed a conceptual model of contribution management based on the analysis of seven OSS systems. They compared the code review processes of Android and Linux and offered some recommendations for practitioners in establishing effective contribution management practices. While Sethanandha et al. [149] and Bettenburg et al. [22] aimed at developing models of contribution management process, we focus on formalizing the lifecycle of a patch.

Nurolahzade et al. [134] examined the patch evolution process of the Mozilla development community. They quantitatively measured parameters related to the process, explained inner-workings of the process, and identified a few patterns pertaining to developer and reviewer behaviour including "patchy-patcher" and "merciful reviewer". While also studying Mozilla Firefox

and exploring patch review process, we modelled the lifecycle of patches rather than the patch evolution process. We also investigated whether the code review is affected by the contributor's participation on a project.

Jiang et al. [99] studied the relation of patch characteristics with the probability of patch acceptance and the time taken for patches to be integrated into the codebase on the example of the Linux kernel. They found that patch acceptance is affected by the developer experience, patch maturity and prior subsystem churn, while reviewing time is impacted by submission time, the number of affected subsystems, the number of suggested reviewers and developer experience.

Summary Community engagement correlates with the success of the open source project. Much of the research on contribution management attempts to develop models of the process, explain and improve existing practices. Previous research suggests that while the patch review process varies from project to project, the core developers play the vital role in ensuring the quality of the patches that get in. We investigate the impact of other potential factors that might determine the code review timeliness and outcome (Section 4). In addition, most of the related studies on code review perform mining on a project's commit history and thus are not able to reason about negative feedback and rejection interval. We address these limitations by extracting information from the issue-tracking and code review systems and providing a more comprehensive view of the code review process. In Section 4.2 we study patch contribution processes with the goal of formalizing the lifecycle of a patch to be able to discover the knowledge about the process and the developer dynamics to support decisions related to the questions: "Should we invest in having our own reviewers on a particular component?", "Do we need to improve our turnaround time to prevent loss of valuable contributions?".

2.1.3 Issue-Tracking Systems

Issue tracking is a key element of any software development project. Issue-tracking tools allow community members to submit bug reports, feature requests, documentation, crash reports, and other development-related artifacts. There are many issue-tracking systems available, both commercial and open-source ones. Section 2.1.3.1 describes and compares popular issue-tracking systems. Section 2.1.3.2 presents relevant research efforts in addressing various limitations of the existing issue-tracking systems. And Sections 2.1.3.3 discusses research around increasing developer awareness in global software development.

2.1.3.1 Survey of Popular Issue-Tracking Systems

Existing research offers a number of surveys about issue-tracking systems [97, 150, 163]. These surveys often compare issue trackers based on their functionality or features including installation,

user interface, technology, security, integration with other tools, accessibility and extensibility. This section provides a brief survey of the five well known issue-tracking systems; we describe their main features, how they support the key tasks of developers, and their main limitations.

Bugzilla [69] is one of the most well-known open-source issue-tracking tools. It is used on many open source projects such as Mozilla, Eclipse, and many Linux distributions, and is well-adopted by other large open source projects.

Bugzilla allows managing multiple products, optionally grouping related projects into "Classifications". A number of components can be defined for a particular product. Bugzilla supports project management and quality assurance tasks by providing functionality of defining versions, development milestones, and release versions.

Bugzilla users can search for existing issues using either a simple and convenient keyword-based search (with optional filtering by product or by status), or by using the more sophisticated Advanced Search, where users can filter on virtually any field in the database (see Figure 2.2). Bugzilla stores detailed metadata on issues including details such as issue's summary, component, severity, product version number, platform, a target milestone, etc. Bugzilla notifies users of any new issues or issue updates via e-mail.

Bugzilla provides some reporting and charting features. Users can generate tables, simple line graphs, bar or pie charts using *Tabular* and *Graphical* reports. However, Bugzilla is complicated and its interface is perceived as being difficult to use.

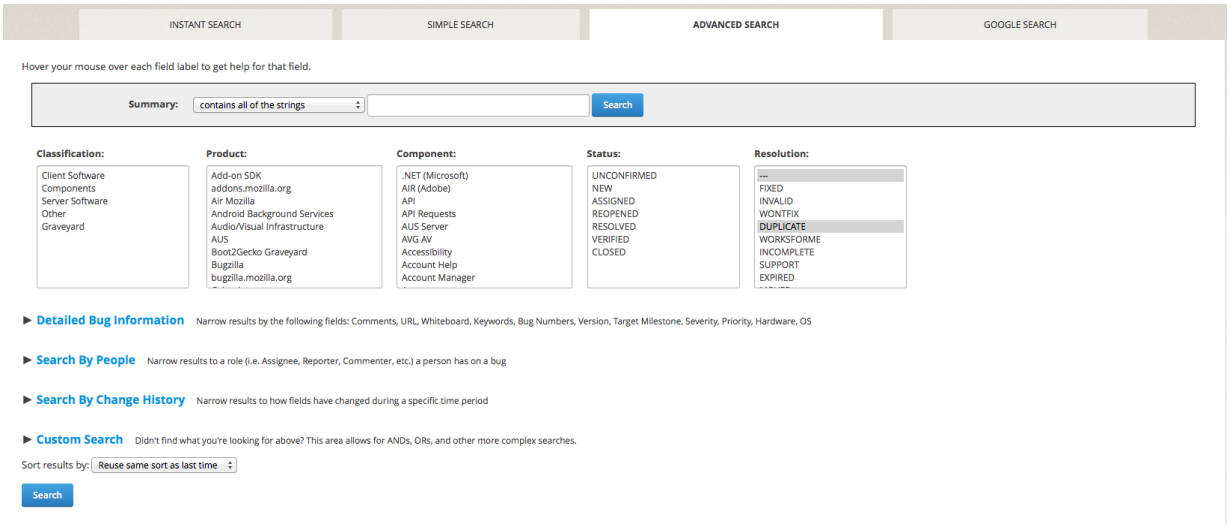


Figure 2.2: Bugzilla issue-tracking system.

Bugzilla has a new feature that offers its users a custom view of their Bugzilla profile. Figure 2.3 presents Bugzilla's My Dashboard as a screenshot taken by the user Mike Conley (a

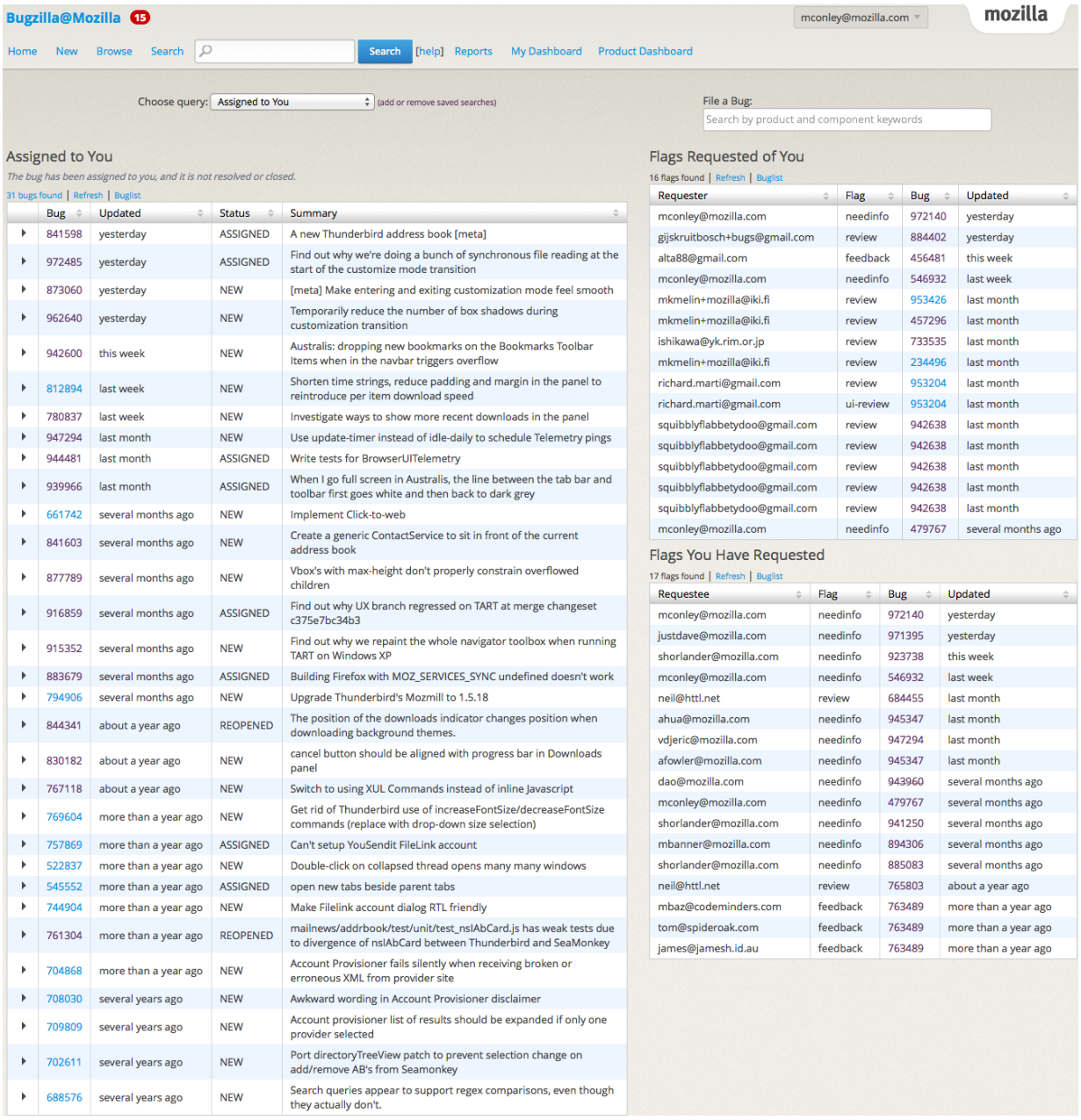


Figure 2.3: Screenshot representing Mozilla developer Mike Conley's My Dashboard in Bugzilla.

Mozilla developer). My Dashboard allows developers to select a query such as issues "*Assigned to You*", "*New Reported by You*", "*In Progress Reported by You*", "*You are CC'd On*", and "*You are QA Contact*". For any of these queries, it displays flags requested by/from a developer on the right-hand side of the page. However, the dashboard presents one query at a time, for example only Assigned issues; thus, developers need to execute a query every time they need to track various tasks. Bugzilla's performance is another main reason of why My Dashboard is ill adopted by the developers (we discuss this in Chapter 5).

Trac [152] is an enhanced wiki and issue-tracking system for software development projects. Trac is in fact a wiki — any page can be modified by users with appropriate rights. The most notable feature of Trac is its close integration with Subversion. The wiki syntax used in Trac lets you reference Subversion revisions, change-sets and even files. For example, a comment in an issue can contain a hyperlink to a particular Subversion change-set. A special syntax allows Trac users to add direct links to various objects. For example, `#925` refers to the ticket No.925, and `[2678]` refers to the changeset (or revision) No.2678. One can browse the source code repository directly from within Trac, displaying both the actual source code and the modification history.

Tickets as a central project management elements in Trac can be used for project tasks, feature requests, bug reports. All tickets can be edited, annotated, assigned, prioritized and discussed at any time. The *Trac Reports* module provides a simple reporting facility to present information about tickets in the Trac database. It relies on standard SQL `SELECT` statements for custom report definition. For example, the following query returns all active tickets, sorted by priority and time:

```
SELECT id AS ticket, status, severity, priority, owner, time, summary
FROM ticket
WHERE status IN ('new', 'assigned', 'reopened')
ORDER BY priority, time.
```

The *View Tickets* view offers a number of predefined reports such as All Active Tickets, My Active Tickets, Active Tickets by Milestone, Assigned Tickets by Owner (shown in Figure 2.4). In addition to reports, Trac provides support for *custom ticket queries*, used to display lists of tickets meeting a specified set of criteria. Trac provides several views that make daily issue-management activities easier, including navigating through the project, going from tickets to revisions to source code, etc. The *Timeline* view presents all activity on the project, including activity involving tickets and changes to the source code repository. The *Roadmap* view allows users to track project progress through milestones. It also provides a graphical view of the number of tickets closed compared to the total number of tickets for each milestone. While Trac offers a full-text search feature to search tickets, as well as wiki and changesets (revision history), it lacks any dashboard-like functionality, there are no graphs or charts, nor ability for Trac users to track ticket data over time. The information filtering can only be accomplished via custom report queries described above.

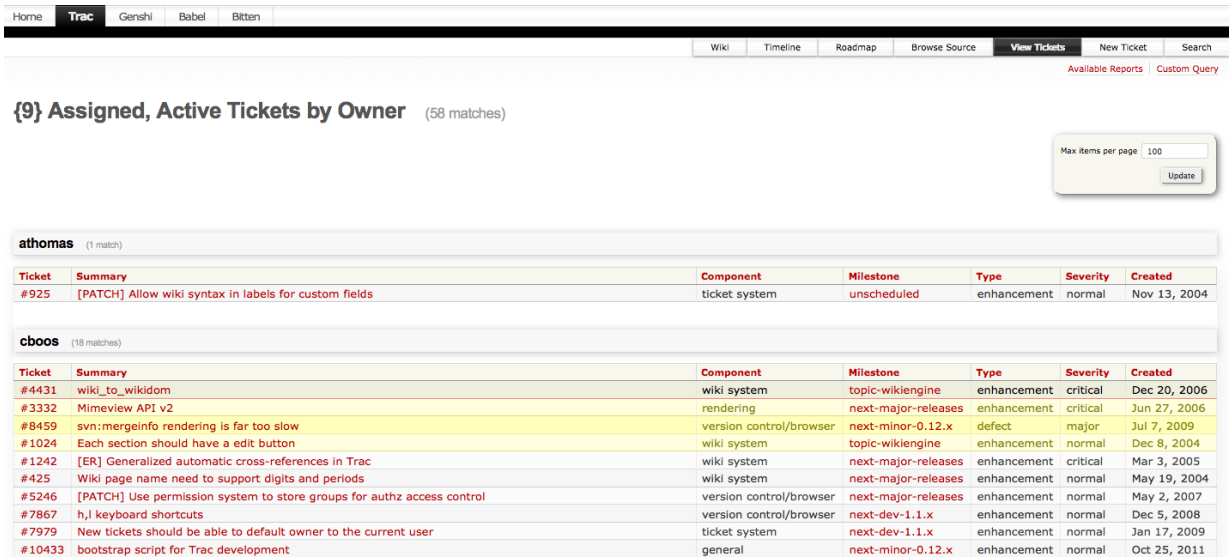


Figure 2.4: The *View Tickets* tab in Trac (from the Trac website [152]).

JIRA [9] is a widely used and well-regarded commercial issue-management tool. The *Issues* view organizes issues according to their status such as "All issues", "Updated recently", "Outstanding", etc. (as shown in Figure 2.5). *Road Map*, *Change Log* and *Versions* views allow developers to manage versions and product releases. One can define versions, track version releases and release dates, and generate release notes for a particular version with the list of all fixed and unresolved bugs for that release. JIRA provides numerous graphical *Reports* designed to give its users a quick overview of the project's current status: the list of issues assigned to a developer, "in-progress" issues, the number of open issues grouped by priority, etc. The JIRA's search functionality allows to perform full-text searches with filtering on key fields such as issue type, status, affected or fix versions, reporter, assignee, and priorities, as well as by date. The JIRA search is simpler and less cluttered with metadata than the Bugzilla's Advanced Search.

Similar to Bugzilla, developers can "watch" issues and receive email notifications about any updates on the issues of their interests. Jira's interface is more intuitive than Bugzilla allowing developers to collaborate on the team, organize issues, assign work, and follow team activity. JIRA integrates with source control programs such as Subversion, CVS, Git, Mercurial, and Perforce. JIRA offer various wallboards and dashboards to keep track of issues, ideas, tasks, stories and activities on the project. An example of the JIRA's dashboard is shown in Figure 2.6. It presents a summary of the latest changes on the project including current releases, issues and their status updates, recent activity of the developers, etc.

Unlike Bugzilla or Trac, JIRA is a commercial tool offering more advanced dashboard and re-

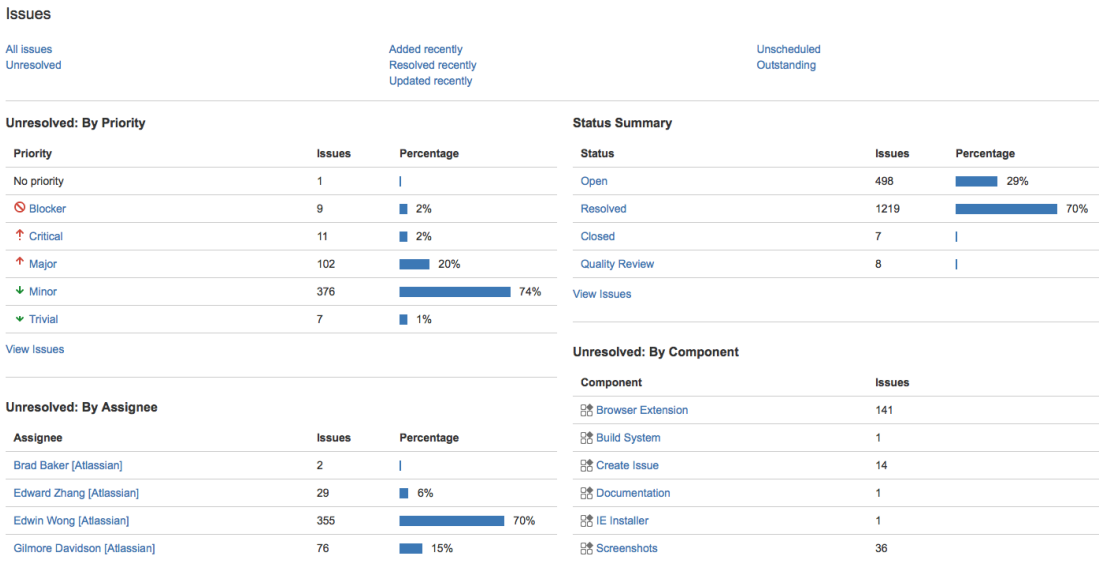


Figure 2.5: Issues view in JIRA.

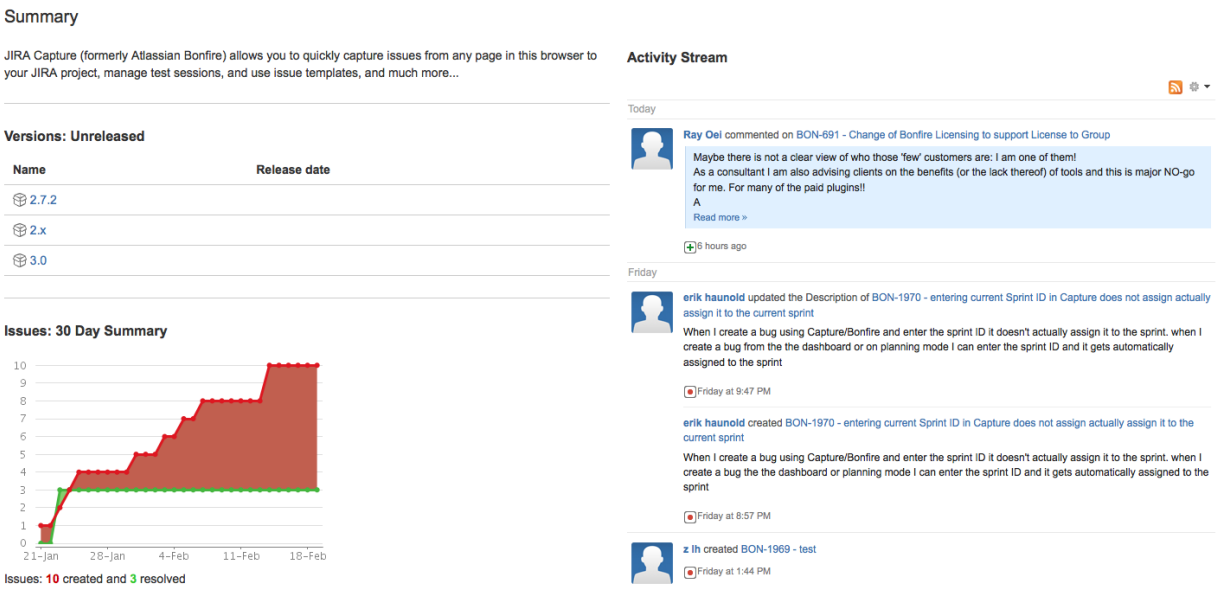


Figure 2.6: An example of JIRA's dashboards (from the JIRA website [9]).

porting features such as drag-and-drop boards, Scrum boards, Kanban boards, real-time analytics and reports, agile reporting, customizable queries, configurable workflow, etc., for additional cost.

Bitbucket [8] is a web-based hosting service for software projects. While it is not an issue-tracking tool, Bitbucket offers its users an integrated issue tracker that allows developers to follow up on the project's feature requests, bug reports and other project management tasks. The Bitbucket's issue tracker has three configurable fields (component, version and milestone). Figure 2.7 illustrates the *Issues* view showing a list of the"Open" issues. One can switch between "All", "My issues", and "Watching" tabs. Similar to JIRA, Bitbucket provides a clean user-friendly interface for tracking and updating issues.

Bitbucket integrates the Mercurial and Git version control systems, providing numerous features for collaborative development such as showing source files, listing all revisions and commits, pull requests, branch management, etc. It also lets users define Components, Milestones, and Versions to the projects supporting simple planning and release management functionality. However, Bitbucket does not offer any reporting functionality. The Bitbucket's *Dashboard* (shown in Figure 2.8) presents developers' repositories, summarizes recent activity on the repositories (e.g., commits and issue updates), and displays issues and pull requests.

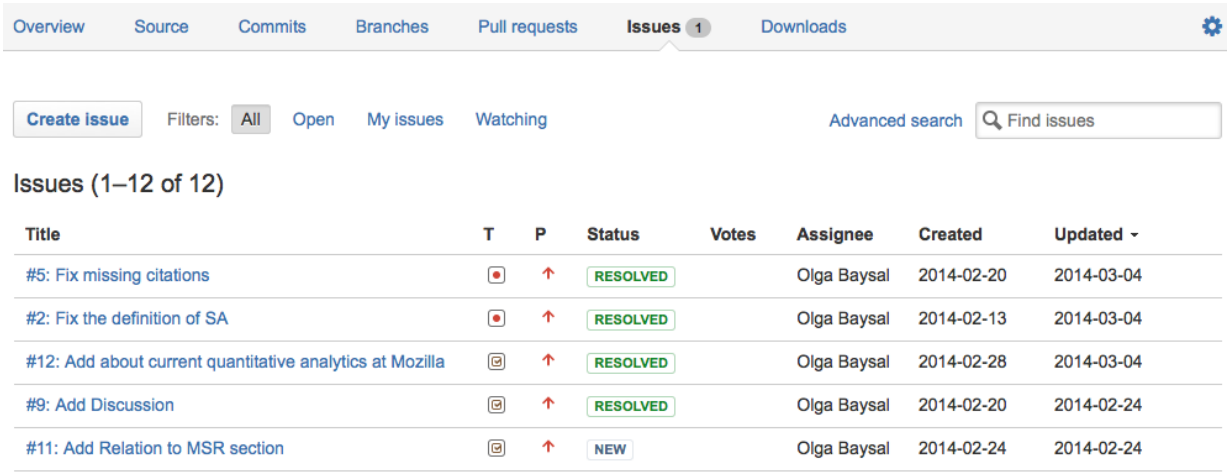


Figure 2.7: Bitbucket's issue tracker.

GitHub [95] is another web-based hosting service for software development projects that use the Git revision control system. Unlike Bitbucket offering free private repositories (which can have up to five users). GitHub provides integrated issue-tracking systems to its users. Figure 2.9 presents the Issue tab for the elasticsearch repository — an open source, RESTful full-text search engine [63]. The *Issues* tab allows users to create and apply labels to issues, to assign them to other users or categorize them, to prioritize issues with the "drag and drop" function, to search, sort and filter issues, and to close issues from commit messages. GitHub users can filter issues

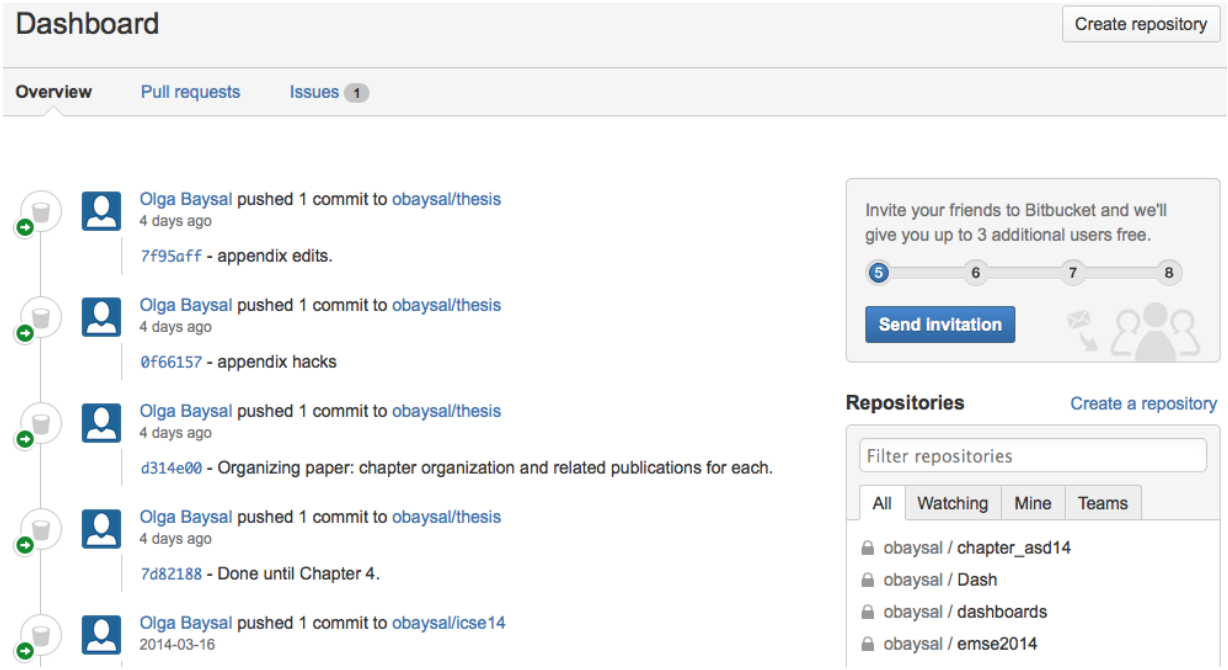


Figure 2.8: Bitbucket's dashboard.

by open and closed items, assignees, labels, and milestones, or sort issues by their age, number of comments, and update time. Issues can be grouped by milestones which allows teams to set up their project goals; labels are another way to organize issues and can be customized with colors. GitHub supports collaborative code review tasks by means of *pull requests* that streamline the process of discussing, reviewing, and managing changes to code. GitHub allows users and teammates to have a detailed discussion about each and every commit that is pushed to the project. Users can comment on each commit as a whole or any individual line of code.

GitHub integrates various graphs such as Pulse and Graphs to help its users explore their GitHub hosted repositories. Figure 2.10 demonstrates the example dashboards generated for the **/elasticsearch** repository. Every repository comes with a set of metrics ranging from contributions made by people to the project, to the frequency of code submitted. The data is rendered and visualized using 3D.js library. *GitHub Pulse* as shown in Figure 2.10(a) provides statistics about who has been actively committing and what has changed in a project's default branch. Pulse view also allows GitHub users to see new and merged pull requests, open and closed issues, and unresolved discussions. *GitHub Graphs*, shown in Figure 2.10(b), displays contribution activity on the project. The `Contributors` view shows who is contributing to a project and their contributions relative to other contributors, it offers a high-level view of everyone that has committed to the codebase, throughout the lifecycle of the project. `Commit activity` displays

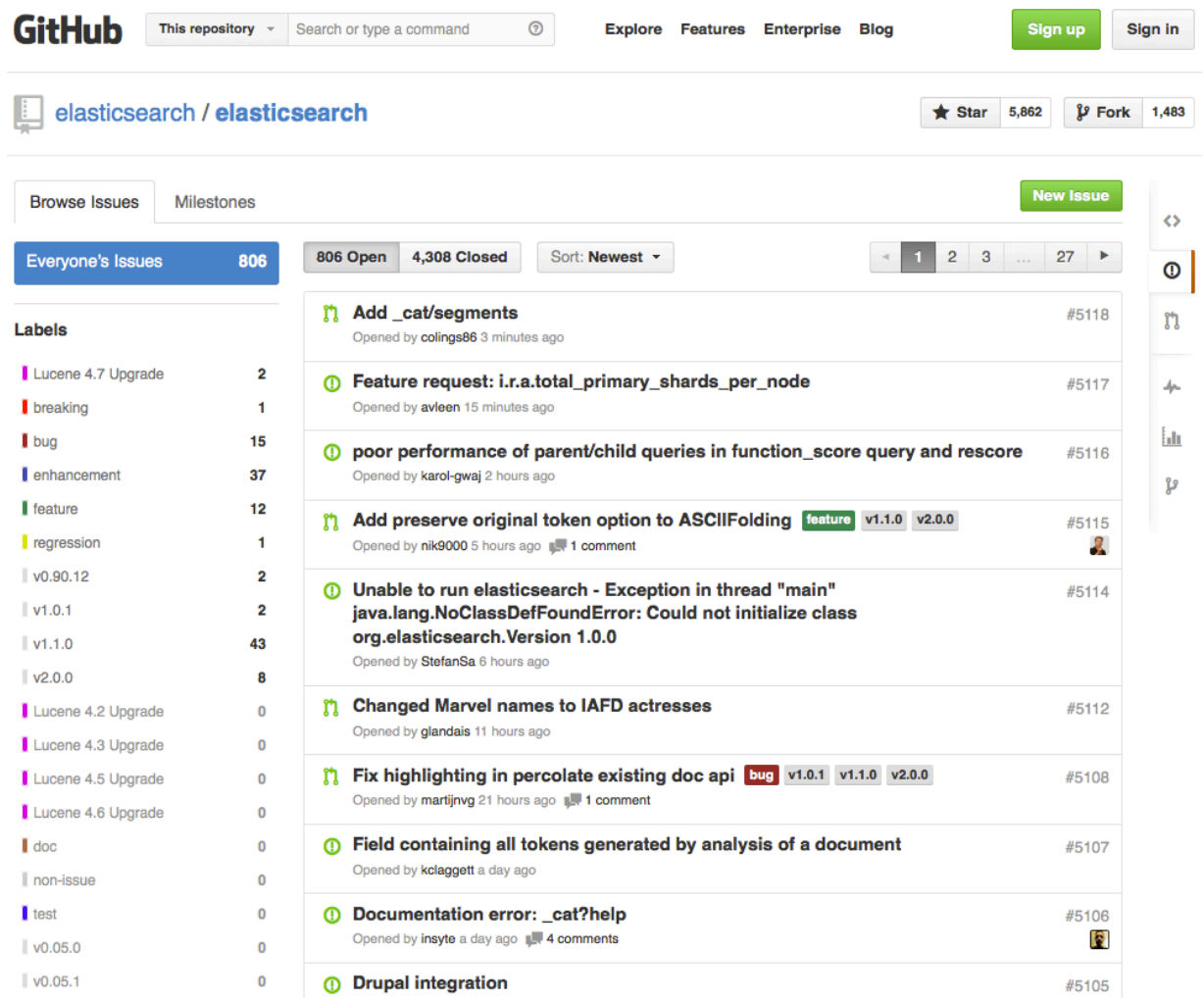
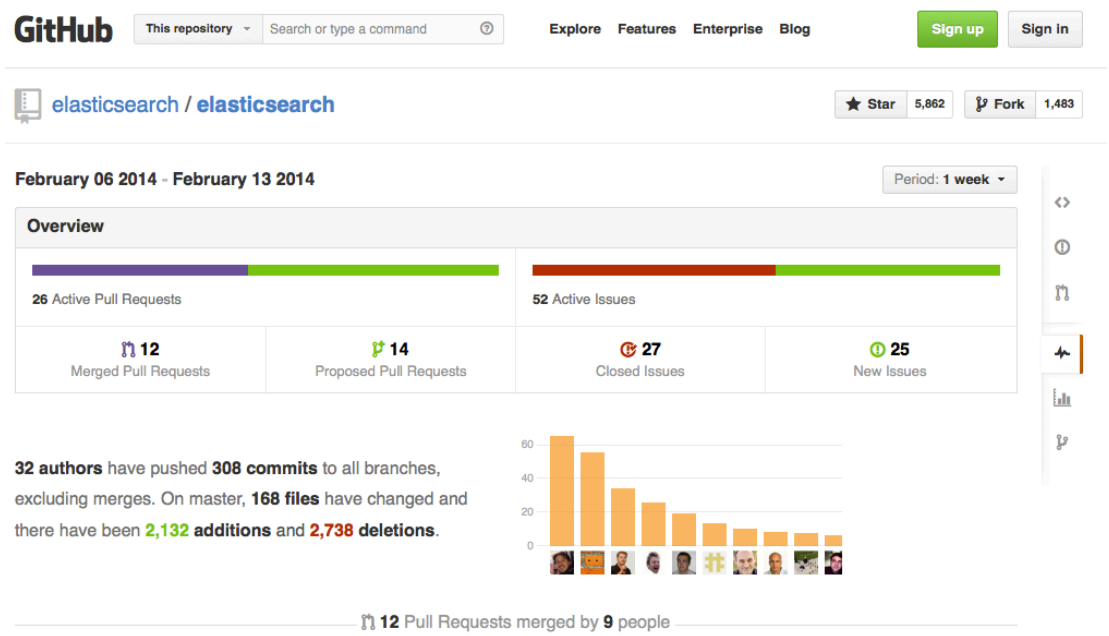


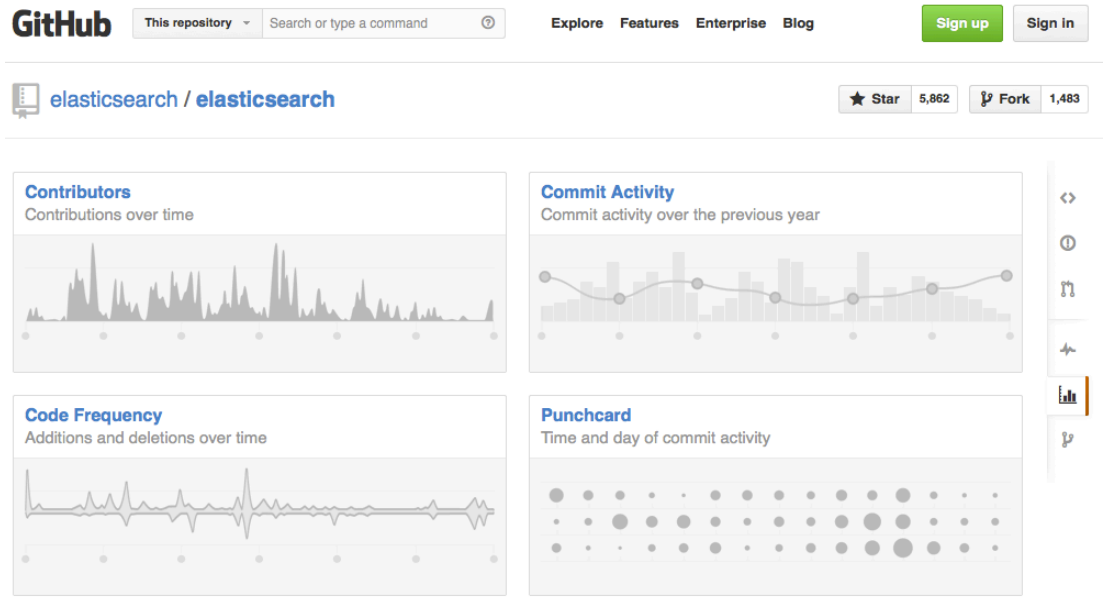
Figure 2.9: Issues view in GitHub.

all the submissions to the repository over the course of the past year with a week by week breakdown. The `Code frequency` tab allows users to see additions and deletions over the lifetime of a project. This tab takes the same information given in `Commit activity`, except it lays them out by actual number of lines changed, rather than simply counting commits. It provides an "average change" line to visually identify the trending change in the repository. And finally, `Punchart` presents the frequency of updates to the repository, based on day of the week and time of day.

Overall, GitHub provides great support for collaborative development and project management tasks and offers several good dashboard-like features including Pulse and Graphs.



(a) GitHub Pulse shows recent activity on projects.



(b) GitHub Graphs displaying contribution activity on projects.

Figure 2.10: Dashboards in GitHub.

Summary Most issue-tracking systems come with reporting functionality and dashboard-like features to help users to track recent activity on their projects. Bugzilla supports large projects and user bases; it offers nice workflow features. On the downside, Bugzilla is not user friendly for reporting and fixing bugs as it stores too much metadata, affecting its performance. Trac is a lightweight issue-tracking system suitable for small teams. Its main shortcoming is the lack of any dashboard-like features or means for tracking changes over time. JIRA provides almost all the features offered by Bugzilla in a more usable form. While JIRA offers real-time analytics and more sophisticated dashboards than any other issue-tracking system, these features might come at a high cost to the organizations wanting to make use of these analytics. Bitbucket is mainly adopted by smaller teams for project collaboration. It comes with simple issue tracking features and a high-level summary about user's repositories including recent activities, list of issues and pull requests. GitHub's dashboards and powerful bug tracking and project management systems make it a popular choice for hosting open source projects, yet industrial projects have to pay for hosting their private repositories.

This survey highlights the need for improved modelling of the key items and developer awareness of the working environment and we use the open source Bugzilla issue-tracking system to investigate this. Since Bugzilla remains the most popular open-source issue-tracking system that is well adopted by many organizations, we plan to overcome it shortcomings such as being not very intuitive and user friendly, as well as storing too much unwanted metadata (in terms of not relevant to the daily developers' tasks) affecting its search performance.

2.1.3.2 Improving Issue-Tracking Systems

The research community has provided several investigations of how issue-tracking systems can be improved. Most work has focused on understanding how issue management systems are used in practice. For example, Bettenburg et al. conducted a survey of 175 developers and users from the Apache, Eclipse, and Mozilla projects to determine what makes a good quality bug report [24]. They developed a tool that measured the quality of new bug reports and recommends ways to augment a report to improve its quality. Bertram et al. performed a qualitative study of the use of issue-tracking systems by small, collocated software development teams and identified a number of social dimensions to augment issue trackers with [20]. They found that an issue tracker serves not only as a database for tracking bugs, features, and requests but also as a communication and coordination hub for many stakeholders.

Several studies have suggested a number of design improvements for developing future issue-tracking tools. Just et al. [100] performed a quantitative study on the responses from the previous survey [24] to suggest improvements to bug tracking systems. Zimmermann et al. addressed the limitations of bug tracking systems by proposing four themes for future enhancements: tool-centric, information-centric, process-centric, and user-centric [179]. They proposed a design for a

system that gathers information from a bug report to identify defect location in the source code. Breu et al. quantitatively and qualitatively analyzed the questions asked in a sample of 600 bug reports from the Mozilla and Eclipse projects [27]. They categorized the questions and analyzed response rates and times by category and project and provided recommendation on how bug tracking systems could be improved. Rastkar et al. [139] and Czarnecki et al. [47] both worked on the problem of summarizing bug reports in the Bugzilla issue-tracking systems.

We note that while existing research has provided recommendations on how such systems can be improved, there is a relative lack of efforts in providing solutions to the developers that can help them overcome shortcomings of the existing issue management systems and assist them with their daily development tasks. To address these gaps, we plan to study how developers interact with issue-tracking systems, understand current limitations and develop tools that improve developer working environments and support their daily tasks and activities.

2.1.3.3 Increasing Awareness in Software Development

Much of the work in this area has focused on how communication and coordination affects awareness in global software development [50, 75, 83, 98, 157]. Several tools have been developed to enhance developer awareness and understanding of large source code bases [35, 55, 56, 66, 72, 77, 147] but none of them specifically targets issue-tracking systems. Existing research also offers a number of tools [25, 29, 45, 92, 106, 165] to assist developers with daily tasks and development activities; these tools are more relevant to our research goals.

Treude and Storey [162] investigated the role of awareness tools — such as IBM's Jazz dashboards and feeds — in supporting development activities. Their findings suggest that dashboards are used mainly to keep track of the overall project status, to provide awareness of the work of other teams, and to stir competition between teams; they found that feeds are used to track work at a small scale and to plan short term development activities. Our qualitative study (described in Section 5.2) also motivates the need for customized dashboards to support developers' awareness of their immediate environment and the issues that they work with. Further, our work led to the creation of a comprehensive model of the informational needs for these dashboards, instantiated it in a prototype tool, and validated this tool with industrial developers.

Cherubini et al. [36] looked at how and why developers use drawing during software development. Furthermore, Fritz and Murphy [71] studied how developers assess the relevancy of these feeds to help users deal with the vast amount of information flowing to them in this form.

FASTDash [25] offers an interactive visualization to enhance team awareness during collaborative programming tasks. Hipikat [45] provides assistance to new developers on the project by recommending relevant artifacts (source code, bug reports, emails) for a given task. The Bridge [165] tool enables full-text search across multiple data sources including source code, SCM repositories, bug reports, feature requests, etc. Mylyn [106] is a task management tool

for Eclipse that integrates various repositories such as GitHub, Bugzilla, JIRA, etc. It offers a task-focused interface to developers to ease activities such as searching, navigation, multitasking, planning and sharing expertise. Yoohoo [92] monitors changes across many different projects and creates a developer-specific notification for any changes in the depend-upon projects that are likely to impact their code. Similarly, Crystal [29] increases developer awareness of version control conflicts during collaborative project development.

Summary Issue-tracking systems play a central role in ongoing software development; primarily, they are used by developers to support collaborative bug fixing and the implementation of new features. But they are also used by other stakeholders including managers, QA, and end-users for tasks such as project management, communication and discussion, and code reviews. Bugzilla is a widely deployed bug tracking system used by thousands of organizations including open-source projects such as Mozilla, the Linux kernel, and Eclipse, as well as NASA, Facebook, and Yahoo![30]. While existing research identifies key design improvements for developing the next generation of issue management systems, there are no attempts at providing better systems. We investigate how developers interact with issue-tracking systems to identify Bugzilla's limitations (Section 5.2) and to provide a solution to assist developers with completing their tasks and development activities (Section 5.3 and Section 5.4.1).

2.2 Data Analysis

In order to transform data from the available artifacts into insights that highlight useful information, the data needs to be gathered, pre-processed, and analysed using various techniques and tools. Our data comes in various forms (both structured and unstructured) and from different artifacts (usage logs, release history, issue-tracking systems, version control repository, project collaboration systems, documentation such as online documents, policies, interview data). We now discuss the main approaches and techniques we have used to tackle our research problems, including statistics, machine learning, natural language processing, information retrieval, and grounded theory.

2.2.1 Statistics

Statistics are used to describe, analyze, interpret and make inferences from data [145, 151]. In data mining, we employ both types of statistical analysis: descriptive statistics and inferential statistics. *Descriptive statistics* deal with frequency distributions and relationships between variables; while *inferential statistics* are used to draw conclusions from the datasets accounting for randomness.

Distributions can be presented using shapes (histograms, boxplots, lune graphs) or functions. To compare two distributions, several statistical tests can be used: t-test, χ^2, Kolmogorov-Smirnov test, Euclidean distance, etc. These tests can be used to asses the differences or similarities between distributions. Software engineering often deals with non-parametric data and thus Mann-Whitney U test or Wilcoxon U test are most appropriate for comparing two distributions, and Kruskal-Wallis test for comparing more than two distributions.

Empirical studies are often done on multiple data sets or include hypotheses testing. And thus, statistical analysis is essential for interpretation the study results or for validity of study conclusions [60]. Statistical analysis allows us to interpret the data we study either by describing and comparing the data or making predictions. By looking at the distribution of values, we could check for the most frequent values, outliers or the overall symmetry of the distributions. Often researchers want to find the central tendency of the data, and thus the median is a measure to report. However, data is rarely collected simply for description. We often want to compare a real to an ideal value or to compare two or more data sets. In such cases, it is important to report the difference between data distributions. Statistical tests of comparison (t-test, ANOVA, Wilcoxon/Mann-Whitney test, Kruskal-Wallis test) are decisions about whether an observed difference is a real one of occurs just by chance.

2.2.2 Machine Learning

Machine learning provides tools for automatically extracting and integrating knowledge from empirical data [168, 172]. Machine learning is a powerful tool since it can build classification and prediction models from the historical data. A training set (historical facts) is used to produce a classifier (naive Bayes, decision trees, linear regression, support vector machines, etc.) for predictive purposes.

Within the field of empirical software engineering and software analytics, Anvik et al. [6] used a supervised learning (support vector machine algorithm) to recommend a list of potential developers for resolving a bug. Past reports in the Bugzilla repository are used to produce a classifier. They developed project-specific heuristics to train the classifier instead of directly using the assigned-to field of a bug report to avoid incorrect assignment of bug reports. Cubranic et al. [44] used supervised machine learning (Naive Bayes classifier) to automatically assign bug reports to the developers. Similar to Bougie et al. [26] and Panjer [135], Hosseini et al. [93] evaluated various classifiers (0-R, 1-R, Naive Bayes, decision tree, logistic regression) on their powers to correctly predict bug lifetimes and found that Naive Bayes algorithm outperforms others by +/-2%.

2.2.3 Natural Language Processing

Most software repositories are text-based artifacts or can be transformed to a textual representation. For example bug reports, emails, commit logs, requirements documents, source code are all considered as unstructured text. Given the large amount of available natural language text, natural language processing (NLP) and text mining techniques can be applied to automatically analyze the natural language text to improve software quality and productivity [161].

NLP is a field of computer science and linguistics that is concerned with adopting machine learning to process the human language for a given task [67]. NLP provides techniques and tools to SE community for linking various artifacts [114]. The common techniques include modelling language, measuring text similarity, finding feature location, topic analysis, part-of-speech tagging, sentiment analysis, etc. [105].

NLP techniques have been successfully adopted by the MSR community to trace artifacts at different levels of abstraction including high level to low level requirements [52, 53], requirements to design [58], requirements to source code [136], high-level features to their implementation [137], functional requirements to Java components [5], requirements to bug reports [146, 176], and change requests to the impacted software modules [4], to learn how developers use identifier names and words when describing concepts in code [86], to measure code complexity [89].

NLP techniques are useful to recover information and associations from various development artifacts. These associations allow us to link various artifacts such as defects, source code, email archives, etc.

2.2.4 Information Retrieval

Information retrieval (IR) is another approach of the classification and clustering of textual units based on various similarity concepts. IR methods have been applied to many software engineering problems such as traceability, program comprehension, and software reuse. For example, Canfora and Cerulo [33] developed a taxonomy of information retrieval models and tools to clarify IR models with respect to a set of basic features and tasks they support. They later demonstrated how bug descriptions and CVS commit messages can be used as input to such models for the purpose of change predictions [32]. Their approach provides a set of files that are likely to change based on only the textual description of a newly introduced bug in the bug repository. A bug report is linked to a CVS commit based on the explicit bug identifier found in that commit message. To draw the similarity between various artifacts, Cubranic et al. employed a vector-based IR method [46, 164]. They developed a tool, Hipikat, that provided assistance to new developers on the project by recommending relevant artifacts (source code, bug reports, emails) for a given task.

2.2.5 Grounded Theory

Grounded theory method (GT) is a systematic methodology in the social sciences involving the discovery of theory through the analysis of data [115]. The basic idea of the grounded theory approach is to develop theory through an iterative process of data analysis and theoretical analysis. The data may come from various sources including interviews, observations, surveys, diaries, letters, document and artifact analysis, etc. The goal of the grounded theory is to develop a theory that emerges from and is connected to that reality. Grounded theory investigates the actualities in the real world and analyses the data without preconceived ideas. It is useful when there is a need to examine relationship and behaviour within a phenomenon from an unbiased perspective.

Open coding [41, 43, 120] is the part of the analysis concerned with identifying, naming, categorizing and describing phenomena found in the data. It involves searching out the concepts behind the actualities by looking for codes, then concepts and finally categories. Written data from notes or transcripts are conceptualized line by line. Each line, sentence, paragraph etc. is read to understand what it is about and what it is referring to [74, 159]. Open coding is often an iterative process; a researcher goes back and forth while comparing data, keeps modifying and improving the growing theory. The researcher continues collecting and examining data until the patterns continue to repeat and no new patterns emerge. The researcher builds the theory from the phenomena, from the data, and the theory is thus built on or "grounded" in the phenomena.

Since qualitative research gives a rich, multi-dimensional view on the phenomenon, it is complementary to any quantitative study. Many software artifacts consist of textual data such as emails, bug reports, project documentation, observations, etc. Therefore, grounded theory can be applied to study an aspect of the software development. Software engineering research shows that grounded theory is a powerful tool to study software process and its improvement, as well as human aspects of software engineering [2, 3, 34, 38, 49, 54, 91, 117, 126, 166], mainly because it allows researchers to build a body of knowledge about a topic of interest (e.g., to get insights into a code review process in open source projects or to study behaviour and interactions between developers collaborating on a project). By analysing the data in a bottom-up fashion, researchers can develop hypotheses and research questions for their new studies.

2.2.6 Software Metrics

Software metrics can be classified into three categories: product metrics, process metrics, and project metrics [104]. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics can be used to improve software development and maintenance. For example, the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process. Project metrics describe the project characteristics and execution. For example, the number

of software developers, cost, schedule, and productivity. Some metrics can belong to several categories.

In order to describe observations quantitatively we need to count and measure them, software metrics allow us to count and measure properties of software systems and their artifacts. Measurement of changes, whether they be revisions, diffs, deltas, or structural deltas has been investigated in work by Ball et al. [10], Mens et al. [118], Lanza [109], German et al. [73], Robbes et al. [143, 144], Hindle et al. [87, 88, 90], and Herraiz et al. [85].

Summary In this section, we provided an overview of the tools that can be used to transform the data from software development artifacts into insights about behaviour and processes of the developers and users when they develop or use software systems. Statistics are used to model underlying data so we can describe the data and reason about it. Machine learning focuses on identifying patterns in the data and making predictions about a new data based on what has been learnt. Natural language processing helps us to deal with the large amount of unstructured and semi-structured natural language artifacts (such as bug reports) that developers and users leave behind. Information retrieval is helpful in sorting and finding information in documents (bug reports, commit history) by matching strings of characters. Grounded theory is effective in developing new theories and understanding phenomena. All of these data analysis techniques have been leveraged in the field of mining software repositories and are relevant to the research problems we are trying to solve.

2.3 Summary

In this chapter, we have reviewed previous research that has inspired much of the work presented within this dissertation. We covered two main areas: mining software repositories and data analysis. We first reviewed the research from the area of mining software repositories, including previous research efforts related to the usage mining, collaboration and contribution management, and issue-tracking systems. Much of our work relies on MSR research since we mine, correlate and analyze various kinds of development artifacts such as issue-tracking repositories, code review repositories, server logs, commit and release histories. MSR-like tools and techniques provide a means to perform repository analysis including fact extraction, formatting, statistics, presentation/visualization, and prediction.

We then reviewed the data analysis tools such as statistics, machine learning, grounded theory and software metrics.

Chapter 3

Understanding User Adoption of Software Systems

Web usage data, an artifact that has yet received little attention, stores valuable information about users and their behaviour related to adoption and use of software systems. In this chapter, we apply software analytics to mine usage log data that can provide teams with the insights into how users adopt a software system and what characteristics of the user adoption can be extracted from these usage logs. In particular, we demonstrate how we can employ available usage data and combine it with more traditional MSR-like data and analysis to study broader questions of how software systems are actually used.

Chapter Organization Section 3.1 introduces the problem and our research questions. Section 3.2 presents the methodology of mining dynamic usage data from web logs and describes the setup of our study. Section 3.3 presents results of the empirical study and Section 3.4 discusses our findings on adoption trends, behavioural characteristics of users, and also addresses threats to validity. And finally, in Section 3.5 we summarize our main findings.

Related Publications

The work described in this chapter has been previously published:

- Olga Baysal, Reid Holmes, and Michael W. Godfrey. Mining Usage Data and Development Artifacts. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2012, pages 98–107.
- Olga Baysal, Ian Davis, and Michael W. Godfrey. A Tale of Two Browsers. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2011, pages 238–241.

3.1 Introduction

With the continued growth of web services, the volume of user data collected by organizations has grown enormously. Analyzing such data can help software projects determine user values, evaluate product success, design marketing strategies, etc. Such analyses involve searching for meaningful patterns from a large collection of web server access logs.

The mining software repositories (MSR) research is generally focused on analyzing, unifying, and querying different kinds of development artifacts, such as source code, version control meta-data, defect tracking data, and electronic communication. Augmenting MSR-like activities with real-world usage data can give insights into deployment, adoption, and user behaviour of the systems. Our previous work suggested that by studying dynamic web usage data, we can infer knowledge on user adoption trends [11]. Therefore, we decided to further study user community and adoption practices. Understanding how users adopt and use systems such as web browsers is important for measuring the success of a product and performing target market analysis. Analysis of adoption trends can also help to identify main trends in user adoption at a global scale. By studying users' dynamic behaviour, we are able to gain knowledge on the deployment environments of users to compare and contrast the use of a software system on different hardware configurations. This knowledge can then be used to assess software acceptance, user experience, or sustainability of a software system.

We have taken a statistical approach to extract dynamic behaviour of the users from web traffic logs consisting of over 143 million entires. We analyzed each entry, representing a single page view by a single user, to determine the browser's name, version, and its host operating system, to map each host IP to a geographical location, and to track user browsing behaviour.

In this work we address three research questions:

1. *Are there differences in platform preferences between browser end-users*?
2. *Is there a difference in geographic distribution between user populations*?
3. *Is there a difference in navigation behaviour between two user groups*?

Our study reveals several notable differences in the Firefox and Chrome user populations. Chrome undergoes continual and regular updates and has short release cycles, while Firefox is more traditional in delivering major updates, yet providing support for older platforms. Our data suggests that Firefox users are primarily centred in North America, while Chrome users are better distributed across the globe. We detected no evidence in age-specific differences in navigation behaviour among Chrome and Firefox users. However, we hypothesize that a younger population of users are more likely to have more up-to-date versions of a web browser than more mature users.

This chapter offers several contributions. First, by mining web usage data we define several characteristics of the user population for empirical evaluation. Second, we analyze the usage patterns and highlight the main differences in how the browsers provide operating system (OS) support to the end-users, appeal to users across the globe, and emphasize age-specific differences among its users in the adoption of new releases. Third, we discuss how characteristics of user population and adoption can provide insights into the sustainability of a product. Our findings may also help improve user experience. First, development team members may consider our findings to target a wider user population. Second, our findings have implications for better support of software applications that appeal to a wider population across the globe, support older and a variety of platforms, and reduce age-specific usability issues. And finally, our work might facilitate further research on user adoption and acceptance of software products.

3.2 Setup of the Study

Label	Chrome Release	Firefox Release
b3	0.2	–
b2	0.3	0.8
b1	0.4	0.9
r1	1.0	1.0
r2	2.0	1.5
r3	3.0	2.0
r4	4.0	3.0
r5	5.0	3.5
r6	6.0	3.6
r7	7.0	–

Table 3.1: Browser release labels.

This section describes browser release histories, usage log data, provides a sample of web server logs, and explains the process of mining dynamic behavioural data from web logs.

3.2.1 Release History

Mozilla Firefox is an older web browser, originally released in November 2004 as the successor of the Mozilla project. Google Chrome is a younger web browser that was first released in December 2008. Chrome is based on the Webkit layout engine, which is also used by Apple's Safari browser.

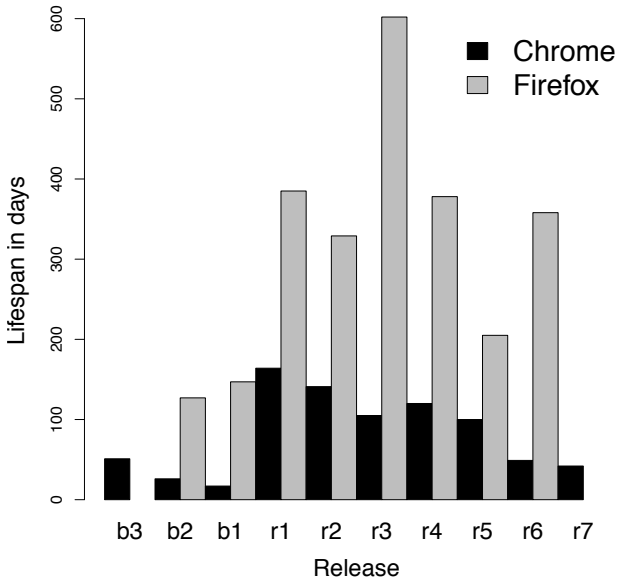


Figure 3.1: Lifespan of major releases for Chrome and Firefox. The difference in release delivery is statistically significant ($p<0.005$).

```
10.0.0.1 - - [20/Oct/2008:23:05:24 -0400] "GET /undergrad/handbook/courses/
waitlist/index.shtml HTTP/1.1" 301 368 "http://www.cs.uwaterloo.ca/
current/" "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_5; en-us)
AppleWebKit/525.18 (KHTML, like Gecko) Version/3.1.2 Safari/525.20.1"

10.0.0.2 - - [26/Oct/2008:16:47:49 -0400] "GET /~fwtompa/.papers/xmldb-
desiderata.pdf HTTP/1.1" 301 365 "-" "Mozilla/4.0 (compatible; MSIE 6.0;
Windows NT 5.1; SV1)"
```

Figure 3.2: An example of server access log.

Strictly speaking, Chrome is not open source but its core base — a project called Chromium — is.

As of November 2010, Firefox had released 8 major versions [171]; by this time there were 10 major releases for Chrome starting with version 0.2 [170]. In this paper, we define labels (see Table 3.1) and use them when comparing releases of the browsers. On average, a new release of the Firefox browser is launched every 10 months, while a new version of the Chrome browser is released every 2.5 months (see Figure 3.1). The difference in release delivery of the browsers is

statistically significant ($p<0.05$).

3.2.2 Web Server Logs

Web server logs are automatically generated by web servers whenever a user navigates through the web pages the server hosts. These logs contain detailed information about the browsing behaviour of visitors to a website. Each HTTP request to the server, called a *hit*, is recorded in the server access log. Each log record may contain the following fields: the client IP address, the time and date of the request, the requested resource, the status of the request, the HTTP method used, the size of the object returned to the client, the referring web resource, and the user-agent of the client. An example of a combined log format obtained from `www.cs.uwaterloo.ca` server is given in Figure 3.2. IP addresses of the visitors have been changed to protect their privacy. The user-agent field identifies the browser's type and version, as well as information about its host operating system.

We obtained web server logs from the University of Waterloo, School of Computer Science (`http://www.cs.uwaterloo.ca`). The web server logs spanned from February 2007 until November 2010; there was 36GB of uncompressed raw textual data, comprising over 174 million entries.

3.2.3 Dynamic Behaviour Mining

Web-based services collect and store large amounts of user data. Analyzing such data can help organizations better understand users and their behaviour, develop marketing strategies, optimize the structure of their sites, etc. Such analysis involves searching for interesting patterns in large volumes of user data. A web server log is one of the primary sources for performing web usage mining since it explicitly records the browsing behaviour of the site visitors [153]. Web usage mining is a process of extracting useful information from web server logs by analyzing the behavioural patterns and profiles of users visiting a web site [40, 124, 153]. Mining users' browsing history provides insights into how users seek information and interact with the web site. Typical web usage mining consists of three stages: pre-processing, pattern discovery, and pattern analysis [153]. Figure 3.3 presents an architecture of a web usage mining process (adapted from [153]).

In the **pre-processing** stage, web logs are divided into transactions representing user activities during each visit. Depending on the analysis, the usage data is then transformed and aggregated at different levels of abstraction (users, sessions, click-streams, or page views). In this paper, we evaluate accesses to the web resources by considering page accesses or "hits" (Section 3.4.2 explains our decision). Our data cleaning process eliminates all the log entries generated by the web agents such as web crawlers, spiders, robots, indexers, or other intelligent agents that prefetch pages for caching purposes. This removed 31,255,963 entries (17%). We also restructured

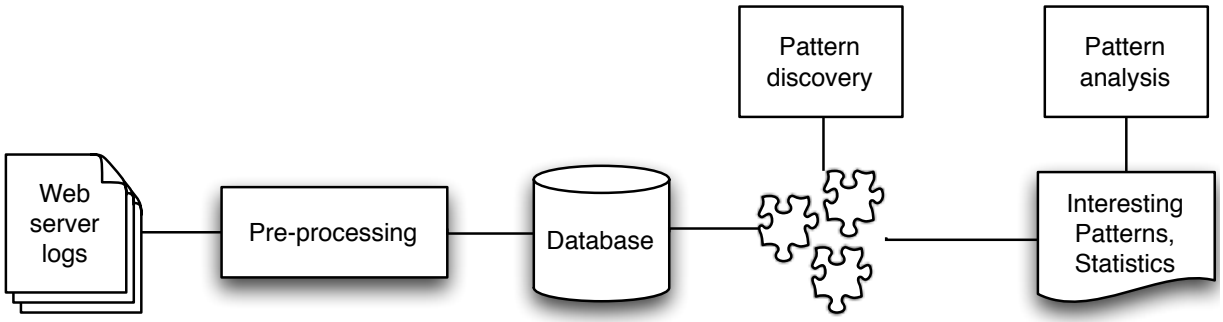


Figure 3.3: High level architecture of web usage mining.

the date and time fields of the log entry to `[year-month-day hour:minute:second]` format. After the cleaning and transformation stages, the traffic data contained 143,613,905 entries, each corresponds to a single page view by a single user, and loaded then into a relational database.

During the **pattern discovery** stage, various operational methods can be applied to uncover patterns capturing user behaviour. The most commonly-used methods are descriptive statistical analysis, association rule mining, clustering, classification, sequential pattern analysis and dependency modelling [153], and prediction. These techniques are typically used for personalization, marketing intelligence, system improvements, and site modification. Statistical analysis provides measures to organize and summarize information. Association rule mining concerns discovery of relationships between the items in a transaction. Clustering is an unsupervised grouping of objects, while classification is supervised grouping. In web mining, the objects can be users, pages, sessions, events, etc. Sequential pattern analysis is similar to association rules but it also considers the sequence of events. The fact that page A is requested before page B is an example of a discovered pattern. All these techniques were designed for knowledge discovery from very large databases of numerical data and were adapted for web mining with relative success. In our work, we performed a descriptive statistical analysis when discovering patterns of user behaviour.

In this research work, we took a statistical approach for studying user behavioural dynamics. We examined the web log data to determine the browser type of our visitors. We analyzed the HTTP user agent strings that web browsers report when requesting a web page. We extracted the name of the browser from the user agent strings and calculated the number of accesses to our web site for each release of a browser. The proportion of accesses for each web browser is shown in Figure 3.4. The left pie chart shows the percentages of the total volume each that browser makes up. To our surprise, Firefox dominated the web traffic (31%), followed by Internet Explorer (24%), while Chrome users contribute only 3%. "Other" represents web traffic from other browsers including Safari, Netscape Navigator, Opera, mobile browsers, etc. To have a fairer picture on the web usage share, we eliminated the "Other" slice and normalized the

cumulative access count per browser by its average market share: Chrome (10.70%), Firefox (20.16%), Microsoft's Internet Explorer or MSIE (52.37%) [156]. As can be observed from the right-hand side pie chart in Figure 3.4, Firefox is an obvious preferred choice among the visitors to our web site.

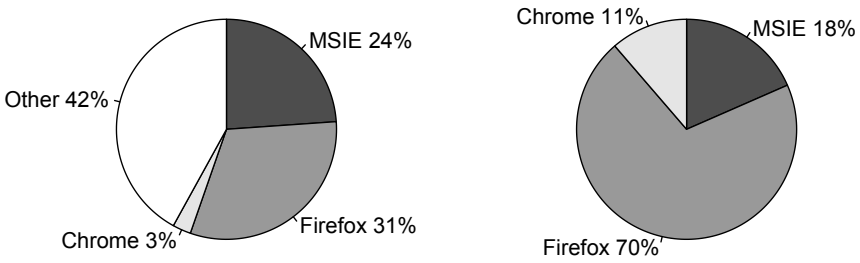


Figure 3.4: Pie charts representing volumes of accesses for a web browser. The pie chart on the left depicts percentages of the total hits per browser, while right-hand side shows "normalized" traffic shares among three leading browsers.

In the final stage, **pattern analysis**, the discovered patterns and statistics are further processed and used as input to applications such as recommender systems, report generation tools, visualization tools, etc. Since we performed only a statistical analysis during the pattern discovery stage, in this step we used GNU R tool [57] and presented graphs to report and explain our findings.

3.3 Empirical Study

This section addresses each research question by describing how we approach the problem and reporting our findings. When applicable, we report results of statistical analysis of the data.

3.3.1 Platform Preferences

We first try to answer research question Q1: *Are there differences in platform preferences between end-users of the browsers?*

Since Chrome and Firefox are developed to run on multiple operating systems, we were interested to compare the browser's adoption and support for different deployment environments.

We examined the choice of computing platform of the visitors to our website. For each release of the browser, we extracted the number of accesses from three operating systems: Windows, Linux, and OS X. This data was then normalized by the operating system market share obtained

from StatOwl.com, which predominately measures United States web sites [154, 155]. Since our server is located in Canada, we consider our choice of market share statistics from StatOwl.com as reasonable to use for our analysis. The OS market share data represents "real" web site browsing community (excludes automated systems like search robots) and excludes mobile usage. For each release of a browser, we calculated an average OS market share percentage (reported every month) for the period of the release's lifespan. Since the market share numbers were reported starting from 2008, we applied the total average market share (Windows – 88.21%, OS X – 11.10%, Linux – 0.54%) to the releases deployed prior September 2008. Table 3.2 provides the percentages we used to normalize our usage data with respect to the users' choice of the platform.

Rel.	Chrome			Firefox		
	Win	**OS X**	**Linux**	**Win**	**OS X**	**Linux**
b3	90.67%	8.87%	0.43%	–	–	–
b2	90.39%	9.13%	0.45%	88.21%	11.10%	0.54%
b1	90.34%	9.00%	0.63%	88.21%	11.10%	0.54%
r1	91.12%	8.21%	0.61%	88.21%	11.10%	0.54%
r2	89.74%	9.59%	0.57%	88.21%	11.10%	0.54%
r3	88.43%	11.06%	0.41%	88.21%	11.10%	0.54%
r4	88.35%	11.07%	0.44%	90.89%	8.49%	0.56%
r5	87.93%	11.40%	0.51%	89.02%	10.37%	0.51%
r6	87.65%	11.66%	0.53%	87.68%	11.65%	0.51%
r7	87.43%	11.84%	0.59%	–	–	–

Table 3.2: Operating systems market share.

Results The distribution of the number of user accesses from a platform is presented in Figure 3.5. A beanplot consists of a one-dimensional scatter plot (aka boxplot), its distribution as a density shape and an average line for the distribution [103]. The left side of a beanplot represents the density of the distribution for Chrome, while the right side of a beanplot shows the distribution for the Firefox browser. Applying the Mann-Whitney statistical test, we compared the distributions of each platform across two browsers. The results show that the difference in density for both OS X and Linux platforms between two browsers is statistically significant ($p<0.05$), while distributions of Windows users across Chrome and Firefox are fairly similar ($p=0.40$). Users do not adopt browsers equally across operating systems. Users on a Linux or OS X choose Firefox over Chrome. On Windows, users equally opt for either one of the two browsers.

We then performed the Kruskal-Wallis statistical test to compare distributions between the three different platforms for each browser (see Figure 3.6). Unlike Chrome ($p=0.23$), the difference in distributions of operating systems between each other for the Firefox browser is statistically significant ($p=0.05$). This suggests that Linux users have an order of magnitude higher rate of

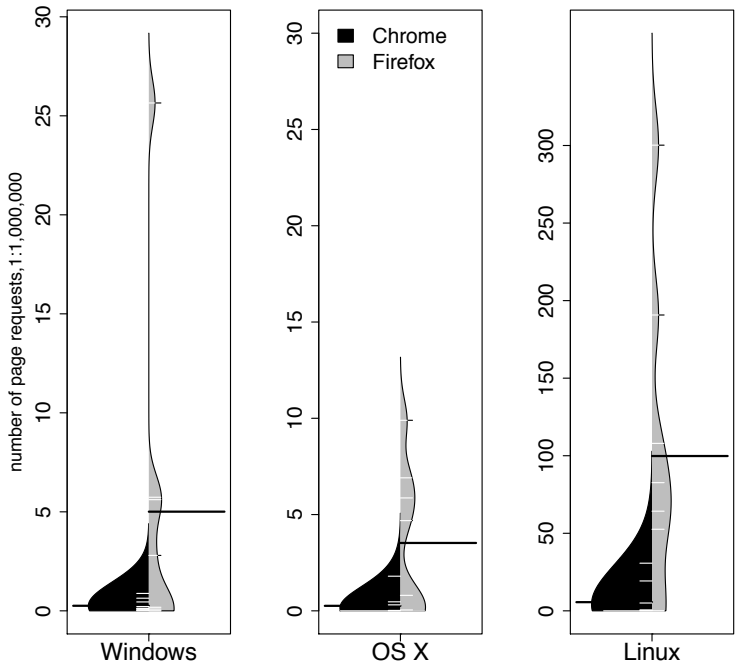


Figure 3.5: Density of the page requests by user's platform. The left side of each bean consists of hits for the Chrome browser, whereas the right side of a bean contains hits for Firefox. The horizontal lines represent the average. The difference in distributions for both OS X and Linux platforms between two browsers is statistically significant ($p<0.05$).

Firefox adoption than OS X or Windows users; while Chrome is being adopted fairly consistently across platforms.

By analyzing historical trends of Chrome and Firefox support for different operating systems (see Figure 3.7), we noticed that Firefox offers outstanding OS compatibility from the very beginning, while Chrome begins to reach for Linux and OS X users only starting from release `r4`, i.e., Chrome 4.0 (Google officially started to offer OS X and Linux OS support with the release of Chrome 5.0). Firefox reaches the peak of its adoption among Windows users in release `r3` (Firefox 2.0), among OS X population releases `r3` (Firefox 2.0) and `r5` (Firefox 3.5) are more well adopted than others, while Linux users seem to favour release `r4` (Firefox 3.0). We should also mention that Firefox can run not only on Windows, OS X and Linux, but also on BSD and other Unix platforms [169]. Therefore, we can say that Firefox provides early and better OS compatibility.

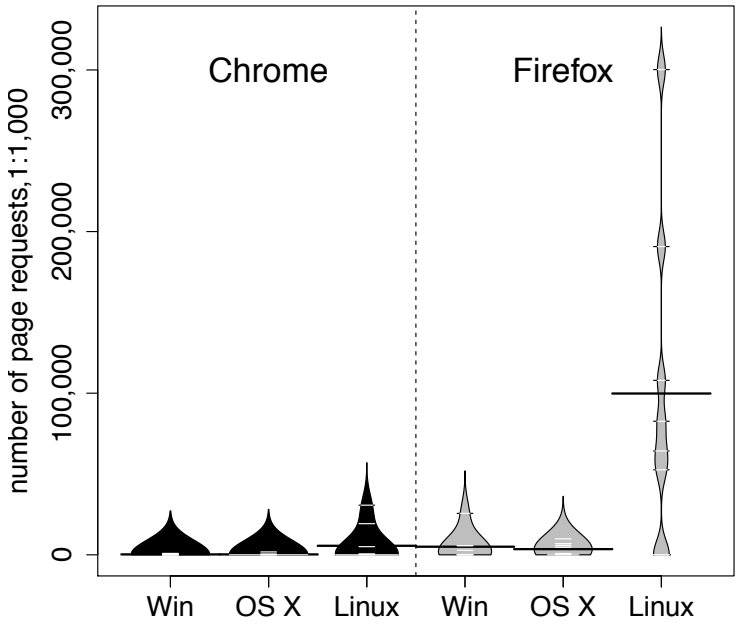


Figure 3.6: Density of the page requests by user's platform within a browser. Black beans represent Chrome, and grey beans represent Firefox. The difference between OS platforms for Firefox is statistically significant (p=0.05).

3.3.2 Geographic Location

The next question is Q2: *Is there a difference in geographic distribution between user populations?*

The previous question has shown that there are clear differences between how two browsers are being adopted across operating systems. We now study geographical location of the users and whether there is a difference in adoption of the browsers across the globe.

We used a geolocation service to track the geographic distribution of visitors to our website. We used Geo::IPfree, a Perl module, to look up a country of an IP address. We used six continents from WorldAtlas.com [173] to map a user's IP address to a geographical location. The list of continents (we call them regions) includes Africa (AF), Asia (AS), Europe (EU), North America (NA), South America (SA) and Australia/Oceania (OC). During the process of mapping IP address to the country and region, we detected a number of private IP addresses (local networks), which we excluded from the analysis.

Results Figure 3.8 illustrates the differences in the distribution of the user populations by world regions. The Geo::IP database contains information from various registry sources. In some

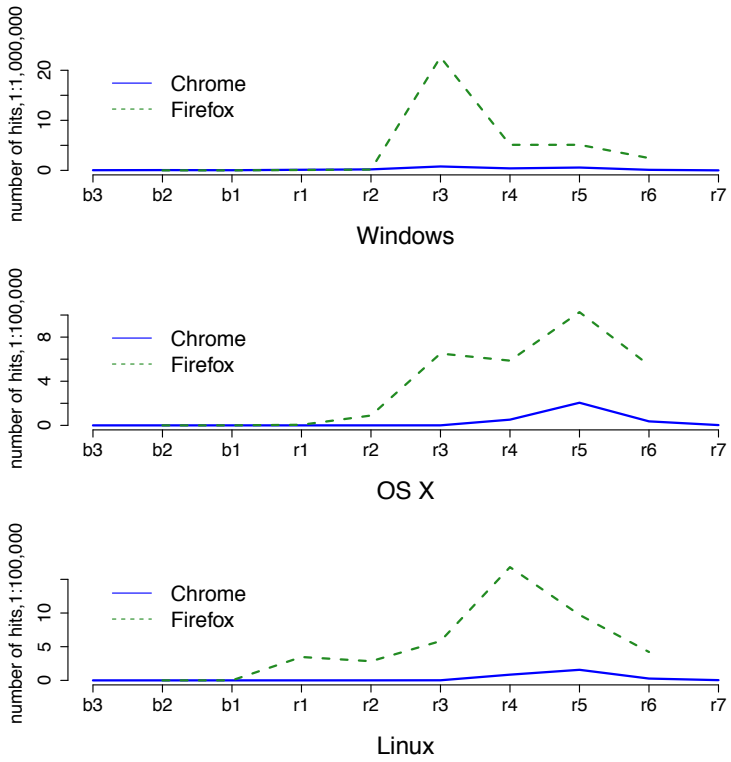


Figure 3.7: Support for Windows, OS X and Linux platforms across releases of Chrome and Firefox.

cases, the location is indicated only as Europe, which means that the requests from such hosts may come from anywhere in the European Union. To bridge the global digital divide — the disparities in the opportunities to access the Internet between developed and developing countries [113], we normalized our user accesses by the world's Internet usage data. The statistics on the distribution of the Internet users by world regions report the following numbers: NA 13.0%, AS 44.0%, EU 22.7%, SA 10.3%, AF 5.7% and OC 1.0% [96]. In our sample, we found that 85% of Firefox users and only 72% of Chrome users are located in North America. Overall, Chrome is better adopted across the globe.

We then compared user adoption of the browsers with respect to the country coverage. From our sample, we found that Chrome is adopted by the users in 187 countries, while the Firefox user population covers 207 countries. Table 3.3 presents statistics on the user accesses by the top 5 countries. We were not surprised to see China and India in the top 5 countries list as these two countries contribute to the majority of the international students in our school's undergraduate

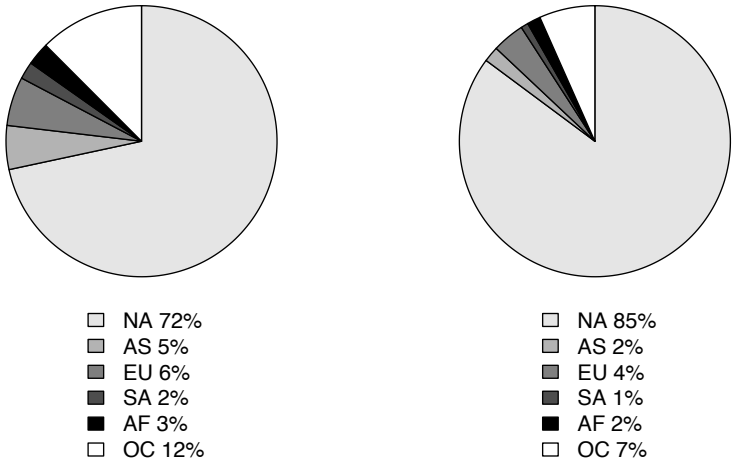


Figure 3.8: Density of the page requests by region for Chrome (left) and Firefox (right).

program.

Chrome	**Firefox**
Canada (1,968,421)	Canada (32,236,878)
USA (603,149)	USA (3,424,990)
India (193,805)	India (670,807)
China (87,058)	Europe (557,854)
UK (73,986)	UK (386,408)

Table 3.3: Top 5 countries of user's accesses

The results suggest that Firefox adoption has been more heavily concentrated in North America, while Chrome users are better distributed across the globe. Thus, it is safe to say that Chrome has a more culturally diverse user population.

3.3.3 Navigation Behaviour

Finally, we address our last research question Q3: *Is there a difference in navigation behaviour between two user groups?*

By looking at the content of the pages requested by the visitors, we wanted to identify whether the user populations of two browsers have different browsing goals and behaviour.

We were interested in classifying users according to their navigation behaviour on the CS website. To investigate patterns in the browsing behaviour of the users, we first determined the types of the web content our web site offers to the visitors. Our school's web site is mainly designated for the following classes of visitors:

1. students – offering information to current and prospective students about the courses, their description, schedules, lectures, assignments, exams, etc.
2. researchers/industry partners – offering information on faculty's and grad students' research interests, current projects, publications, potential collaboration opportunities, etc.

Based on the content of the page requests, we defined two profiles of the user accesses related to undergraduate *teaching* or *research*. We note that not every access is related to either of the two profiles. Table 3.4 defines our rules for classifying visitors' requests into two profiles.

Teaching	**Research**
• requests to any CS 100–600-level undergraduate course	• requests to any CS 700–800-level graduate course
• requests to course descriptions and course schedules for undergrads	• requests to publications
• requests to information for future undergrads and prospective students	• requests to anything under `/research`

Table 3.4: Pattern matching rules to classify user access type

Requests to the publications are defined as ones to any .pdf document located under `/pubs/`, `/publications/` or `/papers/` directories.

Results Figure 3.9 illustrates the differences in the browsing behaviour between Chrome and Firefox users. As we expected, undergraduate teaching pages are accessed more often than research-related ones by both Chrome and Firefox users. While we detected no statistical evidence for age-specific differences in browsing behaviour among Firefox and Chrome users, Figure 3.9 suggests that the browsing habits of Chrome users follow approximately a normal distribution, while for Firefox the distributions of the accesses of both research and teaching pages are more spread out. Since we did not detect any statistical difference between the distributions, we performed the Kolmogorov-Smirnov test to test for the equality of two distributions. The results suggested that for the teaching profile, Chrome and Firefox samples come from the same distribution. This tells us that Chrome and Firefox users behave the same way when navigating to undergraduate teaching content.

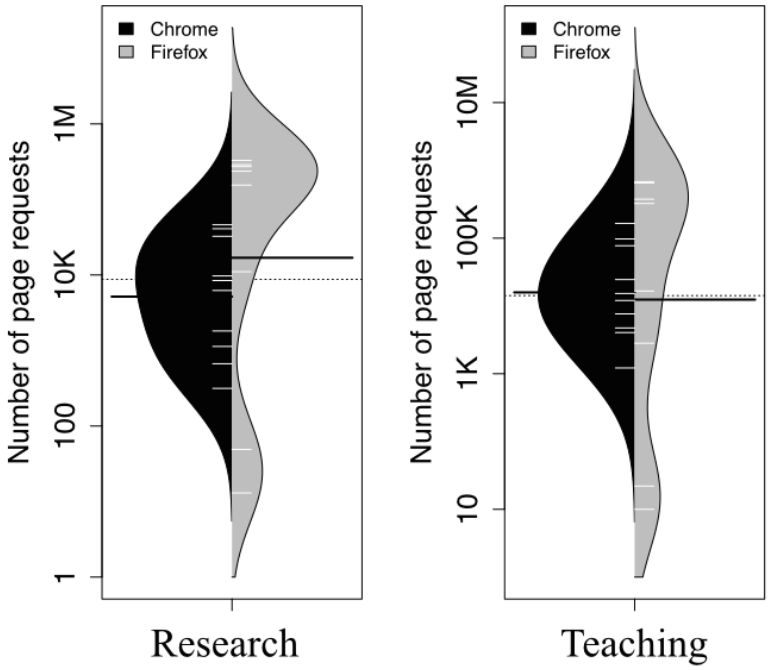


Figure 3.9: Differences in navigation behaviour between Chrome and Firefox users.

Historical trends of the user accesses for each release of a browser are demonstrated in Figure 3.10. Chrome users have similar navigation patterns when viewing both types of the web content: both research- and teaching-related pages were accessed from more recent releases of a browser starting from Chrome 3.0. Unlike Chrome, we found quite different patterns in viewing web content among Firefox users. Most hits to the research-related pages came from Firefox 1.0 (shown as release `r1` in Figure 3.10). We were surprised to see no accesses from the earlier releases of Firefox to the undergraduate teaching content. Firefox 2.0 is the oldest browser used to navigate to the teaching information, while the largest volume of the page views to this content originated from the Firefox releases 3.0 and 3.5. These findings suggest that undergraduate students, a younger population of users, have more up-to-date versions of a web browser (true for both Chrome and Firefox), while researchers, a more mature population of users, do not update their browsers as often as their younger counterparts.

Surprisingly, the first five releases of the Chrome browser have no or comparatively fewer number of hits to both types of the web content on our web site. Since our web traffic data is dated to February 2007 and Google Chrome was released later in 2008, we expected to see our visitors having early releases of Chrome installed on their computers. This observation suggests that Chrome adoption started slowly, with the first wave a year later with Chrome 3.0.

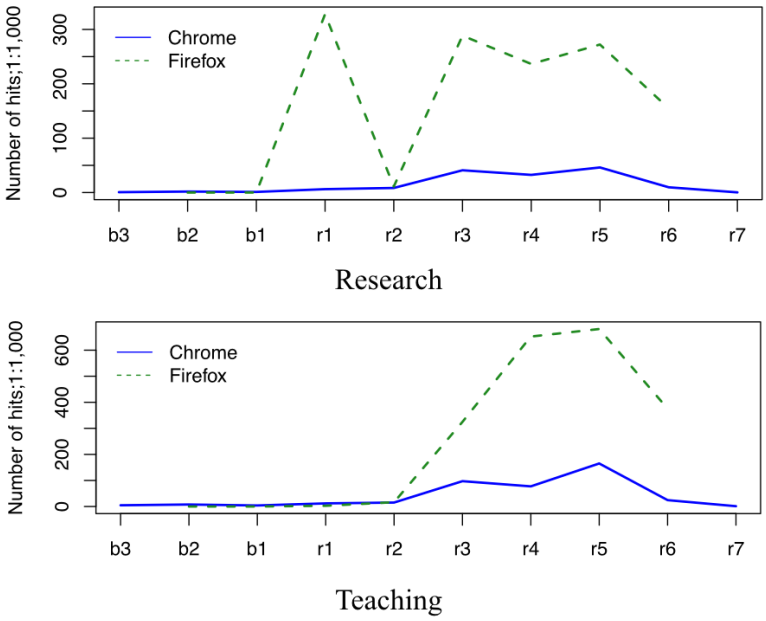


Figure 3.10: Distributions of user accesses to research and teaching content per each release of a browser.

Firefox 1.0 was released in November 2004, yet at the time of the first records in our logs, this release was more than two years old. Thus, Firefox was well adopted from the very first release (mainly due to earlier deployment of the browser under different names - *m/b* (mozilla/browser) under the Mozilla Suite, Phoenix, and Mozilla Firebird) and users stayed quite loyal to the initial release of the browser, hesitating to update to Firefox 1.5 until Firefox 2.0 became available in October 2006.

3.4 Discussion

This section discusses our findings and how they can be used to assess sustainability of software projects. We also discuss some threats to the validity of our study.

3.4.1 Software Sustainability

Software projects collect and store enormous archives of web usage data that are often disregarded or unused. Such archives contain data on user characteristics including user environment, locality,

browsing behaviour. This work shows that by applying mining techniques, we can extract such user characteristics from web logs to study user dynamic behaviour and adoption and combine them with traditional MSR data. By analyzing user behavioural and adoption patterns, we can, for example, assess several properties of sustainability of a software project. A sustainable software system can be defined as being "socially and environmentally bearable, viable economically without introducing impacts to the environment, and socially and economically equitable and accessible to everyone" [1]. Therefore, our empirical findings could provide the following insights on sustainability:

- Development process and practices, in particular release history of a product, can account for the maintenance quality. For example, shorter release cycles underlay better maintenance and delivery of more reliable and defect-free software.
- Firefox's support for older operating systems and platform compatibility fosters hardware sustainability and thus, reduces e-waste.
- Since Chrome users appear to be better distributed through the different regions in the world, Chrome supports larger diversity and ethnicity among its user population. Tracking cultural trends of the user adoption can provide information on the globalization of a project.
- User navigation behaviour can offer insights on user population and age diversity, as well as the success of a software release.

From these findings, we might be able to gain insights into sustainability of a system by measuring not only environmental aspects (e.g., development of software solutions that require lower energy consumption), but also social ones (user behaviour, adoption and interactions).

For open source products, it can be quite challenging to construct dynamic usage trends based on the number of product downloads due to the lack of a central repository to track such downloads. However, analysis of web usage data can provide valuable information on how users adopt and use software projects, analysis of historical trends can justify the popularity of certain releases of a software product. It is important to know the end-users of a product not only from the statistics collected by the marketing surveys but also by analyzing real usage data such as web logs to infer knowledge on user population, their technical environment, locality and navigation behaviour. The knowledge of these usage characteristics can lead to better understanding of how software systems are actually being used in practice.

3.4.2 Threats to Validity

External validity. Our findings are limited by the obtained data set: web server logs. Since we perform a comparative study, our school's web traffic is representative of the world-wide user

population of two browsers. Usage data sets are typically not publicly available due to privacy and business concerns. In our analysis we tried to balance the data representativeness and to avoid being biased by normalizing the number of the page requests by the system's usage share. Further studies may be necessary to confirm our findings.

Internal validity. Our web usage data has a few gaps due to the specifics of the university's backup routine, making the quality of the logs an important threat. A small number of CS undergraduate courses are offered through UW-ACE — a web-based course management system. Our web server logs do not include user accesses to such courses. We also need to mention that CS graduate courses normally reside on the faculty's web space. However, some faculty members have their web sites hosted by the web servers belonging to the Faculty of Mathematics. In such cases, we were not able to track accesses to these courses. For example, the CS846 course has been taught by several professors through the years and its web site is located on both `plg.uwaterloo.ca` and `se.uwaterloo.ca` sub-domains. Neither `plg.` nor `se.` sub-domains are hosted under `cs.uwaterloo.ca`.

Our choice of the granularity in analyzing web logs is determined by the existing challenges to identify users. Accurate tracking of the individual users by IP address is not always possible. A user who accesses the web from different machines (e.g., work vs. home computer) might have different IP addresses. A user that uses multiple browsers on the same machine will appear as multiple users (user agents will differ). ISPs can assign multiple IP addresses to a user for each request or several users might share same IP address.

Unlike Google, which uses cookies to track individual users and their navigation behaviour over the web, we are limited with the data captured in typical web traffic logs.

Since March 2011, Mozilla has accelerated the Firefox release cycle, and has provided several new releases with shorter timespans between them. Since our web logs are spanned from February 2007 until November 2010, this new Firefox update policy is not reflected in our study. Adoption of Firefox beyond release 3.6 is not considered in this paper. Newer web logs would be needed to reflect the influence of Chrome-like rapid release deployment practices of Firefox on its user adoption rates.

Construct validity. We have chosen a set of metrics to quantify the value of the collected data that captures only a part of its potential meaning. Our choices are a function of our interest in exploring the data and the availability and structure of the data sets.

Conclusion validity. We reported findings based on the statistical significance. We applied statistical analysis when needed, and were able to reject null hypotheses and detect interesting patterns.

3.5 Summary

In this chapter, we demonstrated how analysis of real-world usage data can reveal valuable information on how software systems are actually used in practice. In particular, we showed how usage data can be extracted from web server logs and combined with development information to provide insight into user dynamic behaviour, as well as adoption trends across various deployment environment. We took a statistical approach to mine dynamic usage data to determine characteristics of the user population. We analyzed discovered usage patterns and outlined the main differences in user adoption, deployment and usage of the Chrome and Firefox browsers. We also discussed how usage characteristics can help to account for sustainability of a software system.

We found that while Chrome is being adopted at a consistent rate across platforms, Linux users have an order of magnitude higher rate of Firefox adoption. Also, Firefox adoption has been concentrated mainly in North America, while Chrome users appear to be more evenly distributed across the globe. Finally, we detected no evidence in age-specific differences in navigation behaviour among Chrome and Firefox users; however, we hypothesize that younger users are more likely to have more up-to-date versions than more mature users.

Mining usage data is a powerful way to track and assess user dynamic behaviour and adoption trends of a software system.

Chapter 4

Managing Community Contributions

In Chapter 3, we looked into users and demonstrated how analytics can help to discover knowledge about their behaviour, adoption, and use of software systems. In this chapter, we focus on the problem of managing contributions from the greater user community, as well as developers. Open source projects grow and evolve through community participation and engagement. Yet, code review — the process of inspecting code modifications — can promote or discourage these contributions. We investigate how analytics can help developers and their teams to make better informed decisions related to code review tasks and processes.

Chapter Organization Section 4.1 describes the code review process and its role in the project development and maintenance. Section 4.2 introduces our artifact lifecycle model as a pattern for analyzing software data. Section 4.2.3 demonstrates how these lifecycle models can capture how the code review process works in both open-source and industrial projects. Section 4.4 illustrates our empirical studies performed to gain insights into both technical and non-technical factors that can influence the timeliness and outcome of code reviews. Section 4.5 summarizes the main contributions of this work.

Related Publications

- Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. The Secret Life of Patches: A Firefox Case Study. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2012, pages 447–455.
- Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Extracting artifact lifecycle models from metadata history. In *Proceedings of the ICSE Workshop on Data Analysis Patterns in Software Engineering (DAPSE)*, 2013, pages 17–19.

- Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. The Influence of Non-technical Factors on Code Review. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2013, pages 122–131.

4.1 Code Review

Code review is a key element of any mature software development process. It is particularly important for open source software (OSS) development, since contributions — in the form of bug fixes, new features, documentation, etc. — may come not only from core developers but also from members of the greater user community [7, 134, 140, 141]. Indeed, community contributions are often the life's blood of a successful open source project; yet, the core developers must also be able to assess the quality of the incoming contributions, lest they negatively impact the overall quality of the system.

The code review process evaluates the quality of source code modifications (submitted as patches) before they are committed to a project's version control repository. A strict review process is important to ensure the quality of the system, and some contributions will be championed and succeed while others will not. Consequently, the carefulness, fairness, and transparency of the process will be keenly felt by the contributors. In this work, we wanted to explore whether code review is an egalitarian process, i.e., contributions from various developers are being reviewed equally regardless of the developers' involvement on a project. Do patches from core developers have a higher chance of being accepted? Do patches from casual contributors take longer to get feedback? OSS projects typically offer a number of repositories or tools to communicate with the community on their feedback. For example, Mozilla provides Bugzilla for filing bug reports or submitting patches for a known defect. Users are also able to leave their feedback (praise, issues, or ideas) for Firefox on a designated website [130].

Improving community contribution and engagement is one of the challenges that many open source projects face. Thus, some projects establish initiatives to improve communication channels with the community surrounding the project. Such initiatives aim at identifying bottlenecks in the organizational process and creating improved eco-systems that actively promote contributions.

By studying interview data [12], we noticed that there is a perception among developers that the current Mozilla code review process leaves much room for improvement:

> "The review system doesn't seem to be tightly integrated into Bugzilla." (D10)
>
> "There are issues with reviews being unpredictable." (D11)
>
> "People have different definitions of the review process and disagree a lot, different types of requests". (D16)

Developers also expressed interest in having a version control system on patches, as well as better transparency on code review and statuses of their patches.

4.1.1 The Mozilla Project

Mozilla employs a two-tier code review process for validating submitted patches — *review* and *super review* [128]. The first type of a review is performed by a module owner or peers of the module; a reviewer is someone who has domain expertise in a problem area. The second type of review is called a super review; these reviews are required if the patch involves integration or modifies core Mozilla infrastructure. Currently, there are 29 super-reviewers [129] for all Mozilla modules and 18 reviewers (peers) on the Firefox module [131]. However, any person with level 3 commit access — core product access to the Mercurial version control system — can become a reviewer.

Bugzilla users flag patches with metadata to capture code review requests and evaluations. A typical patch review process consists of the following steps:

1. Once the patch is ready and needs to be reviewed, the owner of the patch requests a review from a module owner or a peer. The review flag is set to "*r?*". If the owner of the patch decides to request a super review, he may also do so and the flag is set to "*sr?*".

2. When the patch passes a review or a super review, the flag is set to "*r+*" or "*sr+*" respectively. If it fails review, the reviewer sets the flag to "*r-*" or "*sr-*" and provides explanation on a review by adding comments to a bug in Bugzilla.

3. If the patch is rejected, the patch owner may resubmit a new version of the patch that will undergo a review process from the beginning. If the patch is approved, it will be checked into the project's official codebase.

4.1.2 The WebKit Project

WebKit is an HTML layout engine that renders web pages and executes embedded JavaScript code. The WebKit project was started in 2001 as a fork of KHTML. Currently, developers from more than 30 companies actively contribute to this project; Google and Apple are the two primary contributors, submitting 50% and 20% of patches respectively. Individuals from Adobe, BlackBerry, Digia, Igalia, Intel, Motorolla, Nokia, Samsung, and other companies also contribute patches to the project.

The WebKit project employs an explicit code review process for evaluating submitted patches; in particular, a WebKit reviewer must approve a patch before it can land in the project's version control repository. The list of official WebKit reviewers is maintained through a system of voting

to ensure that only highly-experienced candidates are eligible to review patches. A reviewer will either accept a patch by marking it `review+` or ask for further revisions from the patch owner by annotating the patch with `review-`. The review process for a particular submission may include multiple iterations between the reviewer and the patch writer before the patch is accepted (lands) in the version control repository.

4.1.3 The Blink Project

Google forked WebKit to create the Blink project in April 2013. Google decided to fork WebKit because they wanted to make larger-scale changes to WebKit to fit their own needs that did not align well with the WebKit project itself. Several of the organizations who contributed to WebKit migrated to Blink after the fork.

Every Blink patch is submitted to the project's issue repository[1]. The reviewers on the Blink project approve patches by annotating it "LGTM" ("Looks Good To Me", case-insensitive) on the patch and reject patches by annotating "not LGTM". In this work, we consider WebKit's `review+`/`review-` flags and Blink's "lgtm"/"not lgtm" annotations as equivalent. Since Blink does not have an explicit review request process (e.g., `review?`), we infer requests by adding a `review?` flag a patch as soon as it is submitted to the repository. Since patches are typically committed to the version control system by an automated process, we define landed patches as those followed by the automated message from the "commit bot". The last patch on the issue is likely to be the patch that eventually lands to the Blink's source code repository. Committers could optionally perform a manual merge of the patches to the version control system, although we do not consider these due to their infrequence.

4.2 Artifact Lifecycle Models

Software development projects often face challenges when trying to leverage development data that accrues in various repositories (e.g., issue-tracking systems, version control systems, etc.) during software development. The volume of data in these repositories makes it difficult to manually generate a reasonable understanding of both the state of the data and how it has evolved over time. In this section, we introduce a data pattern called *artifact lifecycle models* that can be used to extract a succinct summary of how specific properties within a development artifact have changed over time.

Lifecycle models can be applied to any data that changes its state, or is annotated with new data, over time. For example, issues often start their lives as `OPEN` and are eventually `RESOLVED` or marked `WONTFIX`. A lifecycle model for issues could build up an aggregate graphical

[1] code.google.com/p/chromium/issues/

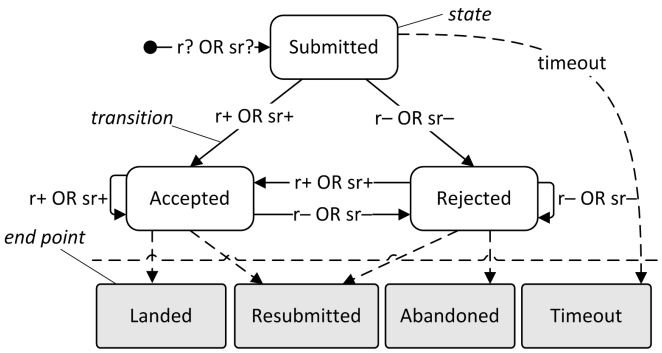


Figure 4.1: The lifecycle of a patch.

representation of how bugs flow through these three states. For example, the lifecycle model would capture the proportion of bugs that are reopened from a `WONTFIX` state that might be interesting for a manager considering changes to their issue triage process. Such lifecycle models often exist in process models (if defined). Therefore, extracting one from the history enables comparing the defined lifecycle with the actual one.

In Section 4.2.3 we will apply lifecycle models to instead capture how the code review process works in both open-source and industrial projects. We demonstrate how these models both capture interesting traits within individual projects, but are also succinct enough to compare traits between projects.

4.2.1 Model Extraction

Lifecycle models can be extracted for any data elements that evolve over time from their metadata history. These elements may include issues' change status, reviewed patches, evolved lines of code. By examining how each artifact evolves over time, we can build a summary that captures common dynamic evolutionary patterns. Each node in a lifecycle model represents a state that can be derived by examining the evolution of an artifact.

The lifecycle model is presented as a simple graph that shows the states the model contains; edges capture the transitions between the lifecycle states. An example of the patch lifecycle model is given in Figure 4.1.

Lifecycle models can be generated as follows:

1. Determine key states of the system or process (e.g., a number of events that could occur) and their attributes.

2. Define necessary transitions between the states and specify an attribute for each transition.
3. Define terminal outcomes (optional).
4. Define and gather qualitative or quantitative measurements for each state transition and if present, final outcomes. In addition to these measurements, the time spent in the state or the transition to another state can also be considered and analyzed.

The pattern provides means to model, organize, and reason about the data or underlying processes otherwise hidden in individual artifacts. While the lifecycle model can be modified or extended depending on the needs, they work best when the state space is well defined. Applying this pattern to complex data may require abstraction. Artifacts may contain unnecessary details or very rich metadata that could impede the model extraction and comprehension. Thus, lifecycle models promote abstracting away from the details and focusing on the key states of the data to ease understanding of a system or process and how it changes over time.

4.2.2 Possible Applications

The lifecycle model is a flexible means that can be used in a variety of software investigation tasks. For example:

- Issues: The state of issues changes as developers work on issues. Common states here would include `NEW`, `ASSIGNED`, `WONTFIX`, `CLOSED`, although these may vary from project to project.
- Component/Priority assignments: Issues are often assigned to specific code components and are given priority assignments. As an issue is triaged and worked upon these assignments can change.
- Source code: The evolutionary history of any line or block of source code can be considered capturing `Addition`, `Deletion`, and `Modification`. This data can be aggregated at the line, block, method, or class level.
- Discussions: Online discussions, e.g., those on StackOverflow, can have status of `CLOSED`, `UNANSWERED`, `REOPENED`, `DUPLICATE`, `PROTECTED`.

4.2.3 Lifecycle Analysis

In this section, we apply lifecycle models to the code review processes to highlight how code reviews happen in practice. The goal of the model in this instance is to identify the way that patches typically flow through the code review process, and also to identify exceptional review paths. We have extracted the code review lifecycle model for Mozilla Firefox (Section 4.2.3.1), WebKit (Section 4.2.3.2), and Blink (Section 4.2.3.3).

4.2.3.1 Mozilla Firefox

We modelled the patch lifecycle by examining Mozilla's code review policy and processes and comparing it to how developers worked with patches in practice. To generate a lifecycle model, all events have been extracted from Mozilla's public Bugzilla instance. We extracted the states a patch can go through and defined the final states it can be assigned to.

All code committed to Mozilla Firefox undergoes code review. First, a developer will submit a patch containing their change to Bugzilla and request a review. Then, after evaluating the patch, reviewers will annotate the patch either positively or negatively. For highly-impactful patches super-reviews may be requested and performed. Once a patch has been approved by the reviewer(s), it can finally be added to the Mozilla code base via the version control repository.

We generated the model by applying the four steps mentioned in Section 4.2.1:

1. **State identification:** Patches under review exist in one of three key states: `Submitted`, `Accepted`, and `Rejected`.
2. **Transition extraction:** Code reviews transition patches between the three primary states. A review request is annotated with $r?$. Positive reviews are denoted with a $r+$. Negative reviews are annotated $r-$. Super reviews are prepended with an s (e.g., $sr+/sr-$).
3. **Terminal state determination:** At the end of its lifetime, a patch can transition into one of four terminal states. `Landed` patches are those that pass the code review process and are included in the version control system. `Resubmitted` patches are patches that a developer decides to further refine based on reviewer feedback. `Abandoned` patches capture those patches the developer opted to abandon based on the feedback they received. Finally, the `Timeout` state represents patches for which no review was given even though it was requested.
4. **Measurement:** For this study we measured both the number of times each transition happened along with the median time taken to transition between states.

Figure 4.1 presents a model of the patch lifecycle for the Mozilla review process [17]. The diagram shows the various transitions a patch can go through during its review process. A transition represents an event which is labelled as a flag and its status reported during the review process.

We considered only the key code review patches having "review" (r) and "super-review" (sr) flags on them. Other flags on a patch such as "feedback", "ui-review", "checkin", "approval_aurora", or "approval_beta", as well as patches with no flags were excluded from the analysis.

The code review process begins when a patch is submitted and a review is requested; the initial transition is labelled as "r? OR sr?", i.e., a review or super review is requested respectively. There are three states a patch can be assigned to: `Submitted`, `Accepted`, and `Rejected`. Once the review is requested (a flag contains a question mark "?" at the end), the patch enters the `Submitted` state. If a reviewer assigns "+" to a flag (e.g., "r+" or "sr+"), the patch goes to the `Accepted` state; if a flag is reported with a status "–" (e.g., "r–" or "sr–"), the patch is `Rejected`.

Both the `Accepted` and the `Rejected` states might have self-transitions. These self-transitions, as well as the transitions between the `Accepted` and the `Rejected` states illustrate the double review process. The double review process takes place in the situations when a reviewer thinks that the patch can benefit from additional reviews or when code modifications affect several modules and thus need to be reviewed by a reviewer from each affected module.

We define and call end points as *Landed*, *Resubmitted*, *Abandoned*, and *Timeout*. These end points represent four final outcomes for any given patch. During the review process each patch is assigned to only one of these four groups:

- *Landed* — patches that meet the code review criteria and are incorporated into the codebase.
- *Resubmitted* — patches that were superseded by additional refinements after being accepted or rejected.
- *Abandoned* — patches that are not improved after being rejected.
- *Timeout* — patches with review requests that are never answered.

The cumulative number of the patches in Landed, Resubmitted, Abandoned, and Timeout is equal to the number of the Submitted patches.

Figure 4.2 illustrates the lifecycle model of the patch review process for core Mozilla contributors. The lifecycle demonstrates some interesting transitions that might not otherwise be obvious. For instance, a large proportion of accepted patches are later resubmitted by authors for revision. Also, we can see that rejected patches are usually resubmitted, easing concerns that rejecting a borderline patch could cause it to be abandoned. We also see that very few patches timeout in practice. From the timing data (shown in green colour) we see that it takes an average of 8.9 hours to get an initial $r+$ review, while getting a negative $r-$ review takes 11.8 hours. Application of this pattern on the code review process of Firefox (and comparing the lifecycle models for patch review between core and casual contributors) has generated a discussion and raised concerns [127] among the developers on the Mozilla Development Planning team:

> "... rapid release has made life harder and more discouraging for the next generation of would-be Mozilla hackers. That's not good... So the quality of patches from casual

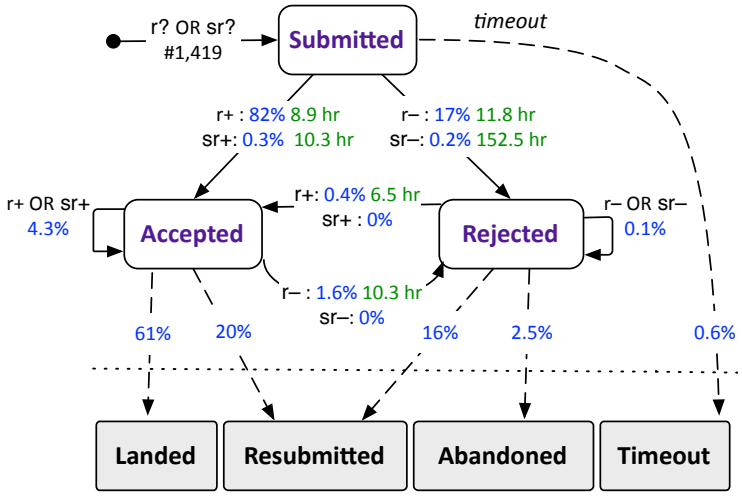


Figure 4.2: Mozilla Firefox's patch lifecycle.

contributors is only slightly lower (it's not that all their patches are rubbish), but the amount of patches that end up abandoned is still more than 3x higher. :-(" [Gerv Markham]

"I do agree that it's worth looking into the "abandoned" data set more carefully to see what happened to those patches, of course." [Boris Zbarsky]

4.2.3.2 WebKit

The lifecycle model can be easily modified according to the dataset at hand. For example, we have applied the pattern to study the code review process of the WebKit project. The model of the patch lifecycle is shown in Figure 4.3. While states remain the same, there are fewer state transitions since the WebKit project does not employ a "super review" policy. Also, the self edges on the `Accepted` and `Rejected` states are absent because while Mozilla patches are often reviewed by two people, Webkit patches receive only individual reviews. Finally, a new edge is introduced between `Submitted` and `Resubmitted` as Webkit developers frequently "obsolete" their own patches and submit updates before they receive any reviews at all.

Since WebKit is an industrial open source project — developed mainly by a small set of companies — we were particularly interested to compare its code review process to that of other open source projects. To do so, we extracted the WebKit's patch lifecycle (Figure 4.3) and compared it with the previously studied patch lifecycle of Mozilla Firefox [17] (Figure 4.2).

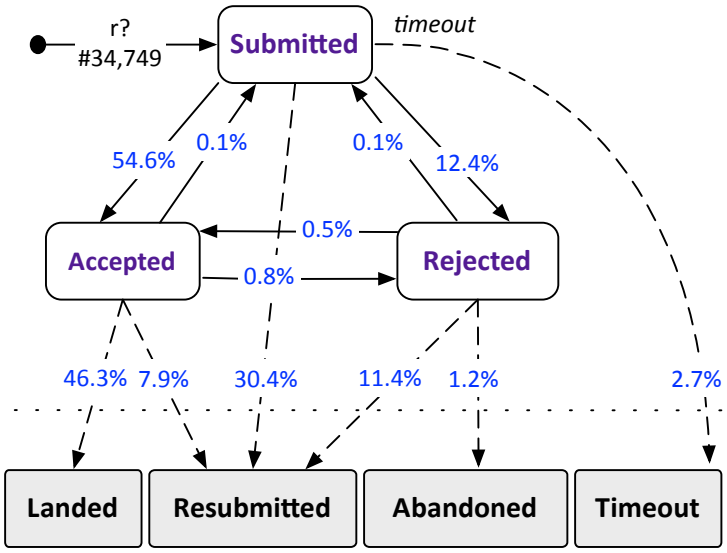


Figure 4.3: WebKit's patch lifecycle.

The patch lifecycle captures the various states patches undergo during the review process, and characterizes how the patches transition between these states. The patch lifecycles enable large data sets to be aggregated in a way that is convenient for analysis. For example, we were surprised to discover that a large proportion of patches that have been marked as accepted are subsequently resubmitted by authors for further revision. Also, we can see that rejected patches are usually resubmitted, which might ease concerns that rejecting a borderline patch could cause it to be abandoned.

While the set of states in our patch lifecycle models of both WebKit and Firefox are the same, WebKit has fewer state transitions; this is because the WebKit project does not employ a "super review" policy. Also, unlike in Mozilla, there are no self-edges on the `Accepted` and `Rejected` states in WebKit; this is because Mozilla patches are often reviewed by two people, while WebKit patches receive only individual reviews. Finally, the WebKit model introduces a new edge between `Submitted` and `Resubmitted`; WebKit developers frequently "obsolete" their own patches and submit updates before they receive any reviews at all. One reason for this behaviour is that submitted patches can be automatically validated by the external test system; developers can thus submit patches before they are to be reviewed to see if they fail any tests. All together, however, comparing the two patch lifecycles suggests that the WebKit and Firefox code review processes are fairly similar in practice.

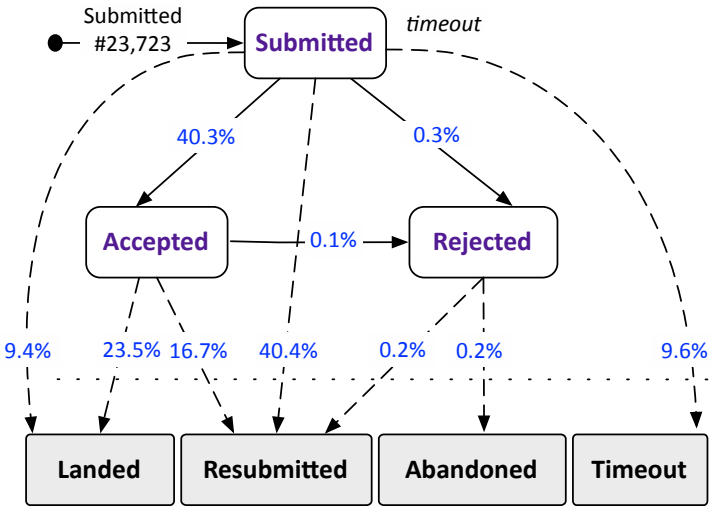


Figure 4.4: Blink's patch lifecycle.

4.2.3.3 Blink

Blink's patch lifecycle is depicted in Figure 4.4, which shows that 40% of the submitted patches receive positive reviews and only 0.3% of the submitted patches are rejected. Furthermore, a large portion of patches (40.4%) are resubmitted. This is because Blink developers often update their patches prior receiving any reviews; as with WebKit, this enables the patches to be automatically validated. At first glance, outright rejection does not seem to be part of the Blink code review practice; the `Rejected` state seems to under-represent the number of patches that have been actually rejected. In fact, reviewers often leave comments for patch improvements, before the patch can be accepted.

The model also illustrates the iterative nature of the patch lifecycle, as patches are frequently `Resubmitted`. The edge from `Submitted` to `Landed` represents patches that have been merged into Blink's source code repository, often after one or more rounds of updates. Developers often fix "nits" (minor changes) after their patch has been approved, and land the updated version of the patch without receiving additional explicit approval. The lifecycle also shows that nearly 10% of patches are being neglected by the reviewers (i.e., `Timeout` transition); `Timeout` patches in Blink can be considered as "informal" rejects.

Comparing the patch lifecycle models of WebKit and Blink, we noticed that Blink has fewer state transitions. In particular, the edges from the `Accepted` and `Rejected` back to `Submitted` are absent in Blink. Since Blink does not provide any indication of the review request on patches, we had to reverse engineer this information for all patches by considering the timestamps on each item (patch) in the series. We automated this process by putting the `Submitted` label to the

patch at the time the patch was filed to the issue repository.

Blink also accepts a smaller portion of patches (about 40% of all contributions compared to the WebKit's 55% of submitted patches), yet officially rejects less than 1%. While timeouts are more frequent for Blink patches than WebKit ones, timeouts can be viewed as "unofficial" rejects in the Blink project where disapprovals are uncommon. Blink appears to exhibit a larger portion of patches being resubmitted (a 10% increase compared to the WebKit patches), including resubmissions after patches are successfully accepted (16.7%).

Finally, a new edge is introduced between `Submitted` and `Landed`, accounting for those contributions that were committed to the code base without official approval from the reviewers; these cases typically represent patch updates. Both WebKit and Blink developers frequently "obsolete" their own patches and submit updates before they receive any reviews at all.

Comparing the two patch lifecycle models suggests that the WebKit and Blink code review processes are similar in practice; at the same time, it appears that Google's review policy may not be as strict as the one employed by Apple on the WebKit project.

4.2.4 Summary

Software developers and managers make decisions based on the understanding they have of their software systems. This understanding is both built up experientially and through investigating various software development artifacts. While artifacts can be investigated individually, being able to summarize characteristics about a set of development artifacts can be useful. In Section 4.2), we proposed *artifact lifecycle models* as an effective way to gain an understanding of certain development artifacts. Lifecycle models capture the dynamic nature of how various development artifacts change over time in a graphical form that can be easily understood and communicated. Lifecycle models enables reasoning of the underlying processes and dynamics of the artifacts being analyzed. We described how lifecycle models can be generated (Section 4.2.1) and demonstrated how they can be applied to the code review processes of three development projects (Section 4.2.3).

4.3 Evaluating Code Review Lifecycle Models

We now present our empirical study on the patch contributions within the Mozilla Firefox project. We evaluate code review process and practice by comparing lifecycles of the rapid release and traditional development models, as well as lifecycle models of core and casual contributors.

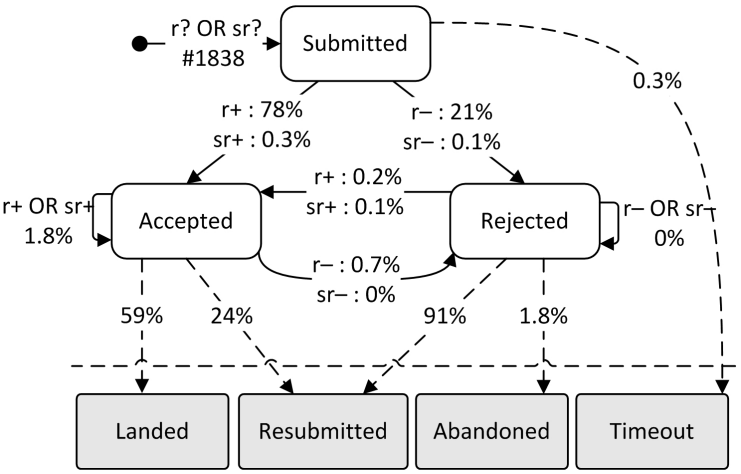


Figure 4.5: Patch lifecycle for core contributors for pre-rapid release.

4.3.1 Rapid Release vs. Traditional Development Model

On April 12, 2011 Mozilla migrated to a rapid release model (with releases every 6 weeks) from a more traditional release model (with releases averaging every 10 months). We were interested to investigate whether the patch lifecycle of the pre-rapid release differs from the post-rapid release process. Therefore, we compared patch contributions between the following time periods:

- pre-rapid release span: 2010-04-12 to 2011-04-12
- post-rapid release span: 2011-04-12 to 2012-04-12.

One of the key properties of open source projects is that community members are voluntarily engaged in a project. Thus, the amount of the patches received from a particular member varies depending on his interest (and ability) in contributing to the project. The total number of patches submitted in pre- and post-rapid release periods are 6,491 and 4,897 respectively.

We looked at the amount of patches each developer submitted during a two-year timespan and compared various groups of the received contributions. We empirically determined the right amount of contributions to define two different groups of contributors: **casual** and **core**. We compared patch distributions within the lifecycle model for different sets: 10, 20, or 50 patches or fewer for casual contributors; and more than 10, 20, 50, or 100 patches for core contributors. We did not find a significant difference in the distribution of patches for casual developers among 10/20/50 patch sets or for core developers among 10/20/50/100 patch sets. However, to avoid

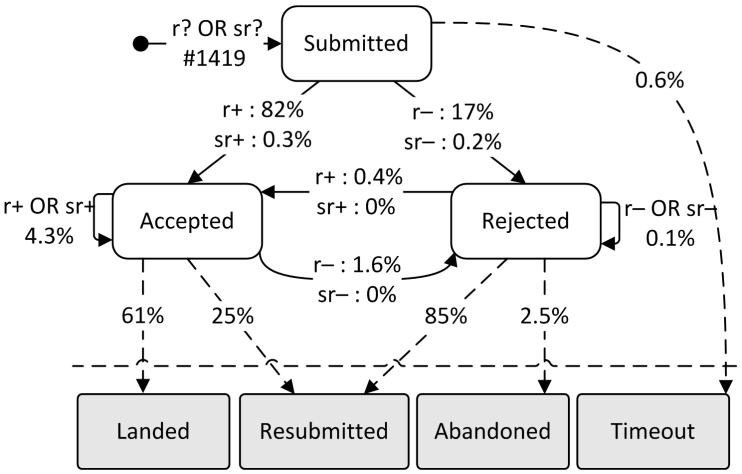


Figure 4.6: Patch lifecycle for core contributors for post-rapid release.

being biased toward the core group when choosing the right threshold for the core group (due to large amount of patches), we decided that 100 patches or more is reasonable to define core contributors. Therefore, we chose to set the casual group at 20 patches or fewer and core group at 100 patches or more. Thus, we considered both the distribution of patches in a patch lifecycle within each group of contributors and the distribution of patches between casual and core groups. The contributors who submitted more than 20 and fewer than 100 patches (12% of all contributors) were out of the scope of the analysis.

We first compared patch lifecycles for pre- and post-rapid release periods; we considered only patches submitted by core contributors.

Figure 4.5 and Figure 4.6 illustrate the lifecycles a patch goes through in pre- and post-rapid release time periods. Edges represent the percentage of the patches exiting state X and entering state Y.

Comparing Figure 4.5 and Figure 4.6, we observed the following. Post-rapid release is characterized by the 4% increase in the proportion of patches that get in and the 4% decrease in the percentage of the patches that get rejected. Although the post-rapid release world has a slightly higher percentage of the patches that pass the review and land in the codebase (61% vs. 59% submitted before April 2011), there are more patches that are abandoned (2.5% vs. 1.8%) and twice as many patches that receive no response. After switching to rapid release, developers are less likely to resubmit a patch after it fails to pass a review – a 6% decrease in the number of such patches.

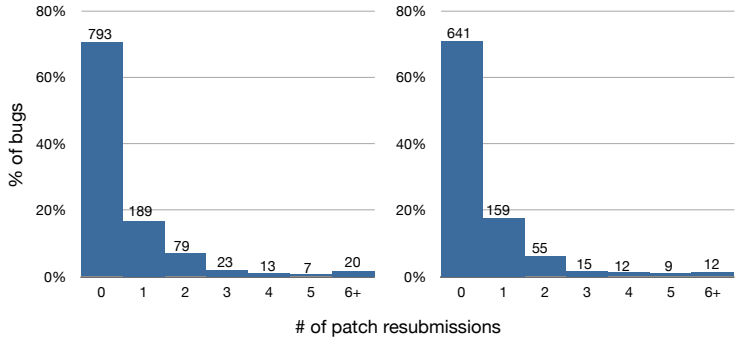


Figure 4.7: Patch resubmissions per bug for pre- (left) and post-rapid (right) release periods.

Super reviews are, in general, sparse – only 0.4% (before April 2011) and 0.5% (after April 2011) of the patches are being super reviewed. The proportion of patches that pass super reviews remains unchanged, at 0.3% for both periods; while post-rapid release super-reviewers reject twice as many patches than before.

For post-rapid world, we found that there are two times more patches that are first rejected but are later accepted and that the proportion of the initially accepted patches that are rejected during the second review phase is increased by a factor of 2.3x, from 0.7% to 1.6%. The situation when a patch goes from r+ to r– occurs quite often in practice. In such a situation, a patch is approved and is checked into the *try* or main tree, but subsequently fails to pass tests on the try server or during the nightly builds. As soon as it crashes the tree trunk it gets flagged "r–".

Patches submitted and accepted after April 2011 are more likely to go through the second round of reviews (4.3% vs. 1.8%). This happens in two cases: 1) when someone else comes up with a better version of a patch or an alternative approach of performing a change; 2) when a reviewer flags the patch as "r+ w/ nits". Such "reviews with nits" are quite common. Reviewers approve a patch even if they are not completely satisfied with the patch. While they comment on what changes (nits), typically small, need to be fixed, they leave it to the contributor to fix them. Such flag assignment requires a certain level of trust between reviewer and developer. "r+ w/ nits" can also happen when a patch requiring small modifications is approved and fixed by the reviewer himself. It is likely that these patches come from casual developers who do not have commit access to checkin the patch themselves.

While the results report some differences in the proportion of patches on each transition of the model in pre- and post-rapid release, the patch lifecycle appears not to have changed much.

We noticed that 39% (pre-rapid) and 36% (post-rapid) of all patches are being resubmitted at least once. We checked all the bugs including ones with no patch resubmissions. Figure 4.7 presents two histograms showing the proportion of bugs and the number of patch resubmissions

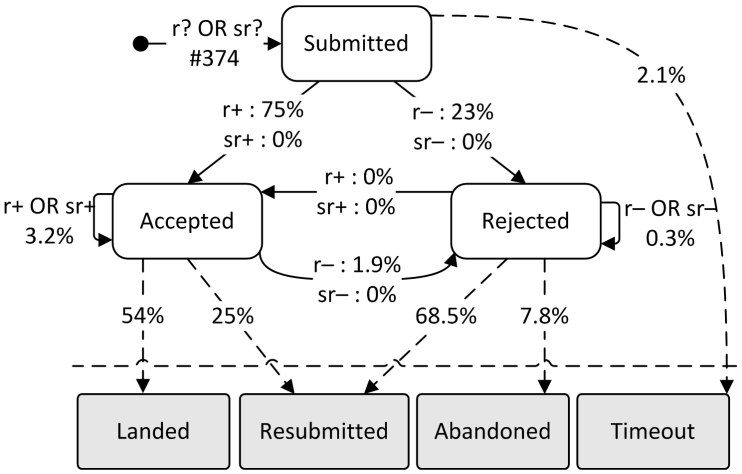


Figure 4.8: Patch lifecycle for casual contributors.

for pre- (left) and post-rapid (right) release periods. There were 1,124 and 903 bugs filed by core contributors during pre- and post-rapid release periods respectively.

The results show that pre and post-rapid release bugs have a similar distribution of the resubmitted patches. 71% of bugs had no patches that were resubmitted. While 1.8% of pre-rapid release bugs had patches that were resubmitted over 6 times compared to 1.3% of the post-rapid release ones, the highest number of resubmissions in a bug, i.e., 25 patches, is found in the post-rapid release world.

4.3.2 Core vs. Casual Contributors

Since the lifecycle of a patch for the post-rapid release phase remains almost unchanged, we decided to explore whether there is a difference between patch lifecycles for core (>100 patches) vs. casual (<20 patches) contributors (for a post-rapid release phase only). Since we found no difference in patch lifecycles of casual and core contributors for pre- and post-rapid release periods, we only report the comparison of casual and core contributors for a post-rapid release model to avoid repetition of the results.

Comparing the lifecycles for core (Figure 4.6) vs. casual contributors (Figure 4.8), we noticed that, in general, casual contributors have 7% fewer patches that get accepted or checked into the codebase and have 6% more patches that get rejected. The amount of the patches from casual contributors that received no response or are being abandoned is increased by the factor of 3.5x and 3.12x respectively. Review requests with timeouts are likely those that are directed to wrong reviewers or landed to the "General" component that does not have an explicit owner. If a review

was asked from a default reviewer, a component owner, the patch is likely to get no response due to heavy loads and long review queues the default reviewer has. Since contributors decide what reviewer to request an approval from, they might send their patch to the "graveyard" by asking the wrong person to review their patch. The process, by design, lacks transparency on the review queues of the reviewers.

Moreover, casual contributors are more likely to give up on a patch that fails a review process – 16% fewer patches are resubmitted after rejection. Unlike patches from core developers, once rejected patches from "casual" group do not get a chance to get in (0% on the "r– → r+" transition) and are three times more likely to receive a second negative response.

The results show that patches submitted by casual developers do not require super reviews, as we found no super review requests on these patches. We found this unsurprising, since community members who participate occasionally in a project often submit small and trivial patches [167].

Our findings suggest that patches from casual contributors are more likely to be abandoned by both reviewers and contributors themselves. Thus, it is likely that these patches should receive extra care to both ensure quality and encourage future contributions from the community members who prefer to participate in the collaborative development on a less regular basis.

4.3.3 Lifespan of Transitions

While Section 4.3.2 demonstrated the difference in the patch distribution on each transition of the model for both core and casual developers in pre- and post rapid release spans, the model lacks the temporal data on how long these transitions take.

A healthy code review process needs to handle patch submissions in a timely manner to avoid potential problems in the development process [22]. Since a patch can provide a fix to an existing software bug, patches should ideally be reviewed as soon as they arrive.

Since the lifecycle of a patch remains almost unchanged after Mozilla has switched to rapid release trains, we wonder whether patches are reviewed in a more timely fashion. To answer this question, we analyzed the timestamps of the review flags reported during the review process. We considered only the review flags that define the transitions of the patch lifecycle model. We looked at the time a review flag is added at and computed deltas (in minutes) between two sequential flags in a patch. These two consecutive flags form a flag pair that corresponds to a transition in the patch lifecycle model.

We were interested to determine the time it takes for a patch to be accepted and rejected for core developers in pre- and post-rapid release phases. Figure 4.9 and Figure 4.10 report the results. About 20% and 23% of pre- and post-release patches are accepted within the first hour of being submitted; and 63% and 67% of the overall patches are accepted within 24 hours. Thus, since April 2011 Mozilla has increased the amount of patches that get in within 24 hours by 3–4%,

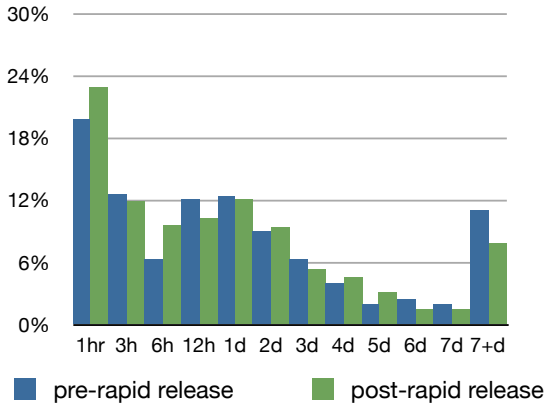


Figure 4.9: Patch acceptance time for core contributors.

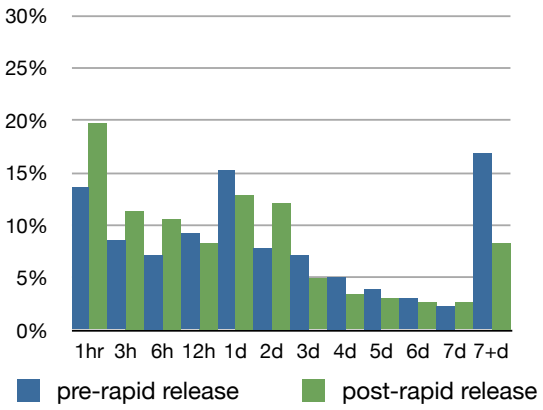


Figure 4.10: Patch rejection time for core contributors.

while decreasing the number of patches that require longer attention (over a week review cycle) by 3% (compared to previous 11%).

Mozilla has also increased the amount of patches that receive a negative decision within first hour after switching to rapid release (20% vs. 14%). The number of patches that stay in a review queue for longer than a week is reduced by more than half (from 17% to 8%). One possible explanation is that Mozilla tries to manage the overwhelming number of the received contributions by enforcing reviewers to address patches as soon as they come in.

We then compared the time it takes for patches to be accepted (Figure 4.11) and rejected (Figure 4.12) for core and casual contributors. The main difference is that patches from casual developers are accepted faster, they are more likely to be approved within first hour (26% vs. 23%). Since contributions from casual developers tend to be smaller, this observation is not surprising. This finding conforms to the previous research that patches from casual contributors are accepted faster than those from core developers, since reviewers tend to favour smaller code modifications as they are easier to review [141]. In general, the acceptance rate and the proportion of patches for each time interval is consistent among two groups of the contributors.

We observed that patches from core contributors are rejected faster, around 20% of these patches are rejected within first hour of their submission. While we did not account for the size of patches, contributions from core developers are generally larger and more complex [7, 140]. By rejecting larger patches early, core developers are notified quickly that their code needs further work to comply with quality standards, thus letting core developers make necessary changes quickly without wasting much time [22]. In contrast to core contributors, casual developers do not receive negative feedback on their patches until a later time, with 13% of all the patches being rejected over a week later. In general, core developers considered as the elite within the

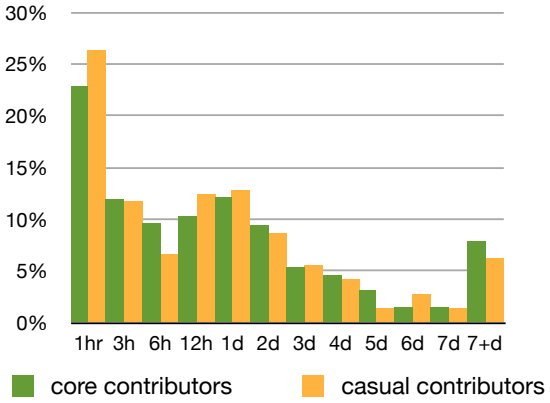


Figure 4.11: Patch acceptance time for the rapid release period.

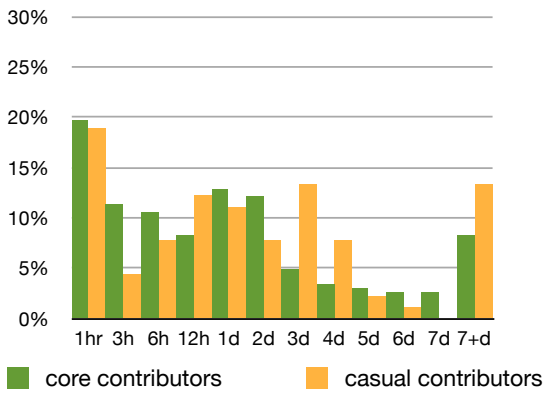


Figure 4.12: Patch rejection time for the rapid release period.

community [149] as their contributions are more likely to affect the quality of the codebase and the development of the OSS project. Therefore, it is unsurprising that their patches receive faster negative response. On the other hand, if a patch fails to attract reviewer's attention, the review is likely to be postponed until the patch generates some interest among reviewers or project team members [142].

We measured the time it takes for a patch to go from one state to another. Table 4.1 reports the median time (in minutes) each transition of the model takes.

Transition	Pre	Post	
		Core	Casual
r? → r+	693*	534*	494
r? → r–	1206*	710*	1024
r+ → r–	1181	390	402
r– → r+	1350	1218	15
sr? → sr+	525	617	n/a
sr? → sr–	6725	9148	n/a

Table 4.1: The median time of a transition (in minutes); * indicates statistical significance.

Statistically significant results are achieved for transitions "r? → r+" and "r? → r–" when comparing pre- and post-rapid release populations of patches (p-value<0.05). This shows that after the switch to a rapid release model, decisions on whether to accept or reject a contribution are made faster.

The transition "r? → r+" is a lot faster than "r? → r–" for all three groups such as pre core, post core and casual. This means that reviewers provide faster responses if a patch is of good quality.

To our surprise, the fastest "r? → r+" is detected for casual developers. Our findings show that contributions from casual developers are less likely to get a positive review; yet if they do, the median response rate is about 8 hours (in contrast to 9 hours for core developers).

Super reviews, in general, are approved very quickly, within 8–10 hours. This finding conforms to the Mozilla's code review policy – super-reviewers do provide response within 24 hours of a super review being requested. However, it takes much longer for a super-reviewer to reject a patch requiring an integration. It takes over 4 to 6 days for a super-reviewer to make such a decision, often through an extensive discussion with others.

"r+ → r–" is a lot faster for the rapid release world (6.5 hours compared to 20 hours), meaning previously accepted patches are more likely to be reviewed first during the second round of a review process. A possible explanation for this is that a reviewer might expect that such patches are of better quality and thus they appeal more to him.

In a post-rapid release phase "r– → r+" is a lot slower for core developers, mainly because there is only one occurrence of this transition for the "casual" group.

4.3.4 Lifespan of Patches

In our study, we define the lifespan of a patch to be the period during which a patch exists, i.e., from the time the patch is submitted until the time it is in one of the three final outcomes (Landed, Abandoned, or Resubmitted). Patches with no review responses (in a Timeout state) are excluded from the analysis since they are "undead" until they attract interest among developers. Table 4.2 reports the mean lifespan of patches in each final outcome.

Final Outcome	Pre	Post	
		Core	Casual
Landed	4.5	2.7	2.1
Abandoned	31.2	11.1	7.1
Resubmitted	3.8*	2.5*	4.3

Table 4.2: Average (mean) lifespan of a patch (in days) for final outcomes; * indicates statistical significance.

Landed patches submitted by casual developers have the shortest average "life expectancy" at 2.1 days; while abandoned patches submitted by core developers prior April 2011 have the longest average "life" at 31.2 days.

The longest average lifespan in Landed group is for pre-rapid release patches at 4.5 days (compared to 2.7 and 2.1 days for post-rapid release patches). The difference between the average lifespans of the patches submitted by core contributors before and after April 2011 was statistically significant ($p<0.005$). As expected, after switching to a rapid release model, the reviews of patches from core developers are done faster (2.5 days vs. 3.8 days).

Decisions to land patches from casual contributors are made very fast ($\sim$2 days) with 26% of receiving an "accept" within the first hour. Since code modifications from the casual group members are of smaller size and less critical [142], if they are found to be enough interesting or important, they are quickly reviewed. In contrast, if these patches fail to generate interest or have resubmissions, the review decisions take longer ($\sim$4 days).

On average, the lifespan of a pre-rapid release patch is 4.6 days, while the lifespans of post-rapid release patches are 2.8 days and 3.4 days for core and casual developers respectively. Therefore, the shortest-lived Firefox patch is a zero-minute patch submitted before April 2011. The review request on the patch was made and self-approved and the actual time delta was 14 seconds. We also calculated the average lifespan of all the patches in our dataset. The average patch "lives" for 3.7 days, which is about the same as the common mosquito.

4.3.5 Discussion

Software projects put considerable effort into defining and documenting organizational rules and processes. However, the prescribed processes are not always followed in practice. Therefore, we tried to detect any disconnects between Mozilla's code review policy and actual practice.

We noticed that two reviewer policies are not being strictly enforced. Only 6.4% of patches from core developers and 5.4% of patches for casual developers are being double reviewed. Mozilla's policies are not consistent due to its commitment to the open communication standards.

We noticed that super reviews happen only rarely. As expected, patches from casual developers do not undergo super reviews as these patches are unlikely to require any integration changes (changes in API, significant architectural refactoring, etc.). One of the Mozilla developers explains: *"You do not want to do large scale changes. If the codebase is moving fast enough, your Bugzilla patch can go stale in that approach"*.

We also looked at single patch contributions and found that while 65% of submitted patches successfully landed to the codebase, 20% of the initial review requests were neglected and 15% of patches were abandoned after receiving "r–" reviews. To promote community contributions, open source projects may wish to be more careful in reviewing patches from the community members who are filing their first patch.

Our findings suggest that rapid release review process offers faster response rates. Within the last year, Mozilla has hired more developers, as well as has started a campaign on providing faster response on contributions (48 hours), in particular for those submitted from the external community members (i.e., not developers employed by the Mozilla Corporation). This shows that Mozilla is learning from the process and being more efficient in managing huge number of contributions from the community.

The code review process is a key part of software development. Like many open source projects, the codebase of Mozilla Firefox evolves through contributions from both core developers as well as the greater user community. Our findings show that Mozilla's effort to reduce the time patches spend waiting for reviews after switching to the rapid-release model was successful. We observed that review of patches from core contributors follows "reject first, accept later" approach, while review of patches from casual developers follows "accept first, reject later" approach. While the size of contributions might account for this difference, further research is needed.

While large portion of submitted patches is accepted by reviewers, there is the risk of alienating valuable contributors when reviewers do not respond to the submitters of received contributions. This is particularly important for the patches submitted by casual developers. Creating a first positive experience with the review process for community members is important to encourage possible future contributions from them. We hope that our research findings can help inform the decisions made by various practitioners, including Mozilla Corporation, in improving their patch management practices.

Threats To Validity While we performed a comprehensive assessment of the patch review process and practice of Mozilla Firefox, our empirical study is a subject to external validity; we can not generalize our findings to the patch resubmission and review processes of other open source projects. However, the described model of the patch lifecycle is general enough to be easily adjusted by other practitioners should they decide to assess their own process.

Currently we assume that patches (code changes only) within a bug are not independent. We treat consequent patches as resubmissions of the initial patch. In most cases, this assumption holds, an older patch becomes obsolete and a new patch is added to the bug report.

Our study considers only the review flags such as "review" and "super-review". A possible extension of the work would be to include other flags into the analysis such as "feedback", "checkin", and "approval" flags. For example, by mapping "checkin" and "approval" flags to the projects' version control system, we could evaluate project's release management process and practice. Unfortunately, these flags are often not accurately set in practice or are not reported by the project's release drivers.

4.4 Factors Affecting Code Review

Many software development projects employ code review as an essential part of their development process. Code review aims to improve the quality of source code changes made by developers (as patches) before they are committed to the project's version control repository. In principle, code review is a transparent process that aims to evaluate the quality of patches objectively and in a timely manner; however, in practice the execution of this process can be affected by many different factors, both technical and non-technical.

Existing research has found that organizational structure can influence software quality. Nagappan et al. demonstrated that organizational metrics (number of developers working on a component, organizational distance between developers, organizational code ownership, etc.) are better predictors of defect-proneness than traditional metrics such as churn, complexity, coverage, dependencies, and pre-release bug measures [133]. These findings provide support for Conway's law [39], which states that a software system's design will resemble the structure of the organization that develops it.

In this section, we describe our empirical studies to gain insight into the different factors that can influence how long a patch takes to get reviewed and a patch's likelihood of being accepted [19]. The factors we analyzed include personal and organizational relationships, patch size, component, bug priority, reviewer/submitter experience, and reviewer load. Since software developers are primarily interested in getting their patches accepted as quickly as possible, we have designed our research questions to align with this perspective:

RQ1: *What factors can influence how long it takes for a patch to be reviewed?*
Previous studies have found that smaller patches are more likely to receive faster responses [99, 141, 167]. We replicate these studies on our data, and extend the analysis to a number of other potential factors.

RQ2: *What factors influence the outcome of the review process?*
Most studies conclude that small patches are more successful in landing to the project's codebase [141, 167]. A recent study showed that developer experience, patch maturity and prior subsystem churn play a major role in patch acceptance [99]. We further extend these results with additional data that includes various non-technical factors.

We study the community contributions and industrial collaboration on the WebKit open source project. WebKit is a web browser engine that powers the Apple's Safari and iOS browsers, was the basis for Google's Chrome and Android browsers, and host of other third-party browsers. WebKit is a particularly interesting project as many of the organizations that collaborate on the project — including Apple, Google, Samsung, and Blackberry — also have competing business interests. In April 2013 — and during the execution of our study — Google announced the

creation of their own fork of WebKit, called Blink. Therefore, we investigated the differences in the velocity of the code reviews on the patches submitted to the Blink project and compare findings of the two case studies.

The rest of the section is organized as follows. Section 4.4.1 provides a description of the methodology we used in our empirical studies. We then present two case studies: WebKit (Section 4.4.2) and Blink (Section 4.4.3). Section 4.4.4 offers a discussion about the possible interpretation of the results and addresses threats to validity.

4.4.1 Methodology

To investigate our research questions, we first extracted code review data from the Bugzilla issue-tracking system (for the WebKit project) and the Chromium code review repository (for the Blink case study); we then pre-processed the data, identified factors that may affect review delays and outcomes, and performed our analysis. To avoid repetition in explaining two similar processes, we describe each step of our approach for the first case study only. For any deviations in the methodology of the Blink project (e.g., description of the extracted dataset and data filtering), we refer readers to the beginning of the Section 4.4.3.

4.4.1.1 Data Extraction

Every patch contributed to the WebKit project is submitted as an attachment to the project's issue repository[2]; we extracted this data by scraping Bugzilla for all patches submitted between April 12, 2011 and December 12, 2012. We defined the same time interval as the one determined in our previous study on Firefox [17] to be able to compare code review processes of two projects. The data we retrieved consists of 17,459 bugs, 34,749 patches, 763 email addresses, and 58,400 review-related flags. We tracked a variety of information about issues such as name of the person who reported the issue, the date the issue was submitted, its priority and severity, as well as a list of patches submitted for the issue. For each patch we saved information regarding its owner, submission date, whether a patch is obsolete or not, all review-related flags along with the files affected by the patch. For each patch we also recorded the number of lines added and removed along with the number of chunks for each changed file.

All fields in the database, except those related to affected files, were extracted directly from the issue tracker. To create the list of affected files we needed to download and parse the individual patches. Each patch file contains one or more `diff` statements representing changed files. In our analysis we ignored `diff` statements for binary content, e.g., images, and focused on textual `diffs` only. From each statement we extracted the name of changed file, number of lines marked

[2] https://bugs.WebKit.org/

added and removed, and number of code chunks. Here a code chunk is a block of code that represents a local modification to a file as it defined by the `diff` statement. We recorded total number of lines added and lines removed per file in total and not separately for each code chunk. We did not try to derive the number of changed lines from the information about added/removed ones.

Almost every patch in our database affects a file called `ChangeLog` (our analysis found 91 different `ChangeLog` files in the data set). Each `ChangeLog` file contains description of changes performed by the developer for a patch and is prepared by the patch submitter. Although patch files contain `diff` statements for `ChangeLog` files and we parsed them, we eliminated this information when we computed the size of the patch later in our study.

There are three possible flags that can be applied to patches related to code review: `review?` for a review request, `review+` for a review accept, and `review-` for a review reject. For each flag change we also extracted date and time it was made as well as an email address of the person who added the flag.

As the issue tracker uses email addresses to identify people, our initial data set contained many individuals without names or affiliations. Luckily, the WebKit team maintains a file called `contributors.json`[3] that maps various developer email addresses to individual people. We parsed this file and updated our data, reducing the number of people in our database to 747.

We next determined developers' organizational affiliations. First we parsed the "WebKit Team" wiki webpage[4] and updated organizational information in our data. We then inferred missing developers' affiliations from the domain name of their email addresses, e.g., those who have an email at "apple.com" were considered individuals affiliated with Apple. In cases where there was no information about organization available, we performed a manual search on the web. For those individuals where we could not determine an affiliated company, we set `company` field to "unknown"; this accounted for 18% of all developers and 6% of the patches in our data set.

4.4.1.2 Data Pre-processing

In our analysis we wanted to focus as much as possible on the key code review issues within the WebKit project. To that end we performed three pre-processing steps on the raw data:

1. We focused only on the patches that change files within the WebCore part of the version control repository. Since WebKit is cross-platform software, it contains a large amount of platform-specific source code. The main parts of WebKit that are not platform-specific are in WebCore; these include features to parse and render HTML and CSS, manipulate the

[3] http://trac.WebKit.org/browser/trunk/Tools/Scripts/WebKitpy/common/config/contributors.json
[4] http://trac.WebKit.org/wiki/WebKit%20Team

DOM, and parse JavaScript. While this code is actively developed, it is often only developed and reviewed by a single organization (e.g., the Chromium code is only modified by Google developers while the RIM code is only modified by the Blackberry developers). Therefore we looked only at the patches that change non-platform-specific files within WebCore; this reduced the total number of patches considered from 34,749 to 17,170. We also eliminated those patches that had not been reviewed, i.e., patches that had only `review?` flag. This filter further narrowed the input to 11,066 patches.

2. To account for patches that were "forgotten", we removed slowest 5% of WebCore reviews. Some patches in WebCore are clear outliers in terms of review time; for example, the slowest review took 333 days whereas the median review was only 1.27 hour (≈76.4 minutes). This filter excluded any patch that took more than 120 hours (≈5 days) and removed 553 patches. 10,513 patches remained after this filter was applied.

3. To account for inactive reviewers we removed the least productive reviewers. Some reviewers performed a small number of reviews of WebCore patches. This might be because the reviewer focused on reviewing non-WebCore patches or became a reviewer quite recently. Ordering the reviewers by the number of reviews they performed, we excluded less active developers, i.e., those conducting 5% of all reviews when taken together. This resulted in 103 reviewers being excluded; the 51 reviewers that remained each reviewed 31 patches or more.

The final dataset[5] consists of 10,012 patches was obtained by taking the intersection of the three sets of patches described above.

4.4.1.3 Identifying Independent Factors

Previous research has suggested a number of factors that can influence review response time and outcome [99, 141, 167]. Table 4.3 describes the factors (independent variables) that were considered in our study and tested to see if they have an impact on the dependent variables (*time*, *outcome*, and *positivity*). We grouped the factors into four dimensions: *patch*, *bug*, *organizational*, and *personal*.

In order to define the component a patch modifies, we initially planned to consider the classification of components from the WebKit's bug repository, where each bug is assigned to a specific component. After discussing the WebKit source tree structure with some WebKit developers, we decided to focus only on WebCore patches (as described in the filter above) and identified the

[5]Extracted data for the WebKit study is stored in a database and made available online: https://cs.uwaterloo.ca/~obaysal/webkit_data.sqlite

Independent Factor	Dimension	Description
Patch Size	patch	number of LOC added and removed
Component	patch	top-level module in /WebKit/Source/WebCore/
Priority	bug	assigned urgency of resolving a bug
Organization	organizational	organization submitting or reviewing a patch
Review Queue	personal	number of pending review requests
Reviewer Activity	personal	number of completed reviews
Patch Writer Experience	personal	number of submitted patches

Table 4.3: Overview of the factors and dimensions used.

modified components from the patches directly rather than use the issue tracker component flag which were often incorrect.

WebKit does not employ any formal patch assignment process. In order to determine review queues of individual reviewers at any given time, we had to reverse engineer patch assignment and answer the following questions:

- *When did the review process start?* We determined the date when a request for a review was made (i.e., `review?` flag was added to the patch). This date was referred as "review start date". While there might be some delay in the actual time a reviewer started working on the patch, we have no practical means of tracking this.
- *Who performed code review of a patch?* The reviewer of a patch is defined as the person who marked the patch with either `review+` or `review-`. Having this we added the assignee to each review request.

We computed a reviewers queue by considering the reviews they eventually completed and walking backwards considering the date the patch was submitted for review and counting the queue length as the number of patches that were "in flight" for that developer.

4.4.1.4 Data Analysis

Our empirical analysis used a statistical approach to evaluate the degree of the impact of the independent factors on the dependent variables. First, we tested our data for normality by applying Kolmogorov-Smirnov tests [116]. For all samples, the p-value was lower than 0.05 showing that the data is not normally distributed. We also graphically examined how well our data fits the normal distribution using Q-Q plots. Since the data is not normally distributed, we applied non-parametric statistical tests: Kruskal-Wallis analysis of variance [108]

Factor	WebKit		Blink	
	Time	Positivity	Time	Positivity
Patch Size	✓	N/A	✓	N/A
Priority	✓	✓	N/A	N/A
Component	✓	×	✓	×
Organization	✓	✓	✓	×
Review Queue	✓	✓	×	×
Reviewer Activity	✓	×	✓	×
Patch Writer Experience	✓	✓	✓	✓

Table 4.4: Statistically significant effect of factors on response time and positivity.

for testing whether the samples come from the same distribution, followed by a post-hoc non-parametric Mann-Whitney U (MWW) test [111] for conducting pairwise comparisons. All our reported results are statistically significant with the level of significance defined as p-value ≤ 0.05. The statistical analysis of the WebKit and Blink data are performed using R [138] and the scripts are available online: https://cs.uwaterloo.ca/~obaysal/webkit_analysis_script.r and https://cs.uwaterloo.ca/~obaysal/blink_analysis_script.r.

Ultimately, we investigated the impact of seven distinct factors on the code review process both in terms of response time and review outcome (positivity) for the WebKit and Blink projects; this is summarized in Table 4.4.

4.4.2 WebKit Study

In this work we considered the top five organizations contributing patches to the WebKit repository. Figure 4.13 provides an overview of the participation of these organizations on the project with respect to the percentage of the total patches they submit and the percentage of the patches they review. It is clear that two companies play a more active role than others; Google dominates in terms of patches written (60% of total project's patches) and patches reviewed (performing 57% of all reviews) while Apple submits 20% of the patches and performs 36% of the reviews.

To answer our research questions, we performed two empirical studies. We start with demonstrating the results of our analysis of the WebKit dataset and highlighting its main findings.

4.4.2.1 Patch Size

The size of the patch under review is perhaps the most natural starting point for any analysis, as it is intuitive that larger patches would be more difficult to review, and hence require more

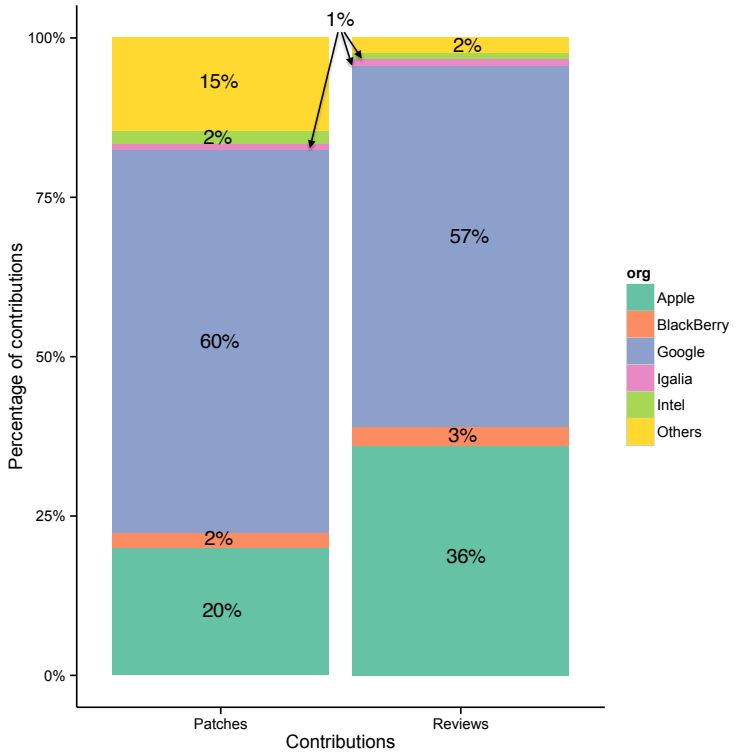


Figure 4.13: Overview of the participation of top five organizations in WebKit.

time; indeed, previous studies have found that smaller patches are more likely to be accepted and accepted more quickly [167]. We examined whether the same holds for the WebKit patches based on the sum of lines added and removed as a metric of size taken from the patches.

In order to determine the relationship between patch size and the review time, we performed Spearman correlation — a non-parametric test. The results showed that the review time was weakly correlated to the patch size, r=0.09 for accepted patches and r=0.05 for rejected patches, suggesting that patch size and response time are only weakly related, regardless of the review outcome.

With a large dataset, outliers have the potential to skew the mean value of the data set; therefore, we decided to apply two different outlier detection techniques — Pierce's criterion and Chauvenet's criterion. However, we found that removal of the outliers did not improve the results.

Next we split the patches according to their size into four equal groups: A, B, C, and D where each group represents a quarter of the population being sampled. Group A refers to the smallest patches (0–25%) with the average size of 4 lines, group B denoting small-to-medium size changes

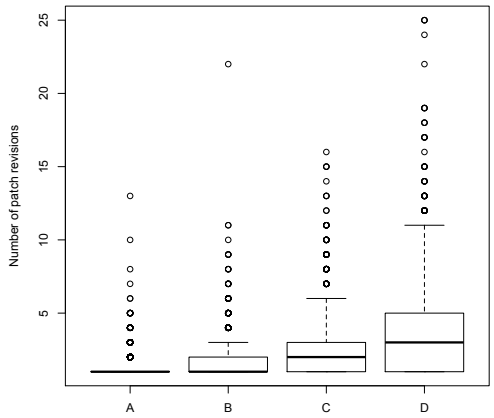


Figure 4.14: Number of revisions for each size group.

(25–50%) on average having 17 lines of code, group C consists of the medium-to-large changes (50–75%) with the mean of 54 LOC, and group D represents largest patches (75–100%) with the average size of 432 lines. A Kruskal-Wallis test revealed a significant effect of the patch size group on acceptance time ($\chi^2(3)=55.3$, p-value <0.01). Acceptance time for group A (the median time is 39 minutes, the mean is 440 minutes) is statistically different compared to the time for groups B (with the median of 46 minutes and the mean of 531 minutes), C (the median of 48 minutes and the mean of 542 minutes) and D (the median is 64 minutes, the mean time is 545 minutes).

In terms of review outcome, we calculated the positivity values for each group A–D, where we define positivity as $\texttt{positivity} = \sum\texttt{review+} \;/\; (\sum\texttt{review-} + \sum\texttt{review+})$. The median values of positivity for groups A–D are 0.84, 0.82, 0.79, 0.74 respectively. Positivity did decrease between the quartiles, matching the intuition that reviewers found more faults with larger patches, although this result was not significant.

However, review time for a single patch is only part of the story; we also wanted to see whether smaller patches undergo fewer rounds of re-submission. That is, we wanted to consider how many times a developer had to resubmit their patch for additional review. We calculated the number of patch revisions for each bug, as well as the size of the largest patch. Figure 4.14 illustrates the medians of the patch revisions for each size group, the median of the revisions for group A and B is 1, for group C is 2, and for D is 3. The results show that patch size has a statistically significant, strong impact on the rounds of revisions. Smaller patches undergo fewer rounds of revisions, while larger changes have more re-work done before they successfully land into the project's version control repository.

4.4.2.2 Priority

A bug priority is assigned to each issue filed with the WebKit project. This field is created to help developers define the order in which bugs must be fixed[6]. There are 5 different priorities currently in the WebKit ranging from the most important (`P1`) to the least important (`P5`). We were surprised when we computed the distribution of patches among priority levels: `P1` – 2.5%, `P2` – 96.3%, `P3` – 0.9%, `P4` and `P5` – 0.1% each. Looking at these numbers one might speculate that the priority field is not used as intended. Previous work of Herraiz et al. also found that developers use at most three levels of priority and the use of priority/severity fields is inconsistent [84]. The default value for priority is `P2`, which might also explain why the vast majority of patches have this value assigned. Also, in our discussion with WebKit developers we found that some organizations maintain internal trackers that link to the main WebKit bug list; while the WebKit version has the default priority value, the internal tracker maintains the organization's view on the relative priority. In our analysis we discarded priorities `P4` and `P5` because they did not have enough patches.

A Kruskal-Wallis test demonstrated a significant effect of priority on time ($\chi^2(2)$=12.70, p-value <0.01). A post-hoc test using Mann-Whitney tests with Bonferroni correction showed the significant differences between P1 and P3 (with median time being 68 and 226 minutes respectively, p-value <0.05) and between P2 and P3 (with median time being 62 and 226 minutes respectively, p-value <0.01). While patches with priority P2 receive faster response that the ones with P1, the difference is not statistically significant.

To analyze positivity we considered each review by a developer at a given priority and computed their acceptance ratio. To reduce noise (e.g., the data from reviewers who only reviewed one patch at a level and hence had a positivity of 0 or 1), we discarded those reviewers who reviewed 4 patches or fewer for a given priority. We found a statistically significant correlation between priority levels and positivity ($\chi^2(2)$=10.5, p-value <0.01). The difference of the review outcome for patches of P1 (median value is being 1.0) compared to the ones of P2 (median is 0.83) is statistically significant (p-value <0.01), indicating that patches of higher priority are more likely to land to the project's codebase. Even though reviewers are more positive for patches that are higher priority, we caution about the interpretation of these results because the vast majority of patches are `P2`.

4.4.2.3 Component

WebCore represents the layout, rendering, and DOM library for HTML, CSS, and SVG. WebCore consists of several primary components:

[6] https://bugs.webkit.org/page.cgi?id=fields.html

- `bindings` — houses the language-specific bindings for JavaScript and for Objective-C;
- `bridge` — bridge is about the bridging to the WebKit framework;
- `css` — the CSS back end;
- `dom` — the DOM library;
- `editing` — the editing infrastructure;
- `html` — the HTML DOM;
- `inspector` — web inspector;
- `page` — contains code for the top-level page and frames;
- `platform` — contains platform-specific code, e.g., mac, chromium, android; thus, as mentioned earlier, excluded from our analysis;
- `rendering` — the heart of the rendering engine.

While it is natural assume that some components are more complex than others, we wanted to find out whether contributions to certain components are more successful or reviewed more quickly. To answer this, we selected the components that undergo the most active development: `inspector` (1,813 patches), `rendering` (1,801 patches), `html` (1,654 patches), `dom` (1,401 patches), `page` (1,356 patches), `bindings` (1,277 patches), and `css` (1,088 patches). The difference in the response time between components was statistically significant ($\chi^2(6)$=29.9, p-value <0.01), in particular the `rendering` component takes longer to review (the median time is 101 minutes) compared to `bindings` (72 minutes), `inspector` (58 minutes), and `page` (58 minutes). The difference in reviewing time of patches submitted to the `page` and `dom` components was also significant with the medians being 58 minutes vs. 91 minutes respectively.

Although the positivity values vary among components and range between 0.73–0.84, we found no relation between positivity and the component factor. From the developer's perspective, we can tell that it is more difficult for developers to land a patch to `page` (the value of positivity is 0.73), while patches to `inspector` are more likely to be successful (the value of positivity is 0.84).

4.4.2.4 Review Queue

Our previous qualitative study of Mozilla's process management practices found that developers often try to determine current work loads of reviewers prior making a decision as to who would be the best choice to request a review from [12]. Thus, we investigated the relationship between

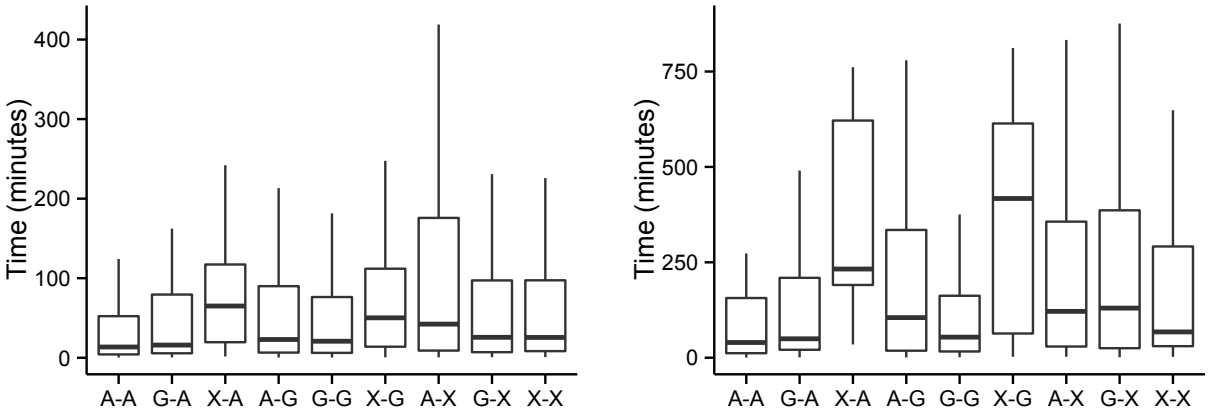


Figure 4.15: Acceptance time (left), rejection time (right). Organization: A=Apple, G=Google, X=Rest.

review queue size and review response time, expecting to find that reviewers having shorter queues would provide quicker reviews.

We calculated queue sizes for the reviewers at any given time (the process is described in Section 4.4.1.3). The resulting queues ranged from 0 to 11 patches.

Since the average queue was 0.6 patches, we distributed patches into three groups according to the queue size: shortest queue length ranging from 0–1 patches (group A), medium length consisting of 2–3 patches (group B) and longer queues ranging from 4–11 patches (group C).

We found a significant effect of review queue size on reviewing time ($\chi^2(2)$=15.3, p-value <0.01). The medians of queue size for group A, B and C are being 0, 2, and 5 patches respectively. A post-hoc test showed significant differences between group A and group C (with median time being 63 and 158 minutes respectively, p-value <0.01) and group B and C (with median time being 90 and 158 minutes respectively, p-value <0.05).

Studying the impact of the queue size on the reviewer positivity (with the Kruskal-Wallis effect being $\chi^2(2)$=15.8, p-value <0.01), we found a significant difference between A and C groups (the median positivity being 0.84 and 1 respectively, p-value <0.01), as well as B and C groups (with median positivity being 0.88 and 1.0 respectively, p-value <0.05).

Thus, we found that the length of the review queue influences both the delay in completing the review as well as the eventual outcome: the shorter the queue, the more likely the reviewer is to do a thorough review and respond quickly; and a longer queue is more likely to result in a delay, but the patch has a better chance of getting in.

4.4.2.5 Organization

Many companies that participate in the WebKit development are business competitors. An interesting question is whether patches are considered on their technical merit alone or if business interests play any role in the code review process, for instance by postponing the review of a patch or by rejecting a patch for a presence of minor flaws. While we analyzed all possible pairs of organization (36 of them), for the sake of brevity we discuss only Apple, Google, and "the rest".

Figure 4.15 represents review time for each pair of organizations. The first letter in the label encodes a reviewer's affiliation, the second encodes a submitter's affiliation; for example, A-G represents Apple reviewing a Google patch. Analysis of the patches that received a positive review showed that there is a correlation between review time and the organization affiliated with the patch writer.

To identify where the correlation exists, we performed a series of pair-wise comparisons. We discovered that there is a statistically significant difference between how Apple approves their own patches (`A-A`) and how Google approves their own patches (`G-G` column). Another statistically significant difference was found between time Apple takes to accept their own patches and time it takes to accept Google patches (`A-G`). However, we found no statistical difference in the opposite direction — between the time for Google to accept their own patches compared to patches from Apple (`G-A`).

The correlation between review time and company was also found for patches that received a negative review. The pair-wise comparison showed almost the same results: statistical difference between Apple-Apple and Apple-Google, and no statistical difference between Google-Google and Google-Apple. At the same time the difference between Apple-Apple and Google-Google is no longer present. Based on these findings, it appears that Apple treats their own patches differently from external patches, while Google treats external patches more like their own. Pairs involving "the rest" group exhibited no statistically significant differences for both review decisions.

Since statistical tests can report only a presence of statistical difference, we also report the means and medians of review time required for each company pair (Table 4.5). To ease comparison of the differences in the response time for organizations, patch reviewers and writers between the two projects, we placed Tables 4.5, 4.6, 4.7, and 4.8 on one page (p. 83). According to the data, Apple is very fast in reviewing its own patches, but is relatively slow in reviewing Google patches (3–4 times difference in medians, 1.5–2 times difference in means). At the same time Google exhibits the opposite behaviour, i.e., provides faster response to the patches from Apple than their own developers. While both means and medians are almost the same for positive reviews, the median and the mean values of review time for negative review for Apple patches are 20 and 200 minutes less respectively than for Google own patches.

To compute the positivity of various organizations we cleansed, the data as we did for the priority analysis above; we removed any reviewer who had reviewed less than 10 patches to

avoid an overabundance of positivities of 0 or 1. The box plot with this filter applied is shown in Figure 4.16. Statistical tests showed that there is a correlation between the outcome of the review and patch owner's affiliation ($\chi^2(2)=10.7$, p-value <0.01). From the pair-wise comparison, we found that there is statistically significant difference between positivity of Apple reviewers towards their own patches (`A-A` column) compared to the patches of both Google (`A-G` column) and "the rest" (`A-X` column). The other pair that was statistically different is positivity of Google reviewers between their own patches (`G-G` column) and patches from "the rest" (`G-X` column).

Quantitatively, there are some interesting results. First, the positivity of Apple reviewers towards their own patches clearly stands out (the median is ≈0.92). Possible explanations for this include that there is a clear bias among Apple reviewers, or that Apple patches are of extreme quality, or that Apple applies some form of internal code review process. We also observed that both Apple and Google are more positive about their own patches than "foreign" patches; while this could be a systematic bias, Apple and Google are also the two most experienced committers to WebKit and this may account for this difference. Finally, the positivity of Apple reviewers towards Google patches (the median is ≈0.73) is lower than the positivity of Google reviewers towards Apple patches (the median is ≈0.79).

Reviewer → Writer	Accepted		Rejected	
	Median	Mean	Median	Mean
Apple → Apple	25	392	60	482
Apple → Google	73	617	283	964
Google → Google	42	484	102	737
Google → Apple	45	483	80	543

Table 4.5: Response time (in minutes) for organizations participating on the WebKit project.

Group	Reviewer		Writer	
	Median	Mean	Median	Mean
A	84	621	102	682
B	76	634	76	632
C	46	516	43	491
D	57	496	48	478

Table 4.6: Response time (in minutes) for WebKit patch reviewers and writers.

4.4.2.6 Reviewer Activity

WebCore has 51 individuals performing code reviews of 95% of patches. The breakdown of the reviewers by organization is as follows: 22 reviewers from Apple, 19 reviewers from Google, 3

Reviewer → Writer	Accepted		Rejected	
	Median	Mean	Median	Mean
Google → Google	57	385	169	716
Google → Other	95	473	428	737
Other → Other	66	351	n/a	n/a
Other → Google	48	399	n/a	n/a

Table 4.7: Response time (in minutes) for organizations participating on the Blink project.

Group	Reviewer		Writer	
	Median	Mean	Median	Mean
A	71	434	106	547
B	71	490	51	384
C	42	338	59	394
D	91	362	56	287

Table 4.8: Response time (in minutes) for Blink patch reviewers and writers.

reviewers from BlackBerry, Igalia and Intel are being represented by one reviewer each, and 5 reviewers belong to the group "others". Comparing reviewing efforts, we noticed that while Apple is predominant in the number of reviewers, it reviews only 36% of all patches; by comparison, Google developers perform 57% of the total number of reviews. Since WebKit was originally developed and maintained by Apple, it is perhaps unsurprising that Apple remains a key gatekeeper of what lands in the source code. However, we can see that Google has become a more active contributor on the project, yet has not surpassed the number of Apple reviewers.

To find out whether reviewers have an impact on review delay and outcome, for each reviewer we calculated the number of previously reviewed patches and then discretized them according to their reviewing efforts using quartiles. Applying statistical tests, we determined that the difference for response time for A and B groups of reviewers (i.e., the less active ones) is statistically significant when compared to C or D groups (i.e., the more active ones). Since the distribution of delays is very skewed, we report both the median and mean values for reviewers' timeliness (see Table 4.6). The results show that the choice of reviewers plays an important role on reviewing time. More active reviewers provide faster responses (with median being 57 minutes and mean being 496 minutes) compared to the individuals who performed fewer code review (the median for time is 84 minutes and 621 minutes for the mean).

Considering that reviewers' work loads appear to affect their response rate, WebKit contributors may wish to ask the most active reviewers to assess their patches in order to get a quick response. With respect to the question whether there are reviewers who are inclined to be more positive than negative, we found that there is no correlation between the amount of reviewed

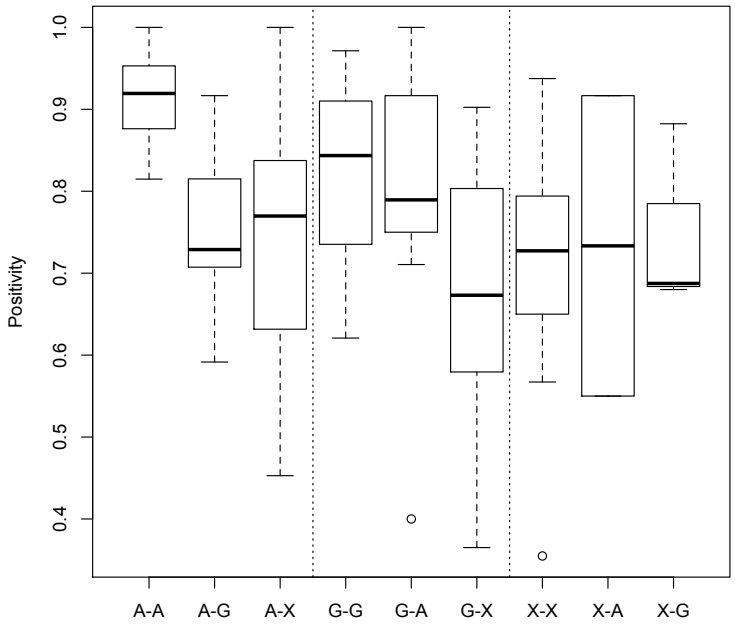


Figure 4.16: Positivity values by organization: A=Apple, G=Google, X=Rest.

patches on the reviewer positivity: 0.83 for group A, 0.84 for group B, 0.75 for group C, and 0.83 for group D. This suggests that WebKit reviewers stay true and unbiased in their role of ensuring the quality of code contributions. This observation is important since reviewers serve as gatekeepers protecting the quality of the project's code base.

4.4.2.7 Patch Writer Experience

The contributions to WebCore during the period studied came from 496 individuals among which 283 developers filing 95% of patches (submitting 5 patches or more). Considering our top five organizations, we identified that WebCore patches were submitted by 50 developers from Apple, 219 individuals from Google, 20 BlackBerry developers, 16 developers from Intel, 10 from Igalia, and 181 developers come from other organizations.

Noticing strong contributor diversity in the WebCore community, we wondered if patches from certain developers have a higher chances of being accepted. To assess whether developer experience influences review timeliness and acceptance, we performed a similar procedure (as described in 4.4.2.6) for calculating the number of submitted changes for each developer and then discretizing patch owners according to their contributions.

We achieved similar results in the differences of response time for A and B groups of submitters

(occasional contributors) is statistically significant compared to more experience developers in C or D groups. From Table 4.6 we conclude that more experienced patch writers receive faster responses (with median in group D being 48 minutes and mean being 478 minutes) compared to those who file fewer patches (the median for time in group A is 102 minutes and 682 minutes for the mean).

Investigating the impact of developer experience on positivity of the outcome, we found correlation between two variables ($\chi^2(3)$=17.93, p-value < 0.01). In particular, statistical difference was found between group A (least active developers) and groups C and D (more active developers) with the median positivity values being 1.0, 0.73 and 0.81 respectively, as well as group B (less active developers) compared to the group D (most active ones) with the median positivity being 0.63 and 0.81 respectively. These findings suggest that the WebKit community has a positive incentive for newcomers to contribute to the project as first-patch writers (i.e., group A with the median number of patches submitted being 1) are likely to get a positive feedback. For developers of group B (where contributions range between 3–6 patches) it is more challenging to get their patches in, while contributing to the project comes with the improved experience of landing patches and as a result with more positive outcomes.

Our findings show that developer experience plays a major role during code review. This supports findings from our previous work, where we have seen faster response time for core developers compared to the casual contributors on the project [17]. This appears to show that active developers are being rewarded with both faster response and more positive review outcome for their active involvement in the project.

4.4.2.8 Multiple Linear Regression Model

To estimate the relationships among factors, we built a multiple linear regression model. Multiple linear regression (MLR) attempts to model the relationship between two or more explanatory variables and a response variable by fitting a linear equation to observed data [37].

We tested whether the explanatory variables x_1, x_2,... x_p (our factors) collectively have an effect on the response variable y (being review time or outcome), i.e:

H_0: $\beta_1 = \beta_2 = ... = \beta_p = 0$.

The results of the MLR model with respect to time reported the F statistic of 9.096 (p-value <0.01), indicating that we should reject the null hypothesis that the variables size, priority, queue, etc. collectively have no effect on time. The results also showed that the patch writer's experience (p-value <0.01) and affiliation (p-value <0.01) variables are significant controlling for the other variables in the model. While we rejected the null hypothesis, the R-square statistic (i.e., the measure of the regression model's usefulness in predicting outcomes) is close to 0 (R^2=0.01435, $R^2_{adjusted}$=0.01277), indicating that the studied factors have no explanatory power.

Estimating the effect of the factors on the review outcome, we obtained the F statistic of 33.71 (p-value <0.01), and thus rejected the null hypothesis. The most significant factors controlling the other variables in the model remain patch writer's experience (p-value <0.01) and affiliation (p-value <0.01), along with the reviewer load (p-value <0.05) and number of components affected (p-value <0.01). The R-square statistic (R^2=0.0512, $R^2_{adjusted}$=0.04968) still shows that the model has low overall predictive power and that the variables do not account for the the variation in the review outcome in the WebKit project.

While it is not feasible to account for all possible factors that might affect the outcome and interval of the code review process, our findings suggest that among the factors we studied non-technical (organizational and personal) ones are better predictors compared to the traditional metrics such as patch size or component, and bug priority. These findings confirm previous empirical studies on code review [99, 133].

4.4.3 Blink Study

Every Blink patch is submitted to the project's issue repository. We extracted all patches by scraping the public Blink issue-tracking system for the patches submitted between April 03, 2013 and January 07, 2014. The extracted dataset consists of 18,177 bugs, 37,280 patches, 721 emails and 64,610 review flags. We extracted the same information about issues, patches, and files as we did for the WebKit data (described in Section 4.4.1.1).

To determine developer's organizational affiliations, we first inferred affiliations from the domain name of their email addresses. Then, we asked a Blink developer to perform confirm our guesses. To be consistent with our WebKit dataset, we marked independent developers contributing to the Blink project as "unknown".

Data Filtering To clean up our our dataset, we performed pre-processing steps on the raw data similar to that of the WebKit study:

1. We considered only patches that affect files within `Source/core` portion (the Blink team refactored `WebCore` to this directory) of the source repository reducing the total number of patches from 37,280 to 23,723.
2. We further eliminated those patches that had not been reviewed, narrowing the input to 9,646 patches.
3. We removed 5% of slowest patches, eliminating those that took more than 80 hours to review.
4. We eliminated the least active reviewers: those who performed less than 10 code reviews; this resulted in retaining a set of 174 reviewers out of the 223 individual reviewers performing code reviews for Blink.

After applying data filtering, the final Blink dataset consisted of 8,731 patches[7].

4.4.3.1 Patch Size

To investigate the correlation between patch size and review response time we again split the patches according to their size into four equal groups (quartiles): A, B, C, and D. Group A refers to the smallest patches (0–25%) with the average size of 4 lines, group B denotes small-to-medium size changes (25–50%) on average having 18 lines of code, group C consists of the medium-to-large changes (50–75%) with the mean of 63 LOC, and group D represents largest patches (75–100%) with an average size of 698 lines of code. A Kruskal-Wallis test revealed a significant effect of the patch size group on acceptance time ($\chi^2(3)$=44.16, p-value <0.01). Acceptance time for group A (the median time is 47 minutes, the mean is 368 minutes) is statistically different compared to the time for group C (with the median of 76 minutes and the mean of 444 minutes), and D (the median of 69 minutes and the mean of 388 minutes).

The median positivity values for groups A–D are all 0.99. Reviewers' positivity remains quite high and does not appear to be affected by the size of the contributions.

Investigating the relationship between patch size and the number of patch revisions, we considered all the bug IDs that we have the patches for after applying our data filters. We calculated the number of patch revisions for each bug, as well as the size of the largest patch. Our results demonstrate statistically significant, strong impact of patch size on the rounds of revisions ($\chi^2(3)$=1473.7, p-value <0.01). The median of the patch revisions for smaller patches of group A (under 22 LOC) is 1, while the highest number of resubmissions is 7. Group B (patch size ranges between 22–71 LOC) and group C (with the size between 72–205 LOC) has the same median value of resubmissions (on average 2 patches per issue), the highest number of patch revisions is 11 for group B and 23 for group C. The largest patches (on average of around 1,000 LOC) have more revisions than smaller patches with a median of 3 and a maximum of 30 resubmissions from group D.

4.4.3.2 Component

Blink's source code is organized similar to the WebKit's except that `bindings` and `bridge` have been removed.

We selected the same number of top actively developed components: `dom` (5,856 patches), `rendering` (5,732 patches), `page` (4,936 patches), `html` (4,934 patches), `css` (4,517 patches), `inspector` (2,938 patches), `loader` (2,305 patches). The difference in the response time between

[7]Our Blink dataset is available online:
https://cs.uwaterloo.ca/~obaysal/blink_data.sqlite

components was statistically significant ($\chi^2(6)$=40.75, p-value < 0.01); similar to the WebKit study, the `rendering` component takes longer to review (the median time is 89 minutes) compared to any other component including `inspector` (49 minutes), `page` (70 minutes), `dom` and `html` (58 minutes), and `css` (63 minutes).

We found no relationship between positivity and the component factor; the average positivity values for the components are quite high (0.98-0.99), suggesting that patches have high chance of being landed to these actively developed components.

4.4.3.3 Review Queue Length

Similar to the WebKit study, we calculated queue sizes for the reviewers at any given time (the process is described in Section 4.4.2.4). The resulting queues ranged from 0 to 14 patches and the average queue was 0.7 patches. Statistical tests showed that there is no significant effect of review queue size on neither reviewing time ($\chi^2(14)$=21.63, p-value = 0.086) nor positivity of the reviewers ($\chi^2(14)$=20.20, p-value = 0.124).

Thus, we found that the length of the review queue affects neither the review response time nor its outcome.

4.4.3.4 Organization

While Google developers submit 79.3% of all patches, other organizations also contribute to the project including Samsung (8.7%), Opera (3.8%), Intel (2.9%), Adobe (1.9%) and Igalia (0.2%), as well as independent individuals (3.1%). To assess whether organization influences review response and outcomes, we grouped non-Google contributions together and labelled them as "other" and then compared this group against Google-only contributions.

We discovered that there is a statistically significant relationship between response time and which organization submits the patch. Regardless of the review outcome, patches from Google developers receive faster responses than patches coming from other organizations (57 minutes vs. 95 minutes). Table 4.7 reports the mean and medians of both accepted and rejected review times for each group. Google patches are accepted or rejected faster that patches from others.

In terms of the positivity, we found no difference in review outcomes for Google vs. "other" patches. This finding is somewhat expected since 98% of reviews are done by Google reviewers who appear to provide mainly positive reviews (see Section 4.4.3.5).

Comparing review response times, we noticed that the median values of both acceptance and rejection increase while the mean values decrease for Google reviewers participating on the Blink project vs. the WebKit project. While we do not have insights on why this happened, we speculate that Google focuses on accepting good contributions (the positivity values being very high) and

providing constructive feedback to patch writers than just relaying quick negative feedbacks to the developers.

4.4.3.5 Reviewer Activity

Blink has 174 active reviewers performing code reviews. While the majority of the contributions to the Blink repository are reviewed by Google (98%), other organizations perform the remaining 2% of code reviews; Intel reviewed 75 patches, Samsung reviewed 41 patches, Adobe reviewed 13 patches, and independent developers reviewed 29 patches.

To determine whether reviewer activity as a factor has an effect on the response time, we calculated the number of previously reviewed patches for each reviewer and discretized reviewers according to their reviewing experience using quartiles (similar to the procedure we performed for WebKit). Statistical test showed that the difference for response time of less experienced reviewers (i.e., A and B groups of reviewers with the median response time of 71 minutes) is statistically significant ($\chi^2(3)$=62.14, p-value < 0.01) compared to more active ones (group C with median value of the reviewing time being 42 minutes). The difference in the response time for group C was also statistically significant compared to group D (median time is 91 minutes). We note that group D consists of one most active reviewer on the Blink project who reviewed 1,392 patches (15% of all reviewed patches). Table 4.8 reports both the median and mean values for reviewers' timeliness.

Looking at the reviewers' involvement on the project and whether it affects their positivity, we found no correlation between reviewers' positivity and their activity on the project. Positivity values remain similar for the group A, B, C and D with medians ranging between 0.99–1.0. This shows that reviewers provide positive feedback to almost all patches they review; it seems that Google reviewers use positive reinforcement when assessing contributions. If a patch is not quite ready to land to the source code, reviewers would discuss potential flaws and expect the patch to be resubmitted again for further review. Such behaviour is likely to increase response from developers submitting their patches.

4.4.3.6 Patch Writer Experience

The contributions to Blink's core during the studied period came from 394 developers. While Blink is developed and maintained mainly by Google — they submit 78% of all patches — other organizations also contribute patches to the Blink repository, including Samsung (757 patches), Opera (328 patches), Intel (256 patches), Adobe (170 patches), and Igalia (21 patches) and other independent developers (274).

To understand whether the experience of a patch writer affects the timeliness and outcome, we grouped developers according to their contributions (similar to the step described in 4.4.2.7). We

found that the differences of response time for group A of submitters is statistically significant ($\chi^2(3)$=109.04, p-value < 0.01) when compared to more experienced developers (B, C and D groups). From the Table 4.8 we conclude that more experienced patch writers are more likely to get faster responses (the median for groups B, C, and D being 51, 59 and 56 minutes respectively) than those who has not gained enough experience in filing project contributions, individuals who submitted fewer than 30 patches (the median for group A of submitters is 106 minutes).

When investigating the impact of developer experience on the likelihood of patch acceptance, we found correlation between two variables ($\chi^2(3)$=32.65, p-value < 0.01). In particular, a statistical difference was found between group A (least active contributors) and groups C and D (more active developers), as well as group B compared to the group D (most active ones). However, the median and mean values for the groups are almost same, 1.0 for the median and mean values ranges between 0.98-0.99. The statistical difference accounts for the distribution of the positivity scores within each group, showing that the developers who are more actively involved on the project almost certainly can expect their patches to be accepted. On the other hand, the least active developers receive a fair amount of rejections. This conclusion also supports the overall code review culture that we have seen from the lifecycle model (shown in Figure 4.4) — Google reviewers are inclined to accept patches with only a very small portion (0.3%) of the submitted patches receiving negative reviews; patches that need reworking are simply resubmitted again, after reviewers provide their comments about the potential flaws.

Our findings show that developer experience is a key factor when it comes to review outcomes and timeliness. Similar to the WebKit study, we see that more active developers on the Blink project receive faster responses and their patches have high chances of being approved.

4.4.4 Discussion

The WebKit community is a complex institution in which a variety of organizations that compete at a business level collaborate at a technical level. While it would be ideal for the contributions of these organizations to be treated equally based on their technical merit alone, our results provide empirical evidence that organizational and personal factors influence review timeliness, as well as the likelihood of a patch being accepted. Some factors that influenced the time required to review a patch, such as the size of the patch itself or the part of the code base being modified, are unsurprising and are likely related to the technical complexity of a given change. Other factors did not seem to fit this mould: for example, we found significant differences in how long a patch took to be reviewed based on the organizations that wrote and reviewed a given patch. We found similar effects influencing the chances of a patch being accepted.

Ultimately, the most influential factors of the code review process on both review time and patch acceptance are the organization a patch writer is affiliated with and their level of participation within the project. The more active role a developer decides to play, the faster and more

likely their contributions will make it to the code base.

4.4.4.1 Threats to Validity

Internal validity concerns with the rigour of the study design. In our study, the threats are related to the data extraction process, the selection of the factors that influence code review, and the validity of the results. While we have provided details on the data extraction, data filtering, and any heuristics used in the study, we also validated our findings with the WebKit developers and reviewers. We contacted individuals from Google, Apple, BlackBerry, and Intel and received insights into their internal processes (as discussed in 4.4.4.2). To ensure that we are on the correct track in interpreting the results of our studies, we talked to the WebKit and Blink developers via email (for Apple, Intel, Google), as well as had face-to-face meetings with Google and Blackberry developers at the respective local offices in Waterloo, Ontario (Canada). Face-to-face meetings included a presentation of the main findings followed by the discussion about possible explanations and insights into the "hidden" factors affecting code review process and practice.

When investigating the relation between patch size and the number of patch revisions, we assumed that patches are independent; this might have introduced some bias since several different patches can often be associated with the same bug ID and "mentally" form one large patch. However, for both studies we considered the size of the largest patch due to the lack of indication of which patches are actually comprising a larger patch and which patches are being resubmits.

Unlike Bugzilla's issue tracking — which is used by both Mozilla and WebKit to carry out code review tasks — Blink's code review system does not support history tracking of patch changes and lacks any explicit review requests. We overcome these limitations by inferring the review start times of Blink patches by considering the most recent patch (in terms of time) in a list of the patches followed by a review flag. This heuristic is a "best effort" approximation; unfortunately, accurate timestamps of the review starting point cannot be determined by scraping the data from the existing code review system.

Our empirical study is also subject to *external validity* concerns; we cannot generalize our findings to say that both organizational and personal factors affect code review in all open source projects. While we compared WebKit's code review process with that of Mozilla Firefox, and found that its patch lifecycle is similar to open source projects, the fact that WebKit is being developed by competing organizations makes it an interesting case yet a rather obvious exception. Hence, more studies on similar projects are needed.

Statistical conclusion validity refers to the ability to make an accurate assessment of whether independent and dependent variables are related and about the strength of that relationship. To determine whether relationships between variables are statistically significant, we performed null

hypothesis testing. We also applied appropriate statistical tests (analysis of variance, post-hoc testing, and Spearman's correlation).

4.4.4.2 Other Interpretations

Drawing general conclusions from empirical studies in software engineering carries risk: any software development process depends on a potentially large number of relevant contextual variables, which are often non-obvious to outsiders. While our results show that certain non-technical factors have a statistically significant effect on the review time and outcome of patch submissions, understanding and measuring the practical significance of the results remains challenging. Processes and developer behaviour around their contributions to the WebKit project depend on the organization, its culture, internal structure, settings, internal development cycles, time pressures, etc.

Any of these "hidden" factors could potentially influence patch review delays and outcomes; for example, let us consider time pressures. It is our understanding that Apple prefers strict deadlines for shipping hardware, and the supporting software needs to match the projected delivery dates of the new hardware. This results in Apple developers prioritizing internal development goals over external ones, and thus prioritizing patches that help them meet their short-term objectives.

Organizational and geographical distribution of the developers may also provide insights into review delays. We understand that WebKit developers at Apple are co-located within the same building, which may account for a better visibility of the patches that their co-workers are working on; conversely, WebKit developers at Google tend to be more geographically distributed, which may result in a poorer awareness of the work of others.

In summary, understanding the reasons behind observable developer behaviour requires an understanding of the contexts, processes, and the organizational and individual factors that can influence code review and its outcome. Thus, while our results may be statistically valid, care must be taken in interpreting their meaning with respect to actual developer behaviour and intent. We consider that much work remains to be done in studying how best to interpret empirical software engineering research within the context of these "hidden" contextual factors.

4.5 Summary

In this chapter, we proposed artifact lifecycle models as an effective way to gain an understanding of certain development artifacts and demonstrated how these models can be applied to assess the code review processes of three industrial projects.

We also describe two empirical studies to gain insight into the different factors that can influence how long a patch takes to get reviewed and a patchs likelihood of being accepted. WebKit is comprised of a complex community to which a variety of organizations contribute; these organizations compete at a business level while collaborating at a technical level. Ideally, the contributions of these organizations to be treated equally, based on their technical merits alone. While some influencing factors include the size of the patch itself or the part of the code base being modified, other non-technical factors have significant impact on the code review process. Our results provide empirical evidence that organizational and personal factors influence both review timeliness as well as the likelihood of a patch being accepted. Additionally, we found significant differences in how long a patch took to be reviewed based on the organizations that wrote and reviewed a given patch along with the final outcome of the review.

Ultimately, the most influential factors of the code review process on both review time and patch acceptance are the organization a patch writer is affiliated with and their level of participation within the project. The more active role a developer decides to play, the faster and more likely their contributions will make it to the code base.

Chapter 5

Personalizing Issue-Tracking Systems

In Chapter 4, we studied the code review process and showed that analytics can help with synthesizing knowledge from development artifacts such as code review and issue tracking systems. These systems together with other development repositories and environments are often complex, general-purpose tools that can provide an immense amount of raw information. Because of this, developers may feel frustrated when they cannot easily access the particular set of details that they need for a given task. The information they seek is in there, somewhere, but not at their fingertips, and the tools often provide only limited support for their specialized, definable, and recurring information needs. In this chapter, we motivate the need for *personalization* in the tools and environments that software developers use and demonstrate it on the example of personalizing issue-tracking systems.

Chapter Organization Section 5.1 provides an introduction describing the challenges developers face when interacting with issue-tracking systems. Section 5.2 describes our methodology and the study setting, and we discuss the results of the qualitative study on the set of interviews with developers. In Section 5.3, we introduce our model of issue tracking together with an illustration of our high-fidelity prototype; we also discuss a qualitative validation of this prototype. Section 5.4.1 describes our final tool, and we discuss its quantitative validation (usage and bug mail compression) and qualitative evaluation (via interviews with developers). Section 5.5 discusses possible further improvements to our tool and also addresses possible threats to validity of our work. Finally, Section 5.6 summarizes the contributions of this work.

Related Publications

- Olga Baysal and Reid Holmes. A Qualitative Study of Mozilla's Process Management Practices. Technical Report CS-2012-10, David R. Cheriton School of Computer Science,

University of Waterloo, Waterloo, Canada, June 2012. Also available online http://www.cs.uwaterloo.ca/research/tr/2012/CS-2012-10.pdf.

- Olga Baysal, Reid Holmes, and Michael W. Godfrey. Developer Dashboards: The Need for Qualitative Analytics. *IEEE Software*, 2013, 30(4):46–52.
- Olga Baysal, Reid Holmes, and Michael W. Godfrey. Situational Awareness: Personalizing Issue Tracking Systems. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2013, pages 1185–1188.
- Oleksii Kononenko, Olga Baysal, Reid Holmes, and Michael W. Godfrey. DASHboards: Enhancing developer situational awareness. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2014.
- Olga Baysal, Reid Holmes, and Michael W. Godfrey. No Issue Left Behind: Reducing Information Overload in Issue Tracking. In *Under review*, 2014.

5.1 Introduction

Software developers work in complex, heterogeneous technical environments that involve a wide variety of tools and artifact repositories [68, 101]. These tools frequently provide generalized interfaces that can be used by many kinds of stakeholders including not only developers but also managers, QA, technical support, and marketing staff. Since these tools aim to provide a unified experience for every user, they may present information in a very general way, perhaps augmented by complex querying mechanisms to aid in narrowing the focus of the results. At the same time, the complexity of these systems continues to increase [180] and the amount of information any single user[1] needs to consider swells rapidly. For example, at the start of 2002 the Mozilla project issue-tracking repository contained around 117,500 issues, but by April 2014 it had reached 998,000 an average of around 201 new issues per day over 12 years (Figure 5.1).

The flood of information that developers need to control is always increasing because any update on an issue triggers an automatic email being sent to the developers involved with this issue (either by reporting it, leaving a comment, being assigned to fix it, or being on the CC list). And because much of this data is related only tangentially to the task at hand, there is an increased risk that a developer may miss something truly important amid the deluge: *"Bugzilla doesn't let you control the flow enough, 5,000 emails in a month and most of it doesn't relate to my work."*[2]

[1] For the remainder of the chapter we will restrict our discussion to developers.

[2] All italicized quotes are verbatim comments made by Mozilla developers as part of the study described in Section 5.2.

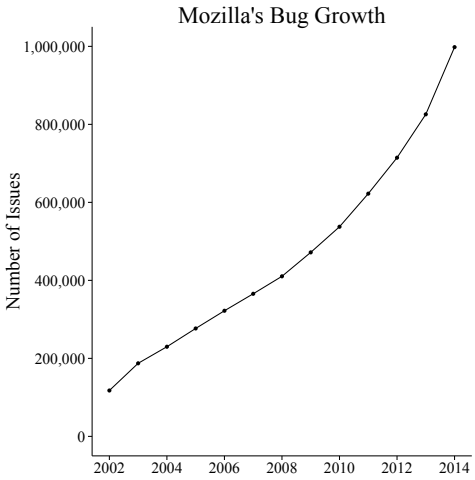


Figure 5.1: Mozilla's bug growth; on average 201 new bugs are reported each day.

One way to help developers manage the increasing flow of information is to provide them with personalizable development tools that work to highlight the details that are most important to their specific needs, while eliding the rest from view [158]. In this chapter, we describe a qualitative study that identified an industrial desire for this kind of customization for issue-tracking systems. From this study we derived a model that captures the notion of the data and tasks developers want to have support for. We then developed and validated a high-fidelity prototype organized around personalized dashboards that provides a custom view of the Bugzilla issue repository; we do this by filtering irrelevant details and metadata from the issue-tracking system, and equipping developers with information relevant to their current tasks. Through qualitative and quantitative measures we evaluated our approach for its ability to reduce information overload while being able to improve developers' awareness on the issues they are working on and support their daily tasks.

Issue trackers have long been used by software teams to report, discuss, and track both defects and new features. While the collaborative demands of this problem space are well met by modern issue trackers [9, 65, 152], some routine kinds of tasks are poorly supported; a particular sore point is the weak support provided to developers in building and maintaining a detailed understanding of the current state of their system: What is the current status of my issues? Who is blocking my progress? How am I blocking others? These limitations can be overcome by adding detailed information that is specific to developers and their tasks, and by providing the technical means for developers to create these personalized views themselves: *"Querying in Bugzilla is hard; have to spend a few minutes to figure out how to do the query."* That is, offering developers customizable means of filtering information — such as via dashboards — can help

them better maintain situational awareness about both the issues they are working on and how they are blocking other developers' progress on their own issues [15]. This finding is similar to the one described by Treude and Storey [162] who demonstrated that dashboards are used mainly for providing developers with high-level awareness (e.g., tracking the overall project status). In this work, we propose a model of the informational needs for issue tracking based on the concepts and themes that emerged from a qualitative analysis of interviews with industrial developers. This model highlights the information needed to support developer daily activities and tasks. From this model we constructed a prototype, refined this prototype as a tool implemented via developer dashboards, and validated the tool with the Mozilla developers. We show that these dashboards can help developers maintain their awareness on the project, as well as overcome the burden of managing a high volume of mostly irrelevant bug mail.

This chapter makes the following contributions:

- A qualitative analysis of industrial developers' perceptions of the shortcomings of their widely-deployed issue tracker (Section 5.2).
- A model that describes the data and tasks developers want to have personalized within their issue-tracking system (Section 5.3), its initial prototype (Section 5.3.4.1) and validation (Section 5.3.4.2).
- A tool that reifies the model for the Bugzilla issue-tracking system (Section 5.4) and an industrial validation of this tool (Section 5.4.1).

5.2 Qualitative Study

We have used a mixed method approach [42] consisting of three main steps as shown in Figure 5.2: 1) data collection (industrial interviews) and analysis (open coding, affinity diagram); during the qualitative analysis a model of issue tracking which supports customizable means of filtering software development tool information has emerged; 2) this model is then instantiated in a prototype which has been validated via interviews; and 3) tool development and its industrial validation (usage data, bug mail compression and interviews).

The initial data collection was performed as a part of the Mozilla Anthropology project [21]; this project was started in late 2011 to examine how various Mozilla community stakeholders make use of the Bugzilla issue-tracking system, and to gain a sense of how Bugzilla could be improved in the future to better support the rapidly-growing Mozilla community. During this process, Martin Best of the Mozilla Corporation conducted 20 one-hour interviews with active developers from various Mozilla projects. These interviews solicited feedback on various aspects of how developers interact with issues throughout their life cycle. The main goal of the Anthropology project was to

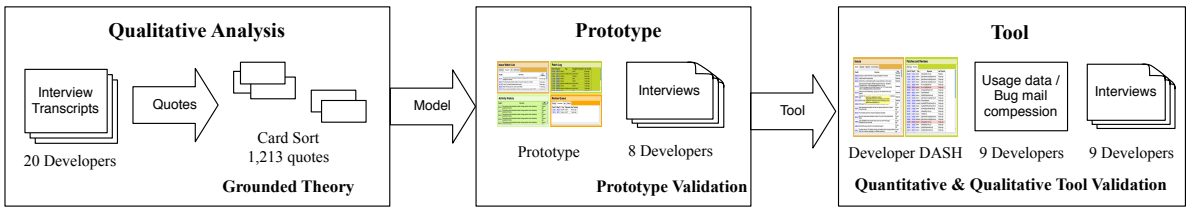


Figure 5.2: Our research method. In a qualitative analysis of the data from the interviews with industrial developers, we uncovered a model of the informational needs for issue tracking based on the concepts of the key limitations of issue tracking that emerged during the open coding analysis (grounded theory). From this model we constructed a prototype, validated it, refined in into a tool that supports developers' situational awareness and validated this tool with the industrial developers.

identify trends that could help locate key problem areas with issue management, as well as best practices related to the use of Bugzilla. The full text of these transcripts is available online [21]; as far as we know ours is the first analysis of this data set.

As the data collection was driven by an industrial initiative and conducted by a Mozilla employee, our analysis of the interview transcripts was exploratory: we had no predefined goals or research questions. Our interest was purely to see if developers had any qualms with their issue-tracking system.

We created all of the "cards", splitting 20 interview transcripts into 1,213 individual units; these generally corresponded to individual cohesive statements, which we call comments. The 1,213 individual comments were analyzed using a grounded theory approach [41]. Grounded theory allowed us to group the comments according to their content using some coding criterion. We applied an an open coding approach [121] to assign codes to the collected data; by not predefining our coding strategy we allowed the categories to be iteratively developed and refined.

We applied an open coding technique to classify 1,213 individual comments into naturally emerging themes. We identified 15 themes and 91 sub-themes, containing between two to 41 comments. Later, the 15 themes were grouped into 4 concept categories. These concept categories consist of two to six themes. All of the concept categories were *organic* — that is, they naturally arose from the data. Each concept category relates to a different aspect of how Mozilla developers interact with Bugzilla to report and resolve bugs and to manage the collaborative process.

We proceeded with this analysis in three steps:

1. The two coders independently performed card sorts on the cards extracted from the interview transcripts of three participants — P1, P2, and P3 — to identify initial card groups (we refer to these groups as sub-themes in Section 5.2). The coders then met to compare and discuss their identified groups.

2. The two coders performed another independent round, sorting the comments of participants P4, P5, and P6 into the groups that were agreed-upon in the previous step. We then calculated and report the intercoder reliability to ensure the integrity of the card sort. We selected two of the most popular reliability coefficients for nominal data: percent agreement and Krippendorff's Alpha (see Table 5.1). Intercoder reliability is a measure of agreement among multiple coders for how they apply codes to text data. To calculate agreement, we counted the number of cards for each emerged group for both coders and used ReCal2 [70] for calculations. The coders achieved a high degree of agreement; on average two coders agreed on the coding of the content 98.5% of the time. For a detailed information about intercoder reliability scores including percent agreement, Scott's Pi, Cohen's Kappa, and Krippendorff's Alpha refer to Appendix A.2.

3. The rest of the card sort — the comment of participants P7–P20 — was performed by both coders together.

To make sense of the groups identified in the card sort, the 91 groups were clustered into 15 related themes, which were in turn organized into four concept categories; this was done using an affinity diagram, which helps to sort a large number of ideas into related clusters for analysis and review [28]. We generated the affinity diagram in three steps: 1) the names of the groups were printed on cards and attached to a whiteboard; 2) the groups were clustered on the whiteboard into related themes, and these themes were further clustered into high-level concept categories; and 3) each of the themes and concept categories was given a representative name. The final organization of concept categories, themes and sub-themes, as well as the results of the open coding and affinity diagram methods are available [12].

	Percent Agreement	Krippendorff's Alpha
Median	98.9%	0.871
Average	98.5%	0.865

Table 5.1: Average scores of intercoder reliability.

Section 5.2.1 provides an overview of the concept categories, as well as their themes and sub-themes. For each theme and sub-theme we provided the number of individual participants commenting on a certain issue and the total number of quotes given. For each sub-theme we developed a synthetic quote that provides a general thought on the topic. Synthetic quotes (SQ) are generated by combining participants' comments into a single statement.

Appendix A.1 contains a detailed 35 page analysis of the qualitative findings from this card sort. In this section, we present an overview of these findings as well as several interesting specific points. For more data and additional analysis, please see Appendix A.

5.2.1 Emerged Concept Categories

The four concept categories that have emerged during the card sort do not necessarily have direct correspondence to the tasks developers perform daily; rather, they are a combination of actions and mental activities developers perform when considering and executing these tasks.

The high-level concept categories are situational awareness (Section 5.2.2), supporting tasks (Section 5.2.3), expressiveness (Section 5.2.4), and the rest (Section 5.2.5).

Table 5.2 provides an overview of the concept categories, 15 themes, and 91 sub-themes. The table also reports the number of developers who made quotes that fell into each theme, the total number of quotes in the theme, and the theme's synthetic quote.

Category/theme/sub-theme	**# Participants**	**# Quotes**
Situational Awareness	**19**	**208**
• Dashboards	18	99
– Public dashboards	16	60
Developer profiles	12	31
Workload transparency	12	22
Communicating interest	8	12
– Private dashboards	15	39
Tracking activity	12	21
Watch list	10	18
• Collaborative Filtering	4	8
– Highlighting important comments	4	8
• Status	18	56
– What is the current status?	8	24
– Next action	11	20
– Bug hand-off	6	12
• Email	17	45
– Overwhelming volume	8	12
– Email is important	11	16
– Email is not important	2	2
– Filtering	6	9
– Formatting	4	6
Supporting Tasks	**20**	**700**
• Code Review	20	130
– Recommending reviewers	6	10
– Patches	10	27
– Importance of review timeliness	8	12

Table 5.2 – *Continued*

Category/theme/sub-theme	**# Participants**	**# Quotes**
– States	12	32
– Process and community	8	11
– Risk/Reward	5	8
– Misc	17	30
• Triage & Sorting	20	259
– Sorting / filtering	15	35
– Bug assignment	12	25
Who gets the bug?	7	10
Self Assignment	6	10
Unassigned	4	5
– Components/products	17	41
– Component owners	3	4
– Last touched	4	7
– Is this bug actionable?	10	16
– Volume of bugs to triage	4	5
– Duplicates	4	7
– Bugs in “General”	6	9
– Bug kill days	3	4
– Triage & community engagement	5	7
– Midair collision	2	2
– Triage meetings	5	5
– Triage of other components	2	4
– Triage process	10	18
– Comments	4	4
– Misc	15	39
• Reporting	20	145
– Submission is harder / more confusing than needed	17	33
– Improving reporting	10	14
– Defects in the reporting experience	4	9
– Metadata	10	23
– The role of the description and summary fields	9	14
– Possible enhancements to improve reporting	12	15
– External tools	2	3
– Intimidating for new users	7	8
– Misc	8	26
• Search	14	53
– Quick search	4	7

Table 5.2 – *Continued*

Category/theme/sub-theme	**# Participants**	**# Quotes**
– Advanced search	5	5
– Saved/shared	5	8
– Hard/confusing	7	10
– Date range	3	3
– Performance	3	5
– Product	2	2
– Dups	3	5
– Misc	6	8
• Testing & Regression	14	37
– Regression	5	9
– Testing and its reliability	8	15
– QA and Validity	6	9
– Fuzzing	2	4
• Tasks	13	76
– Role-specific views	4	7
– Release management	10	27
– Where a bug lands?	10	22
– Statistics	8	12
– Workflow	6	8
Expressiveness	**19**	**188**
• Metadata	20	132
– Tracking flags	14	38
– Whiteboard	17	36
– Keywords	14	22
– Meta bugs	3	6
– Metadata (general)	8	12
– Tagging (general)	3	4
– Status flags	9	14
• Severity/Prioritization	19	56
– Unclear definition of priority/severity	6	8
– Priority	13	14
– Severity	11	16
– Prioritization	8	13
– Misc	5	5
Everything Else	**20**	**117**
• Version Control	8	43

Table 5.2 – *Continued*

Category/theme/sub-theme	# Participants	# Quotes
– Reviewing via Github	3	12
– Checkins	6	13
– Branching	4	6
– Merging	2	4
– Integration of Bugzilla with Hg	4	8
• Bugzilla Issues	16	64
– UI	8	11
– Advanced scripts and tweaks	7	11
– Process	3	3
– Feature pages	8	11
– Performance	2	3
– Culture	4	6
– Misc	12	19
• Useless	9	10

Table 5.2: Overview of the concept categories.

5.2.2 Situational Awareness

Email is a primary way of receiving bugs and communication updates. Developers have to manage the flood of emails they receive daily. As a result, developers find it hard to determine the current status of the bug they are working on where it is bug is in the bug-fixing process.

One of the most surprising concept categories we found clustered a series of themes and sub-themes that relate the idea of *situational awareness*. Situational awareness is a term from cognitive psychology; it refers to a state of mind where a person is aware of the changes in their environment [64]. The term is an apt description of how software developers must maintain awareness of what is happening on their project, to be able to manage a constant flow of information as issues evolve, and be able to plan appropriate actions. Developers often find themselves trying to identify the status of a bug — What is the current status of the issue? What is blocking this issue? What issues am I blocking? Who is the best person to review the patch? — as well as trying to track their own tasks — How many bugs do I need to triage, fix, review, or follow up on?

15 of the 20 participants expressed a desire in having private dashboards that would allow them to track their own activity and quickly determine the what changes had occurred since the last time they had examined the issue: *"[A] gigantic spreadsheet of bugs he is looking at. It would be useful to know how the bugs have changed since he last looked"* (P11). Since developers can track only a limited number of issues in their heads, they desired *"a personal list of bugs without claiming them"* (P8) and ability to *"get info based on what has changed since the last time I looked at it"* (P6).

Since tasks such as code review require collaboration or otherwise depend on others, developers expressed a desire: *"to better understand what people are working on and how it fits with [their tasks]"* (P11). Knowing what others are working on can enable assigning more relevant tasks to people: *"[I] frequently uses the review queue length to see who might be the quickest"* (P17). In addition, workload transparency can enable better load balancing: *"to spread the load on key reviewers"* (P3).

Situational awareness also crosscuts many of the other sub-themes from the other categories such as *Supporting Tasks::Code Review::Recommending Reviewers* where developers wanted the ability to determine the work load of a reviewer before requesting a review; this could be captured in a public dashboard that showed the reviewer's current queue.

5.2.3 Supporting Tasks

Bugzilla is the main venue for developers to collaborate on bug reporting and triage, code review activities, communication, release management, etc. However, the lack of good sorting and filtering mechanisms increases the effort required to perform these tasks.

Developers also had a number of comments that related directly to their daily tasks, including code review, triage, reporting, testing, and release management.

All interviewed developers expressed concerns with current support for tasks related to code review. Code review is an essential part of the development process at Mozilla; every patch is formally reviewed before it can be committed. *"The review system doesn't seem to be tightly integrated into Bugzilla"* (P10).

When a submitting patch for review, a developer must specify the name of a person they are requesting a review from. Deciding who is the "best" reviewer to send a patch to can be challenging since this requires estimating the workloads of others in addition to having an understanding of their expertise. Being able to get a list of reviewers along with their workloads would reduce the amount of time required to get a patch approved: *"He will go to the review requests page and look up people's review queue lengths to see who might be quickest"* (P17).

Developers also want to be sure that they are not blocking others; they want to be able to easily assess their own review queues. One developer explained that he sorts his bug mail: *"so that it shows only the reviews that are asked of him"* (P20).

While developers face challenges managing a large flow of issues they are working with, Bugzilla also lacks good sorting and filtering mechanisms needed for tasks such as bug triage and sorting: *"The lack of ability to filter mixed with the volume makes it an overwhelming task. A lot of rework in sorting bugs rather than actually triaging it and moving it along"* (P9). One of the limitation of Bugzilla is that its interface is *"cluttered with rarely-used fields"* (P5). The amount of the metadata displayed to its users makes it difficult to search for the information that the developers need. Developers wanted to be able to sort information *"based on tag values"* (P5).

5.2.4 Expressiveness

Bugzilla stores a wide variety of metadata that can be overwhelming and intimidating to the developers reporting and fixing issues. A good tagging system could eliminate many fields such as OS, severity, priority, platform, etc.

Developers did not want Bugzilla to prevent them from modifying issues in a way that works for them; in particular, they wanted to be able to express themselves in a variety of ways to convey information to other stakeholders. Expressiveness in Bugzilla is achieved in part through the large number of labelled fields available for entry on issues including whiteboard terms, keywords, tracking flags, priorities, components, etc. These fields are used as *"sort of a tagging system"* (P4) during issue management including tracking status, seeking approval, highlighting important information on a bug, or making version names and numbers. While these fields are important for grouping and organizing issues in addition to communicating awareness and interest on issues, they are often used in a project- or team-specific manner. Therefore, developers desired a good tagging system that *"could get rid of many fields"* (P5) as *"a cross between the keyword field and whiteboard field"* (P5).

Prioritization is another concept that appears to be poorly supported in Bugzilla. Some fields such as `severity` and `priority` do not have a clear definition and thus are often used incorrectly. Developers explained that *"priority and severity are too vague to be useful"* (P5) and *"everyone doesn't use priority levels consistently"* (P11). Instead of *"prioritizing en masse"* (P5), developers seek a means to *"set our own priorities on bugs"* (P15) or team priorities on issues and tasks and sort them based on their importance: *"team decides priority as a group, P1: forget about everything else, P2: have to do, P3: don't look until later"* (P12).

5.2.5 Everything Else

This category includes Mozilla internal topics related to the discussions about interacting with version control systems, other general issues about Bugzilla such as performance, culture, process, etc. Details about this category are in the Appendix A.

5.3 A New Model of Issue Tracking

As a result of our exploratory study, we devised a developer-centric model of issue-tracking systems that addresses many of the key challenges that the study uncovered. The improvements are organized around the concepts that emerged during the card sort process. The model was then instantiated as a high-fidelity prototype (described in Section 5.3.4.1) that addresses the perceived limitations of the Bugzilla platform and its incumbent processes; this was done by providing developer-specific enhancements to aid in task-specific decision making.

We hypothesize that many of the comments made by developers in the concept categories of situational awareness, supporting tasks, and expressiveness can be tackled through an effort of creating custom views of Bugzilla tailored for individual developers that filter relevant information from the issuer tracking system and support their work tasks. In general, the information developers are trying to keep up with is usually stored within the Bugzilla repository, it is just hard to access. The primary means that developers currently do this is through 'bug mail'; that is, the developers subscribe to a large number of bugs and are sent email notifications whenever something in the bug changes. Unfortunately, this results in hundreds of emails per day, more than even a complex array of filters can hope to keep up with without some important updates being lost.

Based on the qualitative analysis, we identified several ways in which we hypothesized that the needs of Bugzilla users could be better met, largely by easing access to key information that already exists within the system but can be hard to obtain through the web interface or email filtering alone.

We identified two groups of work items that developers are engaged with: issues and patches. Developers work with both of these on a daily basis. Issues contain bug reports and new features that need to be implemented, while patches contain the reification of that issue in source code that can then be reviewed by other developers. Both the management of issues and patches are of key concerns to developers.

In this section, we describe the key information elements that developers stated a desire to keep appraised of, and we describe how these elements can change over time. One of the reasons these requests arose is that people themselves are essentially metadata on issues: a person files an issue, an issue is assigned to a person, a person comments on an issue, requests a review, and

files a patch. While the issue is the central artifact, all *actions* on issues are generated manually by people.

5.3.1 Issues

For developers who use Bugzilla, email is the key communication mechanism for discussing bugs and the bug fixing process. Any change on an issue results in an email being sent to the developers whose names are on the issue's CC list. For many developers, these emails are the primary means for maintaining awareness of issue evolution in Bugzilla. Developers receive an email every time they submit or edit an issue, someone comments or votes on a bug, or any metadata field is altered. An individual developer can track only a limited number of bugs in their head; 10 of the developers who were interviewed wanted to be able to watch bugs and sort them by activity date. One said, *"[I] would like to have a personal list of bugs without claiming them"* (P8).

Many developers wanted "watch lists" for indicating their interest in an issue without taking ownership of it. Watch lists provide means to track bugs privately by adding them to their private watch list without developers having to CC themselves on the bugs. Bugs are ordered by "last touched" time as *"last touched time a key metric for tracking if work is being done on a bug"* (P1).

We found that developers face challenges in determining what has happened since the last time an issue was examined; this was noted by 12 participants, whose comments included: *"[I want] to get info based on what has changed since last time I looked at it"* (P6), and *"You look at the bug and you think, who has the ball? What do we do next?"* (P7).

Ultimately, developers wanted to be able to more flexibly track, query, and explore the issues stored in the Bugzilla repository. While Bugzilla provides a web interface that developers must use to modify issues, developers also rely heavily on automated bug mail, as email clients support flexible sorting and filtering of messages. Unfortunately, in an active bug, many changes may be happening simultaneously, resulting in a large number of emails, only some of which may be interesting to the developer; also an email message can be easily missed amid the deluge, causing an issue to "fall on the floor."

To track issues effectively, developers need access to the issue and its metadata presented to them in a meaningful way. From the expressiveness concept category, we know that different developers often desire access to a different pieces of metadata. In reality, these requests arise because developers are thinking about how they will filter the issue's bug mail. What the developers seek is a customized list of issues — implemented through whatever technical means might work — that keeps track of a variety of issues and can specify how "interesting" they consider each one to be. For example, developers want a list of issues that are assigned to them, sorted by when they last changed. Additionally, they would like lists of issues they have commented on, are CCed on, and have voted on. They would also like to be able to have component-level lists

that they can then select issues to move into private watch lists to keep track of. As these issues evolve, the lists should continually update "live" so that the most recently updated issues appear first.

5.3.2 Patches and Reviews

Bug fixing tasks involve making *patches*. While working on an issue, developers will often split a single conceptual fix into multiple patches: *"People are moving to having multiple patches rather than one large patch. This really helps with the review. Bugzilla isn't really setup for this model"* (P16). Ten developers expressed a desire to improve the way Bugzilla handles patches: *"It would be good if [Bugzilla] could tell you that your patch is stale"* (P13). Developers were primarily interested in tracking their own patch activity, as well as determining what patches are awaiting reviews, or who is blocking their reviews.

12 participants indicated that they felt Bugzilla is ill-suited for conducting code review: *"The review system doesn't seem to be tightly integrated into Bugzilla"* (P10). A common task is determining who the "right" reviewer would be to request a review from: this may may be the one having faster review turnaround or the one having a shorter review queue. In order to address this question, developers need to be informed about reviewers' work loads and average response time: *"I can be more helpful if I can better understand what people are working on and how it fits with [their tasks]"* (P11). While Bugzilla keeps track of all of the reviews outstanding on all issues, developers cannot query to find out what the review queue of an individual developer is. Supporting this task is particularly important if a developer is not familiar with the module/-component reviewers: *"When submitting a patch for another component, it's more difficult, he has to try to figure out who is good in that component, look up their review info"* (P8).

Developers also need some means to observe and track their own tasks, such as their review queues: *"He has a query that shows all his open review queue"* (P16), *"The review queues are very useful, he will check that every few days just to double check he didn't miss an email"* (P8).

Code review is a particularly important task, where developers do not want to miss key events related to it. While their patches are awaiting review they are effectively blocked for that issue. If their review request is missed or lost, a significant amount of delay can be incurred. The converse also happens, if a developer misses a review request they can block other developers.

Rather than having code review notifications flow through bug mail, the developers requested a dedicated review queue. As developers usually have a limited number of outstanding review requests, bringing these all together can ensure that none are missing; for example, in a list of 5 requests, the one that is a month older than the other four tends to be noticeable. Developers also wanted to be able to observe the review queues for other developers so they can estimate whether they would be able to turn around a review for them in a reasonable amount of time.

5.3.3 Model Summary

Our conceptual model of issue tracking is derived from the data of interviews with industrial developers, and reflects the concepts that emerged during the qualitative study described in Section 5.2:

Issues are work items that developers are involved with during the active development of a software system; issue tracking issue is one of the key tasks developers perform daily.

Patches are code modifications that developers "produce" to resolve issues or implement new features; a typical developer's daily activities may include writing and tracking their own patches, as well as conducting code reviews of the patches of others.

Both issues and patches relate to the themes of *task support* (Section 5.2.3).

Identifying relevance involves discerning work items, such as issues and patches, that are of concern to a developer; it relates to *situational awareness* (described in Section 5.2.2).

Information reduction concerns filtering out irrelevant metadata fields — such as priority, severity, product, etc. — so that the more important fields — such as bug ID and summary fields for issues, and patch ID, issue ID, flags, and requester fields for patches — are more prominent; it relates to *situational awareness*.

Temporality concerns displaying issues sorted by the "Last touched" field, adding visual clues to the items that require attention (e.g., patches awaiting reviews), adding context to work items (e.g., what has changed on the issue?); it relates to both *expressiveness* (Section 5.2.4) and *situational awareness*.

Roles concern organizing issues by developer role on the project; roles can include bug reporter, bug fixer, and reviewer.

Ownership relates to separating issues that developers are responsible for from those in which they are only observers.

Roles and ownership informational needs relate to both *situational awareness* and *task support*.

This model highlights the key informational needs much desired by the industrial developers for the dashboards that are designed to overcome current limitations of the issue-tracking systems and provide developers with better awareness of their working environment.

5.3.4 Prototype and Its Validation

To validate the concepts that emerged from the qualitative analysis of the interview data, as well as the prototype (see Section 5.3.4.2) we interviewed eight (D1-D8) Mozilla developers for 20-60 minutes. None of these developers were involved in the initial set of interviews.

The developers confirmed that they face challenges keeping track of tasks they are involved with, organizing issues they are working on, *"I have no way of parsing or prioritizing the component information so I can't watch that very well"* (D8). They observed that developers often created their own ad hoc solutions for organizing tasks and keeping track of issues, such as public work diaries, notes on paper, "homebrew" tools filtering bug mail, saved Bugzilla searches, mail clients with better filtering capabilities. For example, one developer explained *"I use etherpad to keep track of my list of bugs for each project listed here and I move them up the chain. I put little notes on something so I can keep track of what I'm waiting for; it's all very ad hoc"* (D1). Another developer tried several applications before he *"switched to paper now; [I use] paper for one-day tasks, Bugzilla for longer tasks"* (D7).

Developers were keen to ensure that patches did not *"fall off the radar"* (D1), and that important issues were not allowed to *"fall through the cracks"* (D5).

Ultimately, these interviews ended up echoing most of the major themes identified during the first round of interviews.

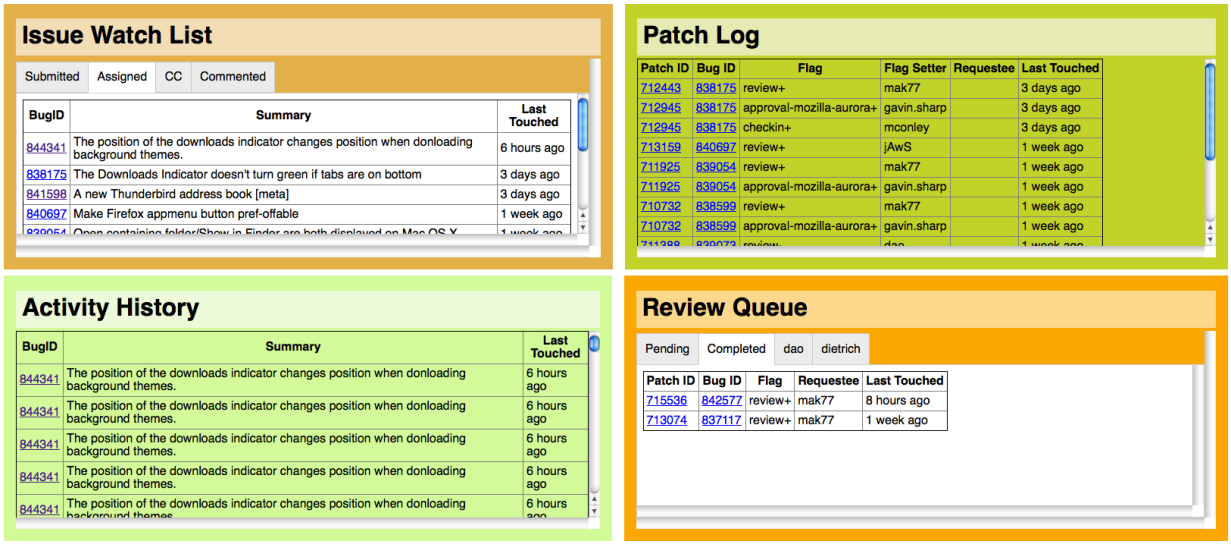


Figure 5.3: High-fidelity prototype.

5.3.4.1 High-Fidelity Prototype

We have implemented a prototype of our model in the form of a personalized dashboard that provides developers with public watch lists, patch logs, and an activity history feature; the active history can help Bugzilla users maintain better awareness of the issues they are working on, as well as other issues that interest them, and common tasks they perform daily. Our solution is organized around custom views of the Bugzilla repository supporting ongoing situational awareness of what is happening on the project. Figure 5.3 illustrates our prototype; it shows a custom view of the Bugzilla repository generated for Mike Conley, a Mozilla developer.

The dashboard serves as a template for displaying information to assist in developers' common tasks. This template contains all the key elements that are important to the developers as they capture the concepts derived from the qualitative analysis (as described in Section 5.3.3). There are four panels: `Issue Watch List`, `Activity History`, `Patch Log` and `Review Queue`. The `Issue Watch List` panel includes four tabs: `Submitted`, for issues that are reported by a developer; `Assigned`, for bugs that a developer needs to resolve; `CC`, for issues that developer expressed interest in following up on either by putting his name on its CC list or by voting on a bug; and `Commented`, for issues that the developer participated in a discussion on. The `Activity History` panel displays the developer's activity on the project. The `Patch Log` panel displays the list of recently submitted patches, the outcome of a code review (positive or negative) with the name of the reviewer, or the current status of the patch (approval for committing the patch to the master tree) with the name of the person who set the flag. Finally, `Review Queue` panel contains several tabs: `Pending` shows patches that await the developer's review, together with the name of the person who made this request; `Completed` lists recently reviewed patches together with the review decision; and `dao` and `dietrich` show the review queues of two other developers of interest (Figure 5.3).

By default, all of the tabs are sorted by `Last Touched`, the timestamp of the most recent change on an issue. If the developer prefers to see the date and time, they can hover over the last touched cell.

5.3.4.2 Prototype Evaluation

This step of our study aimed at validating our high-fidelity prototype. As mentioned earlier (in Section 5.3.4), we conducted eight interviews with Mozilla developers lasting 20–60 minutes each; none of these developers had participated in the original interviews. Our interviews included questions relating to how the developers managed their daily tasks, patches, and code reviews. During the interviews, we provided the developers with our prototype, personalized for their work from the previous day. We asked the developers to comment on the prototype and how it could be extended in the future.

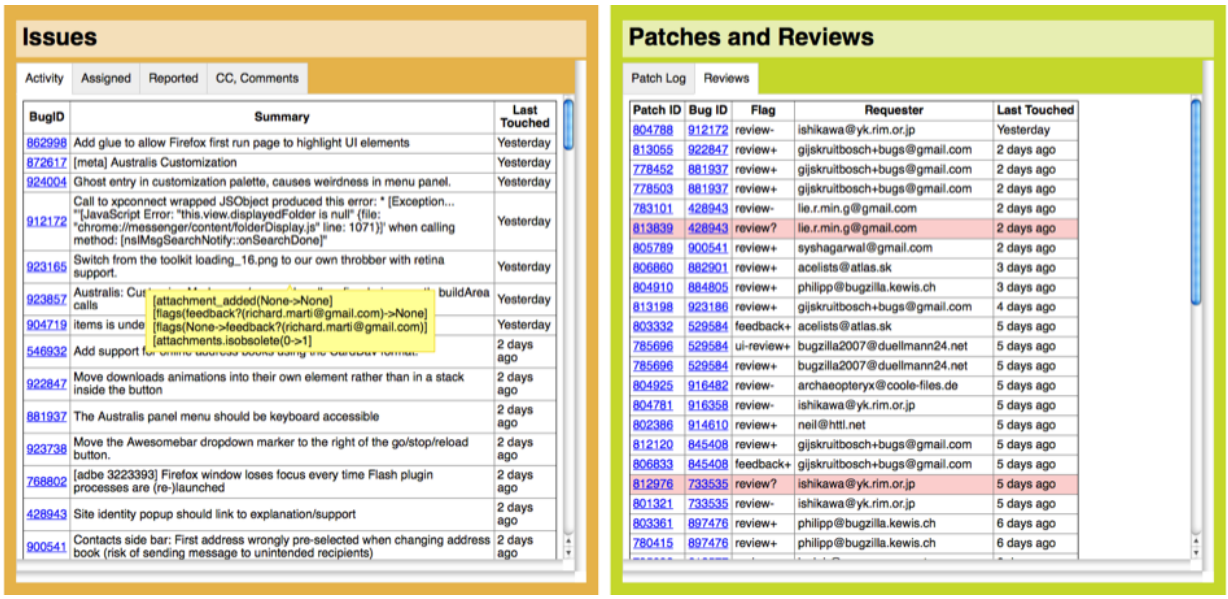


Figure 5.4: Developer dashboard.

Our prototype focused on extracting metadata from Bugzilla and displaying it for users in a meaningful way. The intent was to provide a dashboard that developers could use throughout the day to keep track of their issues. The night before we met with the developers we generated two versions of the prototype (see Figures 5.3 and 5.4) that were specific to the issues they were working on at that time.

The high-fidelity prototype was well received, for example, *"this [prototype] is really cool! I think this is great, something like this would be fantastic for sorting through all the things I need to take care of. When can I start using it?"*D6).

Developers were glad to see that most of the tasks they perform are supported: tracking issues, tracking assigned tasks, assigning reviewers, prioritizing tasks, etc. *"I really like the issue watch list. I have my own custom Bugzilla queries for the four columns [Submitted, Assigned, CC, and Commented]. It's a saved search I have but when I do use it it is hard to look through the list"* (D2). *"I really like the idea of having peers tab"* (D7). *"Oh, and activity history so I can see everything I've contributed to the project"* (D6). *"I like the patch log as well because I find that the bugs I care about often have active patches in them"* (D2). These were all ways in which the prototype helped developers access information that was crucial to their day-to-day development tasks.

Most developers said that being able to determine what issues are assigned to them is useful to have a quick start each day. While some developers would choose to open developer dashboards

once or twice during the day: *"If I have small tasks more often [to visit], one big thing — once every morning and evening"* (D6) , others expressed interests in frequently checking dashboards for any updates: *"I'd like to keep this [dashboard] open all the time, all day long"* (D7), *"Honestly, if we had better dashboards I would keep them open and come back to it frequently, several times a day"* (D8).

During the interviews we demonstrated two versions of our prototype: a two-panel view (similar to the one shown in Figure 5.4) and four-panel view (shown in Figure 5.3). Both versions contained the same information and tabs. The only difference was the grouping of the displayed tabs and the way page was divided into the panels. Developer feedback on the page layout was unanimous that the two-panel layout of the landing page was better: *"The two views make more sense. Reducing the need to scroll is good."* (D1), *"I work on a mobile a lot and the two-column one is nice because I can see a lot more data"* (D6).

The interviews also revealed that further refinement of the prototype's functionality is needed. While we expected requests for richer customization of the base template and the interface, we were surprised to hear that most improvements involved questions of expressiveness, including:

- *A customizable template* — Developers expressed a variety of preferences what tabs should be present by default; providing a template that developers can modify according to their individual needs is important.
- *Search functionality* — Developers wants to be able to perform a quick search on the various bug fields, both the visible ones and the underlying fields that are not shown.
- *Time range options* — While custom pages were generated for the past three weeks, we received a variety of answers with respect to this setting: *"3 month is useful, for thinking about our quarterly goals"* (D7), *"for last month"* (D6), *"every year or every six months is useful ... to recall things I have worked on"* (D7).

Apart from common recommendations on further improvements, we received requests to meet individual needs of the developers or teams. One developer wanted the support of other tasks such as triage of issues of the component they are responsible for: *"We have bi-monthly person for a week who is responsible for bug triage: review bugs, triage them — these bugs are important, these are not, check may be I have missed something"* (D7). Another developer wanted to be able to send email from the landing page, e.g., to reply to a recently added comment on an issue without having to enter Bugzilla.

All of the desired improvements that were discussed in the interviews are easy to support in the current implementation of our high-fidelity prototype. Also, we are actively enhancing the prototype to make it available to the developers as a plugin or extension to the Firefox browser.

5.4 DASH: Improving Developer Situational Awareness

To motivate and define all DASH features, we used the model (described in Section 5.3) that emerged from the qualitative study.

Based on our prototype and the feedback we received while validating it, we implemented a number of changes to our final tool. These modifications include:

- a component-based query to allow developers to organize work items for a particular product they are involved in;
- a time-range option to generate custom views of Bugzilla for a specific time frame; and
- a two-list layout that separates issue-tracking activities from tracking updates for patches and reviews.

These improvements enhance our model of the informational needs for issue tracking (summarized in Section 5.3.3) by including additional features related to the topics of situational awareness and expressiveness. Further details about the architecture and implementation of DASH can be found elsewhere [107].

5.4.1 Tool Validation

To validate our tool we analyzed the collected usage data of developer interactions with the tool, and then conducted interviews with industrial developers and analyzed bug mail compression.

5.4.1.1 Usage Data

To determine how our tool is used, we implemented event tracking mechanisms and collected user data for the period of four weeks. Usage data includes information about the users and their demographics, the content visited, the queries generated, as well as the events triggered such as form submits, tab switching, and clicks on the dashboard. The usage data indicates 61 unique visitors who have used our tool with the vast majority being from the Mozilla Toronto office (36 developers) with the total of 46 visitors from Canada, as well as individuals from the United States (8), Germany (2), France (2), Netherlands (2), and New Zealand (1).

Event tracking allowed us to evaluate and measure how users interacted with our tool; Table 5.3 summarizes these events. We found that tab switching was the most frequent activity, and that `Reviews` was the mostly visited tab, comprising more than 80% of the total events. We also noticed that users did not use dashboards as their landing page for Bugzilla: fewer than 1% of all events were referrals to the issue tracker via external links (issue IDs and patch IDs).

Category	Name	Total Events	%Total Events
Tabs	Activity	105	81.78%
	Assigned	65	
	Reported	107	
	CC, Comments	104	
	Patch Log	105	
	Reviews	111	
Form	Submit	119	17.47%
Links	Issue	3	0.73%
	Patch	2	
Total:		681	100%

Table 5.3: Event tracking measuring user engagement.

Further interviews with the developers revealed that they prefer to copy-paste issue IDs into an already-open Bugzilla window, instead of having another browser tab pop up each time they clicked on an issue or patch presented in their dashboards.

5.4.1.2 Interviews and bug mail compression

While usage data can help answer questions like "who?", "how?" and "how often the tool was used?", it does not provide any indication on the advantages of the tool for the developer over their day-to-day practices. To investigate this, we visited the Mozilla office again and talked to the most active users of our tool (developers A–I in Table 5.4).

During these interviews, we asked them to count the number of work-related emails they received the previous day from Bugzilla (carefully stating the assumption " if yesterday was your typical day") to determine their average daily and weekly bug mail. We then generated developer-specific dashboards for each developer for the period of one week and counted the number of items displayed on the dashboard; these include issues, patches, and their reviews that the developer was involved or interested in. The results can be found in Table 5.4 demonstrating the compression of the received bug mail by over 99%.

We then asked developers for their feedback on the relevance of the filtered information, the correctness of our filtering approach, and the accuracy of the bug mail compression ratio. Interviews with the developers confirmed that vast majority of the received emails are not relevant to their work: *"Easily 200+ bug mails a day ... in fact most of them I do not need to read"* (H).

Seven out of 9 developers (78%) expressed a desire to use the developer dashboard in place of their bug mail, *"... with a performant/live updating dashboard it seems feasible to largely replace bug mail"* (B). From those who wanted to continue receiving all their bug mail, one developer

Developer	Bug mail (one week)	Dashboard (one week)	Reduction Percent
A	435	16	99.96%
B	605	25	99.95%
C	2500	28	99.98%
D	1200	5	99.99%
E	525	9	99.98%
F	1500	26	99.98%
G	235	2	99.99%
H	1000	14	99.98%
I	250	13	99.94%
Average:	917	15	99.97%

Table 5.4: Scale of bug mail and number of items displayed on the dashboard for the week-long period.

needed the ability to see and follow conversations happening on a bug: *"One of the advantages of email is that I have a copy of the conversation that's going on in a bug — so I don't actually have to enter Bugzilla to read the comments. With your dashboard, I can know that a bug had a new comment posted, but then I have to go into Bugzilla to see if it's important. So I think that's a slight deficiency"* (A). Another user wanted to watch component bug mail to identify new bugs he might be interested in fixing: *"99% of this [bug mail] is not about bugs I'm involved in, but Bugzilla components I'm watching to see if any bugs I am interested in have come up"* (D).

5.5 Discussion

In this section, we describe some of the key concerns raised by developers that we have not yet investigated, along with threats to validity.

5.5.1 Threats and Limitations

Our dashboards are implemented using custom views of the issue-tracking system filtering the data to identify developer-specific items. Currently, they do not support developer needs for private watch lists. Public watch lists (aka being on the CC list) provides a mechanism to indicate interest in an issue without taking ownership. In contrast, private watch lists enables developers to privately mark bugs to watch. Unfortunately, implementing these lists is not possible within the current Bugzilla architecture; we are currently investigating ways around this.

Our tool does not currently support personal tagging. Some developers wanted the ability to group issues by whiteboard flags, *"may be you could say what whiteboard flags you're interested in. But we don't want to get it to be replacing Bugzilla at all."* (D2) Others wanted to be able to *"have them grouped by colour and sorted within the colour by date"* (D5). These grouping enhancements were by far the most prevalent: developers wanted to be able to organize their issues in ways that were most relevant to them. Most developers expressed a willingness to manually move issues to specific groups if this preference could be maintained.

Bettenburg and Begel [23] recently described an automatic approach for classifying work items to reveal what is actually happening within a task. Their approach could add contextual rationale to the work items displayed on the dashboards to help developers interpret and differentiate their work activities.

The first limitation lies in the validity of our findings from the qualitative study. However, Bugzilla is widely-deployed and used by thousands of organizations, including open-source projects such as Mozilla, the Linux kernel, and Eclipse, as well as NASA, Facebook, and Yahoo!.[3]. Our investigation has focused strongly on Mozilla developers using the Bugzilla issue-tracking system. While some issue trackers provide better developer-oriented support (e.g., the Bitbucket issue tracker allows users to tag people so they get notifications to issues they are not subscribed to), our work aims to provide guidelines that we hope can improve the information filtering ability of all issue trackers.

As with all exploratory studies, there is a chance we may have biased the categories that were identified. We tried to minimize this by coding the first three subjects independently, performing an inter-coder reliability measurement on the next three, by validating our findings with a separate group of developers along with our high-level prototype and finally by evaluating our final tool both quantitatively (usage data analysis and bug mail compression) and qualitatively (interviews with active users).

5.5.2 Deployment

We are currently in the process of deploying the developer dashboard into the Mozilla development environment. The initial tool is currently offered as a web-based service, hosted by one of our research group's servers. While most of the data stored in Bugzilla is public, Mozilla is interested in moving the tool to their internal network to be able to populate confidential data, such as security bugs. In our first prototype, we relied on the Bugzilla API to get the data; however, we found its performance to be unusably slow. With the help from Mozilla developers, we then changed DASH's backend to use Elasticsearch. Elasticsearch [63] is an open source search engine that provides near real-time search and full-text search capability. The improvement in response

[3] http://www.bugzilla.org/installation-list/

time was remarkable, and it allowed Mozilla developers to interact with the developer dashboard in real time. This effectively turned our proof-of-concept research tool into a practical industrial-strength analytics tool [107].

5.6 Summary

In this chapter, we described a qualitative study of industrial developers that identified a need for improved approaches to issue tracking. The model of these informational needs was derived from the interview data in which it was "grounded". Based on this model we designed our tool implemented in the form of developer dashboards. These developer dashboards can enable developers to better focus on the evolution of their issues in a high-traffic environment, and to more easily learn about new issues that may be relevant to them. We have proposed an approach that improves support for particular tasks individual developers need to perform by presenting them with the custom views of the information stored in the issue management system. By improving issue management systems, developers can stay informed about the changes on the project, track their daily tasks and activities. We validated our initial prototype and later our tool with Mozilla developers and received feedback on the desired information and features. We are actively improving our tool to enable it to be deployed in an industrial setting.

Chapter 6

Discussion

Chapter Organization Section 6.1 discusses the main limitations of employing software analytics. Section 6.2 suggests extensions to this research. Section 6.3 highlights some possible future directions in the software analytics research.

6.1 Limitations

There are two main limitations of our approach to employing software analytics:

- **Many other development artifacts are not analyzed**

  We have applied software analytics to extract facts and insights from only issue repositories, version control systems, and usage logs. We did not consider other possible sources to support development decisions such as mailing lists, social media, test suites, configuration management systems, etc. Our intent was to show that some of the decisions related to the key processes (e.g., code review, completion of daily tasks such as bug fixing, patch writing, etc.) can be informed by utilizing software analytics.

- **Exploratory research rather than a mature software analytics tool**

  Our initial work demonstrates that by applying techniques and approaches from the field of mining software repositories, we were able to extract information about specific aspects of software development process. We plan to continue to investigate how analytics can be utilized to support better decision making. In particular, we want to explore the power of predictive analytics to anticipate future trends and patterns, for for example, developing a prediction model of community contributions, recommending best "next action" to

a developer for being more productive in completing tasks. While this work is only a preliminary effort in the area of integrating analytics into the software development setting, our developer DASH tool has recently been adopted for everyday use within the Mozilla project.

6.2 Possible Research Extensions

- **Combining different kinds of analytics**

  We have demonstrated how different analytics, including usage data (Chapter 3), summarizations in a form of lifecycle models (Chapter 4) and customization (Chapter 5) can inform better decision making. While we have studied each kind of analytics, we have not looked into combining them together. We believe that integrating different kinds of analytics would be beneficial. For example, we can combine the work described in Chapter 3 and Chapter 4 to augment and evaluate Chapter 5. One way to enhance personalized development tools is to add summarizations in the form of patch lifecycles to the customized dashboards to increase developer awareness of how patches evolve and change over time. Adding patch lifecycles as reports to the dashboards can further support developers' code review tasks providing better transparency on the velocity of their patches. Dashboards can further be improved by integrating quantitative analytics such as developer's review queues that can provide improved support for performing code review tasks. Saying this, we can design and implement a dashboard-like system for code reviews.

- **Porting DASH to other systems**

  In this work, we have only focused on improving Bugzilla issue tracking system. We could adopt our dashboard-like approach to other issue tracking systems. For example, we can generate our DASH for GitHub to compare it with the current GitHub's dashboard. We then can conduct usability studies with developers to evaluate the benefits and the limitations of the two dashboards.

- **Measuring situational awareness**

  Chapter 5 demonstrated that the dashboard-like approach to issue tracking can improve situational awareness. However, it is difficult to measure situational awareness. While we analyzed the collected usage data of developers' interactions with our tool, providing better evidence of the improved awareness remains a challenge. We need to figure out the type of experiments that could allow us to demonstrate the effectiveness of dashboards in enhancing awareness on the projects and issues developers are involved with.

6.3 Future Work

Our initial work demonstrates that software analytics are useful in recovering the knowledge about the software processes and practices, as well as developer and user behaviour. We mostly focused on analyzing data from the issue-tracking systems, code review systems, server logs and release history. There are many potential questions to address next, e.g., how can we make data useful to anyone involved in software project and not just to developers and project managers? How can we better tie usage analytics to development analytics? How can we make analytics solutions more generic and not only project-specific?

The future work in applying software analytics within the area of mining software repositories is broad and includes:

- **Framework for large-scale data analysis**

  Develop a framework for large-scale data analysis of software repositories. With the increasing variety and volumes of software artifacts, traditional offline read-only databases may not be suitable to store large data sources or may not be able to handle the processing demands.

  New big data technologies and architectures such as Hadoop or NoSQL databases have emerged to support these big data analytics environments. We could utilize emerged technologies to support mining and analysis of the software repositories. We have showed that by adopting Elasticsearch — a distributed full-text search engine — we could overcome Bugzilla performance when developing real-time developer dashboards.

- **Study commercial systems**

  Our current research is validated by conducting empirical studies on open-source systems. Conducting case studies of commercial systems, in particular what are the repositories that are available to commercial systems that we could leverage. These repositories might include various sorts of logs and field reports.

- **Study social dynamics**

  While we studied the influence of non-technical factors on code review response and outcome, we only considered some of the factors. We could also expand our research by studying social dynamics among developers collaborating on the project. By analyzing how developers interact and communicate with each we might get insights into organizational and collaboration dynamics in an open-source community.

Based on all our conversations with Mozilla developers throughout this research process, we have identified three design requirements that our developer dashboard needs to consider:

- **Personal tags to group issues**

  These grouping enhancements were by far the most prevalent: developers really wanted to be able to organize their issues in ways that were most relevant to them. Most developers expressed a willingness to manually move issues to specific groups as well.

- **Support for other tasks**

  The current tool is mainly designed to support code review tasks including patch status updates, review flags, etc. We could think of the ways to support other development tasks such as issue triage, release management, etc.

- **Context relevance**

  For example, developers may only wish to receive updates relevant to the issues they are interested in, not every issue that they are involved in.

Chapter 7

Conclusion

Analytics have been demonstrated to be effective in leveraging large volumes of data from multiple sources to support better decision making. Organizations commonly apply analytics to *business data* to describe, predict, and improve business performance. Unfortunately, developers need better support to complete their daily tasks and make daily decisions.

The thesis of this dissertation is that by providing practitioners with software analytics that summarize trends, synthesize knowledge from the development artifacts and visualize facts and tasks, we can support them with better insights into their processes, systems, products, users and help them make better *data-driven development* decisions.

To validate these claims, we have provided three different examples of employing software analytics that assisted developers and teams in understanding the big picture, discovering insights from the data and making decisions related to user adoption, managing community contributions, and supporting developers' daily activity and tasks.

In Chapter 3, we showed how analysis of real-world usage data can be used to assess user dynamic behaviour and adoption trends of a software system by revealing valuable information on how software systems are used in practice.

In Chapter 4, we described a lifecycle model that synthesizes knowledge from software development artifacts, such as reported issues, source code, discussions, community contributions, etc. We demonstrated how lifecycle models can be generated, and we presented three industrial case studies where we apply lifecycle models to assess projects code review process.

In Chapter 5, we presented a developer-centric approach to issue tracking that aims to reduce information overload and improve developers' situational awareness. Through qualitative study of interviews with 20 Mozilla developers, we identified four conceptual areas of how Mozilla developers interact with Bugzilla to report and resolve bugs, as well as to manage the collaborative

process. The study also revealed that developers face challenges maintaining a global understanding of the issues they are involved with, and that they desire improved support for situational awareness that is difficult to achieve with current issue management systems.

Based on the results of the qualitative analysis, we identified several ways in which the needs of Bugzilla users could be better met, largely by easing access to key information that already exists within the system but can be hard to obtain. We have recovered a model of the kinds of information elements that we understood to be essential for developers in completing their daily tasks, and from this model we have developed a prototype tool organized around customizable dashboards supporting ongoing situational awareness on what is happening on the project. Further quantitative (usage data and bug mail compression) and qualitative (interviews) evaluation demonstrated that this dashboard-like approach to issue tracking can reduce the flood of irrelevant bug email by over 99% and also improve support for individual tasks.

7.1 Contributions

This dissertation has made several main contributions:

- A model of extracting usage data from the log files to measure user adoption.
- A demonstration of how analysis of real-world usage data can provide insight into user dynamic behaviour and adoption trends across various deployment environment.
- A data analysis pattern called *artifact lifecycle model* for synthesizing knowledge from software development artifacts.
- Evidence of the effectiveness of patch lifecycle models in providing actionable insights into the contribution management practice.
- Empirical evidence that organizational and personal factors influence review timeliness and outcome on the projects featuring both competition and collaboration.
- A categorization of the key issue-tracking design aspects to consider when implementing next-generation tools.
- The developer DASH tool that allows developers to explore their daily task items and keep abreast of the changes.
- Evidence that our personalized dashboard-like approach to issue tracking can reduce the flood of irrelevant bug email by over 99%, increase developer's awareness on the issues they are working on and also improve support for their individual tasks.

7.2 Closing Remarks

Our work contributes to the field of software engineering by demonstrating that software analytics should be better adopted and integrated into the development processes as they provide practitioners with powerful means that support more informed data-driven decisions.

Software analytics is not a problem that has been solved in this thesis. A very small portion of it has been explored and we suggest there is a broad field of research ahead of us.

References

[1] William M Adams. The future of sustainability: Re-thinking environment and development in the twenty-first century. Technical report, IUCN Renowned Thinkers Meeting, January 2006.

[2] Steve Adolph, Wendy Hall, and Philippe Kruchten. A methodological leg to stand on: lessons learned using grounded theory to study software development. In *Proceedings of the Annual Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2008, pages 13:166–13:178.

[3] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Journal of Empirical Software Engineering*, 2011, 16.

[4] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Identifying the starting impact set of a maintenance request: a case study. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2000, pages 227–230.

[5] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering (TSE)*, 2002, pages 970–983.

[6] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the International Conference Software Engineering (ICSE)*, 2006, pages 361–370.

[7] Jai Asundi and Rajiv Jayant. Patch review processes in open source software development communities: A comparative case study. In *Proceedings of the Annual Hawaii International Conference on System Sciences (HICSS)*, 2007, pages 166–166.

[8] Atlassian. Bitbucket. https://bitbucket.org/.

[9] Atlassian. Jira. http://www.atlassian.com/software/jira/overview.

[10] Thomas Ball, Jung min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk. In *Proceedings of the International Conference Software Engineering (ICSE)*, 1997.

[11] Olga Baysal, Ian Davis, and Michael W. Godfrey. A Tale of Two Browsers. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2011, pages 238–241.

[12] Olga Baysal and Reid Holmes. A Qualitative Study of Mozilla's Process Management Practices. Technical Report CS-2012-10, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, June 2012. Also available online http://www.cs.uwaterloo.ca/research/tr/2012/CS-2012-10.pdf.

[13] Olga Baysal, Reid Holmes, and Michael W. Godfrey. Mining Usage Data and Development Artifacts. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2012, pages 98–107.

[14] Olga Baysal, Reid Holmes, and Michael W. Godfrey. Developer Dashboards: The Need for Qualitative Analytics. *IEEE Software*, 2013, 30(4):46–52.

[15] Olga Baysal, Reid Holmes, and Michael W. Godfrey. Situational Awareness: Personalizing Issue Tracking Systems. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2013, pages 1185–1188.

[16] Olga Baysal, Reid Holmes, and Michael W. Godfrey. No Issue Left Behind: Reducing Information Overload in Issue Tracking. In *Under review*, 2014.

[17] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. The Secret Life of Patches: A Firefox Case Study. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2012, pages 447–455.

[18] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Extracting artifact lifecycle models from metadata history. In *Proceedings of the ICSE Workshop on Data Analysis Patterns in Software Engineering (DAPSE)*, 2013, pages 17–19.

[19] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. The Influence of Non-technical Factors on Code Review. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2013, pages 122–131.

[20] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, 2010, pages 291–300.

[21] Martin Best. The Bugzilla Anthropology. https://wiki.mozilla.org/Bugzilla_Anthropology.

[22] Nicolas Bettenburg, Bram Adams, Ahmed E. Hassan, and Daniel M. German. Management of community contributions: A case study on the android and linux software ecosystems. http://nicolas-bettenburg.com/?p=399, 2010.

[23] Nicolas Bettenburg and Andrew Begel. Deciphering the story of software development through frequent pattern mining. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2013, pages 1197–1200.

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

[25] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. Fastdash: A visual dashboard for fostering awareness in software teams. In *Proceedings of the Conference Human Factors in Computing Systems (CHI)*, 2007, pages 1313–1322.

[26] Gargi Bougie, Christoph Treude, Daniel M. Germn, and Margaret-Anne D. Storey. A comparative exploration of freebsd bug lifetimes. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2010, pages 106–109.

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

[28] Galen C. Britz. *Improving Performance Through Statistical Thinking*. 2000.

[29] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Crystal: Precise and unobtrusive conflict warnings. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2011, pages 444–447.

[30] Bugzilla. Installation list. http://www.bugzilla.org/installation-list/, November 2012.

[31] Raymond P.L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2012, pages 987–996.

[32] Gerardo Canfora and Luigi Cerulo. How software repositories can help in resolving a new change request. In *Proceedings of the Workshop on Empirical Studies in Reverse Engineering*, 2005.

[33] Gerardo Canfora and Luigi Cerulo. A taxonomy of information retrieval models and tools. *Journal of Computing and Information Technology*, 2007, 12:175–194.

[34] Jeffrey Carver. The use of grounded theory in empirical software engineering. In *Proceedings of th International Conference on Empirical Software Engineering Issues: critical assessment and future directions*, 2007, pages 42–42.

[35] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazzing up eclipse with collaborative tools. In *Proceedings of the International Conference Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003, pages 45–49.

[36] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the Conference Human Factors in Computing Systems (CHI)*, 2007, pages 557–566.

[37] Jacob Cohen. *Applied Multiple Regression - Correlation Analysis for the Behavioral Sciences*. 2003.

[38] Gerry Coleman and Rory O'Connor. Using grounded theory to understand software process improvement: A study of irish software product companies. *Information and Software Technology (IST)*, Jun 2007, 49(6):654–667.

[39] Melvin E. Conway. How Do Committees Invent? *Datamation*, 1968, 14(4):28–31.

[40] R. Cooley, B. Mobasher, and J. Srivastava. Web mining: information and pattern discovery on the world wide web. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*, 1997, pages 558–567.

[41] Juliet Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 1990, 13(1):3–21.

[42] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2003.

[43] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 2009.

[44] Davor Cubranic. Automatic bug triage using text categorization. In *Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 2004, pages 92–97.

[45] Davor Cubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2003, pages 408–418.

[46] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering (TSE)*, June 2005, 31(6):446–465.

[47] Krzysztof Czarnecki, Zeeshan Malik, and Rafael Lotufo. Modelling the "hurried" bug report reading process to summarize bug reports. In *Proceedings of the International Conference Software Maintenance (ICSM)*, 2012, pages 430–439.

[48] Jacek Czerwonka, Nachiappan Nagappan, Wolfram Schulte, and Brendan Murphy. Codemine: Building a software development data analytics platform at microsoft. *IEEE Software*, 2013, 30(4):64–71.

[49] Barthélémy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. Moving into a new software project landscape. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2010, pages 275–284.

[50] Daniela Damian, Luis Izquierdo, Janice Singer, and Irwin Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proceedings of the International Conference on Global Software Engineering (ICGSE)*, 2007, pages 81–90.

[51] Thomas H. Davenport, Jeanne G. Harris, and Robert Morison. *Analytics at Work: Smarter Decisions, Better Results*. Harvard Business School Press. Harvard Business Review Press, 2010.

[52] Andrea De Lucia, Rocco Oliveto, and Paola Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *Proceedings of the International Conference Software Maintenance (ICSM)*, 2006, pages 299–309.

[53] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. Assessing ir-based traceability recovery tools through controlled experiments. *Empirical Software Engineering*, 2009, 14:57–92.

[54] Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2008, pages 241–250.

[55] Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. Code thumbnails: Using spatial memory to navigate source code. In *Proceedings of the International Symposium Visual Languages and Human-Centric Computing (VLHCC)*, 2006, pages 11–18.

[56] Robert DeLine, Mary Czerwinski, and George Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the International Symposium Visual Languages and Human-Centric Computing (VLHCC)*, 2005, pages 241–248.

[57] R Development Core Team. The R Project for Statistical Computing. http://www.r-project.org/. [Online; accessed 10-July-2011].

[58] Chuan Duan and Jane Cleland-Huang. Clustering support for automated tracing. In *Proceedings of the International Conference Automated Software Engineering (ASE)*, 2007, pages 244–253.

[59] Thomas Duebendorfer and Stefan Frei. Why Silent Updates Boost Security. Technical Report 302, TIK, ETH Zurich, May 2009.

[60] Tore Dyba, Vigdis By Kampenes, and Dag I.K. Sjoberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 2006, 48(8):745–755.

[61] David Eaves. Developing community management metrics and tools for mozilla. http://eaves.ca/2011/04/07/developing-community-management-metrics-and-tools-for-mozilla/, April 2011.

[62] Mohammad El-Ramly and Eleni Stroulia. Mining software usage data. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2004, pages 64–68.

[63] Elasticsearch. Elasticsearch. http://www.elasticsearch.com/.

[64] Mica R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human factors*, 1995, 37(1):32–64.

[65] ENTP. Lighthouse. https://lighthouseapp.com/.

[66] J. Alberto Espinosa, Sandra Slaughter, Robert Kraut, and James Herbsleb. Team knowledge and coordination in geographically distributed software development. *Journal of Management Information Systems*, July 2007, 24(1):135–169.

[67] Davide Falessi, Giovanni Cantone, and Gerardo Canfora. A comprehensive characterization of nlp techniques for identifying equivalent requirements. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pages 18:1–18:10.

[68] C. Fernstrom, K.-H. Narfelt, and L. Ohlsson. Software factory principles, architecture, and experiments. *IEEE Software*, March 1992, 9(2):36–44.

[69] Mozilla Foundation. Bugzilla. http://www.bugzilla.org, October 2012.

[70] Deen Freelon. ReCal2: Reliability for 2 coders. http://dfreelon.org/utils/recalfront/recal2/.

[71] Thomas Fritz and Gail C. Murphy. Determining relevancy: how software developers determine relevant information in feeds. In *Proceedings of the Conference Human Factors in Computing Systems (CHI)*, 2011, pages 1827–1830.

[72] Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2004, pages 387–396.

[73] Daniel M. German and Abram Hindle. Measuring fine-grained change in software: Towards modification-aware change metrics. In *Proceedings of the International Software Metrics Symposium (METRICS)*, 2005, pages 28–.

[74] Barney Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, 1967.

[75] Claude Godart, Pascal Molli, Grald Oster, Olivier Perrin, Hala Skaf-Molli, Pradeep Ray, and Fethi Rabhi. The toxicfarm integrated cooperation framework for virtual teams. In *Distributed and Parallel Databases*, 2004, pages 67–88.

[76] Michael W. Godfrey, Ahmed E. Hassan, James Herbsleb, Gail C. Murphy, Martin Robillard, Prem Devanbu, Audris Mockus, Dewayne E. Perry, and David Notkin. Future of mining software archives: A roundtable. *IEEE Software*, Jan 2009, 26(1):67–70.

[77] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, 2004, pages 72–81.

[78] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: Msr for app stores. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2012, pages 108–111.

[79] Ahmed E. Hassan. Mining software repositories to guide software development, 2005.

[80] Ahmed E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance (FoSM)*, Sept 2008, pages 48–57.

[81] Ahmed E. Hassan and Tao Xie. Mining software engineering data. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2010, pages 503–504.

[82] Ahmed E. Hassan and Tao Xie. Software intelligence: Future of mining software engineering data. In *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research (FoSER)*, 2010, pages 161–166.

[83] James D. Herbsleb and Rebecca E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, September 1999, 16(5):63–70.

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

[85] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2007, pages 21–.

[86] Emily Hill. Developing natural language-based program analyses and tools to expedite software maintenance. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2008, pages 1015–1018.

[87] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2007, pages 19–.

[88] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Indentation as a proxy for complexity metric. In *Proceedings of the International Conference Program Comprehension (ICPC)*, 2008, pages 133–142.

[89] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, May 2009, 74(7):414–429.

[90] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, May 2009, 74(7):414–429.

[91] Rashina Hoda, James Noble, and Stuart Marshall. Using grounded theory to study the human aspects of software engineering. In *Human Aspects of Software Engineering*, 2010, pages 5:1–5:2.

[92] Reid Holmes and Robert J. Walker. Customized awareness: Recommending relevant external change events. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2010, pages 465–474.

[93] Hadi Hosseini, Raymond Nguyen, and Michael W. Godfrey. A market-based bug allocation mechanism using predictive bug lifetimes. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pages 149–158.

[94] Kenneth Hullett, Nachiappan Nagappan, Eric Schuh, and John Hopson. Data analytics for game development. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2011, pages 940–943.

[95] GitHub Inc. Github. https://github.com/.

[96] InternetWorldStats.com. Internet usage and population statistics. http://www.internetworldstats.com/stats.htm. [Online; accessed 20-September-2011].

[97] Jiří Janák. Issue tracking systems. Master's thesis, Masaryk University, Brno, The Czech Republic, 2009.

[98] Chyng-Yang Jang, Charles Steinfield, and Ben Pfaff. Virtual team awareness and groupware support: an evaluation of the teamscope system. *International Journal of Human-Computer Studies*, Jan 2002, 56(1):109–126.

[99] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? and how fast? – case study on the linux kernel. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013, pages 101–110.

[100] Sascha Just, Rahul Premraj, and Thomas Zimmermann. Towards the next generation of bug tracking systems. In *Proceedings of the International Symposium Visual Languages and Human-Centric Computing (VLHCC)*, 2008, pages 82–85.

[101] R. Kadia. Issues encountered in building a flexible software development environment: lessons from the arcadia project. In *Proceedings of the ACM SIGSOFT Symposium on Software Development Environments (SDE)*, 1992, pages 169–180.

[102] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*, March 2007, 19(2):77–131.

[103] Peter Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 2008, 28(1):1–9.

[104] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2003.

[105] Paul B. Kantor. Foundations of statistical natural language processing. *Information Retrieval*, 2001, pages 80–81.

[106] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, 2006, pages 1–11.

[107] Oleksii Kononenko, Olga Baysal, Reid Holmes, and Michael W. Godfrey. DASHboards: Enhancing developer situational awareness. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2014.

[108] William H. Kruskal and W. Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 1952, 47(260):583–621.

[109] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, 2001, pages 37–42.

[110] Steve LaValle, Eric Lesser, Rebecca Shockley, Michael S. Hopkins, and Nina Kruschwitz. Big data, analytics and the path from insights to value. *MIT Sloan Management Review*, 2011, 52(2):21–31.

[111] Erich L. Lehmann and H.J.M. D'Abrera. *Nonparametrics: statistical methods based on ranks*. Springer, 2006.

[112] Paul Luo Li, Ryan Kivett, Zhiyuan Zhan, Sung-eok Jeon, Nachiappan Nagappan, Brendan Murphy, and Andrew J. Ko. Characterizing the differences between pre- and post- release versions of software. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2011, pages 716–725.

[113] Ming-Te Lu. Digital divide in developing countries. *Journal of Global Information Technology Management*, 2001, 4(3):1–4.

[114] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2007, 16(4).

[115] Patricia Yancey Martin and Barry A. Turner. Grounded theory and organizational research. *The Journal of Applied Behavioral Science*, 1986, 22(2):141–157.

[116] Jr. Massey, Frank J. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 1951, 46(253):68–78.

[117] Laurie Mcleod, Stephen G. Macdonell, and Bill Doolin. Qualitative research on software development: a longitudinal case study methodology. Aug 2011, 16(4):430–459.

[118] Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, 2001, pages 83–86.

[119] Tim Menzies and Thomas Zimmermann. Software analytics: So what? *IEEE Software*, 2013, 30(4):31–37.

[120] Matthew B. Miles and Michael Huberman. *Qualitative Data Analysis: An Expanded Sourcebook*. Sage Publications, 1994.

[121] Matthew B. Miles and Michael Huberman. *Qualitative Data Analysis: An Expanded Sourcebook*. SAGE Publications, 1994.

[122] Roberto Minelli and Michele Lanza. Samoa – a visual software analytics platform for mobile applications. In *Proceedings of the International Conference Software Maintenance (ICSM)*, 2013, pages 476–479.

[123] Roberto Minelli and Michele Lanza. Software analytics for mobile applications–insights & lessons learned. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2013, pages 144–153.

[124] Bamshad Mobasher, Namit Jain, Eui-Hong (Sam) Han, and Jaideep Srivastava. Web mining: Pattern discovery from world wide web transactions. Technical report, 1996.

[125] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2002, 11(3):309–346.

[126] M.A. Montoni and A.R. Rocha. Applying grounded theory to understand software process improvement implementation. In *Proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2010, pages 25–34.

[127] Mozilla. The Mozilla Development Planning Forum.

[128] Mozilla. Code Review FAQ. https://developer.mozilla.org/en/Code_Review_FAQ, June 2012.

[129] Mozilla. Code-Review Policy. http://www.mozilla.org/hacking/reviewers.html#the-super-reviewers, June 2012.

[130] Mozilla. Firefox Input. http://input.mozilla.org/, June 2012.

[131] MozillaWiki. Modules Firefox. https://wiki.mozilla.org/Modules/Firefox, June 2012.

[132] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, Ankle Sprains, and Keepers of Quality: How Is Video Game Development Different from Software Development? In *Proceedings of the International Conference Software Engineering (ICSE)*, 2014.

[133] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2008, pages 521–530.

[134] Mehrdad Nurolahzade, Seyed Mehdi Nasehi, Shahedul Huq Khandkar, and Shreya Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *Proceedings of the Joint International Workshop on Principles of Software Evolution and Workshop on Software Evolution (IWPSE/EVOL)*, 2009, pages 9–18.

[135] Lucas D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2007, pages 29–.

[136] Massimiliano Di Penta, Sara Gradara, and Giuliano Antoniol. Traceability recovery in rad software systems. In *Proceedings of the International Conference Program Comprehension (ICPC)*, 2002, pages 207–216.

[137] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering (TSE)*, June 2007, 33(6):420–432.

[138] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

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

[140] Peter Rigby and Daniel German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, Canada, January 2006.

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

[142] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2011, pages 541–550.

[143] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, Jan 2007, 166:93–109.

[144] Romain Robbes, Michele Lanza, and Mircea Lungu. An approach to software evolution based on semantic change. In *Proceedings of the International Conference Fundamental Approaches to Software Engineering*, 2007, pages 27–41.

[145] Jarrett Rosenberg. Statistical methods and measurement. In Forrest Shull, Janice Singer, and Dag I. K. Sjberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 155–184. Springer London, 2008.

[146] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2007, pages 499–510.

[147] Anita Sarma, Zahra Noroozi, and Andre van der Hoek. Palantir: Raising awareness among configuration management workspaces. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2003, pages 444–454.

[148] ScienceDaily. Big data, for better or worse: 90% of world's data generated over last two years. www.sciencedaily.com/releases/2013/05/130522085217.htm, May 2013.

[149] Bhuricha Deen Sethanandha, Bart Massey, and William Jones. Managing open source contributions for software project sustainability. In *Proceedings of the Portland International Conference on Management of Engineering & Technology (PICMET)*, 2010, pages 1–9.

[150] VB Singh and Krishna Kumar Chaturvedi. Bug tracking and reliability assessment system (btras). *International Journal of Software Engineering & Its Applications*, 2011, 5(4).

[151] Mark R. Sirkin. *Statistics for the Social Sciences*. Sage Publications, 1995.

[152] Edgewall Software. Trac. http://trac.edgewall.org/.

[153] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: discovery and applications of usage patterns from web data. *ACM SIGKDD Explorations Newsletter*, January 2000, 1:12–23.

[154] StatOwl.com. About our data. http://statowl.com/about_our_data.php, July 2011.

[155] StatOwl.com. Operating systems market share. http://statowl.com/operating_system_market_share.php, July 2011.

[156] StatOwl.com. Web browser market share. http://www.statowl.com/web_browser_market_share.php, August 2011.

[157] Igor Steinmacher, Ana Paula Chaves, and Marco Aurelio Gerosa. Awareness support in global software development: a systematic review based on the 3c collaboration model. In *Proceedings of the International Conference on Collaboration and Technology (CRIWG)*, 2010, pages 185–201.

[158] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, 2006, pages 195–198.

[159] Anselm L. Strauss and Juliet M. Corbin. *Basics of qualitative research: grounded theory procedures and techniques*. Sage Publications, 1990.

[160] Lorenzo Strigini. Limiting the dangers of intuitive decision making. *IEEE Software*, 1996, 13:101–103.

[161] Lin Tan and Tao Xie. Text analytics for software engineering: Applications of natural language processing. In *Proceedings of the European Software Engineering Conference/-Foundations of Software Engineering (ESEC/FSE)*, 2011.

[162] Christoph Treude and Margaret-Anne Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2010, pages 365–374.

[163] Ravi Vadugu. Comparison of opensource bug tracking tools. http://www.toolsjournal.com/item/187-comparision-of-opensource-bug-tracking-tools, 2011.

[164] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2003, pages 408–418.

[165] Gina Venolia. Textual allusions to artifacts in software-related repositories. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2006, pages 151–154.

[166] Geoff Walsham. Interpretive case studies in is research: nature and method. *European Journal of Information Systems*, 1995, (4):7481.

[167] Peter Weissgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2008, pages 67–76.

[168] Weka Machine Learning Project. Weka. http://www.cs.waikato.ac.nz/ml/weka/.

[169] Wikipedia. The Comparison of Web Browsers — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Comparison_of_web_browsers. [Online; accessed 14-Aug-2011].

[170] Wikipedia. Google Chrome — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Google_Chrome. [Online; accessed 28-November-2010].

[171] Wikipedia. Mozilla Firefox — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Mozilla_Firefox. [Online; accessed 28-November-2010].

[172] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011.

[173] WorldAtlas.com. Countries listed by continent. http://www.worldatlas.com/cntycont.htm, July 2011.

[174] Tao Xie. Bibliography on mining software engineering data. https://sites.google.com/site/asergrp/dmse.

[175] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. Data mining for software engineering. *Computer*, Aug 2009, 42(8):55–62.

[176] Suresh Yadla, Jane Huffman Hayes, and Alex Dekhtyar. Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 2005, pages 116–124.

[177] Dongmei Zhang, Yingnong Dang, Jian-Guang Lou, Shi Han, Haidong Zhang, and Tao Xie. Software analytics as a learning case in practice: approaches and experiences. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering (MALETS)*, 2011, pages 55–58.

[178] Dongmei Zhang, Shi Han, Yingnong Dang, Jian-Guang Lou, Haidong Zhang, and Tao Xie. Software analytics in practice. *IEEE Software*, 2013, 30(5):30–37.

[179] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. Improving bug tracking systems. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2009, pages 247–250.

[180] Robert W. Zmud. Management of large software development efforts. *MIS Quarterly*, June 1980, 4(2):45–55.

APPENDICES

Organization of Appendices Appendix A provides supporting materials related to the qualitative study described in Section 5.2. Appendix B presents interview materials that we used during the prototype validation process described in Section 5.3.4.2. And Appendix C includes materials related to the review of our application at the Office of Research Ethics at the University of Waterloo.

Appendix A

Qualitative Study: Card Sort

This chapter includes Section A.1 providing a detailed analysis of the qualitative findings from the card sort (described in Section 5.2) and Section A.2 reporting reliability coefficients achieved among coders during the open coding process.

During the card sort, we applied an open coding technique to classify 1,213 individual comments into naturally emerging themes. We identified 15 themes and 91 sub-themes, containing between two to 41 comments. Later, the 15 themes were grouped into 4 concept categories. These concept categories consist of two to six themes. All of the concept categories were *organic* — that is, they naturally arose from the data; the developers were not answering specific questions that were asked of them. Each concept category relates to a different aspect of how Mozilla developers interact with Bugzilla to report and resolve bugs and to manage the collaborative process.

We provide an overview of the concept categories, as well as their themes and sub-themes. For each theme and sub-theme we provided the number of individual participants commenting on a certain issue and the total number of quotes given. For each sub-theme we developed a synthetic quote that provides a general thought on the topic. Synthetic quotes (SQ) are generated by combining participants' comments into a single statement.

The four high-level concept categories are situational awareness (discussed in Section 5.2.2), supporting tasks (Section 5.2.3), expressiveness (Section 5.2.4), and the rest (Section 5.2.5).

A.1 Concept Categories

A.1.1 Situational Awareness [19p, 208s]

A.1.1.1 Dashboards [18p, 99s]

[16p, 60s] Public Dashboards

- **[12p, 31s] Developer profiles (people as first-class Bugzilla entities)**
  Includes: patch-writer reputation from Code review, trustworthiness from Severity.

  SQ: A means to learn more (e.g., their activity, duration with project, reputation, modules/products, roles) about an individual would improve assignment, triage, and code review assignments.

  * **P15**: "Need some way to figure out who you are so we can treat each other better."
  * **P6**: "Severity can be misused because anyone can set it."
  * **P14**: "It is completely based on the person whose code is being reviewed and his sense of them."
  * **P8**: "People are not first class objects which makes gauging a contributor difficult."

  Idea: P8: "Can't mention a person's name in a comment and have that link to their profile."
  Idea: P15: "Are they a module owner or a peer?"
  Idea: P17: "Security bugs have a different colour background, maybe something similar for first-time patch authors / bugs?"

- **[12p, 22s] Workload transparency**
  Includes: Reviewer load from Code review, Time estimate from Dashboard, Transparency from status.

  SQ: Knowing what others are working on can enable assigning more relevant tasks to people as well as enabling better load balancing.

  * **P11**: "...he can be more helpful if he can better understand what people are working on and how it fits with [their tasks]."
  * **P3**: "[it would be better to] spread the load on key reviewers."
  * **P17**: "[frequently uses the review queue length] to see who might be quickest"
  * **P6**: "Order things by time estimate and priority."

- **[8p, 12s] Communicating interest**
  Includes: Getting people's attention.

  SQ: Being able to indicate interest in a bug without taking ownership is useful.

  * **P8**: "Assign it to himself and then makes a note that others can take over."

  Idea: public watching list (feels similar to CC though).

[15p, 39s] Private dashboards

- **[12p, 21s] Tracking activity**
  SQ: It is hard to determine what has happened since the last time an issue was examined.
  - **P11**: "Gigantic spreadsheet of bugs he is looking at. It would be useful to know how the bugs have changed since he last looked.
  - **P4**: "Sometimes will write reminders in the comments; this is dangerous because he may not go back."
  - **P3**: "They will let it go a few weeks, if there is not traction they will start to follow up."
  - **P6**: "Review status on the dashboard."
- **[10p, 18s] Watch list**
  Includes: Personal priorities.

  SQ: Developers can only track a limited number of bugs in their heads; it would be nice to be able to watch bugs and sort them by activity date.
  - **P8**: "Would like to have a personal list of bugs without claiming them."
  - **P6**: "Wants to get info based on what has changed since last time I looked at it."
  - **P7**: "Private comments for himself."
  - **P15**: "Need a way for people to set their own priorities for bugs."

A.1.1.2 Collaborative Filtering [4p, 8s]

[4p, 8s] Highlighting important comments

SQ: It is difficult to determine what comments are good and important.

- **P14**: "Just finding the right comments among dozens of comments is difficult."
- **P7**: "Hard bugs that turn out to be complicated - 72 comments, and it's important, you have to read all that history. Similarly, bugs could benefit from a way to mark which comments are good."
- **P10**: "The biggest thing that would their flow would be to pull out important comments on STR."

Idea: P7: "Strike bad information"
Idea: P11: "Being able to mark something as important fact in the comment field"
Idea: P11: "Would like to see nested conversations, so that you can collapse threads that are not important".

— —

A.1.1.3 Status [18p, 56s]

[8p, 24s] What is the current status?

Includes: Evolution of a bug
SQ: It is hard to determine the current status of a bug, where the bug is at in the process.

- ∗ **P9**: "Very daunting to pickup a bug that you have no idea where it is in the process."
- ∗ **P5**: "Because there are so many fields, its not clear where the bug is in the process."
- ∗ **P6**: "No way to mark a bug in fine grain status if a bug is waiting on something."
- ∗ **P4**: "We can't figure out what state the bug is in."
- ∗ **P4**: "Sometimes people will see an issue and start working on it not knowing about if a bug exists or not."

Idea: P9: "Triage needs a way to tag a bug to know where it's at."
Idea: P13: "He wants tools that help to understand the evolution of the bug."
Idea: P1: "Bug progress would be good to track."

[11p, 20s] Next action

SQ: Developers want to be able to set/see the state of a bug - is a bug waiting for an approval? are repro steps missing? where the bug stands in the process?

- ∗ **P5**: "Having the ability to add a few next actions would be good."
- ∗ **P17**: "Would be good to have some way of having a next action that pings them [reporters]."
- ∗ **P16**: "We can usefully annotate every bug with a next action field."
- ∗ **P7**: "You look at the bug and you think, who has the ball? What do we do next?"

* **P5**: Wants next action, next action assignee rather than a status.

Idea: P10: "Jesse Ruderman's ideas on next steps strikes a cord on him"
Idea: P17: "Next action field would be very helpful (i.e., on whom the bugs is waiting for next action: reporter, triage, dev, QA, etc.)"
Idea: P3: "Would be good to have interface that allowed you to set up 3 things - next step, review, person"

[6p, 12s] Hand-off

SQ: Handoff of bugs is often forced by following up and giving people nudges.

* **P7**: "Assignment often gets ignored. What really works is giving people little nudges."
* **P15**: "You have to watch a bug in a lifecycle and make sure that the hand off is happening."
* **P15**: "You have to push bugs. There isn't a clean handoff and the bugs will stall."
* **P12**: "They have a bug master. This is an experiment called bug master, the person's main task is to watch bug mail and get more detail, make bug actionable. Nag developers when triage follow up is needed."
* **P7**: "If bugs come up that is important, you work them in. Someone comes and asks you about it, you interrupt your project to go work on that."

Idea: P14: "There has been talk in the past of having some way to flag who the next person is for handling a bug."

A.1.1.4 E-mail [17p, 45s]

SQ: Email plays a fundamental role in how developers keep apprised of what's going on inside Bugzilla.

[8p, 12s] Overwhelming volume

SQ: Email is a primary way of receiving bugs, as well as communicating the whole bug fixing process.

* **P4**: “Sometimes he will just skip a day because he can’t keep up with it”.
* **P11**: “Way too much information flow coming at him. Bugzilla folder has 8000+ unread and only can look at it a couple of times per month”.
* **P17**: “The person that reports the bug gets hammered with spam with no way to stop. This causes people to avoid filing the Orange bugs as they knowing it’s going to spam them!”
* **P17**: “Email flow is huge, CC changes cause email...”

Idea: P14: “He would like to be able to watch only incoming bugs coming into a component”.
Idea: P17: “It would be handy to have people that edit a bug to say that this isn’t worth sending out to everyone.”

[11p, 16s] Email is important

* **P16**: “Primary way to see new bugs is bug mail. Workflow is very bug mail focused.”

[2p, 2s] Email is not important

* **P11**: “People contact him through back channel like IRC, send a personal email or talk to him.”

[6p, 9s] Filtering

SQ: Some mail clients (e.g., Gmail) do not have good filtering functions, as a result it’s hard to distinguish important emails due to overwhelming number of emails received daily.

* **P17**: “Bugzilla doesn’t let you control the flow enough, 5000 email in a month and most of it doesn’t relate to his work”.
* **P17**: “Not being able to say that you don’t want email from a specific bug is really annoying, especially if you report something on an Automated test”.
* **P6**: “Now uses Gmail that lets you go through bugs quicker but can’t filter by headers in the emails. He has to remember too much information about why he was sent the information as a result”.
* **P14**: “He would like to see a way to stop receiving email from a bug he filed, no way to do that”.

Idea: P6: “Get a better mail client. No solution that works with Gmail”.
Idea: P15: “He uses the xheader filter, it tells you why it was sent to you”.

[4p, 6s] Formatting

* **P17**: "Email formatting makes it very hard to read when a long text field has changed, Gmail doesn't use fixed width font so table distorted. "

A.1.2 Supporting Tasks [20p, 700s]

A.1.2.1 Code Review [20p, 130s]

[6p, 10s] Recommending reviewers

SQ: Being able to get a list of recommended reviewers along with their review loads would make it easier to request approval on another component.

* **P15**: "You can always type in a name but a list will help new people."
* **P17**: "Has been asked to do reviews in the past even though he is not a peer."

Idea: P17: Populate reviewer list with info from the wiki Modules page.
Idea: P8: "Potentially a default queue or fake person for a component where reviewers can come and take reviews from".

[10p, 27s] Patches

SQ: Improving the way Bugzilla handles patches would ease and improve the code review process.

* **P13**: "You could find out if the patch will land cleanly, GitHub does this".
* **P14**: "interdif is unreliable so you can't trust it".

Idea: P4: "Would like to compare the two versions of the patch".
Idea: P14: "Would be good to see which comments are attached to what version of a patch".
Idea: P14/P16: "[Supporting] series of patches would be very good".

[8p, 12s] Importance of review timeliness

SQ: Developers rely on code reviews turning around rapidly.

* **P7**: "Pretty good norm that reviews have high priority".
* **P3**: "He will by the end of the day go through his [email for review requests]".
* **P1**: "Review queue is a top priority, keeps it at zero".

Idea: P17: "Review ETA would be good".

[12p, 32s] States

SQ: Due to possible social frictions, review approval states are used inconsistently/wrong and thus increasing risks and vulnerabilities.

* **P7**: "r+ with nits is common. Fix is usually trivial, or simple, full confidence that people can fix it. This could be a mistake."
* **P8**: "Even with regular contributors, there is a negative social aspect to r-".
* **P2**: "Believes that r+ with nits is a huge risk."
* **P16**: "He will use r-. Different people have different habits. People bring up that r- can be a very negative experience, a rejection of all the work you have done."

[8p, 11s] Process & community

SQ: Review process is sensitive due to its nature of dealing with people's egos. People take criticism better if they know a reviewer who is giving the feedback. Differentiating between feedback vs. review can help with this.

* **P2**: "There is no accountability, reviewers says things are addressed, there is no guarantee that the person fixed the changes or saw the recommendations."
* **P11**: "He thinks that the culture around reviews is a bit unhealthy, how do we get that down from few days to few hours... Understanding the human element."
* **P7**: "Someone on the team said that because they know a review is coming, they tighten up their code."

Idea: P20: "He does like a fact that we could make a difference between feedback and review. "

[5p, 8s] Risk / Reward

SQ: Patch approval requires some estimation on the risks involved and how much time should be spend for a review.

* **P2**: "Patch approval needs risk and severity explanation."
* **P16**: "They want a justification. What's the motivation, Risk vs. Reward."
* **P7**: "he tries to make an estimation on the risk when deciding on how much time to spend."

Idea: P3: "Risk analysis should be in the bug when put up for approval. Bugzilla should enforce that if possible, at least suggest it."

[17p, 30s] Misc

SQ: Issues related to the code review process including approval queues, lack of a good system, review response time, etc.

* **P10**: "The review system doesn't seem to be tightly integrated into Bugzilla."
* **P13**: "You could have the test results so that if any fail, the review is automatically rejected."

A.1.2.2 Triage & Sorting [20p, 259s]

[15p, 35s] Sorting/filtering

SQ: Bugzilla does not provide good sorting and filtering mechanisms and thus these tasks require lot of effort.

* **P13**: "They have a custom tool that doesn't have good sorting capabilities."
* **P9**: "The lack of ability to filter mixed with the volume makes it an overwhelming task. A lot of rework in sorting bugs rather than actually triaging it and moving it along."

Idea: P5: "if users could sort based on tag values, Bugzilla admins wouldn't have to add fields frequently, and Bugzilla's interface wouldn't become cluttered with rarely-used fields."
Idea: P18: "In triage they use a bucket approach, they get through 10 bugs in half an hour. They want to be fair about what bugs they look at. Would be nice to have some form of random sort."

[12p, 25s] Bug assignment:

- **[7p, 10s] Who gets the bug?**

  SQ: Developers usually pick bugs up as they see them. If they do not feel competent, they will assign a bug to someone with the right expertise to fix it. Some bugs are assigned to the community members who are interested.

  * **P16**: "When you connect a bug to the right person who is familiar with it, it gets fixed quickly."
  * **P6**: "If there is a critical bug he will either do it or assign it to someone better suited."
  * **P17**: "Will set assignee to the patch author."

- **[6p, 10s] Self Assignment**

  SQ: Self assignment is pretty common and used as an indicator to others that someone is working on it. Developer who files a bug is often the one who writes a patch for this bug.

  * **P6**: "Sometimes he will create a bug and assign it to himself right away to signal to others he will take care of it."
  * **P4**: "When someone starts working on a patch, they often will assign it to themselves."
  * **P1**: "If you have enough drive to file a bug, you likely have the drive to own it."

- **[4p, 5s] Unassigned**

  SQ: It can be hard to figure out who is working on a bug since some bugs do not have clear owners because developers do not set Assignee field.

  * **P15**: "Many people don't assign a bug to themselves. When you attach a patch, it should get assigned to you."
  * **P6**: "Some people never set the assign field. A fixed bug with no Assignee make life hard."
  * **P9**: "Assigned could help there but there are a lot of bugs that go from new to fixed without ever being assigned."
  * **P9**: "Sometimes they forget the assigned field and the bug gets marked fixed without ever having an owner. This last one is pretty common."

  Idea: P15: "It would be nice if the system told that you have to attach a patch to this or review. Bug hygiene is something we should do."

[17p, 41s] Components/products

SQ: Figuring out what product/component a bug should go is very confusing since Mozilla has so many of them listed and there is no good documentation available. Updating product/component field is also hard.

* **P8**: "If he doesn't know the product, it's a pain point to find. A lot of them are overlapping and vague."
* **P6**: "It's not clear why certain bugs go under certain areas."
* **P11**: "The component and the product categories are extremely confusing."
* **P15**: "When you want to move from one product to another, it doesn't let you change the component."
* **P17**: "Sometimes it's very hard to figure out in which product/component it should go."

Idea: P17: "The UI doesn't update when you change a product, component should refresh."

[3p, 4s] Component owners

SQ: Having component owners might be useful since their operational knowledge can speed up the triage process.

* **P2**: "Many components don't seem to have clear owners or the buckets are too big to work through."

Idea: P16: "ideally he would like to have explicitly a group of developers watching components and having ownership of those."
Idea: P15: "He thinks that we need to find people that care about certain area. if you split them up, you get a benefit from being familiar with what's coming in and that speeds things up."

[4p, 7s] Last touched

SQ: Ordering bugs by last touched date allows developers to monitor recent changes.

* **P10**: "When he was more involved to triaging Firefox General, he used Last Touch Date for figuring out when the last time a bug had changed."
* **P9**: "Currently the most effective way to triage a bug, look at the most recently touched bugs."

[10p, 16s] Is this bug actionable?

SQ: Developers would like to have mechanisms that trigger an action from the end-users to provide missing information on a bug, e.g., a test case.

* **P7**: "Decent number of bugs that are not actionable. Crash bug that doesn't have a repro case but is common. Not sure what to do with that as it's not actionable."
* **P6**: "Rather than try to make a repro case, he will request one and move one. Often this can lead to bugs getting lost."
* **P17**: "the frustrating bugs are the bugs that don't have enough information and you know that it's very likely that a bug will not be fixable and it's a losing battle."

Idea: P14: "What would be good for triage, would be good to have a way to show what is missing."
Idea: P17: "If the bug is not actionable and there is no follow up, they just need to be closed out."
Idea: P12: "Ubuntu, they have a life cycle, they automatically delete bugs after a certain amount of time."

[4p, 5s] Volume of bugs to triage

SQ: Triage is very difficult due to the overwhelming volume of bugs.

* **P1**: "Feels that he is not aware of all the bugs he should be due to lack of visibility. Currently has about 2500 bugs, no one is aware of the full range of issues in that list due to volume."
* **P16**: "As a general rule, they don't look at the bug list because its just too big."
* **P18**: "They have 1000 open bugs. They have difficulty dealing with that much volume."

[4p, 7s] Duplicates

SQ: Dups are created to track old bugs but the cloning process is harder than needed. Filtering dups needs to be automated.

* **P15**: "He will sort by the resolved and look for dups. Lots of dups show that the bug is out there."

* **P2**: "The cloning process is poor, it's not lightweight and requires you to re-define components as well as making it difficult to clear out information that is not required in the new duplicate."
* **P10**: "Dups were especially useful for looping back to old bugs."

Idea: P15: "It would be awesome if you could compare bugs and come up with a list of potential bugs. This would speed up bug dup clearing."

[6p, 9s] Bugs in "General"

SQ: Bugs filed to Firefox General are likely to be missed since no one owns this component.

* **P9**: "There are also bug that are legitimately general, so it makes it difficult."
* **P1**: "Firefox General - bugs filed under this heading will often go under the radar."
* **P9**: "General is a treasure trove and a graveyard at the same time."

[3p, 4s] Bug kill days

SQ: Bug kill days is a new practice to manage overwhelming number of open bugs.

* **P1**: "Currently working to get this number under control via bug kill days. During this process, they did uncover bugs that they should have been aware of, included patch that had sat around for 5 years but went unnoticed."
* **P9**: "Often bug kill days are mainly about closing bugs."

[5p, 7s] Triage & community engagement

SQ: Engaging the community can help to speed up the triage process by filtering the bugs, determining duplicates, prioritizing bugs, etc.

* **P13**: "People on the mailing list tend to find things pretty quickly. It's pretty easy to get people to test Firefox Aurora and Beta. Its harder for Jetpack, driving engagement is an ongoing issue."
* **P7**: "There could be filtering on some of the bugs by non-engineers. Not sure what would happen, it would take some learning and training."
* **P15**: "It [triage] can be easily distributed. They should find ways to engage volunteers. Incremental approach better than letting a bug log build."
* **P16**: "It would help to have people whose job is just to organize triage efforts."

[2p, 2s] Midair collision

SQ: Miscommunication on who is working on what bug can lead to a midair collision.

* **P10**: "Bugzilla does have a problem with multiple people doing something at the same time, mid air collision. It will happen 1-2 times in a triage session of 100 bugs.Usually this is caused by someone not being clear on who would make the update."
* **P6**: "Midair collisions - someone submits before you do. It's too sensitive."

Idea: P6: "Etherpad has solved it [midair collisions], so it can't be that hard to Bugzilla."

[5p, 5s] Triage meetings

SQ: Triage meetings happen on a weekly basis for most teams. People attend them to discuss issues needing attention.

* **P10**: "Each week they go through all the new bugs."
* **P11**: "He will go to the operational level meetings where e.g. triage is done to talk about it."
* **P8**: "He will attend triage if there are particularly important issues he is working on that are seeking approval."

[2p, 4s] Triage of other components

SQ: Triaging other components is hard since each component has its own policies on the triage process.

* **P9**: "It's hard to triage other groups. Different components have different rules that the developers want triagers to follow."

Idea: P9: "What could help is a guide per component on who they want it done. If that was publicly available, this could help."

[10p, 18s] Triage process

SQ: Current triage process is not manageable as it's very time consuming, yet developers have heavy workloads.

* **P7**: "Spends about a couple hours a week, needs more time than that, but too many other things to do. He does want to change the bug triage process so we can reduce the workload on the engineers."
* **P6**: "This largely causes to be no systematic triage. Doing triage in a systematic way won't get him anything unless it saves time in future."

Idea: P17: "Much of the work in triage could be better if we just asked the right questions up front (e.g., have you tried safe mode, what is the crash ID, etc.)"

[4p, 4s] Comments

SQ: Any activity on a bug is communicated through comments. However, it can be quite time consuming to read through a big list of comments.

* **P8**: "Comments, they work pretty well. There is a strong culture, people have adapted to make comments work well."
* **P4**: "Reading through 30 comments is time consuming."

Idea: Refer to Collaborative Filtering.

[15p, 39s] Misc

A.1.2.3 Reporting [20p, 145s]

This bucket concentrated on challenges surrounding reporting. Triage-related issues are separated into their own bucket (Triage); while some metadata is here if it is really a reporting issue, a whole separate metadata bucket exists as well. *Includes*: Intimidating for new users from Bugzilla.

[17p, 33s] Submission is harder/more confusing than needed

SQ: Submission process can be hard due to the overwhelming number of fields, non-user-friendly form layout, unknown component/product structure, targeting two groups of users - end-users and experienced developers.

* **P6**: "He thinks it's too much, only should have the summary field and description, possibility to add attachments. Anything else is too much for newcomers."

* **P5**: “Needs to be way faster. Entering a big could be fewer steps.”
* **P17**: “He often feels that you never know if a field is important or not. It wastes a lot of time for people to fill in a lot of garbage data when you don’t know if anyone is using it.”

Idea: P12: “Redesigning the usability, placement and layout would help, not removing things.”
Idea: P4: “What shows up on the form biases people to do or not to do certain things. Teaching people what a bug system is, and what information is needed.”
Idea: P13: “How the eye is drawn to the more important fields could be improved by changing the layout.”
Idea: P4: “It might be better for people to choose between more complex and simple version, getting people to file better bugs.”

[10p, 14s] Improving reporting

SQ: Filing bugs can be improved by informing users on how to get their crash data, providing repro steps, etc.

* **P5**: “is attaching a test case and a stack trace, the process of doing one at a time is frustrating. Would be better to do both things at once.”
* **P14**: “Would be nice to get a stand-alone website that shows the bug.”
* **P6**: “The things that really bugs him about filing a bug - go through product, then go through component. Doesn’t always know where to put it.”

[4p, 9s] Defects in the reporting experience

SQ: User reporting experience can be improved by fixing issues with bug submission form.

* **P5**: “If the CC is about to be submitted in error, it should tell you that right away, not after you submit it.”
* **P15**: “Why is the crash data not filling in the bug? This should be a tool suite.”
* **P10**: “The back button duplicates would be nice to fix. People can create multiple bugs by hitting the back button, you often see this as a chain of the same bug submitted several times in a row.”
* **P15**: “URL field. Minor thing but frustrating, it often will add http:// and it will double it if you paste an URL into it. When you select the field it should highlight the HTTP.”

[10p, 23s] Metadata

SQ: Some metadata fields are more important than others, some are irrelevant. Removing redundant and useless fields can help to improve reporting.

* **P17**: "OS is being set wrong could cause problems for people verifying a bug."
* **P13**: "Some things that cause friction. As a developer he knows that platform should be all."
* **P4**: "Sometimes there are a lot of fields that don't seem needed but every now and then will be needed, like platform, often changes them from a specific platform to all platforms."

Idea: P13: "In past version he has seen bug submission tools that will import info about the environment that you are working on when you report a bug. It would be nice to have this kind of Add bug command."
Idea: P17: "It would be really good to get people together to figure out which fields are necessary to them and why."

[9p, 14s] The role of the description and summary fields

SQ: The summary and description are most important fields and thus should be placed on top of the page. The summary field needs to be space limited since developers would like to see a good short summary.

* **P4**: "People will write very general summaries that lead other people that have different issues to assume it's related and start posting conflicting information. Often too broad - Firefox is too slow, Firefox doesn't work right, Firefox uses too much memory, Firefox stopped responding, Firefox crashed."
* **P10**: "There is a big blank box problem with the summary. Some people will use 2000 words into the summary."
* **P7**: "Gets annoyed when he has to go through bugs that are random thoughts."
* **P15**: "Summarizing the bug first seems backwards, so he puts the description field on top of the summary. "

Idea: P11: "Being able to modify the original description is important."
Idea: P13: "He would like to rearrange the fields so that the summary and the description are higher on page. "

[12p, 15s] Possible enhancements to improve reporting

SQ: Developers provide a number of suggestions on how to improve Bugzilla.

* **P9**: "Ideal would be to have friendly UI for end users, very basic version. Harvest bugs out of that system and put them into Bugzilla."
* **P16**: "We could definitely do a better job at filtering incoming reports before they hit the Bugzilla database."
* **P7**: "He has a template that helps fill in information for js bugs. It fills in the component and a few other fields automatically."
* **P1**: "It would be useful to have a primary assignee and a list of secondary assignees."
* **P2**: "CC is so overloaded it doesn't tell you why you are there. Could mean you wrote the bug, you edited the bug, you wants you to be aware of it, someone commented, someone voting by cc on a bug, needs to be teased out a bit more. "
* **P15**: "Auto-component is useful."

[2p, 3s] External tools

SQ: Bugzilla should not be used as a user feedback tool.

* **P9**: "In terms of end user submission, Bugzilla is not the right tool."

Idea: P2: "input.mozilla.org better for feedback, keeping clutter out of Bugzilla."
Idea: P9: "Input is a good way to help with this."

[7p, 8s] Intimidating for new users

SQ: Bugzilla is quite intimidating to new users. End users often treat Bugzilla as a feedback submission tool.

* **P4**: "A lot of people see it as a complaint form rather than an issue-tracking system."
* **P19**: "New people don't know where to file bugs, it should be easier to figure that out."
* **P17**: "He has so many hacks around issues that he forgets how it would be for a new user."

Idea: P9: "Bugzilla, in the long run, should stop being the end user bug tool."
Idea: P15: "We need to make it pretty, we should include Java script."

[8p, 26s] Misc

SQ: Developers are talking about what and how many bugs they receive/file, or why and when they file certain bugs.

* **P7**: "Sometimes people file bugs to talk about ideas. Some optimization we could do to a feature or a similar suggestion."
* **P7**: "Sometimes people will send an email rather than a bug."
* **P4**: "He uses the submit process in a few different ways. Sometimes filing things that he heard from developers in other context, web developers don't always want to deal with the bug system, may be a comment from a mailing list. "

A.1.2.4 Search [14p, 53s]

[4p, 7s] Quick search

SQ: Quick search is not being used much. It is used for searching through whiteboards, keywords or flags.

* **P6**: "There is a quick search, this helps you with the complexity."
* **P5**: "Quick search, needs a lot of info to get good results."

[5p, 5s] Advanced search

SQ: Advanced search is used by most developers as it's more flexible on what you can search by.

* **P5**: "Advanced search, which is what most developers use."
* **P10**: "He uses the advanced search almost exclusively."
* **P16**: "Some people like quick search, he likes the advanced version."

[5p, 8s] Saved/shared

SQ: Developers want to be able to save and share their search lists, as well as the sort order with others.

* **P10**: "Would be nice to be able to push it to them and have them accept the lists, specifically shared saved searches."
* **P13**: "He would like to have saved searches, he can't save the sort order."
* **P18**: "Provide a link that you can share the same random sorted list. "

[7p, 10s] Hard/confusing

SQ: It's hard and confusing to query Bugzilla.

* **P19**: "When you look at bug lists all day and have to do searches in Bugzilla, it's a pain."
* **P9**: "Running searches on Bugzilla s kind of scary sometimes."
* **P6**: "Querying in Bugzilla is hard. He has to spend a few minutes to figure out how to do the query."

Idea: P19: "It should be really easy to search in Bugzilla, is there a way to do Goggle version of a Bugzilla search?"

[3p, 3s] Date range

SQ: Date range is used to track changes on bugs and to improve search performance.

* **P10**: "He uses the time frame to narrow the scope of search results."
* **P15**: "He will put -7d in the search field and only include bugs that have changed in the last week."

[3p, 5s] Performance

SQ: Search is very slow in Bugzilla.

* **P6**: "Bugzilla is too slow, this is wasting a lot of time, very frustrating."
* **P15**: "If a bug list is too long in the URL, then search fails. It has a problem with a volume in performance. "

Idea: P10: "Restricting the range shortens the query time".
Idea: P15: "We should let Google index the whole thing."

[2p, 2s] Product

SQ: Bugzilla doesn't let you search against several products/components.

* **P10**: "He finds that he wants to search against multiple products. When last he checked, Quick Search couldn't handle that."

[3p, 5s] Dups

SQ: Search is often used to detect duplicates.

* **P15**: "Simple search is good for clustering duplicates. He will limit search to two or three duplicates."
* **P10**: "Summary search is good to avoid duplicates."
* **P5**: "Search to make sure we don't make dups."

[6p, 8s] Misc

SQ: Search is limited by what fields you can search by.

* **P5**: "Can't reliably find bugs to help with."
* **P9**: "Should be able to search by point in triage and keyword."
* **P6**: "No good way to query certain information."
* **P15**: "A whole other category of search to find groups of bugs".

– –

A.1.2.5 Testing and Regression [14p, 37s]

[5p, 9s] Regression

SQ: Knowing whether and where the bug has regressed is important for tracking bug fixes. If something is a regression of their patch, they will fix it or find the person who broke it.

* **P5**: "If it's a regression will CC the person that broke it."
* **P8**: "Always trying to keep an eye if this bug regressed on a release train. Figuring out where the bug regressed. Making sure that fixes have tracking."

* **P17**: "MozRegression does bisection of nightly downloads. You input a date range and it downloads the nightly builds, it gives you a change set range that you can drop into the bug."

Idea: P7: "If Bugzilla knew it was a regression, then it could help you find where it was. There is just enough data to figure it out, but very manual process in figuring it out."

[8p, 15s] Testing and its reliability

SQ: Complete testing is infeasible. Developers rely heavily on automated testing, more experts are needed to write manual tests.

* **P8**: "We don't have the infrastructure to reliably test things."
* **P15**: "It has to pass the checking test, Thinderbox pass."
* **P12**: "They do a mix of Test Driven Development and writing the test after the fact."
* **P13**: "They have a big test suite that they run through everything. Most of it is automated, they had a hot fix last week that could have been avoided if there was more manual testing."

Idea: P14: "There is very little attempt to broaden the tests. You can't do this without understanding the code well. The way to fix this is to have someone that knows something about what should be going on and can do some exploratory testing."

[6p, 9s] QA & Validity

SQ: Bug verification process is inconsistent. QA involvement varies from team to team. Some QAs do, others do not double check every bug.

* **P15**: "Verification should be eliminated unless it has a clear visibility."
* **P17**: "Most of the bug he has worked on don't need QA verification, since code cleanup/dead-code removal/build system work."
* **P8**: "They will mark a bug as verified once they have checked it. Not every QA team will do that but it's very useful."

Idea: P14: "QA metrics should not be the number of bugs verified, bugs found might be good."

[2p, 4s] Fuzzing

SQ: Fuzz testing bugs are typically fixed right away.

* **P7**: "Fuzz bugs come in, that get fixed right away, sometimes they hang around and there is a risk there that it could be something important."
* **P5**: "Fuzzers are a form of randomized testing."
* **P7**: "Fuzz bugs are important, the source of the bug tells you a lot."

– –

A.1.2.6 Tasks [13p, 76s]

[4p, 7s] Role-specific views

SQ: Role-specific views (e.g., triage, developer, review) would make it easier to accomplish common tasks for that view.

* **P17**: "Different Bugzilla view modes for different use cases".
* **P10**: Default/New user, QA/Triage, Developer, Review.

[10p, 27s] Release management

SQ: While release schedule is known ahead of time, release drivers are challenged by the uncertainty of what bug fixes are going into next release.

* **P11**: "Bugzilla is terrible for release management... Release manager - it's a burnout job, no one could have it for more than a few months."
* **P12**: "They are on a train cycle: Release branch, stabilization branch, development branch, same 6 week cycle."
* **P11**: "What actually in a release is very hard to comply in Bugzilla."
* **P18**: "Their code development is not directly tied into the train model."

Idea: P13: "It would be nice if there was a large scale version of what fixes are going into the upcoming release."
Idea: P18: "They have a wiki page for every release where they keep notes and will add links to it to maintain history."

[10p, 22s] Where a bug lands?

SQ: It's hard to figure out what version a bug lands. Setting up target milestone can be confusing, error-prone.

* **P13**: "We currently see when it was supposed to land. By looking at it a year later, you have to figure out if that's actually where it landed."
* **P2**: "Doesn't tell you if it landed on Beta or Aurora."
* **P2**: "There is no way to be sure that the version on the bug is correct. A lot of room for failure, no checks and balances."
* **P8**: "Bugzilla only has one resolution which is closed. We need to know what version it is fixed on. Resolved, fixed on this version. "
* **P17**: "Target milestone is set as the landing field rather than the target which is a bit confusing to newcomers."
* **P19**: "There might be uncertainty as to whether to land a fix or not for a branch, and it is because of the differences in opinion about the risk involved."

Idea: P17: "Why not just call it "Fixed for: ""
Idea: P17: "Security team not using target milestone and instead using the status-firefoxN:fixed flags."
Idea: P18: "They are working on a project that will help land patches from Bugzilla [Build:Autoland]"

[8p, 12s] Statistics

SQ: Being able to flexibly create high-level reports (e.g., lists of custom statistics) could create views that must be manually created through searches.

* **P5**: "Statistics would be good; how are things progressing on a larger scale?"
* **P11**: "JS is a good team... they want a graph that is going down."
* **P18**: "It would be interesting to know what the churn rate is on a daily and weekly basis."
* **P19**: "It would be interesting to know the ratio of bugs marked as fixed vs. open."

[6p, 8s] Workflow

* **P11**: "Big flaw is that workflow is not covered by on an individual level."
* **P14**: "His personal workflow is that he writes a patch and then exports it."

A.1.3 Expressiveness [19p, 188s]

A.1.3.1 Metadata [20p, 132s]

Includes: Status flags from Status

[14p, 38s] Tracking flags

SQ: Tracking flags are used to raise awareness for a specific bug.

* **P10**: "Flagging is used to raise bugs to Dev and PM teams."
* **P3**: "The flag indicates to engineering that it needs to be looked at."
* **P2**: "Tracking is meant to make sure the bugs don't get lost."
* **P15**: "The tracking bug is a way to make sure it will be looked at several times a week."

[17p, 36s] Whiteboard

SQ: Whiteboards are used a lot during the process management including tracking status, seeking approval, highlighting important information on a bug, marking the version number, or setting a new keyword.

* **P3**: " Use the whiteboards for next step and who will do it, tracking problem areas."
* **P3**: "Using whiteboard when using approval or review."
* **P4**: "Highlight comments that are important to read before commenting," "free-form tagging".
* **P5**: "Whiteboard to order bugs impact."
* **P12**: "They now use whiteboard for triage follow up."
* **P1**: "Using whiteboard flags to manage the process."
* **P18**: "Public shaming is the key way to maintain the whiteboard flags."

Idea: They are free-form and thus hard to search on.

[14p, 22s] Keywords

SQ: Keywords add additional workflow information.

* **P15/P1**: "Keywords as calls for action."
* **P4**: "Keywords are sort of like a tagging system but more restricted."
* **P11**: " Keywords is a substitute for workflow."
* **P2**: "Component owners should be able to set up keywords specific to their area."

Idea: Lightweight keyword system; Christian's keyword system.

[3p, 6s] Meta bugs

SQ: Meta bugs are used for tracking related issues and dependencies among bugs.

* **P5**: "To find bugs again, he uses meta bugs. Has filed 59 meta bugs. Finds them useful for grouping."
* **P1**: "Meta bugs - a master bug per feature, no work goes on in this bug, it tracks all other bug activities going on via dependencies."

Idea: P18: "It will auto create a meta bug for you if it's missing and do a lot of connections for you."

[8p, 12s] Metadata (general)

SQ: There are too many fields to be filled in when reporting a bug, dealing with meta data can be quite time consuming and intimidating.

* **P6**: "The amount of info we need from users is too much."
* **P4**: "Sometimes what scares them off is that there are all sorts of fields they didn't know what to do with."
* **P5**: "Platform means too many things."

Idea: P4: "We need better metadata, but we don't want people to have to enter meta data because they won't do it."
Idea: P17: "Do we even need some of the data he is updating, we should not have to do some of this work by hand. A lot of meta data work for the sake of it."

[3p, 4s] Tagging (general)

SQ: Tagging can eliminate other metadata fields like OS, platform.

* **P5**: "Tagging could get rid of a lot of fields."
* **P5**: "Tagging as a cross b/w the keyword field and whiteboard field."

Idea: A good tagging system.

[9p, 14s] Status flags

SQ: Status flags are misleading and not well understood by developers.

* **P15**: "He doesn't know what "New" in Bugzilla means."
* **P4**: "What does verified state means?"
* **P1**: "Status is only reliably to say fixed or not."

Idea: P4: "It might be better to have a better way to convey clarity of the bug. Unconfirmed, new, assigned are limited. A few things we want to convey: we don't know what you are talking about; we get what you mean, can't repro; we get what you mean, it's not sure it's a bug; we get what you mean, but we are quite enough detail; and many more."

A.1.3.2 Severity/Prioritization [19p, 56s]

[6p,8s] Unclear definition of priority/severity

SQ: Priority and severity are not well defined.

* **P11**: "Everyone doesn't use the priority levels consistently."
* **P8**: "Priority categories are not well defined... No clear definitions."
* **P5**: "Priority and Severity are too vague to be useful."

Idea: P4: "Needs a 10 word phrase rather than a number 1 to 6."

[13p, 14s] Priority

SQ: Teams can set up their own priority fields. Priority levels are not used consistently.

* **P19**: "Priority is not used at all. People might add P1 if something is really burning."
* **P1**: "Security bugs are always top priority."
* **P20**: "Generally P1 is a blocker, P2 is pretty important. P3 is important, P5 is when you get to it."
* **P12**: "P1 is need, P2 is we want, P3 is nice to have."
* **P8**: "They are using priority field to help organize the large volume."

Idea: P3: "Use tracking for priority. Tracking +".

[11p, 16s] Severity

SQ: Severity s rarely used, mostly to highlight common crash bugs.

* **P5**: "Severity is not used, just something people argue about."
* **P8**: "Severity if rarely used, a few bugs he will set to crash and/or blocker."
* **P6**: "It's usually very evident from the bug if it's severe. "

Idea: P3: "Severity is not used. They used the tracking system. It's either + - or cleared."

[8p, 13s] Prioritization

SQ: Individuals, as well as teams prioritize bugs based on their importance.

* **P1**: "Priorities are heavily influenced by quarterly goals."
* **P5**: "You don't want to prioritize en mass."
* **P5**: "Priority is useful for prioritizing an aspect. Aspect of quality, security, MemShrink, stability."
* **P1**: "Often bugs are ordered on a wiki page during team meetings. Team decides priority as a group."
* **P6**: "He would go after the bigger issues first."

[5p, 5s] Misc

SQ: There is a sense that using priority and severity can be beneficial if there was a well defined system.

* **P5**: "Priority should only come in on large projects."
* **P6**: "He would like to see a priority system, but he doesn't have a good way to do that now."

A.1.4 The Rest [20p, 117s]

A.1.4.1 Version Control [8p, 43s]

Issues related to interacting with version control systems, branching, merging, etc.

[3p, 12s] Reviewing via Github

SQ: Some code reviewing is done via GitHub not Bugzilla.

* **P13**: "GitHub lets you do line by line comments. This means you can have a discussion around issues right in the code on each issue very fast. This leads to better fixes, by the time you are done with the review process on GitHub, you likely to have better code."
* **P13**: "They use GitHub more these days. Github has more systems to fill in comments on code."
* **P12**: "Conversations happen in Bugzilla and then in Github once the development starts, so you lose part of the conversation."
* **P12**: "It's hard to use Github mixed with Bugzilla. Sometimes there is development that isn't filed in Bugzilla."

Idea: P13: "It would be good if you could point at specific bits of codes, revisions, delta."
Idea: P12: "He thinks that there is a desire to integrate Bugzilla and Github."
Idea: P12: "The Bugzilla Github add-on is a very useful hack that helps you reference between the two. "

[6p, 13s] Checkins

SQ: Bugzilla doesn't have a mechanism of checking in code.

* **P2**: "No version control on patches."
* **P17**: "If they changed the commit message header after r+ and then re-upload, the process of checking it is manual, the interdiff is really bad, it lies."
* **P15**: "Check in from Bugzilla possible? It could submit the bug number and summary to the HG. HG logs have a lot of typos."

Idea: P13: "Github allows you to get a link to the actual revisions and see how the bug was actually fixed."

[4p, 6s] Branching

SQ: A set of patches normally forms a new branch.

* **P16**: "They will push a branch to Bugzilla as a set of patches."
* **P13**: "When they release a new version, they move trunk to a stabilization branch."

[2p, 4s] Merging

SQ: It's hard to do merging since it requires update of many metadata fields.

* **P17**: "When doing merging, it's a huge job to update all the bug fields. He would merge 20 to 100 changesets per merge, typically 100+ a day."
* **P8**: "He does merges, it's hard to do, because there is no bulk way to change many bugs at once. "

Idea: P17: "Getting this tool [BZexport, BZimport] fixed would be a very good step, since saves having to set assignee on ma bugs after merging inbound. "

[4p, 8s] Integration of Bugzilla with Hg

SQ: There is no direct linkage between Bugzilla and Hg and thus hard to track source code changes.

* **P10**: “Bugzilla doesn’t let you dig down, it doesn’t give you the changes that happened in a HG range.”
* **P13**: “There isn’t much integration between Bugzilla and the code they are working with.”
* **P20**: “To get richer interaction between the bug and the repository.”

Idea: P13: “Bugzilla would need deeper integration with the source control tool.”

– –

A.1.4.2 Bugzilla Issues [16p, 64s]

This is a high-level bucket for random Bugzilla issues that didnt really fit in the other categories.

[8p, 11s] UI

SQ: Bugzilla’s UI is not user-friendly or designed for process management.

* **P2**: “Bugzilla is not setup to triage, UI not setup for it.”
* **P12**: “Design, usability improvements would make Bugzilla give more information.”
* **P15**: “The Bugzilla interface is bad, too many fields.”

Idea: P5: “Would prefer to replace Bugzilla. Not a particular software package in mind. Fast, Ajaxy, something that doesn’t suck.”
Idea: P6: “If Bugzilla stopped working on new features and focused on the big existing problems, he would be a happy person.”

[7p, 11s] Advanced scripts and tweaks

SQ: Developers use their own scripts or other 3d party tools to help them through the process and workflow.

* **P2**: “Own tools, Python scripts for common tasks. They build tools to work around limitations using the REST API.”
* **P17**: “He has seen people with grease monkey scripts on RedHat Bugzilla/addons doing same thing.”
* **P8**: “3d party tools, he uses the Bugzilla Tweaks.”

[3p, 3s] Process

SQ: The lack of well structured and defined process makes developers' work harder.

* **P2**: "Process is good, tooling bad."
* **P4**: "I think the biggest issue with going through the bugs is that we don't go through the process at all."

[8p, 11s] Feature pages

SQ: The purpose of having feature pages is not well understood, thus they are rarely used and out of date.

* **P11**: "The feature pages are such high overhead they are not always up to date."
* **P3**: "Feature pages, they have been awful. They are out of date the moment they are finished editing. The biggest issue is that there has been poor communication on what they are for."
* **P6**: "He lost track of what was happening on the development of feature pages. Not sure if they are the best way to track that information."

Idea: P15: "Feature pages, need links from a bug from a blog post, then the bug should be aware of that."

[2p, 3s] Performance

SQ: Performance wise, Bugzilla is slow.

* **P14**: "The speed of Bugzilla is the major issue."
* **P18**: "Bugzilla is slow, takes too long to load a page."

[4p, 6s] Culture

SQ: Since Mozilla heavily relies on community contributions, it's important to provide transparency on how Mozilla operates and establish a culture where communication and education of the community is a priority.

* **P15**: "People are not first class citizens in Bugzilla, bugs are."

* **P4**: "If contributors don't really know how Mozilla works, it can be hard to find out how to edit a bug."

Idea: P14: "If you don't need to have one person do the work, you can have a person that attaches a test case, another one that minimizes it, a third one that finds a regression range, and then a fourth one that fixes the bug. We should encourage people to make a bug take any step forward is good. Community members are often the key resource for doing this work."

[12p, 19s] Misc

A.1.4.3 Useless [9p, 10s]

These are quotes that really didn't have any specific value. (e.g., "I like turtles").

A.2 Intercoder Reliability Scores

The intercoder reliability scores reported in the qualitative study discussed in Section 5.2 are presented in Figure A.1. The table presents several reliability coefficients including percent agreement, Scotts Pi, Cohens Kappa, and Krippendorffs Alpha.

	Percent Agreement	**Scott's Pi**	**Cohen's Kappa**	**Krippendorff's Alpha (nominal)**	N Agreements	N Disagreements	N Cases	N Decisions
Variable 1 (cols 1 & 2)	98.9%	0.894	0.894	0.894	180	2	182	364
Variable 2 (cols 3 & 4)	100%	1	1	1	182	0	182	364
Variable 3 (cols 5 & 6)	99.5%	0.886	0.886	0.886	181	1	182	364
Variable 4 (cols 7 & 8)	98.9%	0.851	0.852	0.852	180	2	182	364
Variable 5 (cols 9 & 10)	98.9%	0.962	0.962	0.962	180	2	182	364
Variable 6 (cols 11 & 12)	98.4%	0.888	0.888	0.888	179	3	182	364
Variable 7 (cols 13 & 14)	99.5%	0.854	0.854	0.855	181	1	182	364
Variable 8 (cols 15 & 16)	98.4%	0.658	0.659	0.659	179	3	182	364
Variable 9 (cols 17 & 18)	96.2%	0.84	0.84	0.841	175	7	182	364
Variable 10 (cols 19 & 20)	98.9%	0.851	0.852	0.852	180	2	182	364
Variable 11 (cols 21 & 22)	96.2%	0.891	0.891	0.891	175	7	182	364
Variable 12 (cols 23 & 24)	97.8%	0.788	0.788	0.789	178	4	182	364
Variable 13 (cols 25 & 26)	97.8%	0.739	0.739	0.739	178	4	182	364
Variable 14 (cols 27 & 28)	100%	1	1	1	182	0	182	364

Figure A.1: Intercoder reliability scores for participants P4 – P6.

Appendix B

Interview Materials

This chapter includes supporting materials from the interviews conducted with the Mozilla developers at the Mozilla Toronto office. The intent and findings of these interviews are described in Chapter 5. We include the recruitment email, information/cover letter, as well as verbal consent script in Appendix B.1. The interview script is presented on page 182, and the notes taken during the interviews with the developers are summarized in Section B.3.

B.1 Recruitment Materials

Recruitment Email
(to be sent to Mozilla developers via toronto-all@mozilla.com mailing list)

Dear Mozilla Developers,

My name is Olga Baysal, a PhD student from the University of Waterloo. I am interested in conducting a user study that aims at providing personalized views of issue tracing systems. Those willing to help us out will be provided with a custom view of the Bugzilla repository that provides various views on your daily activities including review queues, bugs in progress, submitted patches, etc. We will ask you a few questions about how you manage your daily bug mail and keep track of changes on the issues you are working with (15-30 minutes). Later will be provided with a prototype (a web-based service) and have a week or so to use it on a daily basis. After interaction with the prototype, you will be asked to provide feedback on the tool. We are aiming at about 15 to 20 minutes to chat with you in person or over Skype about the prototype's features and possible improvements.

If you are interested in participating in this study, please email Olga at obaysal@cs.uwaterloo.ca.

Thank you very much!

Olga and Reid
{obaysal, rtholmes}@cs.uwaterloo.ca
David R. Cheriton School of Computer Science
University of Waterloo

About this project:

Dash is a project led by Olga Baysal from the University of Waterloo to help provide Mozilla developers with customized dashboards that summarize their current Bugzilla issues. The project was started after investigating the Mozilla Anthropology interviews. The project is in active development; if you have any questions, comments, or suggestions, please get in touch <obaysal@cs.uwaterloo.ca>. For more information on the goals of the study, please refer to the Information/Cover Letter attached.

Information/ Cover Letter

Student Investigator: Olga Baysal
Principle Investigator/ Project Supervisor: Dr. Reid Holmes
School of Computer Science, University of Waterloo

Overview
You are being asked to volunteer in a study. The purpose of this study is to investigate how users interact with issue tracking systems. Our study investigates how people organize, manage and keep informed of their daily work items. Specifically, we look at workflow barriers in organizing, filtering and completing daily tasks associated with maintaining situational awareness on the project. We hypothesize that current issue trackers do not support overwhelming amount of bug mail the developers have to sort out on a daily basis. The complexity and poor performance of Bugzilla is frustrating to many developers affecting their productivity, as well as their understanding of working environment and issues they work on.

Your Rights as a Participant
Participation in this study is completely voluntary, and you may decline to answer questions if you wish, and you can withdraw from participation at any time by advising the researcher.

Study Description

Interviews on current practice. During first round of in-person interviews (face-to-face or over Skype), we will first ask some general questions like your experience with issue tracking systems and then ask you about your interactions with Bugzilla, in particular how you stay aware of your daily activities and how you organize and complete your daily tasks such as bug fixing, patch reviews, patch writing, etc. We will ask you to show us how you keep informed about issue status changes or any new issues assigned to you. This task will take place at your working station. We will ask you to share your thoughts with us, i.e. to "think aloud" as you perform tasks on your PC/laptop. We will also ask you questions during and after the task to understand what aspects of the user experience can be improved. For example, we may ask, "What would you suggest we do to improve user's experience with current issue tracking?"

Prototype testing. After interviews you will be provided with a tool that aims at supporting your daily tasks and activities. This tool is provided as a web-based service hosted on the university network. You will be given a week or so to use this tool during your development activities. Your interactions with the tool will be collected and stored on a encrypted laptop to protect your confidentiality.

Prototype evaluation. During the second round of interviews, you will be ask to provide your feedback on the tool (15-30 minutes). Since our goal is to improve user's experience with issue tracking systems, as well as developer awareness on the issues they are involved with, we encourage you to be forthright with your comments, criticisms, and suggestions. A typical interview lasts between 15 and 30 minutes.

Tool evaluation. At a later time once the tool is fully implemented, we may conduct one or more follow-up studies to see what other comments, criticisms, or suggestions you may have for personalized views of the issue tracking system (aka developer dash). These follow-ups will include an interview and, optionally, observations of your use of the developer dash. You may decide at that time whether or not you wish to continue to participate in the study.

Remuneration
You will not be paid for participation in the study.

Benefits of the Study
The information you provide will lead to better designs of issue tracking systems and deployment of the developer dash providing possible improvements of user's experience with their development environment.

Confidentiality and Data Retention
All information you provide will be kept confidential. Neither your name nor demographic details will be revealed. All data will be de-identified and will be kept indefinitely and securely stored in Dr. Holmes's locked office.. With your permission, anonymous quotations may be used in the course report or any publications.

Questions
If you have any questions about participation in this study, during or after the study, please contact Olga Baysal at obaysal@cs.uwaterloo.ca, or Dr. Reid Holmes at (519) 888-4567 ext. 36242.

This study has been reviewed and received ethics clearance through a University of Waterloo Research Ethics Committee. However, the final decision about participation is yours. Should you have any comments or concerns resulting from your participation in this study, please contact Dr. Maureen Nummelin, the Director, Office of Research Ethics, at 1-519-888-4567, Ext. 36005 or maureen.nummelin@uwaterloo.ca.

Verbal Consent Script

Hello, my name is Olga Baysal, and I'm a PhD student at the University of Waterloo. I am interested in talking to you as I am conducting a study about improving developer's awareness of their working context (aka developing developer dash). I would like to ask you a few questions about your interactions with Bugzilla and how you manage and prioritize your daily tasks. The questions will take 15-30 minutes and the information you provide will only be used in the study. Would it be okay to begin with my questions?

B.2 Interview Script

Metadata:

- How long have you been with Mozilla?

- How long have you been using Bugzilla?

- How often do you use Bugzilla?

Card Sort:

- How do you decide what to work on next (next task)?

- How many different issues do you work on in a day?

- How often do you content switch?

- How much email do you get each day?

Personalization:

- Do you need a personalized view of Bugzilla?

- If you had a magic wand, what would you ask for to ease your daily development tasks?

Prototype:

- How likely are you to use developer dashboards?

- How often would you visit developer dashboards during the day?

- What features do you like most? What is useful?

- What changes or additional features would you suggest? What is missing?

B.3 Interview Transcripts

Interviews

Tuesday, Feb 26, 2013 (Mozilla Toronto office)

Mike Conley (D1)

start: 11:00

2 years

bugzilla every day

how to choose a task:

'usually go through emil to see what happened overnight; i usually have a lot of bug mail. i try to see if there are any fires i need to put out. i spend the first 20 minutes prioritizing the day'

'once those are done i go through the tasks that are assigned to me and work on those'

'i use an etherpad to keep track of my list of bugs for each project listed here and i move them up the chain' 'i also have sections for keeping track of other people; it's all very ad hoc'

'i put little notes on something so i can keep track of what i'm waiting on'

use etherpad

'every now and then i reach that point 'wait, what am i supposed to be doing next?' i go back to the etherpad to see what's next on the list'

'bugzilla doesn't have a mechanism for keeping track of what the next action for a bug is' 'these states are the kinds of steps the bugs go through'

task switching:

'some tasks require long builds or test runs so while something is compiling (e.g., 2 minutes) i context switch to try something less complex on another bug' 'i use two machines, three monitors, and run a VM on each machine (6 images in total)'

'i probably receive between 50-100 pieces of email per day' 'a lot of mail filtering in thunderbird' 'more than half, probably 75% is bug mail'

'atul varma has a dashboard that has four columns that has proven to be useful sometimes but it was too strict' 'i want to be able to define the columns' 'what i'm interested in is having a series of bugs that move between columns in a meaningful way' 'i don't necessarily want to make changes to bugzilla to support my workflow that are just for me that i don't want other people to worry about' 'having a way for me to organize the bugs in a way that makes sense for me'

'what might get lost by only focusing on the issues is efforts from new contributors. it's still too easy for them to fall off the radar because they don't really know what the steps are to get a patch from being written to being committed'

'i've seen it happen too many times where a new committer adds a patch to a bug but doesn't set a review flag and it'll sit there for two years without being dealt with. how can we make it easier to make sure that new committers are taken care of'

'a release engineer differently than a developer, and a triager and qa look at it differently as well. people have different slices on it, all of which are visible. if there was a developer view that i could customize further then [role specific views] could work; everything that happens on the bug happens in the commentary; compressing the automated stuff down might be useful.'

'focusing on bugs instead of people is a problem at first, but once you've gone through the gauntlet of using bugzilla you can deal with it ok'

'lots of people work on lots of different component. but sometimes it is hard to know even what component my bug is on. it's pretty ad hoc to determine who should review patches by module owner, it's on the wiki page'

'there's also looking at the blame of the files i'm touching; who recently touched and reviewed the files i'm working on. blake hinton built a tool that provides recommendations by looking at this. i don't use this anymore because i know who to ask now, but when i first started i used it a lot'

'we've been showing new contributors bugs ahoy (joshmatthews.net). there are whiteboard flags for bugs that are mentored but are accessible. these can help get people going. still too easy to fall through the cracks if they're not persistent enough; if they send mail and they never come back to me things fall through the cracks. losing track of new contributors is the biggest tragedy'

prototype

'not only is it important to see how many reviews they have pending, but knowing when they're going to hit a burst would be handy. but i know that dao is going to start a burst in a week or something so knowing that in advance would be helpful. or having a way for him to say that he has a burst coming up so i can be ready for that'

'order bugs by my own prioritization of how important the bug is. i just cut and paste it up to the top of the etherpad'

'i'm not sure how useful it is to know a bout the bugs i've submitted. once it's submitted i don't want to know about it anymore. what success is is seeing a patch with an r and knowing it is taken care of.'

'there's this entanglement between patches and bugs. sometimes there are no patches, sometimes a bug can have many patches. if I have a new r+ then that means an action is needed for me. my own review queue would always give me a sense of guilt because it never goes away. i have tiers of projects; it might handy to have firefox bugs to come to the top of the queue since it's my day job. but if it's thunderbird i can only work on it when i'm not totally exhausted. there's a lot of noise in there because there are a lot of

thunderbird and firefox patches in there. being able to group by product or component would be useful. [and would like to be able to order them]'

'the two views makes more sense. reducing the need to scroll is good. i have a nice big fat monitor so more columns is good. usually has to do with forensics where a bug resurfaced again; so i hit ctrl-f and searched the etherpad [on the summary field]. i know who was involved, i know what component it was on, i know what was said. which one was it?'

[this feels a lot like the expressiveness play]

'if we could take the etherpad (which is a hacky way of representing what is in my head) and break them into lists like things that i need to work on, things that i am working on, things i'm blocked on (this is a big list; i can be blocked because i'm waiting for review, information, the tree is closed), things that are waiting to be committed (are r+ and the tree is closed or i'm waiting for some other dependent patch to land), things waiting for integration'

'things i'm working on, something was reopened and it should go to the top of the list. it's about my own notion of priority'

'i'd be happy to add metadata, to mess around with metadata and flags as long as they're not in bugzilla itself'

'i find it useful to add notes to a bug. usually just a one sentence thing that says 'i need to investigate this thing'. in order to get right back to the mental state i was in on that bug i'll make a note to myself [doesn't need to be private, but just not on the bug]; i would rather not send people bug spam by adding private notes'

'i know that certain actions will send mail and there's a cost to this. i don't want to perform an action unless i'm really sure and i'm ok with this. if it's just a note to myself it's kind of a selfish use of bugzilla but i don't want to spam everybody with it'.

[activity only for forensics]

'there are tiers here. the things i watch i do care about. if i were to be shown a list of bugs i'd like the ones that i'm assigned. assigned, CCed, commented (may be a duplicate of CCed)'

'it's important for me to have an individual dashboard for each project. create a new dashboard and manually add issues to them. my list of blockers are the patches that are blocking the project that i'm working on (50 - 100 bugs for a project; the resolved ones stay in the resolved bin at the bottom). in my head it's columns; it's essentially just move right, move left'

'i really want to emphasize how important it is that each of these dashboards is project-oriented. maybe it is nice to see everything that has to do with me and that's fine but i would prefer to work with individual projects and i'm fine with switching between multiple dashboards'

'almost always i do this by myself, and sometimes someone will suggest something but it doesn't happen often'

———————

Andrew Howberstat (D2)

start: 12:06

2 years with mozilla; bugzilla for the same time
use bugzilla every day (that i'm at work)

task selection:

'usually i have longer-running tasks going on. my main way of interacting with bugzilla is through the awesome bar using keywords. if i'm not really sure what to do next i'll look at my assigned bug list.'

'i do test automation so often things will break so when something breaks that'll take priority for the week. i'll have bugs that pop up out of nowhere. someone will just say on IRC 'hey, this is broken' and i'll go file a bug and track it'

'i have a couple of projects i work on and some are higher priority than others. i usually work on two or three in a day on average. sometimes the same issue persists for several days in a row. if i'm waiting for a review or waiting for a build i'll go work on another issue'

'i probably get 70-80 bug mails in a day'

'so we have an ongoing project so anytime that past project has a problem i keep track of it'

[[should we do a 'bug mail filter study'?]]

prototype

'i really like the issue watch list. i have my own custom bugzilla queries for the four columns. it's a saved search i have. when i do use it it is hard to look through the list.'

'i would use it if i couldn't find what i was looking for in the awesome bar right away. this frequently happens because bugs have similar keywords'

'i like the patch log as well because i find the bugs i care about often have active patches in them. i wouldn't use the review queue very often because i don't like to have them sit around.'

'this might be specific to my needs but i find lately we have a lot of different tracking flags with b2g that has spawned its own process. it might be nice to have a dashboard for different keywords. for b2g they have tracking flag and approval flags. these are the bugs that have trackingb2g18 on them. just open bugs would be the most useful'

'prefer the two column better because the larger space is easier to use. i only care about the bugs i most recently used. i do go by last touch. anything that is open could be in the list. i'd probably rather have one combined dashboard. assigned, reported, CCed or are important to me'

'tracking flags, but maybe if you could say what whiteboard flags you're interested in. we don't want to get it to be replacing bugzilla at all'

done: 12:25

Katz Gupta (D3)

start: 12:28

mozilla since october 2011 (1.5 years); bugzilla every day since then

'use filers to scrape out the bug mail and put it on the web page. i figure out how to prioritize it on the page'

'usually i start my day working on whatever i was working on yesterday. i'll look at my bugmail dashboard to see what new things are coming in'

'there are four columns here. the left one is about patches and reviews. the next one is about ??? and the last one is about things I'm CCed on'

'i usually have two or three tasks that i work on in any given day. when i'm building on one i'll switch to the other one while i'm blocked'

'i watch the dashboard during the day; even on my phone i keep track of it sometimes'

'i manually close bugs when i'm done looking at them. they never go away on their own'

prototype (12:34)

'i usually watch widget-android and firefox for android. i want my components to be grouped together'

'i don't think i would use the completed review tab; once it's done that i would like to forget about it. i like seeing other reviewers because i want to be able to assign reviews to someone who isn't too busy. i prefer to pick someone who has less things on their plate.'

'i think the most common use case is i have someone in mind and i want to see what their current queue looks like. i don't need a permanent tab but to have some way to get that information would be useful'

'i find that sometimes a plain text search over the DB which contains everything that is in the database'

'i prefer the two column format. in terms of history i think i would collapse things for the same bug, i don't need to see individual items. when i'm uploading patches there'll be changes or something.'

'i don't really sort them in that way [with last touch]. it would be good to be able to tag specific bugs in a specific way. i guess if i have a lot of bugs in the same specific area of the code i could work on them in a batch. i don't really care if they're in bugzilla. it would be good to sometimes to have notes. i usually maintain a text file of notes for bugs. if i'm half way through bugs i'll put a state dump so it's easy to pick up on it again.'

'when i remove a card all the context that goes away. sometimes getting rid of a note can get rid of it all. generally i care about bugs that are assigned to me, bugs that i'm tagged for a needinfo or review request

(need additional info from), and there's another class of bugs where new bugs get filed and i see them in my summary and i want to investigate them bug i'm not tagged on them (filed on component). e.g., another column that contains all the list for that component and then i could dismiss things i don't care about. i could have an option to move them to a watch list (the ones i care about) or i could dismiss the ones i don't care about. (private watch list ask).'

'along with CC i also care about things i voted on. other people can CC me on bugs, but they can't vote for me. it could be mixed in with the CC or could be put in another column. i don't know if others do this but it is how i do it myself'

'occasionally i care about past stuff, but sometimes when i want to look at the history of the code i care about that stuff. i use the git history for that though, not the dashboard.'

'bug assigned to me i want to see everything, i don't care how old they are. history and commented just two or three days is ok. being able to move anything into a watch list could be helpful. usually the mechanism for that is to CC, although sometimes i would vote.'

end: 12:53

Ty Flanagan (with creative) (D4)

start: 13:05

bugzilla every day, for 2 years

'bugzilla is an app tab, i look at my bugs (the ones assigned to me) and that's where i start the day. the project manager reassigns relevant bugs to me'

'i work on one solid task or up to twelve different ones. these don't clear out each day. this can just involve commenting or adding things to a bug'

'my bug list i don't have pri1 pri2 or soforth, my manager will indicate when things need to happen. we have deadlines posted into the bugs'

'the deadlines are in the comments in the bugs, it's not a great system for creative. for designers and PMs in a creative place. layervault is one option. we can't upload and share large files in bugzilla because they get rejected. our file sizes can be 200-300 megs. it doesn't allow us to... one of the failures in bugzilla is that you have a comment thread you can't have a real way to have the comments associated with a specific attachment. you lose the nesting'

'creative works on the brand / engagement product'

'when i have a lot of projects going on i don't get too much bugmail (5 assignments per day), these are generally long-lived. internal pojects often don't even get bugs'

'conceptshare is another collaborative online site we looked at. it's like track changes for word so you can upload design and comment on them etc. it keeps track of approvals etc, including conditional approvals that have descriptions on them'

'we've also been working with basecamp for the mobile world congress project because it was easier vendors to interact with us and upload/share/communicate better. you can always attach certain people to groups so they can get notifications when details about the project have changed'

'would like to have in-line commenting from the dashboard'

=== updating parking ===

'layervault is pretty cool. take a look. there are a lot of things in bugzilla that designers just don't need'

'for us simplicity is a key so we can get right to looking straight at the [assets] as easily as possible'

done: 13:34

mike: 'mozilla pulse' might send some kind of json announcements

_____

Chelsea Novak (D5)

start: 13:58

chelsea; community engagement. 3.5 years at mozilla

'i'm a campaign manager; i create bugs and others have to fix them'

'i manage them in two ways. i file all my bugmail into a folder and i tag these with a colour. in order to keep track of active bugs in a tab i have the bugs that i have filed and are still open'

'for major creative projects we have tracking bugs and check the status of the sub bugs'

'for projects i have specific assets to keep track of the status in spreadsheets'

[keep track of things in a text file]

'so basically i have these steps for pre-launching a site and in a google doc i have a visual idea of what's going. in this place i can see what is happening, what it's status is, etc. google doc is better here because lots of things aren't tracked in bugzilla (but instead are stored in etherpad or other sites)'

'want to make sure everything gets done and things don't fall through the cracks. people don't check their bug mail and they ignore it until i ping them until they get things done for me. i scan the bug mail to see what has the colours that indicate action.'

'five colours: personal, saved, important, todo, later. good naming is key so i can search back on things. all manually tagged and triaged.'

'i don't triage in bugzilla because i don't know how to use the search params very well. i'm not experienced with those kinds of searches. bugs i've filed, copied on, or on the components i care about. i wish i could reply to conversations inline'

'i need to be able to search really fast. i have three years worth of bugs in there. the quick filter is super fast. right now it's mainly searching the title. we want to be able to reuse things or have questions of process we want to be able to query on past issues'

'nice. that's cool how the bugs are listed'

'i would have tags by campaigns would show up'

'i probably would like to have them grouped by colour and sorted within the colour by date'

'what about attachment log? we deal with a lot of those and then people comment on that'

'we don't use flags very much, just the developers do that.'

'i'd love to know who filed the bug, priority can be useful, bug status since i look at closed bugs too'

'[role specifc views] would be excellent. lots of these fields just aren't relevant to non-developers and it can be daunting for us. role-specific views would be really helpful and being able to toggle between them'

'it's the managing of active bugs that is the hardest part for people'

'it is very intimidating for first time users and it's really not friendly and approachable. you have to jump through a whole lot of hoops'

end: 14:22

Wednesday, Feb 27, 2013 (via Skype)

Mike Habicher (D6)

start: 10:06

with mozilla for 1 year, bugzilla the whole time

use bugzilla every working day

start the day:

'usually i have a couple of custom [bugzilla] filters that pulls up firefox bugs that are related to camera related bugs for firefox os'

'i get all the bug mail so i can keep an eye on what is going on. usually in the day i get a couple of hundred emails so i can go through and find the interesting ones and delete the rest. it's probably not the most efficient, but i find that this is the best way to avoid missing something important'

'searches on product [b2g] and basically looks in the summary for the keyword camera. i could have searched for component camera, but i find that the summary works better'

'on bugzilla there's the search link and under advanced search i use that. those are the only two fields i set [product + summary]'

'so for one of the project stages called basecamp there were bugs called blocking-basecamp+'

'on a typical day i probably don't work on more than two tasks; that'll probably change as the bigger tasks are done and smaller ones are getting going. in the last big push i was doing 4-6 tasks per day'

'i don't multiask well'

'what else makes it [bugmail] interesting is just that it seems interesting to me'

'if i need to get status of bugs i'll almost always go back to bugzilla. sometimes if i'm looking at bugs in my inbox i can tell what i'm interested in. the rest are bugs that are more interesting and i'll look through those to see if there's something significant that has changed.'

--prototype--

'this [prototype] is really cool'

'oh, and activity history so i can see everything i've contributed to the project'

'wow!'

'looking at mike's the only thing that could be an issue for me is that the activity log could be really long'

'i think this is great, something like this would be fantastic for sorting through all the things i need to take care of'

'i use the queries more as a todo list to figure out what i need to work on'

'one thing, under the commented section (that i think is great), one thing that would be interesting is whether there was any other followup comments to my comment. like from a FB point of view it bumps it up when someone has responded to you'

'this makes much more sense now [with the stylesheet]'

'this is fantastic, although there's a second set of scroll bars'

'so here's my thing, i work on mobile a lot and the 2 col one is nice because i can see a lot more data'

--after Reid left

Suggestions:
Activity History/Commented - group by bug id
Add Patch description for Patch Log
Obsolete patches not to show

How often would come back to dash?
'it depends, small tasks more often, one big thing - once every morning and evening'

'looks fantastic! really impressed!'

'for long lists - option to indicate that you looked at bug already , may be red mark', 'may be to add 2 sections: new and viewed'

Time range: add option to set this: last month, year

'when can i start using it?'

end: 10:45

Rail Aliiev (D7)

start 10:45

meta:
3 years with mozilla
3 years with BZ and even more, previous experience

'every single day, more than one hour a day when trying to triage, looking for bugs assigned'

tasks:

'first thing in the morning to review assigned bugs, go through all bugs'
'deciding what I have to work on today is something i start each day with'
'define my own priority: P1 - forget about everything else, P2 - have to do, P3 - don't look until later'

'revise this list once a week'

20-25 bugs a day working one, closing 1-2 bugs each day.
task switch depends on if i need to run something: go for coffee/beer', 'prefer to work on one task at a time'

'Thinking Rock app used a lot in the past for organizing self'

'now switched to paper, paper for one day tasks, BZ - for longer tasks'

'remember milk - for organizing thoughts on the go, plane, etc.'

wiki for group projects priorities, our group, release eng, release procedure - each release a separate page; bugs assigned correspond to a wiki page for that release'

'make sure your project manager have same priorities as yours'

Organize incoming bug mail into folders, 8 of them:
- interested in (cc), i read this more carefully
- new bugs, 20-3- bugs a day for release eng.
- 2 sub-component folders
- everything else, not yet resolved
- resolved
- sub-component watching
- assigned bugs these are forwarded to mobile phone as notifications

prototype:

'that's interesting!'
checked-in flag is something only release engineering team defines. would like to separate bugs that are not checked-in yet to a separate tab.

'personal tags - checked-in bugs'

'we have bi-monthly build-in person for a week who is responsible for bug triage - review incoming bugs, triage them, to get a whole picture on the project, these bugs are important, these are not. check may be I have missed something'
triage list has 1000s emails

triage is to make component more manageable

feature to add: list bugs for component, option to group by white board tag

Assigned column - will look at often, CC/Commented - once a week
'history is pretty useful'
'really like the idea of having peers tabs'

'i'd like to keep this [dashboards] open all the time, all day long'

'do i need to refresh it? i am ok with that, will refresh every 5 minutes then'

prefers 2-column layout

'having an option of setting time range would be great'
'2 months is useful, quarterly goals'. 'every year or every 6 months is useful for performance review procedure (formal and informal) to recall things i have worked on, dash would help with this'

'sort by personal priority, by the time the bug was closed'

'when do you plan to have it[tool] ready?' 'looking forward to having it'

end: 11:40

_____

Benoit Girard (D8)
start: 13:31

mozilla for 2 years, using bugzilla for 3 years (every day)

'usually it's bugs that have specific tracker flags that get the highest priority; at least once a day i go through my review queue and go through those as well. sometimes [infrequently] we go through the list of bugs and reprioritize important ones that have been demoted'

'i check the bug mail, it's on my phone. i check it every hour, even on the weekend'

'i experimented with this and would have 5-6 copies of the tree going at once but i found that i got less done. now i focus on around 2 tasks at a time'

'i don't really use the bugzilla assigned-to field very much, i just have too many bugs assigned to me. the priorities that are assigned to them are not very useful. what we do is we'll meet every quarter to discuss what we're going to get done. if i'm assigned a tracking bug i will work on those right away. we also have emails that go out every monday to check progress on tracking bugs'

'i get about ~40 bugmails that make it to my inbox; there are a bunch of others on interesting components that go into other folders that i rarely check'

---prototype---

'that's actually one problem i have encountered a lot with bugzilla. i'll submit a patch for a bug to a low-priority bug and i'll miss the r+ and if i miss it i might not see it for a year or more. this view can help me avoid missing those'

'i think this is very interesting. have you seen kat's dashboard?'

'i think being able to keep patch of patches that are r+ that are ready to go; or not even approved but have just been responded to. i don't know that you can do this with a query. this way i would know that i need to take action. also, being able to check the queue of others to see how many outstanding requests they have. perhaps even just a number for the size of their queue and the number could pull up the queue. if the queue is over 10 maybe the row would get a different colour; it could tell me that i should find someone else to review my patch'

'sometimes i'll stop working on a bug and i will attach a patch that is like 'this is where i left off' and i don't care on the bug anymore. i'll still apply a work-in-progress patch but i still want to see it. if the review has been answered i _REALLY_ _REALLY_ want to have that in the patch log so I can followup. I think hiding obsolete patches is a good idea, but I think you need to be a bit careful'

'i'm inclined to say the two column layout. i would recommend switching the ordering of the tabs; in order of importance i think comments are more important than submitted (e.g., when i submit to a component just so they know). bugs that i'm commenting on and am posting patches to are the most

important ones. and bugs that i'm assigned to that are tracked. i don't care about bugs that i track that i'm cced on, but if i've commented/assigned/posting patches on a bug that is tracking are absolutely the most important.'

'i don't even pay attention to the priority; we use the tracking flags for everything'

'i think that [last touched] is good'

'honestly if we had better dashboards i would keep them open and come bak to it frequently, several times a day, i think'

'preferably as far back as possible and maybe limit it to 25 elements. for me i'd like to see the 100 most recent items'

'it might be interesting to add a column to most of these to show the tracking information; it's complicated because we have many tracking flags. it might be a good start to just have a tracking column and if there are any tracking flags at all show it and if there are multiple just say tracking+'

'i might still be interested to look through patches that don't have ? or answer it would be nice for them to be there. it would be good to be able to see patches that are works in progress and be able to see those. that gets a little bit confusing because sometimes we'll put r? or clear the flag instead of an r-'

'i have no way of parsing or prioritizing the component information so i can't watch that very well. being able to dissect a component to keep track of what is going on would be very valuable. in components, i wouldn't even break it down by component. so sometimes we'll be working on a feature that crosses multiple components and we'll do that with a tracking bug or a whiteboard tag that we use, but really what i watch is the feature that spans the component. descendent bugs especially would be useful here. i might be interested in watching how people are responding to mentored bugs and see if one of them is not going the way we would hope. sometimes it is a tracking bug, sometimes it is a whiteboard. here's one that the gfx team is involved in right now 792576. so the tree view ordered by which ones are getting comments or patches would be helpful.'

done: 14:15

———————

Appendix C

Waterloo Office of Research Ethics Approval

PANEL ON
RESEARCH ETHICS
Navigating the ethics of human research

TCPS 2: CORE

Certificate of Completion

This document certifies that

Olga Baysal

has completed the Tri-Council Policy Statement:
Ethical Conduct for Research Involving Humans
Course on Research Ethics (TCPS 2: CORE)

Date of Issue: **12 November, 2013**

UNIVERSITY OF WATERLOO

OFFICE OF RESEARCH ETHICS

Notification of Ethics Clearance of Application to Conduct Research with Human Participants

---

Faculty Supervisor: Reid Holmes **Department:** Computer Science, School of
Student Investigator: Olga Baysal **Department:** Computer Science, School of

ORE File #: 19398

Project Title: Improving Issue Tracking Systems by Implementing Personalized Developer Dashboards

---

This certificate provides confirmation the above project has been reviewed in accordance with the University of Waterloo's Guidelines for Research with Human Participants and the Tri-Council Policy Statement: Ethical Conduct for Research Involving Humans. This project has received ethics clearance through a University of Waterloo Research Ethics Committee.

***Note 1**: This ethics clearance is valid for one year from the date shown on the certificate and is renewable annually. Renewal is through completion and ethics clearance of the Annual Progress Report for Continuing Research (ORE Form 105).*

***Note 2**: This project must be conducted according to the application description and revised materials for which ethics clearance has been granted. All subsequent modifications to the project also must receive prior ethics clearance (i.e., Request for Ethics Clearance of a Modification, ORE Form 104) through a University of Waterloo Research Ethics Committee and must not begin until notification has been received by the investigators.*

***Note 3**: Researchers must submit a Progress Report on Continuing Human Research Projects (ORE Form 105) annually for all ongoing research projects or on the completion of the project. The Office of Research Ethics sends the ORE Form 105 for a project to the Principal Investigator or Faculty Supervisor for completion. If ethics clearance of an ongoing project is not renewed and consequently expires, the Office of Research Ethics may be obliged to notify Research Finance for their action in accordance with university and funding agency regulations.*

***Note 4**: Any unanticipated event involving a participant that adversely affected the participant(s) must be reported immediately (i.e., within 1 business day of becoming aware of the event) to the ORE using ORE Form 106. Any unanticipated or unintentional changes which may impact the research protocol must be reported within seven days of the deviation to the ORE using ORE form 107.*

Maureen Nummelin, PhD
Director, Office of Research Ethics

11/28/2013
Date

OR
Susanne Santi, MMath
Senior Manager, Research Ethics

OR
Julie Joza, MPH
Manager, Research Ethics