Towards Next Generation Bug Tracking Systems

by

Rafael Velly Lotufo

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

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this 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.

Abstract

Although bug tracking systems are fundamental to support virtually any software development process, they are currently suboptimal to support the needs and complexities of large communities. This dissertation first presents a study showing empirical evidence that the traditional interface used by current bug tracking systems invites much noise—unreliable, unuseful, and disorganized information—into the ecosystem. We find that noise comes from, not only low-quality contributions posted by inexperienced users or from conflicts that naturally arise in such ecosystems, but also from the difficulty of fitting the complex bug resolution process and knowledge into the linear sequence of comments that current bug tracking systems use to collect and organize information. Since productivity in bug tracking systems relies on bug reports with accessible and realible information, this leaves contributors struggling to work on and to make sense of the dumps of data submitted to bug reports and, thus, impacting productivity.

Next generation bug tracking systems should be more than a tool for exchanging unstructured textual comments. They should be an ecosystem that is tailored for collaborative knowledge building, leveraging the power of the masses to collect reliable and useful information about bugs, providing mechanisms and incentives to verify the validity of such information and mechanisms to organize such information, thus, facilitating comprehension and reasoning.

To bring bug tracking systems towards this vision, we present three orthogonal approaches aiming at increasing the usefulness and realiability of contributions and organizing information to improve understanding and reasoning.

To improve the usefulness and realibility of contributions we propose the addition of game mechanisms to bug tracking systems, with the objective of motivating contributors to post higher-quality content. Through an empirical investigation of Stack Overflow we evaluate the effects of the mechanisms in such a collaborative software development ecosystem and map a promissing approach to use game mechanisms in bug tracking systems.

To improve data organization, we propose two complementary approaches. The first is an automated approach to data organization, creating bug report summaries that make reading and working with bug reports easier, by highlighting the portions of bug reports that expert developers would focus on, if reading the bug report in a hurry.

The second approach to improve data organization is a fundamental change on how data is collected and organized, eliminating comments as the main component of bug reports. Instead of comments, users contribute informational posts about bug diagnostics or solutions, allowing users to post contextual comments for each of the different diagnostic or solution posts. Our evaluations with real bug tracking system users find that they consider the bug report summaries to be very useful in facilitating common bug tracking system tasks, such as finding duplicate bug reports. In addition, users found that organzing content though diagnostic and solution posts to significanly facilitate reasoning about and searching for relevant information.

Finally, we present future directions of work investigating how next generation bug tracking systems could combine the use of the three approaches, such that they benefit from and build upon the results of the other approaches.

Acknowledgments

First of all, I'm immensely grateful for Erika's love, support, patience, and advice. Her arms and words were fundamental in helping me find the light at the end of the tunnel. I'd also like to thank my parents, Roberto and Valeria, for all the unconditional love and acceptance. I'm privileged to be their son.

Second, my supervisor, Krzysztof Czarnecki who was indeed a great supervisor. Even if I did not take advantage of working on a topic on his long expertise list, he provided me with the guidance I needed at the time I needed it. His commitment to detail, work ethics, and paper-writing skills have significantly influenced the way I think and research.

I would also like to thank my external examiner, Prof. Ahmed Hassan, for finding time to visit Waterloo during the summer, and for his insightful comments on this dissertation. I am very grateful to my internal thesis committee, Prof. Patrick Lam, Prof. Michael Godfrey, and Prof. Lin Tan, for helping me shape this research.

I'm also grateful for my colleagues who collaborated with me on my research papers, Zeeshan Malik and Leonardo Passos. Without Zeeshan's fantastic app and execution of the empirical evaluation of bug report summaries, we would have never earned the best paper award at ICSM. I'm also very grateful for Kacper's and Michal's interest in BugBot, for their valuable insights, and for allowing me to study its usage. I'd also like to thank Steven for helping me out in the very early stages of my research with feature models.

Finally, a big thanks to the Generative Software Lab for all the insightful (and not so insightful) discussions, workshops, lunches, badminton, and research meetings at the gradhouse.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Bug tracking systems are a fundamental component in any software development project or process, serving as a communication channel for contributors to collaborate on bug resolution. Since software engineering has still not provided a silver-bullet against bugs, as much as 50% of software development efforts are spent in resolving them [14]. Valuable developer time is, thus, spent in consulting and analyzing bug report data, discussing solutions for the reported issues, and coordinating bug resolution [6, 15, 29, 57, 66]. Large open source bug tracking systems, for example, receive hundreds of bug reports per day [5, 29], all of which need to be addressed appropriately.

It is commonly believed that the larger the bug tracking community, the better, since larger communities are believed to get more bugs fixed than smaller communities—"with enough eyeballs, all bugs are shallow" [53]. Larger communities, however, lead to more disorder: more contributors posting their comments, their own bug symptoms for their execution environment, their own perspectives, ideas, and hypothesis. While more information is arguably valuable, if such information is not useful, reliable, and organized, bug tracking system users will find it difficult to locate, understand, and work with the information they need to complete their tasks.

Current bug tracking systems, however, provide no means to ensure, motivate, or facilitate building bug reports with reliable, useful, and organized information. Since current bug tracking systems only provide rudimentary facilities to prioritize bug reports, and absolutely no facilities to organize and structure information within a bug report, the overwhelming quantity of bug reports and comments received leave developers struggling to decide on which bug report or comment is most worthy of their attention and relying on their own heuristics to select reports and comments to focus on [5, 29].

We claim, thus, that bug tracking systems, in their current form, are suboptimal to support the needs and complexities of large communities. We consider that the two main factors that make bug reports difficult to work with are the simplistic way that complex, contextual data is organized in bug reports and the substantial amount of incomplete or erroneous content posted to bug reports.

The first factor—simplistic data organization—arises from the fact that virtually all bug tracking systems, today, resemble traditional Internet forums. Similar to a forum thread, a bug report is structured as an initial problem description—often a description of a scenario in which the software system does not behave as expected—followed by a linear sequence of messages ordered by submission time—also known as *comments*—exchanged by a variety of contributors collaborating to resolve the reported software issue.

The second factor—low-quality contributions—arises from the fact that bug reports reports accept unrestricted free-form text as comments, attracting largely informal and unstructured content. Since projects struggle to keep up with the high activity levels and do not have the resources to check all the data sent by contributors [5], the lack of strong motivation or guidance to help contributors post useful information results in bug reports ripe with incomplete, unuseful, and misleading data [10, 15, 61].

The result of this simplistic approach for organizing information and of low-quality contribution are **noisy** bug reports: bug reports with *unreliable*, *unuseful*, and *disorganized* information.

Since collaborative bug resolution is a complex, multi-faceted process that is hard to fit into a simple, linear representation, such as a sequence of comments, information is not organized in a way that facilitates its understanding and reasoning. First of all, since comments are sorted by time, information about similar topics is scattered throughout bug report comments. Understanding all of the symptoms and diagnostic information about a bug, for example, thus, requires a reader to read the entire bug report, since such information is posted by multiple users reporting their own account of the bug.

Furthermore, bug resolution involves much testing of assumptions and hypotheses, which might lead to progress on understanding the bug or to a dead-end, requiring users to withdraw, review, or make new assumptions and formulate new tests [18]. In addition, besides the discussions aimed at problem-solving, there is much off-topic content, such as triaging-related discussions and conflicts. Triaging-related discussions occur for coordination purposes: contributors must agree on the bug priority and who will fix the bug, for example. Conflicts also naturally arise in bug reports and add to the off-topic content in bug reports [9, 33, 38].

To understand a bug report, readers must, therefore, attempt to rebuild the non-linear problem-solving process from the linear sequence of comments, requiring them to keep track of the current assumptions, the variety of tests performed, their results, and their contexts, still taking care not be distracted from off-topic content. The challenge of creating a mental-model of the bug history from such a noisy, linear, sequence of comments, thus, obstructs contributors' ability to consult, understand, and reason about information in bug reports [33, 38]. Since reasoning is fundamental when trying to pose hypotheses for what is causing the bug and for what might solve the bug and when comparing different solutions to a bug, this makes bug tracking systems a suboptimal communication channel for collaborative problem-solving and decision-making.

Productive bug-resolution, however, depends on reliable and useful—high quality—contributions. [29, 35]. Low-quality contributions can reduce productivity for at least three reasons: first, users will need more time to read through the bug report to locate useful information; second, in the case of incomplete information, contributors will need to ask for more information [15, 35] or investigate for themselves; and finally, in the case of wrong information, users will be directed to unproductive directions of investigation. As we present in Chapter 4, developers significantly value tools to help them locate and digest useful information in bug reports.

Low-quality contributions not only reduce productivity by leading developers to wrong assumptions and hypotheses, but they also add more non-linearity to bug reports. As bug tracking systems do not have a specific or automated mechanism for assessing the quality of contributions, contributors must use the comments to warn others about misleading or erroneous data. Since the warning or correction might be in a comment much further down, readers will only notice the error when, and if, they continue reading through the comments.

Evermore, bug reports continue to be consulted much time after their submission date. The life-span of a bug report lasts often weeks or months [65]. Users should, therefore, be able to easily understand and locate relevant information the first time they consult a bug report or after much time since last consulting it.

1.1 Improving Productivity in Bug Tracking Systems

Next generation bug tracking systems should be more than a tool for exchanging unstructured textual comments. They should be an ecosystem that is tailored for *collaborative knowledge building*, leveraging the power of the masses to collect reliable and useful information about bugs, providing *mechanisms and incentives to verify the validity* of such information and *mechanisms to organize* such information, thus, facilitating comprehension and reasoning.

Next generation bug tracking systems, as a result, should be ecosystems in which users should be able to open up a bug report and clearly understand it in a few seconds. Contributors should clearly understand, for example, what is the reported issue; in which environments it has been confirmed in; what are the existing solutions and workarounds for the problems and the environments they are applicable for; and understand what are the issues preventing a bug from being fixed.

This dissertation presents and evaluates three different approaches for bringing bug tracking systems towards this vision and making them more productive. The solutions we present focus on addressing the two main factors that we consider contribute most to making bug reports difficult to work with: low-quality content and the difficulty of fitting the non-linear problem-solving nature of bug resolution into a linear sequence of comments by providing mechanisms and incentives to increase contribution quality and mechanisms to organize information in bug reports. As a result, developer time should not be wasted in looking for and trying to understand information in bug reports, but spent in effectively using such information to complete important tasks in bug tracking systems, such as triaging or solving bugs, detecting duplicate bug reports, finding a workaround to an existing problem, or understanding the issues preventing a bug from being fixed.

1.1.1 Motivating Better Contributions

One of the presumed causes of low-quality content in bug reports is the lack of motivation for contributors to post higher quality content. To address this issue and increase productivity in bug tracking systems, the first approach we present attempts to use *game mechanisms* to create a healthy, engaging ecosystem in which contributors feel motivated to post high quality, useful, content.

Game mechanisms use reputation and rewards systems to encourage desirable behavior and have previously been shown to increase trust, motivate participation, and push participants to new levels of achievements in online communities [20, 22, 43, 45]. Reputation and rewards have also been acknowledged by mainstream online discussion forums, such as Google Help and Slashdot, for instance, which rely heavily on game mechanisms to stimulate user participation. Our work is also motivated by a recent trend in software development ecosystems to adopt reputation systems. Visual Studio, for example, now has achievements, badges, and leader-boards; Launchpad calculates a reputation score for users based on the quantity of contributions and also has an accomplishment board, but does not offer rewards.[1] We are not aware, however, of previous work studying the benefits of such game mechanisms for software development ecosystems.

1.1.2 Identifying Useful Contributions

As previously introduced, bug reports contain many conversation threads that are off-topic, lead to dead-ends, or have low-quality content. The second approach to increase productivity in bug tracking systems improves organization in bug reports by identifying or highlighting useful information and eliminating less-relevant content.

We present two methods for identifying useful information: a manual method, that relies on human evaluation and works in conjunction with game mechanisms; and a fully automated method that creates bug report summaries composed of the most useful and informative content from the original bug reports.

The bug report summarization approach attempts to model how an expert would read a bug report when pressed with time, assuming the reader will have to overlook many sentences and focus on the most important ones. Since we model how developers would read a bug report, in general, regardless of the project and its domain, it does not suffer the drawbacks and limitations of the previous bug report summarization approach [52] that has a high setup cost of having experts create a large set of summaries to train a classifier.

1.1.3 Reducing the Importance of Comments

The final approach to increase productivity in bug tracking systems works on overhauling the bug report communication interface and structuring content to make bug reports better suited as a communication channel for collaborative problem-solving.

While it is difficult to locate, understand, and reason about data in conversational content [33, 38, 43], informational content is objective and to the point [43]. We, thus, propose a significant change on how bug report data is collected and organized and create a bug tracking system that replaces the traditional forum-like user interface of today's bug tracking systems with an interface that encourages users to focus on the most relevant information for bug reports.

By encouraging contributors to focus on informational content creation, this approach provides better guidance for contributors to post higher quality content. Furthermore, lowering the importance of conversational content allows users to focus only on informational

[1] https://wiki.ubuntu.com/Accomplishments/

content, reducing the need for contributors to spend valuable time on following the many different threads of conversation.

1.2 Evaluation

To evaluate these approaches we present three empirical studies, covering a total of six real-world software development ecosystems.

1.2.1 Game Mechanisms

To study and evaluate the use of game mechanisms in software development communities, we perform a systematic empirical investigation into three years of Stack Overflow, a successful developer community enriched with game mechanisms.

We find that Stack Overflow's game mechanisms instill a sense of healthy competition, which increases contributor participation and motivates contributors to post higher quality contributions and to review and improve their peers' contributions. Our analysis shows that Stack Overows game mechanisms implement a formal meritocracy, which is well accepted by its developer community, and is effective: they lead to quality improvement and motivate increases in contribution frequency of up to three times. Rewards also increase the number of competing answers for best contributions by as much as 50%, thereby improving resolution rates by 10%.

Given the similarities between Stack Overflow and bug tracking systems, we propose how to add these game mechanisms to bug tracking systems and, based on an empirical analysis of four large bug tracking systems, present evidence for and the conditions in which game mechanisms would be successful in bug tracking systems. Our study shows that for game mechanisms to be successful in a community, its reward system should be carefully tailored to its community's interests so that the prizes offered are not only attractive to the community, but also motivate contributor behavior that will further improve content quality.

We also find that game mechanisms can also be effective in filtering out unuseful content and identifying reliable contributions. As contributors review their peers' contributions, the higher-quality contributions are highlighted, while the lower-quality contributions can be ignored. We also investigate how peer-reviewing currently works in existing bug tracking systems and find that it is not as effective at filtering out low-quality content as Stack

Overflow's manual filter, since bug tracking systems are not as democratic as Stack Overflow.

1.2.2 Bug Report Summarization

To evaluate our bug report summarization approach—the second approach at identifying useful contributions—we compare the quality of the summaries our approach creates with the quality of the summaries created by the previous bug report summarization approach [52]. Our results show that our summarizer creates summaries with as much as 12% higher evaluation measures compared to the previous approach.

We also perform a very insightful evaluation of our summarization approach with 58 developers from four important open-source bug tracking systems. Our findings show that our approach not only produces higher quality summaries, compared to the previous approach, but that 80% of the developers find that the summaries would be very useful in performing common tasks, such as searching for duplicate bug reports, since they allow them to focus their attention only on the most relevant information.

1.2.3 Conversational to Information Content

Finally, to evaluate our approach at changing the focus of bug reports from conversational to information content, we create BugBot, a novel bug tracking system that eliminates comments as the main component and instead asks contributors to post either diagnostic information or solutions to the bug. We evaluate BugBot during 6 months on a small software development project and find that, while the learning curve to this new paradigm is steep, users find that BugBot makes locating, understanding, and reasoning about information easier than in traditional bug tracking systems and would prefer to use BugBot in future software development projects.

1.3 Research Contributions

This dissertation presents three main research contributions for improving productivity in bug tracking systems: it presents the first systematic study showing how game mechanisms can improve software development ecosystems; it prodvides a deeper understanding of what makes information relevant in bug reports and how to automatically identify it; and presents a study of a novel interface for bug reports that should be better suited to support bug resolution communication.

The Effects of Game Mechanisms in Software Development Ecosystems

- we present a systematic empirical investigation of Stack Overflow's game mechanisms in a software development ecosystem, showing correlations suggesting that such a formal meritocratic system can not only motivate participation but also improve contribution quality (Section 3.5);
- we propose the use of game mechanisms in bug tracking systems; our findings present the conditions in which the game mechanisms should produce its benefits and valuable insights for how to tailor a the game mechanisms so that the conditions are met (Section 3.6);
- we are the first to investigate current voting mechanisms in bug tracking systems, how effective they are at filtering low-quality content, how they can be used with game mechanisms, and how duplicate bug reports can be used to improve filtering (Section 3.6.3).

Identifying Useful Information in Bug Reports

- our study with open source developers show how they acknowledge tools to help them consult and work with bug reports (Section 4.3.3);
- our qualitative investigation of bug reports finds that simple comments are inappropriate to organize information about the bug resolution process since it is a complex, non-linear process (Section 2.2);
- we show how importance evaluation comments in bug reports are and how they can be used to identify important content (Section 4.2.2);
- we present a novel, lightweight, approach for summarizing bug reports with many improvements over previous approaches, together with natural language processing techniques, such as topic similarity and sentiment analysis, to identify the most important information in bug reports (Chapter 4).

Communication Interface Suited for Bug Resolution

- we present the evaluation of a bug tracking system that eliminates comments as the most important element of bug reports and attempts to shift contributors' attention from conversational content to informational content (Chapter 5);
- to facilitate bug report browsing, we also evaluate two proposals for viewing and navigating through bug reports (Section 4.3.3).

1.4 Outline of the Dissertation

In this chapter, we have introduced the problems that reduce productivity in bug tracking systems and outlined three approaches to overcome these problems: motivating higher quality contributions, identifying useful contributions, and shifting contributors' attention from conversational content to important informational content. The remainder of this dissertation is organized as follows. Chapter 2 presents background on previous work studying and improving productivity in bug tracking systems, together with a qualitative investigation of the factors that affect productivity. Chapter 3 motivates and presents our empirical investigation of the use of game mechanisms in Stack Overflow and its usage in bug tracking systems. Chapter 4 presents our bug report summarization approach and its evaluation. Chapter 5 presents the novel bug tracking system we propose, that transforms conversational content into informational content, and its evaluation. Finally, Chapter 6 summarizes the main findings and limitations of our work, the future directions that our work motivates, together with our vision for the next generation of bug tracking systems.

1.5 Publications

This dissertation contains material from the following publications:

- Rafael Lotufo, Krzysztof Czarnecki. Improving Bug Report Comprehension. Technical Report, *University of Waterloo*, 2012.
- Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. Best paper in *International Conference on Software Maintenance*. IEEE, September 2012.

- Rafael Lotufo, Leonardo Passos, and Krzysztof Czarnecki. Towards improving bug tracking systems with game mechanisms. In *9th Working Conference on Mining Software Repositories (MSR'12)*, Zurich, Switzerland, June 2012. IEEE.

Chapter 2

Bug Tracking Systems are Noisy

This chapter investigates how noise—unreliable, unseful, and disorganized information—in bug reports affects software development communities. We start by discussing the importance of reliable information in bug reports and by showing how low-quality contributions significantly affect productivity in bug tracking systems. We then present a qualitative study on a total of 55 bug reports showing the sources of noise in bug reports. Finally, we present the state-of-the-art in overcoming the issues caused by noisy bug reports.

2.1 The Importance of Dependable Bug Reports

2.1.1 Bug Reports are Essential for Software Development

Bug tracking systems have been used since 1970 as a collaborative ecosystem to report and resolve bugs. Typically, bug reports are filed by users or developers when they encounter a system failure. Since virtually every software development project has undiscovered bugs, bug reports are essential for projects to receive feedback from users who uncover such failures. Important open-source bug tracking systems receive hundreds of new bug reports per day [5, 29].

Bug tracking systems have become ever more important, given the increasing number of people knowledgeable and interested in software development, in particular open source software development [9, 55, 67]. Opposed to most closed-source projects, in open-source projects, bug tracking systems are not only accessible, but are also open to outside participation, and are often used by projects to crowd-source quality assurance efforts for development of alpha, beta, and final releases of their products.

Software reuse has also increased the importance of bug tracking systems. Upstream bugs—which are caused by errors in software from different projects or branches—are another common reason for developers to consult bug reports. Since Webkit is the HTML rendering engine adopted by Chrome, for example, many of the bugs from the Chrome project refer to Webkit bugs. The same is valid for software components that have been repackaged and ported to Debian, Ubuntu, or other operating systems. GitHub's success confirms this trend of code sharing and reuse.[1] Thus, as developers share more and more code, they also share more and more bugs, requiring users to work not only on bug reports of their own software, but also on bug reports from upstream channels.

2.1.2 Bug Resolution Depends on Reliable Information

Bug reports, in general, end up with one of three different resolution statuses: *fixed*, *won't fix*, and *duplicate*. When submitted to bug tracking systems, bug reports generally contain a summary of the failure, the environment settings in which it was triggered, the steps to reproduce it, and possibly other *diagnostic* information. Other contributors often share their own diagnostic data about a failure, allowing the community to better characterize the bug and understand its causes. Using the data contained in the bug description and comments, the development team will try to diagnose and confirm the failure. Only once others have managed to reproduce the reported failure will the development team confirm the bug's existence and proceed with its correction [6], often resulting in a fixed bug report; when the failure cannot be reproduced or when there is no agreement that it is a relevant or real failure, the bug often ends up as 'won't fix'; and when the failure is found to have already been reported in another bug report, the bug is closed as a duplicate. Unresolved bug reports remain *open*.

For bug reports to be fixed, it is, thus, important that contributors are able to *easily consult* bug reports and find the information they are looking for, or at least quickly note that such information is not present. Most importantly, developers must be able to *quickly find diagnostic information* [10] so that they can reproduce the bug, or quickly identify if the issue has already been reported in another bug report. They must be able to clearly understand the current knowledge about the bug, the scenarios that have succeeded and the scenarios that have failed, so that they can continue the investigation from where it was left off by other contributors. Developers also need to reason about the execution results of test cases for a series of software revisions, for example, to conclude that a bug is a regression. Contributors must also be able to find and compare the solutions or

[1] http://www.wired.com/opinion/2013/03/github/

workarounds, if any, to the reported issues, together with their advantages, disadvantages, and the environment settings for which they are applicable to.

Diagnosing and confirming the bug depends mostly on the usefulness of information in bug reports and how easy it is to understand such information. When the development team cannot reproduce the bug with the provided information, developers often ask the reporter for more information. In fact, Breu et al. [15] find that 48% of questions in bug reports come from developers asking for missing information and for clarification and further details of existing information; Hooimeijer et al. [35] find that the probability of a bug being fixed increases the faster these questions are answered. Thus, a bug report with missing, irrelevant, confusing, or hard-to-find information is likely to be a noisy bug report.

2.2 The Reality of Noisy Bug Reports

Bug tracking systems are ripe with noisy bug reports. Evidence of this comes mostly from interviews with developers. Developers report, for example, that the number of bugs they receive is far greater than what the development team can handle [5, 29]. As a result, most of the content in bug reports isn't verified or checked. Developers also claim that the large number of bug report reassignments that commonly occur is also caused by bug reports with unclear, unreliable information [30].

Another important evidence of noisy bug reports comes from our study on bug report summarization, which we will present in Chapter 4. From a survey of 58 open-source developers, 80% of them affirmed that they consider that summarization is very important to help them filter through much irrelevant data in bug reports. One developer even affirmed that he uses his own heuristics to filter through comments, looking first at comments sent from high profile contributors. The Debian community also recognizes this problem and allows users to set a summary for bug reports. They claim: "*This is useful in cases where . . . the bug has many comments which make it difficult to identify the actual problem*".[2]

We now present, in more detail, the various factors that make bug reports noisy. These findings result from a literature review of work on improving bug tracking systems, combined with a qualitative investigation of bug report content. We first describe the methodology for this qualitative investigation and then proceed to present our findings.

[2] http://www.debian.org/Bugs/server-control#summary

2.2.1 Qualitative Investigation Method

Our qualitative investigation uses *grounded theory*, as proposed by Strauss and Corbin [58], to attempt to answer the question "*How does knowledge evolve in bug reports?*". By reading the bug report from top to bottom and marking the most relevant concepts to try and answer this question, we get a better understanding of the types of noise found in bug reports and of the difficulties bug report readers face when trying to reconstruct the bug resolution process from the linear sequence of comments. We find that noise comes from not only low-quality comments but also from the problem-solving nature of the bug resolution process, itself.

We start by investigating a random sample of 40 bug reports, with at least 10 comments each, from the Chrome, Launchpad, Mozilla, and Debian bug tracking systems. After going through this random sample, we move to a theoretical sampling approach, as recommended by Strauss and Corbin, and stop sampling when new samples do not deepen the understanding of the problem, but fit into our current theory. The final sample of bugs we have used for this study was 15 from Chrome, 13 from Launchpad, 16 from Mozilla, and 11 from Debian.

2.2.2 Low Bug Report Quality

As we have introduced, one of the most acknowledged sources of noise in bug reports are low-quality contributions, which result in a low bug report fix-rate [10, 35]. Prior work on evaluating bug report quality finds that it is very common that bug report descriptions do not include all information expected by developers. Bettenburg et al. [10] find that the fields in a bug report that are most valued by developers are *steps to reproduce*, *expected behavior*, and *actual behavior*, fields that also are the most neglected by bug reporters, and are often missing, incomplete, or incorrect. They also find that bug reports often lack the appropriate attributes in describing the execution environment, such as the versions of all related software and hardware.

Breu et al. [15] provide more evidence of the significant gap between the information provided in bug reports and the information that developers require to resolve such reports. They surveyed the communication among bug report stakeholders by analyzing the questions from 600 bug reports from Mozilla and Eclipse projects and find that 45.7% of questions in bug reports are aimed at improving the understanding of bug reports, either asking for more information of for clarification.

2.2.3 The Ineffectiveness of Collaborative Filtering

Most bug tracking systems today allow contributors to vote for the bugs they consider to be important to be resolved. While this is intended to assist developers in prioritizing bugs and identify the bugs that are more important to be resolved, we find that such collaborative filtering mechanism is not as effective as it could be and that many developers intentionally ignore such signal.

We perform a systematic review of discussions about voting in the Mozilla repository by searching for comments containing the words 'vote' and 'voting'. We find that the community itself is not clear on the benefits of votes or how to take advantage of them. Some developers claim that they try to take votes into consideration when selecting a bug report to fix, but they find that votes don't correspond well to bug severity, as perceived by the development team, and contributors continue to vote for bug reports even after developers have decided on not fixing a bug. Given this status quo for votes, many contributors don't bother to vote—some aren't even aware of its possibility. As a result, there is a large number of bug reports that are fixed without having received any votes and a large number of bug reports that are not fixed, even after receiving many votes. Thus, as we will show in more detail in Section 3.6.3, the number of votes a bug receives is not a strong indication if the bug will eventually be fixed or not.

2.2.4 Conflicts and Wars are Noisy

Conflicts in bug tracking systems are also a large source of off-topic, irrelevant content. Such conflicts are fueled by the uncertainty of failures, the conflicts of interests between developers and users, power disputes among developers, and the discussion medium itself [9, 33, 38]. In much of these cases, well-known developers often post large amounts of unfair critique to less known contributors. As this can discourage participation from beginners [9, 25], projects seeking to grow their communities try to manage such conflicts. Mozilla, in fact, has a team specialized in conflict resolution.[3] The following is a comment from a Mozilla contributor trying to resolve a certain conflict:

> *"I think this is a good example of how free software projects often fail to receive bug reports well . . . It's basically insulting to the reporter, implying that his time is of little value . . . People who take the time to clearly report bugs are a tiny fraction of a percent of the userbase. Making them jump through hoops*

[3] https://wiki.mozilla.org/Conductors

> *repeatedly will quickly dissuade them from reporting any more bugs . . . This kind of problem is especially prevalent in large projects, such as Mozilla...".*[4]

Conflits, therefore, significantly increase the noise in bug reports, since such ramblings have little to do with the reported error itself or contribute to its resolution. Furthermore, it often spurs an endless stream of further off-topic comments, resembling a pointless debate [19].

2.2.5 Bug Tracking Systems Have Many Novices

Most contributors initiate their participation in open source projects in bug tracking systems, often helping out in bug diagnosis, triaging, or submitting patches to fix bugs. Aspiring developers, thus, use bug tracking systems as an opportunity to establish trust and credibility within a project's community [39, 56]. Indeed, Sinha et al. [56] find that, other than personal ties with core developers, activity in bug tracking systems is the most significant factor for being accepted as a developer with commit privileges.

Bug tracking systems, therefore, have a large number of novices and inexperienced contributors trying get familiarized with and showing their value to the project and its community. This large number of novices, however, serve as a additional fuel for low quality bug reports and for conflicts [9, 61], adding further noise to bug tracking systems.

2.2.6 Problem-Solving is Noisy by Nature

The most interesting finding of our qualitative investigation is that bug reports are noisy not only due to low-quality and off-topic content, but that the problem-solving nature of bug resolution is noisy itself. Since contributors often take many directions for diagnosing and resolving a bug, many of these directions lead to unfruitful findings and are, thus, abandoned. In order to be able to understand a bug's current status and the decisions taken for its resolution, contributors must try to build up the current knowledge about the bug from the log of events stored in the comments.

In general, we find that comments revolve around three types of information about a bug: *claims*, *hypotheses*, and *proposals*. A claim is a general affirmation made by a participant, such as "*I can reproduce this on 4.11*", or "*The function returns -1 for me*". Participants post hypotheses about, for example, the cause of the bug or a possible solution:

[4] Source: https://bugzilla.mozilla.org/show_bug.cgi?id=254714#c93

"*since I cannot reproduce this on Wheezy, the problem might be caused by the render_screen function*" or "*I think that removing that call should fix the crash*". As for proposals, they are generally used when discussing different approaches to resolve an issue: "*I think we should move the button at the end of the screen, not at the top*", or "*How about using json instead of xml?*".

The information introduced by claims, hypotheses, and proposals evolve over time. Participants frequently post *evaluation* comments that confirm or dispute previous claims, support or reject previous hypotheses, and evaluate previous proposals. We also find that at least 27% of comments in bug reports result from the evaluation of other comments. Gasser and Ripoche [26] find that the bug resolution process is much about the 'stabilization' of the knowledge about a bug, which can only be achieved through the evaluation of claims, hypotheses, and proposals.

Readers, therefore, need to keep track of each of these evolving threads of knowledge. It is only from understanding these threads that a reader will be able to comprehend, for example, what the outstanding issues preventing a bug's resolution are, what the different verified solutions or workarounds are, and for which environments each of these solutions and workarounds are suited for.

2.2.7 Comments Hinder Ability to Reason and Analyze Data

Our qualitative study has shown that knowledge about bug reports evolve in a complex, non-linear fashion. The simplistic, linear sequence of comments in bug reports, however, is a poor way to organize such knowledge.

A bug report often contains a variety of diagnostic information, hypotheses, and solution proprosals. Ko and Chilana [38], however, find the lack of a better model for information in bug reports hinders the ability to view, compare, and make decisions about these different design proposals, test results, critiques, and software behavior, which is fundamental for making progress in bug resolution.

The linear thread of comments also makes it difficult for users to work with a bug report, the higher its number of contributors. We've found that, for bug reports with more than 3 contributors, the topic of comments changes frequently, since each contributor has a different perspective about the bug or purpose for contributing: some contributors might be concerned about convincing others that the priority of the bug is low, for example; others might be trying to present evidence that the issue also occurs in other environments than those already known; and other contributors might be trying to coordinate who will be responsible for resolving the bug. As a result, the more contributors discussing a bug report, the more the conversation becomes interwoven and multi-threaded. Since comments are only indexed by author and submission time, the linear thread of comments makes it difficult for readers to follow a particular thread of topic they are interested in, requiring them to read through the whole bug report and to keep track of multiple contexts [40, 43].

2.3 Noisy Bug Report Management, State of the Art

We now present existing work on managing noise to improve productivity in bug tracking systems.

2.3.1 Detecting Missing Information

Prior work on evaluating bug report quality looks primarily at the presence or absence of structural attributes of bug reports. Bettenburg et al. [10] use natural language processing, machine learning, and heuristics, to detect the lack of steps to reproduce, stack traces, and attachments and create a tool that is able to identify such missing attributes and ask the bug reporter for such missing information.

This automated approach, however, treats all bug reports as equal and is not able to discern that bugs of some particular nature might require diagnostic details that are irrelevant to bugs of other nature. A computer's timezone setting, for example, is a specific diagnostic information that is relevant for bugs about event scheduling, but likely to be irrelevant for bugs about memory leaks. Thus, although useful, Bettenburg's approach should still require developers to ask for specific debugging information for most bugs [15, 35].

2.3.2 Duplicate Bugs

Also important to reducing noise in bug tracking systems is the detection of duplicate bug reports. Wang et al. [64] present the current, most effective, approach to identify duplicate bug reports. They build on the previous duplicate bug report detection attempts using only the textual bug description [32, 54] and use both textual descriptions and bug execution traces to disambiguate bug reports. The proposed solution first calculates bug report textual similarity and execution trace similarity and then uses heuristics to combine the two measures. The approach is reported to detect as much as 90% of duplicate bug reports.

Bettenburg et al. [11], however, presents a study claiming that duplicate bug reports help bug resolution productivity. Although most projects still cancel out duplicate bug reports, they claim that duplicate bug reports are valuable since they bring additional information to existing bug reports. This suggests that it would be useful to automatically combine the information from duplicate bug reports to form one master bug report. However, the linear thread of comments makes it difficult to perform such operation, since this would require merging multiple related, but independent, conversations.

2.3.3 Enriching Bug Reports with External Content

To aid answering questions that commonly occur during bug resolution, Ankolekar et al. [3] enrich bugs with links to external resources mentioned in bug report comments, such as commit logs and developer profiles. This approach, thus, complements data in bug reports with external information and suppresses the need of users to leave the bug tracking system to search for such information. It does not, however, solve the problem of locating or reasoning about relevant information within bug reports. Similary, to help developers understand a bug report's context, Cubranic and Murphy[63] propose a recommendation system to identify artifacts related to particular bug reports, such as design documents and source code files.

2.3.4 Identifying Which Bugs Should be Fixed

Guo et al. [29] and Hooimeijer et al. [35] study the characteristics of bug reports that are chosen to be fixed by developers. The correlations they find suggest that developers choose to fix bugs opened by well-known contributors, and that severity is an decision important factor. Their prediction model achieves an accuracy rate of around 60% in detecting bug reports that will be fixed.

While this predictor has reasonable accuracy, using such a tool to help developers prioritize bug reports will only re-enforce that bugs should be prioritized as they currently are today—by contributor reputation and proximity—which may not be optimal at all.

2.3.5 Improving Bug Report Readability

Rastkar et al. [52] recognize the similarity between bug report messages and email threads and use a preexisting summarization technique, created to summarize email threads and conversations [48], to summarize bug reports. The approach creates an *extractive summary*, which is built by selecting a set of sentences from the original bug report, to compose an allegedly informative and cohesive summary. The approach uses a logistic regression classifier that is trained on a corpus of manually created reference bug report summaries—*golden summaries*.

The results presented by Rastkar et al., however, show that the quality of the generated summaries is sensitive to the training corpus, suggesting that the approach is mostly applicable when trained on a corpus of golden summaries from the target bug tracking system and that the training corpus should be adjusted to reflect the types and nature of bugs as a project evolves. Since creating golden summaries requires significant manual effort and should be done by experts, creating a reasonably-sized training set of golden summaries could be considered an impediment for the use of such technique.

Dit [21] proposes a recommendation engine to find the comments that a comment is related to, to improve the readability of bug reports. They do not implement such a system, only acknowledging that detecting these links is a difficult problem.

Guo et al. [30], after studying the causes of bug report reassignements, suggest that bug tracking systems should include better visualizations of reassignment patterns to help developers identify problematic patterns. This is another indication of how only unstructed, textual information hinders reasoning in bug reports.

2.4 Summary

In this chapter, we have shown the importance of useful, dependable information in bug reports and exposed the effects and the causes of noise in bug tracking systems. We have also hypothesized that the most evident symptoms of noise in bug tracking systems is the low fix-rate of bugs and the value that contributors give to tools that help them find and work with relevant information. We have also found that the two primary sources of noise in bug reports are low-quality content and the difficulty of fitting the non-linear problem-solving nature of bug resolution into a linear sequence of comments. Furthermore, the large numbers of inexperienced contributors participating in bug tracking systems increase conflicts and low-quality contributions, adding further noise to bug reports. Existing work to address such issues, however, merely scratch the surface of the problem. Work on improving contribution quality is only able to suggest contributors to fill in missing information in bug report descriptions or pull in related information about resources mentioned in bug reports. Work on noise reduction is also lacking: existing models for prioritizing bug reports are not ready to be used in practice and current bug report summarization approaches have significantly high usage costs.

Chapter 3

Using Game Mechanisms to Increase Contribution Quality and Filter Out Noise

In this chapter, we present an empirical study of Stack Overflow, investigating if and how game mechanisms can be used to improve contribution quality and filter out noise—unreliable and irrelevant contributions—in collaborative software development ecosystems. Our study finds that reputation and rewards systems—two very common game mechanisms—not only engage contributors to post higher quality contributions, but also to improve existing content by means of peer-reviewing. Another desirable effect of peer-reviewing is noise reduction, since lower-quality contributions are marked as such and left in the background.

Our study also allows us to infer the necessary conditions for Stack Overflow's game mechanisms to produce its benefits. We then propose how these mechanisms could be used in bug tracking systems. To ascertain that the game mechanisms, when applied to bug tracking systems, would produce benefits similar to the ones found in Stack Overflow, we empirically check if the necessary conditions should also be valid for most bug tracking systems.

3.1 The Potential of Game Mechanisms

McGonigal [45] studies games and finds that competition, great challenges, collaboration, and sense of accomplishment are the factors that make games so compelling. McGonigal argues that it is possible to make tasks more compelling and rewarding by introducing small changes—mechanisms—that add game-like characteristics to tasks.

Enriching tasks with game mechanisms can also be effective to spur massive crowdsourcing efforts. The Guardian's "Investigate Your MP" game, for instance, successfully motivated thousands of people to examine hundreds of thousands of documents, uncovering serious irregularities in British Parliamentary expense claims.[1]

We, thus, expect that game mechanisms should have similar effects in software development ecosystems and processes, if appropriately used. We hope that contributors should feel more engaged in executing tasks that are not so well appreciated, such as bug reporting. We are not aware, however, of studies performing such a systematic investigation of the effects of game mechanisms in a software development setting. We investigate, therefore, Stack Overflow, which is one of the first and very successful software development collaboration ecosystem to use game mechanisms.

3.1.1 Reputation and Rewards

Stack Overflow uses a reputation and rewards system as the main drivers of its game mechanisms. Reputation and rewards systems have been used extensively in games and in on-line ecosystems and have been shown to encourage desirable behavior, to increase trust, motivate participation, and push participants to new levels of achievements in on-line communities [20, 22, 43, 45].

Reputation is, most often, a numeric score or label given to players, that can be used to measure how much they have achieved in a game, thus, making it easy to compare the progress of different players. A player with 100,000 reputation points, for example, has achieved much more than a player with 1,000 reputation points; a player with a badge titled 'master samurai ninja' has probably achieved more than a player with a badge titled 'baby amateur ninja'. Reputation has been previously found to increase players' participation levels for two reasons. First of all, by comparing their own reputation with the reputation of more advanced players, users feel the urge to increase their reputation to elevate their self-esteem [44]. Second, reputation also increases participation rates by increasing trust

[1] Source: http://mps-expenses.guardian.co.uk/

among users. Dellarocas [20] finds that the reputation system in eBay measures how much the community trusts a user, thus, encouraging more transactions between users.

In on-line community games, the key to earning reputation is *recognition*: a player's actions have to be recognized by the community in order to receive reputation points. The community generally recognizes contributions based on an evaluation of its quality and usefulness. In eBay, for example, sellers are recognized by buyers when the buyers acknowledge they have received the product in good condition. In Wikipedia, the reputation of a user, although not formal, is generally considered to be the number of contributions made by the user that were not deleted by the community. Thus, an editor is recognized by the community when the content that the editor has contributed is not deleted [1, 2].

As a player's reputation increases during the game, players earn rewards. Rewards are generally privileges that players will appreciate, such as reduced service cost, or privileges that will give the player certain advantages in the game. When the user reaches 100,000 reputation points or receives the 'master samurai ninja' badge, for example, a game could award the player with an invisible sword or give the player the ability to predict the opponent's next moves.

Most games employ a carefully crafted reward system that motivates players seeking rewards to reach for higher reputation levels and, therefore, to engage in more participation. Ducheneaut et al. [22] studies game mechanics in massively multi-player on-line games (MMOG) by investigating World of Warcraft and finds that the design of the rewards system and levels is engineered to keep users in frequent participation and that players participate more actively just before reaching levels that awards them privileges.

3.2 Stack Overflow

Stack Overflow is a web application created for developers that provides a collaborative ecosystem aimed at the resolution of specific computer programming problems. Users post questions about these problems and the community posts answers attempting to resolve them. It is considered one of the most successful technical question-and-answer applications today [43]. Stack Overflow's success has spurred the creation of Stack Exchange, a network of question-and-answer websites using the same application as Stack Overflow, with over 50 installations for a wide variety of communities, such as Statistical Analysis, Physics, Cooking, Photography, and Philosophy.

Mamykina et al. [43] qualitatively and quantitatively characterize Stack Overflow and find that questions are answered in just a few minutes. Through interviews with Stack

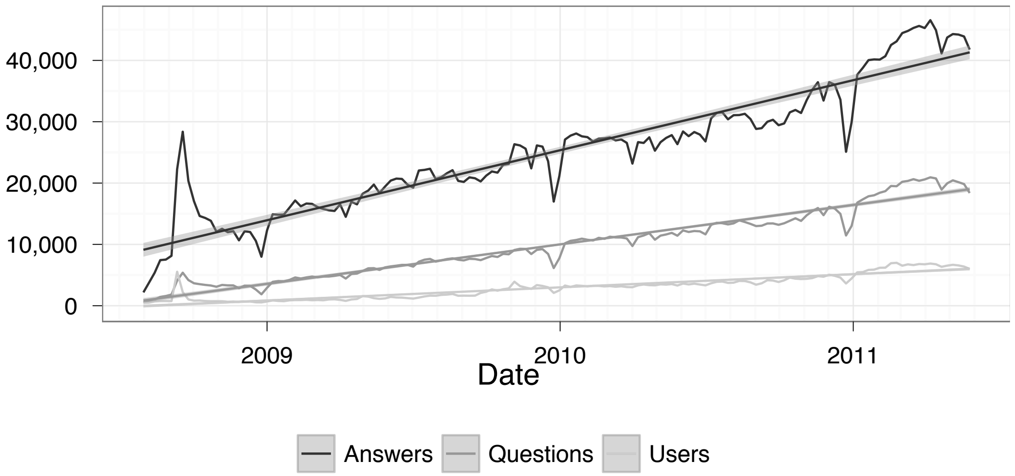


Figure 3.1: Number of new answers, questions, users per week

Overflow's design team and users, they find that Stack Overflow was built with the intention of using productive competition to increase participation and quality.

Figure 3.1 shows how Stack Overflow's usage has grown since its origins in mid-2008. It shows that the number of new answers, questions, and users per week follows an almost linear progression, indicating the total number of answers, questions, and users increases quadratically over time. As of July 2011, Stack Overflow had more than 400k users, 1.7 *mil.* questions, and over 3.5 *mil.* answers.

3.2.1 Stack Overflow Implements a Formal Meritocracy

Arguably, the game mechanisms used in Stack Overflow are one of the top reasons for its success [43]. Registered users start with 1 reputation point and are able to gain more as their contributions are recognized by the community. 'Recognizable' contributions are primarily those of questions and answers: users recognize useful questions and answers by voting, which rewards five and ten reputation points to the submitters of voted questions and answers. Furthermore, the owner of a question can 'accept' an answer as the best, rewarding 15 reputation points to its owner and two reputation points to themselves for selecting a best answer. By doing so, the owner indicates his satisfaction and considers the question has been *resolved.* Users can also *down*-vote questions and answers they consider to be invalid or incorrect, decreasing users' reputation by two points and their own by one—taking one point from voters attempts to assure users only down-vote contributions they really believe have no value. Users can also comment on questions and answers to suggest improvements or ask for clarification. Furthermore, similar to wiki posts, questions and answers can be edited, allowing other contributors to improve them. Finally, users can offer larger amounts of reputation points—*bounties*—for questions they are not satisfied with the resolution of. Users who offer bounties—which are multiples of 50 reputation points—pay them from their own reputation points.

Not all users can perform all these actions, however: they have to *earn each privilege*. New registered users, for instance, can only post questions and answers. They cannot vote, comment, or edit others' posts. Privileges are earned by accumulating reputation points and achieving *reputation levels*: 15 points to vote up; 50 to comment; 125 to vote down, etc. Users need 2000 points to edit others' posts; 3000 points are needed to vote to close or reopen questions.[2] By awarding contributors with privileges as they show their value to the community, Stack Overflow implements a *meritocracy*.

As can be seen, Stack Overflow uses a reputation and rewards systems that aim to spur competition and a sense of accomplishment. Users are expected to compete for the best answers and questions, while earning reputation and privileges should give users a sense of accomplishment and reward. Finally, having well planned reputation levels should challenge users to reach, for example, the 10000 reputation level.

3.2.2 Stack Overflow and Bug Tracking System Similarities

Stack Overflow is appropriate for this investigation, as its community is formed of software developers and its questions have some similarity to bug reports: in both cases, users report problems asking the community to discuss and propose solutions. Stack Overflow's community corroborates this claim, as they consider a "*'language-specific programming problem' ... that exists in code and that can be resolved with correct code*" as the most appropriate type of question the community can help resolve.[3]

We, thus, consider that a question in Stack Overflow is equivalent to a bug report in bug tracking systems: they both provide a description of a software issue for resolution. Furthermore, we consider that answers in Stack Overflow are similar to comments in a bug report, since they are posted by contributors to share their work and knowledge and help resolve the issue.

[2] Find full set of privileges at http://stackoverflow.com/privileges

[3] Source: http://meta.stackoverflow.com/questions/12373

In fact, many software enterprises, such as Facebook and Canonical, use Stack Overflow for crowd-sourcing efforts, engaging the community to answer questions and solve development issues related to using their software API.[4]

3.2.3 Applying Game Mechanisms to Bug Tracking Systems

The mechanisms we investigate and aim to apply to bug tracking systems are:

M1: *rewarding reputation points for good contributions*,

M2: *reducing a user's reputation points for poor contributions*, and

M3: *awarding privileges to users as they reach reputation levels.*

As these mechanisms simply reward and penalize users based on the quality of their contributions as judged by the community, these mechanisms can be easily added to bug tracking systems. For bug tracking systems to implement M1 and M2, they need only to allow users to recognize and down-vote bug reports and comments, the equivalents of Stack Overflow's questions and answers. In fact, most bug tracking systems already allow users to vote for bug reports they consider important to be fixed. To implement M3, bug tracking systems need only to reward existing privileges—such as marking a bug as duplicate and deleting inappropriate comments—to users as they reach certain reputation levels.

3.3 Methodology

We select three of Stack Overflow's game mechanisms—presented in Section 3.2.3—that can be easily added to bug tracking systems and pose four research questions to assess if they can achieve the goal of increasing contribution quality in Stack Overflow. The results of our analysis enable us to identify the *conditions* that allow such mechanisms to achieve the goals in Stack Overflow. Finally, we assess if these conditions are also valid in bug tracking systems, thereby allowing game mechanisms to produce their benefits.

[4] http://askubuntu.com and http://stackoverflow.com/questions/tagged/facebook

3.3.1 Data Sets

To answer our research questions, we analyze Stack Overflow over its three-year lifetime, using the StackApps API, provided by Stack Overflow, to retrieve usage information. Due to restrictions imposed by the API on download allowances, we used simple random sampling to download 80% of the 1.7 *mil.* questions and all 3.5 *mil.* answers for those questions. Finally, we randomly sampled 60k users and downloaded their entire contribution timeline—all of the questions, answers, comments, and edits they had posted.

Our data sets for bug tracking systems comprise of 12k bug reports for Android from Nov-2007 to May-2011, 50k bug reports for Chrome from Aug-2008 to Jun-2010, 50k for Launchpad from Jan-2008 to May-2011, and 140k for Mozilla from Jan-2008 to Jun-2010.

3.3.2 Estimating Reputation Over Time

As the StackApps API does not provide the reputation of users at certain points in time, which is a crucial information for many of our questions, we estimate this value using the number of up and down votes users have received for their questions and answers, and the number of times their answers were accepted as the best solution. Given a user u, his reputation at time t is estimated as in (3.1), where $A_{u,t}$ and $Q_{u,t}$ are the sets of all answers and questions posted by user u until time t; functions $V_{\uparrow}$ and $V_{\downarrow}$ return the number of up and down votes of sets of questions and answers; and function $V_{\checkmark}$ returns the number of accepted answers for a set of answers. As estimating a user's reputation requires the data for all questions and answers posted by a user, for questions requiring this estimation, we limit our investigations to users in our timeline data set.

$$R_{u,t} = 10V_{\uparrow}(A_{u,t}) + 5V_{\uparrow}(Q_{u,t}) - 2V_{\downarrow}(A_{u,t} \cup Q_{u,t}) + 15V_{\checkmark}(A_{u,t}) \tag{3.1}$$

3.3.3 Correlations

When comparing certain metrics dependent on reputation, we discretize reputation scores using the reputations levels selected by Stack Overflow to award privileges: 250, 500, 1000, 1500, 2000, 3000, 5000, 10000, 15000 and 20000. When testing for rank correlations, we use the non-parametric Spearman's test; to check if two independent samples contain equally large values, we rely on the non-parametric MannWhitney U test [7]. Even though all our statistical tests were significant, for each test we present the p-value to support our claims.

3.4 Research Questions

We pose three questions to assess if game mechanisms in Stack Overflow can be used to increase contribution quality and one question to investigate if game mechanisms can be used to reduce noise in Stack Overflow.

Our questsions attempt to look for *causation* relations between rewards and reputation and the benefits we seek with game mechanisms. To answer these questions, we have carefully identified tests that, to our knowledge, when positive, indicate a high probability of causation. We consider, thus, that such causations are *suggest* that causation is possible. Nevertheless, this is a threat to validity, that we expose further in Section 3.7.

3.4.1 Increasing Contribution Quality

To investigate whether rewards are effective in increasing contribution quality, we ask:

RQ 1 *Does the reward system drive contributors to post better answers?*

Peer-reviewing can also bring further improvements to contributions. We, therefore, ask:

RQ 2 *Does the reward system increase peer-reviewing frequency?*

To keep users motivated in participation and avoid common conflicts found in open-source discussion forums [25], conflicts must be managed. As moderation should allow inadequate interactions to be detected as soon as possible and should not depend on the attention of few moderators, we ask:

RQ 3 *Does the reputation and rewards system contribute to create an agile and dependable moderation system?*

3.4.2 Reducing Noise

Filtering out low quality contributions can significantly reduce noise. Filtering noise in bug reports is important so that users are not distracted from looking at unreliable or unuseful information.

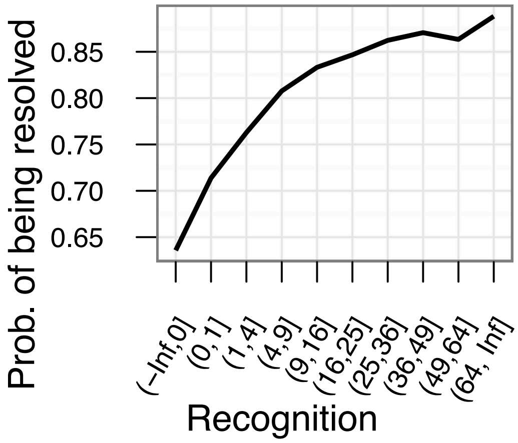


Figure 3.2: Resolution probability

We seek, therefore, a signal that accurately identifies useful contributions. Two candidate signals are the reputation of the contribution's owner and the number of recognition-votes a contribution receives. This leads us to our final research question:

RQ 4 *Are reputation or recognition-votes good signals for identifying useful contributions?*

3.5 Findings

3.5.1 Rewards motivate better answers

RQ 1 asks if rewards drive contributors to post better answers. We find that questions offering higher rewards receive more answers and have a higher likelihood of being resolved. Questions offering bounties—large amounts reputation—receive 50% more answers than non-bounty questions—bounty questions receive an average of 3.34 answers (median of 3), while non-bounty questions receive only 2.53 (median of 2)—and bounty questions are 10% more likely to be resolved than non-bounty questions—all claims are substantiated by the Mann-Whitney U test, with p-values $< 2.2 \times 10^{-16}$.

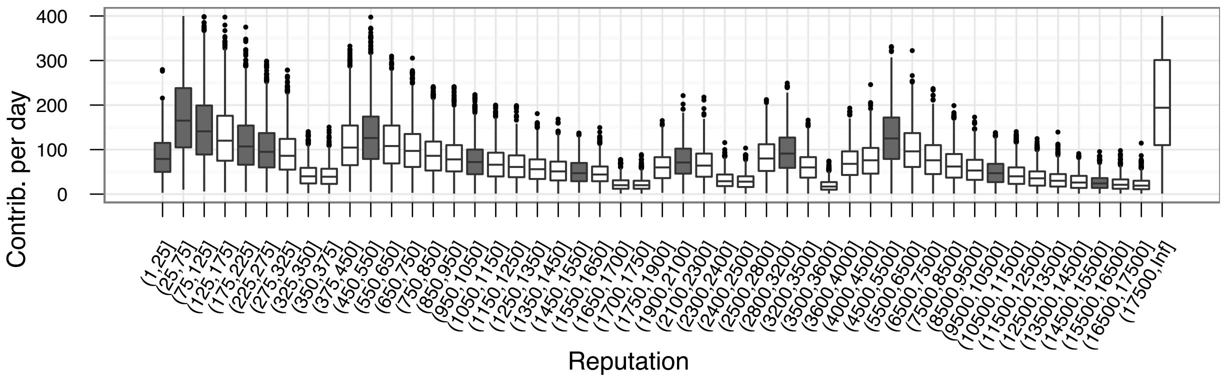


Figure 3.3: Contribution frequency by reputation, showing that users contribute more when they are close to receive new privileges.

We also find that questions with higher reward-potential also receive more answers and are more likely to be resolved, even when not offering bounties. Considering a question's reward is the sum of the number of recognition-votes its answers has received, a correlation rank (ρ) of 0.52 between this value and a question's number of recognition-votes shows that a question's number of recognition-votes is a good indication of its reward potential. Then, similar to bounty questions, non-bounty questions with more reward-potential receive more answers ($\rho = 0.34$), and are also more likely to be resolved ($\rho = 0.13$)—in all cases, the p-values were $< 2.2 \times 10^{-16}$. Figure 3.2 shows how the probability of a question being resolved increases with its reward-potential and confirms these findings.

These correlations indicate that resolution likelihood increases as a result of rewards attracting more participation. Increased participation, however, may spur a large number of low-quality answers. Instead, we find that users restrain themselves from submitting low-quality answers to avoid receiving down-votes: there is a 63% chance of users deleting their own low-quality contributions that received 3 or more down-votes. Consequently, **as users are motivated to earn reputation points, rewarding good contributions and penalizing bad contributions is an effective means to improve contribution quality**.

3.5.2 Rewards Increase Peer-Reviewing

RQ 2 asks if the reward system increases peer-reviewing frequency. By peer-review, we consider contributions that are the result of an evaluation of answers and questions, with suggestions for improvement. In Stack Overflow, we consider all edits to questions and answers to be improvements. Comments are also used as an instrument of peer-reviewing: a sample of 400 comments chosen randomly shows that 52% of them suggest improvements to others' questions and answers, bringing alternative solutions, additional information, and explanations of why answers are incorrect. Thus, when looking at frequencies of comments and edits, we consider that more comments and edits implies more peer-reviewing.

Ducheneaut et al. [22] finds a convincing indication that game mechanisms motivate higher levels of participation. They find that players spend more time playing games when they are are close to achieving an important milestone in the game. This suggests that players, eager to achieve such milestones, put significant additional effort—and often time—into the game.

We also find similar usage patterns in Stack Overflow when looking for increases in participation levels when users are close to reputation levels that award privileges. Figure 3.3 presents the number of answers and questions submitted to Stack Overflow per day, by user reputation. The figure shows increases in contribution frequency just before reputation scores that award privileges—reputation ranges in which privileges are awarded are colored in gray. For example, contribution frequency is tripled—from 40 to 120—to earn privileges rewarded at 500 points. Our data, however, does not allow us to infer if users increase contribution frequency as a sprint to gain privileges, or if users are motivated to increase contribution frequency as a result of receiving such privileges. Nevertheless, every increase in contribution frequency occurs close to a reputation level that awards privileges. This correlation indicates that users are interested in gaining privileges, resulting in increased participation frequency.

More importantly, a similar analysis shows that rewards also encourages more peer-reviewing. As shown in Figure 3.4, peer-reviewing frequency increases for the exact same reputation levels in which we found increases in contribution frequencies in Figure 3.3, showing **that the rewards system also motivates increases in peer-reviewing frequency**.

The increases in peer-reviewing and contribution frequencies for the same reputation levels suggest that **peer-reviewing occurs as a result of the contribution process**. A high correlation between contribution and peer-reviewing rates—Spearman's $\rho = 0.69$, p-value $< 2.2 \times 10^{-16}$—supports this and indicates a dependency between the two rates: as

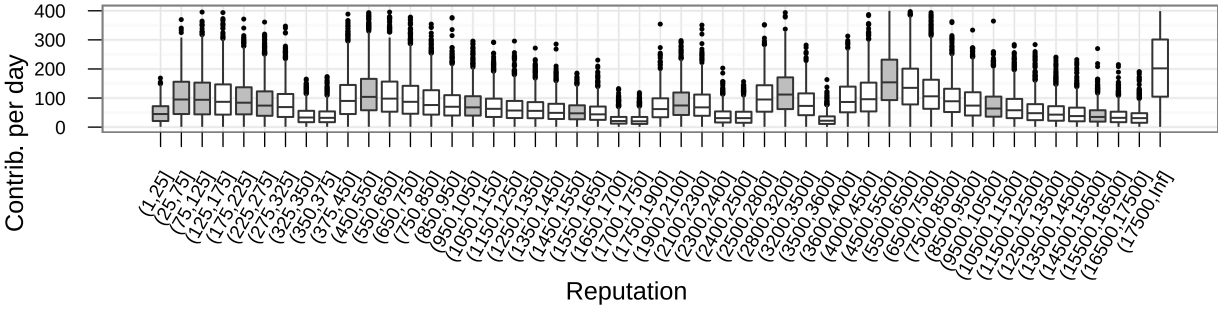


Figure 3.4: Peer-reviewing frequency by reputation, showing that users contribute more when they are close to receive new privileges.

users contribute, they evaluate their peer's contributions, offering suggestions for improvements and corrections. Users who do not evaluate previous contributions before posting their own and submit inappropriate contributions, such as duplicate answers, suffer the risk of being down-voted or flagged.

Not All Rewards are Equally Attractive

Besides showing strong indications of the influence of rewards in increasing contribution frequency, these results also show that not all reputation levels awarding privileges result in increases in contribution frequency. This suggests the importance of setting up reputation levels at intervals and with privileges that are compelling to the community. Curiously, all reputation levels that had an increase in contribution frequency reward privileges related to reviewing and moderating other user's contributions: commenting on other's posts; retagging questions; editing other's posts, voting to approve editions; voting to close or reopen questions; voting to approve tag wiki edits. Reputation levels awarding privileges not related to moderating or reviewing did not show an increase in contribution frequency—for example, reduced advertising, voting to close one's own questions, creating new tags. This correlates with Bergquist's claims that contributors in open-source often seek reputation in order to assert relationships of power over lower-reputation users [9].

As can be seen, rewards are a central point to Stack Overflow in keeping its users active: without rewards, users' participation interest would not be refueled, and would simply decrease. Finally, these results also indicate that the clear road-map provided by Stack Overflow of how users can gain reputation and privileges is also beneficial to

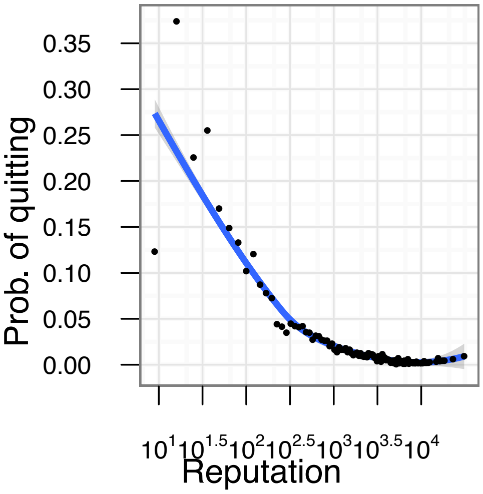


Figure 3.5: Resolution probability

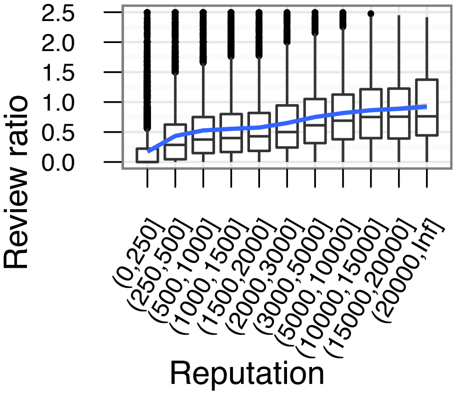


Figure 3.6: Review ratio

motivating participation: without such a road-map, users will not know the conditions for gaining rewards or if they are close to being rewarded.

Higher Reputation, Higher Commitment

We also find that users perform more peer-reviewing the higher their reputation. Here, we consider as peer-reviews all comments and edits made to answers or questions that a contributor did not ask or answer, and therefore has no stake in. We then calculate the *review ratio* as the number of reviews each user has posted in a week, divided by the number of answers and questions they posted in that same week, thereby effectively comparing the number of reviews to the number of answers and questions.

Figure 3.6 presents box-plots of the review ratio of users by reputation, along with a LOESS regression, and shows that it increases with higher reputation. Similar assessment on individual users also finds that this correlation holds for over 85% of users. These results match Maslow's hierarchy of needs [44], suggesting that contributors with low reputation are interested in gaining more reputation: their efforts are focused on posting questions and answers. As users' reputations increase, they are ever more likely to review questions and answers they have no stake in, suggesting that they are more concerned in maintaining the system and evaluating other contributors' posts, instead of publishing their own.

Besides the influence of privilege rewards on contribution frequency, we also find that users with more reputation and privileges are less likely to quit than those with fewer privileges. We consider a user has quit, if he has not posted a question, answer, comment, or edit since the previous 60 days from this investigation. Figure 3.5 shows how the probability of users quitting decreases with their reputation. We calculate the probability of users quitting at reputation r as the number of users that quit at that reputation, divided by the number of users that ever reached reputation r. As Figure 3.5 shows, there is a steep decrease in the probability of quitting from 10 to 300 reputation points, and then a stabilization of this probability below 5% after 300 points. Interestingly, there is a slight increase in the probability of quitting for users with more than 10000 points, suggesting a decrease in user motivation after "completing the game".

These findings suggest that users feel more committed to the community as they gain experience, privileges, and reputation. It is likely that such commitment occurs as a result of the significant efforts they have put in to achieve high reputation levels, and will be, thus, hesitant to leave the community.

3.5.3 Rewards System Creates Dependable Moderation System

To answer RQ 3, we investigate if the reputation and reward system contribute to create an agile and dependable moderation system—qualities we have defined in Section 3.4. Stack Overflow's community is advised to identify and moderate inappropriate contributions: contributions that do not adhere to the community's principles, are offensive, are not the type of questions or answers the community expects, or is a duplicate question or answer. These guidelines were agreed on by the community itself, using Stack Overflow's own meta-discussion forum, which, to encourage participation, uses the same game mechanisms as Stack Overflow.[5]

Stack Overflow's moderation system is tightly tied to its rewards system. As higher reputation offers users more moderation privileges, there is naturally a large number of users with low moderation privileges, and then a progressively smaller number of users with higher moderation privileges—a *population pyramid* of privileges. The first level of moderation is the flagging of contributions, an action that every user with 15 points or more can perform, and brings a flagged contribution to the attention of high reputation moderators. A second moderation level is the down-vote, a judgment that every user with more than 125 reputation points can make. As shown in Section 3.5.1, the down-vote is effective because it discourages users from posting low-quality contributions. The next

[5] http://meta.stackoverflow.com/

moderation level is available to users with more than 3000 points, allowing them to vote to *close*, *reopen* or *migrate* questions to other Stack Exchange sites. These are the users who will be notified when the first-level moderators flag contributions for attention. Next, users with 10000 points can vote to *delete* questions and access other moderation tools.

Although Stack Overflow does not provide data on contributions that suffered moderation, thereby not allowing us to assess its performance, we argue that **its rewards mechanisms, by building a population pyramid of privileges, enable a dependable moderation system**. The system can rely on the attention of the large user base with low moderation privileges to take the first moderation actions on inappropriate contributions, only then relying on a smaller base of high-reputation moderators to take further actions. We also argue that **it is agile, since flagging is available to the large majority of users, increasing the probability of users quickly detecting improper interactions.**

3.5.4 Recognition-Votes Identifies Relevant Content

RQ 4 asks whether reputation or recognition can be used to filter useful contributions. Here we will evaluation if we can use reputation or recognition to filter useful answers.

To evaluate whether reputation or recognition are good signals for identifying useful answers, we rely on an information-retrieval approach to evaluate the precision rates of using reputation and recognition for selecting the best answers. We calculate $P@n$ [17], as shown in Equation (3.2), to measure the percentage of questions q whose best answer can be found by looking at its top n answers, ranked by σ (reputation or recognition). As does Stack Overflow, we consider the answer accepted by the question's owner as the best answer.

$$P@n = \frac{|\{q \in \mathit{questions} : \text{best\_answer}(q) \in \text{top}(n, q, \sigma)\}|}{|\mathit{questions}|} \tag{3.2}$$

Figure 3.7 shows $P@n$ for questions which have an accepted answer. It compares the precision when ranking answers by number of recognition-votes and by user reputation at the time the question was posted. As shown, recognition has considerably higher precision rates compared to reputation: the answer with most votes is the best answer in around 70% of questions, compared to 50% for user reputation. This indicates that **both reputation and recognition-votes are good predictors of useful contributions, with recognition-votes still having as much as 20% higher precision**, considering the top answer.

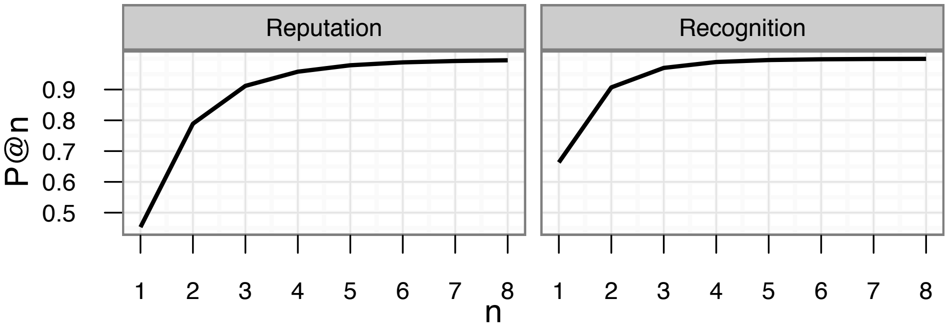


Figure 3.7: $P@n$ for reputation and votes

3.5.5 Summary of Findings

Our investigations have shown that in Stack Overflow, game mechanisms can increase content quality by achieving four outcomes:

O1: *increasing contribution quality*;

O2: *increasing peer-reviewing frequency*;

O3: *creating a dependable and agile moderation system*; and

O4: *filtering out low-quality content.*

By inviting users to compete for reputation points and rewards, Stack Overflow manages to push users to post more and higher quality contributions, as presented in RQ 1. At the same time, the down-vote seems to keep low quality contributions reduced, effectively increasing the overall content quality in Stack Overflow.

In Stack Overflow, peer-reviewing also increases contribution quality through comments or edits. As shown in RQ 2, the increased participation rates, caused by the strive to gain more reputation and rewards, result in higher peer-reviewing rates, since in Stack Overflow, peer-reviewing seems to be a natural by-product of participation: in order to avoid submitting a duplicate answer and receive a down-vote, users should evaluate existing answers to a question before posting a new answer. We have also shown, however, that not all rewards seem to be compelling to users and that, thus, it is important to carefully craft the rewards system with the rewards that will attract high user interest.

We also claim that conflict management is important to not only keep users engaged in submitting high-quality contributions, but also to keep the ecosystem clean of such noise. As presented in RQ 3, Stack Overflow's rewards system creates a population pyramid of privileges that enables a moderation system that should be able to keep conflicts in check since it is dependable and agile. It is dependable, since the moderation system does not rely on the attention of a small subset of users, as the majority of users have flagging privileges. Agility also comes from the fact that majority of users have flagging privileges, since it increases the probability of users quickly detecting inappropriate contributions.

To reduce the noise and allow users to focus on high-quality content, RQ 4 has shown that recognition-votes can be used as a signal to identify useful contributions and is a better signal than the author's reputation. As a result, users can limit themselves to look only at the top-voted answers in a question for the best answers.

3.6 Mapping Back to Bug Tracking Systems

We now evaluate if, once Stack Overflow's game mechanisms are applied to bug tracking systems, as described in Section 3.2.3, they will increase contribution quality and reduce noise as in Stack Overflow, as summarized in Section 3.5.5. To do so, we identify the conditions to achieve the four outcomes (O1, O2, O3, and O4) and if these conditions are valid for bug tracking systems.

3.6.1 Conditions for Achieving Game Mechanism's Benefits

The findings for our research questions (Section 3.5) allow us to identify the conditions that allow game mechanisms M1, M2, and M3 (rewarding reputation points for good contributions, reducing reputation point for poor contributions, and awarding privileges as users reach higher reputation levels) to produce outcomes O1, O2, O3, and O4 (increasing contribution quality, increasing peer-review frequency, creating agile and dependable moderation system, and filtering low-quality contributions) required to increase contribution quality and filter out noise in Stack Overflow.

Figure 3.8 summarizes these findings, with arrows linking each outcome to the game mechanisms and the conditions required to achieve it.

To increase contribution quality (O1), as found in RQ 1, the mechanisms of *rewarding users for good contributions* (M1) and *penalizing them for poor contributions* (M2)

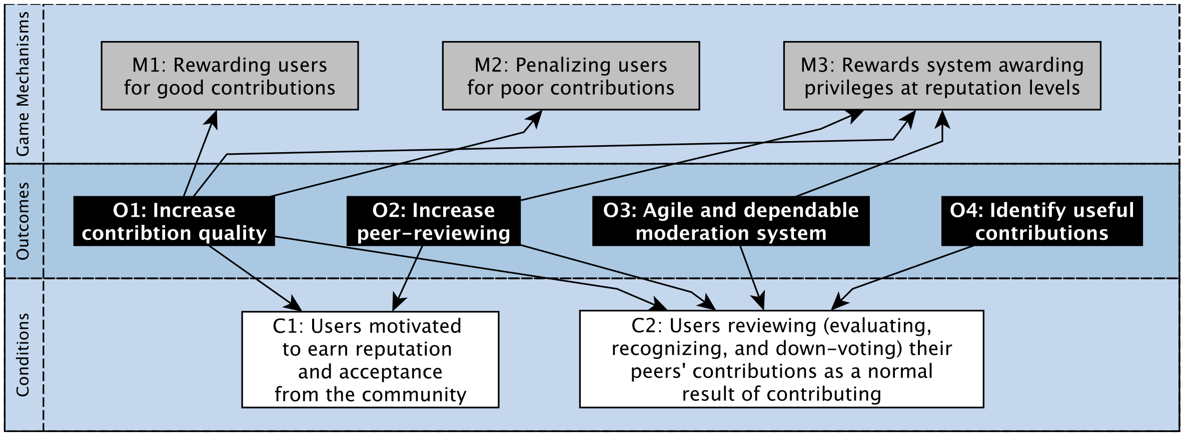

Game Mechanisms
Outcomes
Conditions
M1: Rewarding users for good contributions
M2: Penalizing users for poor contributions
M3: Rewards system awarding privileges at reputation levels
O1: Increase contribtion quality
O2: Increase peer-reviewing
O3: Agile and dependable moderation system
O4: Identify useful contributions
C1: Users motivated to earn reputation and acceptance from the community
C2: Users reviewing (evaluating, recognizing, and down-voting) their peers' contributions as a normal result of contributing

Figure 3.8: Conditions for game mechanisms to achieve goals

increases contribution quality. This occurs because *users are motivated to earn reputation points* (C1) and will, thus, compete to send high-quality answers to gain recognition. Furthermore, in order to maintain users' interests in contributing, it is important that *the community evaluates and recognizes good contributions* (C2) so that users are rewarded.

To increase peer-reviewing frequency (O2), RQ 2 has shown that, first of all, users need to be interested in gaining reputation and the offered rewards (C1) so that they are engaged into higher levels of participation. As a result, since peer-reviewing occurs as a result of normal participation and contribution (C2), higher participation levels will induce higher peer-reviewing levels.

As seen in RQ 3, as *users evaluate and recognize good contributions* (C3), the well-crafted rewards system *creates a population pyramid of privileges* (M3) that enables an agile and dependable moderation system (O3).

As for identifying useful contributions, RQ 4 shows that, as *users routinely evaluate and recognize contributions they find to be useful* (M1 and C3), a question or answer's number of recognition-votes offer a very good signal to judge its usefulness (O4).

3.6.2 Asserting Conditions for Bug Tracking Systems

To assert that conditions C1 and C2 are valid for bug tracking systems, we analyze a random set of 400 bug reports (100 from each project in our data sets). We find, similar to Breu [15], that at least 27% of comments in bug reports result from the *evaluation of other* *comments* (C2), asking for clarification and additional information. Evermore, the number of peer-reviews—users correcting and improving their peer's comments—increases with the number of comments—$\rho = 0.63$, p-value < 0.003—indicating that *peer-reviewing indeed occurs as a by-product of contributing* (C2). Furthermore, users *evaluate and recognize both bug reports and comments* (C2): most bug tracking systems allow users to recognize, by voting, a bug report they consider important to be fixed; as for comments, we find that an average of 22% of comments recognize the validity and usefulness of other comments.

While condition C2 is valid for most bug tracking systems, condition C1 depends on a project and its community to accept a formal merit-based reputation system and to be motivated to earn reputation and privileges. We acknowledge that some projects and contributors might not be interested in such. Small teams, for example, in which members know each other well, might not be motivated to strive to earn reputation to differentiate themselves from others.

Large open source projects, however, have many similarities to Stack Overflow's community. The population pyramid of privileges and Stack Overflow's meritocratic system is similar to the ones in open-source software projects. Similarly to Stack Overflow, each contributor has a position in the community's hierarchy of power based on his reputation and associated privileges [55]. Core developers, for example, have commit privileges, while others can only read from the code repository [56]; other contributors have moderation privileges in developer forums or bug tracking systems, others have none. Open-source projects also grant privileges to contributors by merit: the more valuable contributions one makes to the community, the more one will be allowed to make [24].

The differences between open source projects and Stack Overflow mainly reduce to differences in levels of transparency and democracy. Stack Overflow's meritocratic system seems to be more transparent and more democratic than the average open source project. In Stack Overflow, every recognition-vote awards users a predefined reputation score and the reputation levels at which users gain new privileges are clearly advertised. Most importantly, in Stack Overflow, it is the entire community that recognizes, and not a only subset of high-reputation contributors that decide, when a contributor is worthy of moving on to the next level in the hierarchy of privileges.

Despite the differences in transparency and democracy, both open-source projects and Stack Overflow are attempts at meritocratic systems. As Stack Overflow's formal reputation system—with a concrete reputation score and clear rules for gaining reputation and privileges—simply adds transparency, this work assumes open-source communities would accept such a formal reputation system. As a consequence, given the will of contributors for acceptance, power, and recognition [9, 41, 47, 56, 67], *many contributors should be*

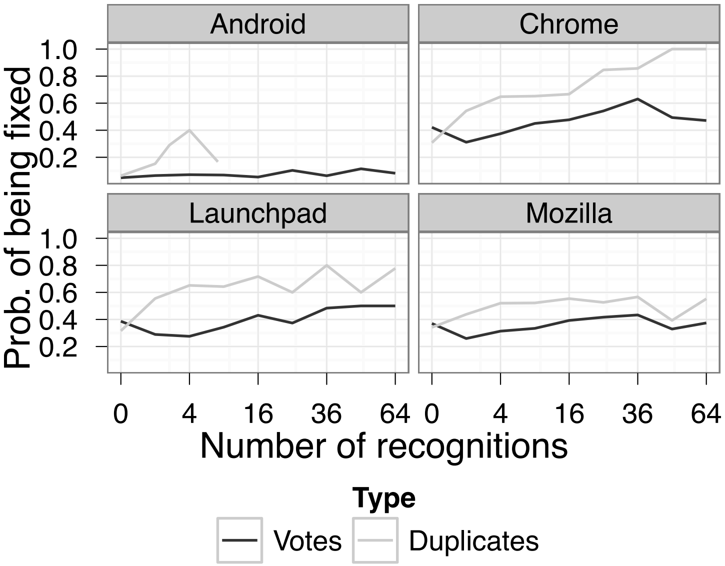


Figure 3.9: Fix-probability by recognition-votes

invested in earning reputation and privileges through such formal system (C1).

3.6.3 Stack Overflow and Bug Tracking Systems Differences

Although Stack Overflow and bug tracking systems share many characteristics, there are two important differences that affect, but do not invalidate, the effects of game mechanisms.

Identifying Useful Bug Reports

The first difference is a direct result of the difference in democracy levels between Stack Overflow and software projects in general. This is a fundamental difference that affects which issues are resolved. In Stack Overflow, it is the community that decides the resolution of a question; in bug tracking systems, it is ultimately the most influential contributors—the minority of developers—that decide if a bug will be fixed, as it is they who accept changes into the code base. Consequently, developers might disagree with the high number of recognition-votes the community has given to particular bugs, rendering the number of recognition-votes a suboptimal signal for developers to identify important bug reports.

To verify this, we investigate if, similar to our findings for Stack Overflow, shown in Figure 3.2, the probability of a bug being fixed increases with its number of recognition-votes. We evaluate two forms of bug recognition: votes and duplicate bug reports. Voting is a mechanism available in most bug tracking systems and is designed to allow users to vote for bug reports they consider important to be fixed. Duplicate bug reports can also be considered a form of recognition, as they imply that the bug has been triggered by more than one user. We find that, as shown in Figure 3.9, for all projects, except Android, the probability of a bug being fixed increases from 2 to 36 recognition-votes for both votes and duplicates—it seems Android is an outlier, perhaps due to its somewhat closed nature.[6] Furthermore, the probability for a bug being fixed is higher for duplicates than for votes, suggesting that duplicates bring more evidence of a bug's relevance than a simple vote and casting more doubt on the harmfulness of duplicates [11].

This confirms that votes and duplicates can be used to identify useful bug reports. It also confirms, however, that developers have different opinions from users, since even bug reports with as much as 64 votes have only a 50% chance of being resolved. While this reduces the utility of such signal for developers, Launchpad has managed to successfully use recognition-votes to help identify useful bug reports: when a bug report receives a certain minimal number of votes, it automatically changes status from 'unconfirmed' to 'confirmed', alerting contributors that a bug should be further investigated. Our findings show that, in addition to votes, duplicate bug reports should also be used to identify important bug reports.

Identifying Useful Comments

Another important difference lies in how knowledge is organized in questions and bug reports. In questions, knowledge is organized as independent answers, allowing users to read only the most useful ones. In bug tracking systems, contributions are organized as a conversation. As a result, although allowing comments to be recognized will identify good comments, users might still need to read previous comments to understand the context of a highly recognized one [43].

The impact these differences bring to the use of game mechanisms in bug tracking systems is limited: bug report recognition might not be an optimal signal for developers, but can still be used by the community, as is done in Launchpad; comment recognition will identify useful comments, but might not reduce the need to read other comments.

[6] http://www.theregister.co.uk/2011/04/12/google_says_android_both_open_and_closed/

3.7 Threats

3.7.1 Internal Validity

Our model for estimating a user's reputation assumes that all votes for a post occur within the first 24 hours of its creation. This is corroborated by our finding that 98% of posts receive all but two of its votes within its first 24 hours. Still, Stack Overflow does not provide information on reputation points awarded by bounties or edits, nor on reputation points lost by down-votes. Nevertheless, as bounties are uncommon (only 2% of questions have offered bounties), edits award only two points, and down-votes cost only one point, we consider our estimation of reputation accurate enough for the purposes of this work.

Although we claim that rewards are the cause of increased participation and competition, we recognize that we have nothing but strong correlations and indications of such causation. Other, unknown factors, might alternatively cause increases in participation and higher number of answers for questions with higher reward potential.

3.7.2 External Validity

We do not perform an experimental evaluation of using game mechanisms on bug tracking systems. Considering the challenges of setting up a valid experiment to perform an extensive evaluation, we consider Stack Overflow and its open data to currently be the best existing surrogate for a bug tracking system with game mechanisms, as it is an organic ecosystem of thousands of software developers participating due to their real needs, and focused on resolving real software issues. We have carefully identified Stack Overflow's and bug tracking systems' similarities and differences and argued that the differences do not invalidate the applicability of game mechanisms to bug tracking systems.

3.8 Summary

In this chapter, we have investigated the potential benefits brought by game mechanisms to improve contribution quality and filter out unreliable and irrelvant content in a collaborative software development ecosystem, and proposed how to enrich bug tracking systems with such game mechanisms. We've shown that Stack Overflow's game mechanisms are effective in increasing contribution quality and in filtering contributions. Game mechanisms are able to improve contribution quality by pushing contributors to post a higher number of high quality contributions and to review and suggest improvements to their peers' contributions. We've also presented the conditions required for game mechanisms to produce its benefits and that the most important condition is that the game mechanisms are carefully tailored so that the community finds the rewards compelling to fight for. In particular, we find that contributors seem to prefer rewards that give them moderation privileges, such as the ability to review, edit, or delete their peers' contributions. When mapping these mechanisms to bug tracking systems, we find that, despite the differences between Stack Overflow and bug tracking systems, by adding game mechanisms to current open-source bug tracking systems, the benefits of increasing contribution quality and filtering noisy contributions should be readily accessible.

Chapter 4

Summarizing Bug Reports to Filter Out Noise

This chapter presents a new, unsupervised, bug report summarization approach that is able to detect the most relevant contents of a bug report. By eliminating irrelevant content, this summarization approach reduces noise—unreliable and unuseful contributions—in bug reports and should, thus, increase productivity in bug tracking systems.

The bug report summarization approach we present attempts to model how a user would, hypothetically, read a bug report when pressed with time and had to skim past content that the user finds to be less relevant. Inspired by the findings of our qualitative investigation on bug reports (Section 2.2.6), we pose three hypotheses on what makes a bug report content relevant: discussing frequently discussed topics, being evaluated or assessed by other sentences, and keeping focused on the problem as stated in the bug report's title and description. Our evaluation suggests that these hypotheses are valid, since the generated summaries have as much as 12% improvement in standard summarization evaluation metrics over the previous bug report summarization approach [52]. Our evaluation also asks developers to assess the quality and usefulness of summaries we create for bug reports they have previously worked on. Feedback from developers not only show the summaries are useful, but also point out important requirements for this, and any, bug report summarization approach.

Differently from Rastkar's bug report summarization approach [52], which has high setup cost, the summarization approach we present is light-weight and unsupervised. Since it follows a model of how contributors would read bug reports in general, the summarization approach should be readily applicable to virtually any bug tracking system, without need for configuration nor of a corpus of manually created summaries.

4.1 Modeling the 'Hurried' Bug Report Reading Process

Summaries are useful for readers who, lacking time, want to be able to comprehend the main points of a text without having to read it in its entirety. It, thus, assumes that the reader trusts that the summary will contain the most important points of the text, leaving out the details.

4.1.1 Extractive Summaries

An extractive summary is a summary which is composed of a set of sentences from the original bug report. Since a good summary should be both informative and cohesive, the goal, when creating an extractive summary, is to select the most relevant and informative sentences from the original text.

This is similar to the task of reading a text in a hurry: the reader must select the most relevant and informative sentences to focus on, leaving the details and irrelevant data aside. Hence, in order to create an extractive bug report summarization approach, we must, first, understand what makes a sentence in bug reports relevant and informative.

To do this, we use the findings of our qualitative research on bug reports, presented in Section 2.2.6, to hypothesize how a reader would read a bug report when limited by time and, thus, needed to select the most relevant sentences to read.

4.1.2 Using a Markov Chain to Model the Reading Process

As with most extractive summarization approaches, we want to rank sentences by relevance and select the n most relevant sentences to compose the summary with. For our summarization approach, we estimate the relevance of a sentence based on the probability of a reader focusing his attention on that sentence, if the reader were only allowed to focus his attention on a limited number of sentences while skimming through the bug report and still wanted to maximize his knowledge about the bug.

We consider this should resemble, in fact, how users would read a bug report when in a hurry: they would have to skip less important portions of the bug report, moving back and forth to portions that will complement their current understanding, following a single topic or moving their attention to different topics, until they are satisfied with the knowledge they have acquired.

We can model this process with a Markov chain. A Markov chain is a weighted directed graph, where nodes represent states and edges between the nodes model the transition probability between the states. To model the hurried bug report reading process, we can represent each sentence in a bug report as a node in a Markov chain $\mathcal{M}$ where each edge $m_{i,j}$ represents the probability of transitioning from sentence s_i to sentence s_j. Thus, each sentence i will have outgoing edges to all other sentences j, weighted by the probability of sentence j being the next sentence to be read after i. The relevance of a sentence can then be approximated by calculating the probability distribution of each state in the Markov chain—the probability of a reader reaching a state if transitioning through the chain according to the transition probabilities of each edge.

As with most models, the Markov chain is only an approximation of our hypothetical bug report reading process. The approximation is given by the fact that, while intuitively the probability of the next sentence does depend on all the previous sentences that were read, Markov chains are memoryless: the probability of the next sentence to be selected will be given only by the current sentence and will not consider the other sentences that have been read.

To complete this model, however, we must estimate the probability transitions from one sentence to another. Estimating such probabilities requires us to understand what sentences users find important to read, based on the sentence they have just read—the *links* users follow from one sentence to another.

4.1.3 How Knowledge Evolves in Bug Reports

As most problem-solving tasks, bug resolution is a process of reducing the uncertainty about a software issue, until the knowledge that has been gathered is enough to resolve the issue. Comments are used, therefore, to share information that could be used to improve the current knowledge about a bug. Thus, for readers to understand a bug report, it is important that they are able to follow the threads of evolving knowledge.

We now recall the findings of our qualitative study of how knowledge evolves in bug reports, presented in Section 2.2. In summary, our study finds that a fundamental source of noise is the collaborative problem-solving nature of bug resolution. First of all, there are many different topics of discussion, ranging from conversations about bug diagnosis, about who should fix the bug, about promising approaches to resolve the bug, and even posts from contributors stating how unsatisfied they are with the project or bug resolution process. More interestingly, there is much content evaluation. As we have presented in Section 3.6, it is common that contributors review their peers' contributions, looking for problems and often suggesting improvements to others' claims and ideas.

These findings, thus, suggest that, in order to understand a bug report, users should follow three general heuristics:

(i) to avoid being distracted by the frequent changes of topics, users should follow the threads of conversation containing the topics they are interested in, from start to finish;

(ii) users should give particular attention to sentences that have been evaluated by other sentences, since they set the context for much of the following comments;

(iii) for users with limited time, in order to focus on the most important points of the bug, users should focus their attention mostly on comments that discuss the problem that was introduced in the bug's title and description and should not follow into parallel topics.

4.1.4 Modelling the Heuristics

We now present how we propose to model each of these heuristics using a Markov chain. Figure 4.1 presents an example of a bug report that we will use to explain the approach. The figure shows the bug report title followed by the sentences $s_{i,j}$ in the bug report, where i is the index of the sentence in the bug report and j is the index of the comment a sentence belongs to—with the bug description being comment 0. Figure 4.2 presents the Markov chain for our running example. In the following sections, we will explain the edges shown in the Markov chain.

Topic Similarity

The first heuristic, that users should follow important threads of conversation from start to finish, implies that after reading a sentence, the most relevant sentences to read next are sentences that talk about the same topics as the previous sentence. This can be easily modeled in a Markov chain: the probability transitions between sentences that talk about the same topics should be higher than the probability transitions of sentences that don't talk about the same topics. Let's assume that $\ell_{\text{tp}}(s_i, s_j)$ measures how much sentence s_i

title **Crash when opening preview window in Squeeze**

$s_{0,0}$ I'm running XX on Debian Squeeze, and its been running fine since last update.

$s_{1,0}$ The crash occurs when I open up the preview window.

$s_{2,1}$ I could not reproduce this, I'm running on Debian Wheezy, and I do not face this crash when opening the preview window.

$s_{3,2}$ Hi, thanks for submitting this bug.

$s_{4,2}$ I also updated my system today to version 2.28.6.

$s_{5,2}$ Now, when I open up the preview window on 2.28.6 my system crashes, so I did manage to reproduce this problem.

Figure 4.1: Bug report for running example.

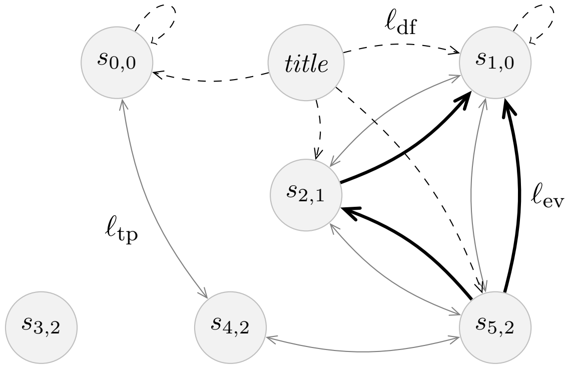


Figure 4.2: Graph showing ℓ_{tp}, ℓ_{ev}, and ℓ_{df} links.

and s_j talk about the same topics. For now, we consider ℓ_{tp} to be binary: it will return 1 if the two sentences share some topic and 0 otherwise:

$$\ell_{\mathrm{tp}}(s_i, s_j) = \begin{cases} 1 & \text{if topic-sim}(s_i,\, s_j) > \tau \wedge\ i \neq j, \\ 0 & \text{otherwise,} \end{cases} \tag{4.1}$$

From the running example, $s_{0,0}$ and $s_{4,2}$ have similar topics, since they talk about updating versions, hence $\ell_{\mathrm{tp}}(s_{0,0}, s_{4,2}) = 1$. In Figure 4.2, gray edges represent the symmetric ℓ_{tp} link between two sentences, so we can see bidirectional edges between sentences $s_{0,0}$ and $s_{4,2}$. Similarly, sentences $s_{1,0}$, $s_{2,1}$, and $s_{5,2}$ all talk about the application crashing when opening the preview window, even if $s_{2,1}$ states that it does not crash when opening the preview window. Therefore, Figure 4.2 shows gray bidirectional edges between these sentences.

Evaluation Sentences

The second heuristic suggests that users should pay attention to sentences that have been evaluated by other sentences. For our Markov chain, this means that the probability transitions from a sentence that evaluates another should be higher than between other sentences. Let $\ell_{\text{ev}}(s_i, s_j)$ be a function that indicates if sentence s_i evaluates sentence s_j:

$$\ell_{\text{ev}}(s_i, s_j) = \begin{cases} \ell_{\text{tp}}(s_i, s_j) & \text{if } s_i \text{ evaluates } s_j, \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

As can be seen, we have defined ℓ_{ev} dependent on ℓ_{tp}: if two sentences do not talk about similar topics, one will never evaluate the other. For our example, sentence $s_{2,1}$ and sentence $s_{5,2}$ are evaluation sentences: sentence $s_{2,1}$ evaluates sentence $s_{1,0}$ since the user says that he could not reproduce the bug described in $s_{1,0}$; therefore, $\ell_{\text{ev}}(s_{2,1}, s_{1,0}) = 1$. In Figure 4.2, thick black edges represent the ℓ_{ev} relationship, so we can see one of these edges from sentence $s_{2,1}$ to $s_{1,0}$. Similarly, sentence $s_{5,2}$ evaluates sentence $s_{2,1}$ since it disagrees with the statement in $s_{2,1}$ that the bug cannot be reproduced; and evaluates sentence $s_{1,0}$ since it agrees with the statement in $s_{1,0}$ that the bug occurs when opening the preview window. While the ℓ_{tp} link is symmetric, the ℓ_{ev} link is unidirectional: from the evaluator sentence to the evaluated sentence in a previous comment. Thus, although $s_{4,2}$ and $s_{5,2}$ share a same topic and $s_{5,2}$ is an evaluation sentence, $\ell_{\text{ev}}(s_{4,2}, s_{5,2}) = 0$ since they are in the same comment.

Similarity to Title and Description

The final heuristic suggests users should focus on sentences that discuss the problem that was initially reported in the bug title and description. To boost the relevance of sentences with similar topics to the bug description, we can add a link from each sentence in the description to itself. There should be two effects of adding self links to the description: first, the relevance of sentences in the description will be increased; second, as a result of sentences in the description being increased, the relevance of sentences with similar topics to the bug description will also increase. As can be seen in Figure 4.2, the two sentences in the description for our example, $s_{0,0}$ and $s_{1,0}$ have edges to themselves.

We also want to boost the relevance of sentences with similar topics to the bug report title, since previous work has shown that the title can be a very good summary of a bug report [65]. To this end, we can artificially include the title as a node in our graph, and add edges from the title t to all sentences s for which $\ell_{\text{tp}}(t, s) > \tau$, for some $\tau > 0$. Figure 4.2 shows links from the title to sentences $s_{0,0}$, $s_{1,0}$, $s_{2,1}$, and $s_{5,2}$ since they all share some topic with the bug title.

The bug report title is not, however, one of the sentences we are trying to rank, so in practice we cannot add the title as a node in the Markov chain and links from the title to the sentences with similar topics. To workaround this issue, for every sentence s_i that shares topics with the bug report title, we add a link from every other sentence to s_i. As a result, to measure the similarity to title and description, we define ℓ_{df} as in (4.3), where S_D is the set of sentences within the bug description, and t is the bug report title.

$$\ell_{\text{df}}(s_i, s_j) = \begin{cases} 1 & \text{if } i = j \wedge s_i \in S_D, \\ \ell_{\text{tp}}(t, s_j) & \text{otherwise,} \end{cases} \tag{4.3}$$

Combining Heuristics

Each of our links target different characteristics of sentences in bug reports. Thus, if these heuristics are valid, we hypothesize that the combination of these links should at least produce similar results to the best heuristic and hopefully improve the results of the best heuristic.

Figure 4.2 shows that we can easily compose ℓ_{tp}, ℓ_{ev}, and ℓ_{df} to rank sentences by combining the three heuristics simply by summing each of the weights. Since our main interest is to verify if a combination of the links does indeed improve the results of the individual links, we combine them in the most straight-forward way: a self vote of value 1.0 for sentences in the bug report description and a linear combination of the ℓ_{tp}, ℓ_{ev}, and ℓ_{df} for other pairs of sentences:

$$\ell_{\text{all}}(s_i, s_j) = \begin{cases} 1 & \text{if } i = j \wedge s_i \in S_D, \\ [\alpha\, \ell_{\text{tp}}(s_i, s_j) & \\ \;+\beta\, \ell_{\text{ev}}(s_i, s_j) & \\ \;+\gamma\, \ell_{\text{tp}}(t, s_j)]\frac{1}{3} & \text{otherwise.} \end{cases}$$

Since we want to verify if each of these heuristics are valid and if their combination is also valid, we use the most straightforward parameters for each of the coefficients: 1. We leave the work of optimizing the weights of these parameters for the future. We note that, since our previous definition of ℓ_{df} only depends on the ℓ_{tp} function, which we have already defined, there is no need to redefine it.

Hypotheses

To evaluate if the proposed heuristics are suitable for summarizing bugs reports, we formulate three hypotheses, one for each heuristic.

From Figure 4.2, due to the nature of Markov chains, the relevance of each sentence—the probability of a reader reaching each sentence—will be greater *the higher the transition probabilities from other sentences to that sentence weighed by the probabilities of each one of those sentences*. For example, a sentence s_i that has only one topic in common with another sentence s_j, that has a low probability of being read, will also have low probability, since it can only be reached from s_j, even if the probability of transitioning from s_j to s_i is high.

We can now pose the following hypotheses for how to rank sentences by relevance for an extractive summary:

Hypothesis 1 *the relevance of a sentence is higher the more topics it shares with other relevant sentences;*

Hypothesis 2 *the relevance of a sentence is higher the more it is evaluated by other relevant sentences;*

Hypothesis 3 *the relevance of a sentence is higher the more topics it shares with the bug title and description.*

4.1.5 Calculating Probability Distribution

The graph in Figure 4.2 is not yet a Markov chain since, until now, the weights for the edges are not probabilities, but simple weights measured by the link functions ℓ_{tp}, ℓ_{ev}, and ℓ_{df}. To transform the graph into a proper Markov chain, we must calculate the probability transition value for each edge. From the weights for the links $\ell(s_i, s_j)$ between the sentences, calculating the probability transition is trivial: for every outgoing edge in a node, we divide its weight by the sum of the weights for all outgoing edges for that same node, as shown in (4.4). As a result, the sum of the weights (probabilities) for the outgoing edges of each node will be 1.

$$m_{i,j} = \frac{\ell(i,j)}{\sum_{\forall k} \ell(i,k)} \tag{4.4}$$

Thus, considering only the ℓ_{tp} links, the adjacency matrix for a graph G for the running example and the resulting Markov chain $\mathcal{M}$ would be the following:

$$G = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \qquad \mathcal{M} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.33 & 0.33 & 0 & 0.33 & 0 \end{pmatrix} \tag{4.5}$$

where the order of nodes in the matrices are given by the index of the sentence in the bug report. Elements $g_{4,0}$ and $g_{0,4}$ have value 1, for example, since they represent the ℓ_{tp} links between sentences $s_{0,0}$ to $s_{4,2}$.

Using PageRank to Calculate Sentence Relevance

Brin and Page [16] develop PageRank to rank web pages by relevance using a very similar model. They estimate the relevance of a web page as the probability of a user reaching that page for a user who surfs the web randomly following hyperlinks from one web page to another—a random surfer. PageRank takes as input a graph G, where web pages are nodes and a link from page i to page j is modeled as a directed edge from i to j with weight $l(i,j) = 1$. PageRank then calculates the Markov chain $\mathcal{M}$ for the random surfer model using (4.4) and outputs a probability distribution $\mathbf{R}$, where each r_i is the probability of a user eventually reaching web page i after a large number of clicks.

Given the Markov chain $\mathcal{M}$, the probability distribution $\mathbf{R}$ for the elements in $\mathcal{M}$ is its principal eigenvector, such that $\mathbf{R} = \mathcal{M}\mathbf{R}$. By the Perron-Frobenius theorem, if $\mathcal{M}$ is an irreducible and aperiodic stochastic matrix, we can use the *iterative power method* to compute $\mathbf{R}$, since the Markov chain is guaranteed to converge to a unique stationary distribution.

An irreducible Markov chain is one in which all states are reachable from any other states, e.g., all states have at least one transition to it with probability > 0. An aperiodic

Markov chain is one in which all states are aperiodic: the minimum common divisor of the number of transitions required to return to any state is 1.

We cannot guarantee, however, that the Markov chain for the web is irreducible and aperiodic. In fact, it is known that it is not: there are many web pages that are not linked to from any other web pages. Similarly for sentences in a bug report, we cannot guarantee there will not be any sentence that does not share any topic with any other sentence.

To transform $\mathcal{M}$ into an irreducible and aperiodic Markov chain $\widehat{\mathcal{M}}$, Brin and Page consider that with probability δ a user will not follow any hyperlink but will randomly go to *any* web page in the Internet. Since any web page is now reachable by any other web page in one transition with probability δ, the chain is now irreducible and aperiodic. The formula for PageRank, in matrix form, is shown below, where $\mathbf{U}_{n\times n}$ is a square matrix of ones, and $\mathcal{M}$ is the Markov matrix.

$$\mathbf{R} = \left[\frac{1-\delta}{n}\mathbf{U} + \delta\mathcal{M}\right]\mathbf{R} = \widehat{\mathcal{M}}\mathbf{R} \tag{4.6}$$

Now that $\widehat{\mathcal{M}}$ is aperiodic and irreducible and the convergence of $\mathbf{R}$ is guaranteed, we use the iterative power method to calculate the principal eigenvector of $\widehat{\mathcal{M}}$. The iterative power method calculates $\mathbf{R}$ starting as a uniform distribution $\mathbf{R} = \frac{1}{n}\mathbf{1}$, and updates $\mathbf{R}$ at each step as $\mathbf{R}' = \widehat{\mathcal{M}}\mathbf{R}$. The algorithm stops when $|\mathbf{R} - \mathbf{R}'| < \epsilon$, which, for aperiodic and irreducible stochastic matrices, is guaranteed to occur for $\epsilon > 0$.

Given the similarity of the random surfer model and the bug report reading model that we have proposed in Section 4.1, applying PageRank to calculate the probabilities of sentences being read requires only that we identify the links—the transitions—between sentences in a bug report. We can then derive the Markov chain $\mathcal{M}$ using (4.4) and calculate the probability distribution $\mathbf{R}$, just as in PageRank.

The only change our original model suffers when using PageRank is that now we must consider that, with probability δ, a user might jump to any sentence in a bug report without following our links—we use $\delta = 0.85$, just as recommended by Brin and Page in PageRank. We can use (4.4) directly to calculate the Markov chain, which is applicable even if $\ell(s_i, s_j)$ returns values different from 0 and 1 and instead returns any value $\geq$ 0 as the weight of links. This will be the case when we rank sentences considering both Hypotheses 1 and 2, for example, by combining ℓ_{tp} and ℓ_{ev} as $\ell(s_i, s_j) = \ell_{\text{tp}}(s_i, s_j) + \ell_{\text{ev}}(s_i, s_j)$ and when ℓ_{tp} and ℓ_{ev} measure the *strength* of these links.

For the running example, considering only the ℓ_{tp} links, transforming G into the stochastic matrix $\mathcal{M}$ by dividing each row by the sum of the row, as shown in (4.4), results in

(4.5). After making it irreducible and aperiodic by multiplying δ and then adding $\frac{1-\delta}{n}\mathbf{U}$, as shown in (4.6), and solving for $\mathbf{R}$ using the iterative power method, we get the following probabilities for the sentences in our example: $[0.11, 0.18, 0.18, 0.02, 0.20, 0.27]$, effectively ranking sentences as $[s_{5,2}, s_{4,2}, s_{1,0}, s_{2,1}, s_{0,0}, s_{3,2}]$.

4.2 Estimating Probability Transitions Between Sentences

We have shown how to model the hurried bug report reading process and estimate sentence relevance using a Markov chain, assuming we can measure ℓ_{tp}, ℓ_{ev}, and ℓ_{df}. The following sections details how we use natural language processing to measure how much two sentences talk about the same topics and to identify sentences that evaluate other sentences and, thus, quantify the weights for links ℓ_{tp}, ℓ_{ev}, and ℓ_{df}.

4.2.1 Measuring ℓ_{tp}

There exists much work on measuring how much two documents discuss the same topics. Most of these first identify the topics contained within documents and then measure topic similarity by considering how much topic overlap there exists between the documents. Sun [59], for example, measures the changes in mutual information from one chunk of a document to another to detect topics. Latent Dirichlet Allocation (LDA) [13] and Probabilistic Latent Semantic Analysis (PLSA) [34], on the other hand, identify topics using word co-occurrence knowledge extracted from documents.

While arguably these approaches are state-of-the-art in identifying topics, they are not lightweight and generally require the tuning of several different parameters, most importantly, the number of topics to be identified. Since we aim for a solution that does not need such parametrization, we choose a more direct and lightweight approach to measure topic similarity: we will consider that sentences that talk about similar topics should have many common words. We will approximate, therefore, topic similarity by lexical similarity.

While there are many textual similarity metrics, such as Levenstein edit distance and Euclidean distance, the *cosine similarity* function is one that has shown consistent results in measuring the similarity of content, and is used, for example, to classify and cluster documents by author, topics, and writing style [17]. The cosine similarity is defined as below, where x and y are the vectors of term frequency for a sentence.

$$\text{cosine-sim}(x, y) = \frac{x \cdot y}{|x|.|y|}$$

As is commonly done when measuring textual similarity, we scale the term frequency (tf) by the inverse document frequency (idf) of the term, diminishing the importance of terms that occur in most documents, since they do not help in differentiating two documents. The term frequencies in the vector for each sentence, scaled by the inverse document frequency is calculated as shown below, where $n_{t,s}$ is the number of times term t occurs in s, N is the total number of sentences, and n_t is the number of sentences that contain term t.

$$\text{tf-idf}(t, s) = n_{t,s} \log \frac{N}{n_t}$$

Before building the vectors for the sentences using tf-idf, we must first tokenize the text into its terms. Based on our previous experience with tokenizing text from bug reports, we tokenize the sentences using the following regular expression: '`[\w-]+(\.[\w-]+)*`' should correctly identify words, but preserve most function and variable names, urls, and software version numbers. After tokenization, we move all characters to lowercase and stem the tokens using the standard Porter stemmer [51]. The sentence "Would it make sense to just use 'CC = $(DEB_HOST_GNU_TYPE)-gcc' unconditionally, for simplicity?", for instance, would be tokenized into the following terms: 'would', 'it', 'make', 'sens', 'to', 'just', 'use', 'cc', 'deb_host_gnu_typ', '-gcc', 'uncondit', 'for', and 'simplic'. We can now redefine ℓ_{tp} as:

$$\ell_{\text{tp}}(s_i, s_j) = \begin{cases} \text{cosine-sim}(s_i, s_j) & \text{if } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

4.2.2 Measuring ℓ_{ev}

The relations of evaluation between sentences we are interested in are those where a sentence evaluates or verifies the validity of the content of another sentence. From our example, $s_{2,1}$ fits this relation, as it suggests that the content of $s_{1,0}$ is not valid, since its author could not reproduce the crash. Similarly, $s_{5,2}$ evaluates both $s_{1,0}$ and $s_{2,1}$, since it says that the bug is reproducible, confirming the content of $s_{1,0}$ and unconfirming the content of $s_{2,1}$.

Although we are not aware of prior work that tries to identify evaluation relations between sentences in a bug report, *polarity detection* through *sentiment analysis* might be a good approximation of this relation. Polarity detection has previously been used to detect the polarity of reviews of movies [8], of products, politicians, and almost anything [62]. We consider polarity detection might be a reasonable approximation, since, in general, it first filters evaluation sentences, and then tries to detect if the evaluation is positive or negative. For our purposes, we consider a sentence is an evaluation sentence if its polarity is different from neutral.

The best results for polarity detection have been achieved using classifiers such as support vector machines that are trained on a corpus of texts that have had its polarity previously annotated. Since we are looking for an approach that is completely unsupervised, however, having to manually classify the polarity of sentences of a bug report would not suit here. We use, therefore, the same approach as Bo and Bhayani [27] who create a large training corpus of sentences for polarity prediction by using the *emoticons* present in Twitter messages to automatically infer the polarity of each sentence. We use a training set composed of $800,000$ Twitter messages with positive polarity, $800,000$ with neutral polarity, and $800,000$ with negative polarity and use the resulting classifiers to predict the polarity of sentences in bug reports—we include into the training set sentences with neutral polarity since we also need to identify the sentences that are not evaluation sentences. This classifier uses a linear support vector machine in which the feature vector for a sentence represents the presence or absence of a word in the comment. We leave the reader to consult Bo and Bhayni's [27] report for further details on this approach.

Simply identifying evaluation sentences is not enough, however, since we want to identify an evaluation *link* from one sentence to another. Our definition of the evaluation link from Section 4.1.4, however, hints to a solution to identify such relation: the evaluation link requires first that two sentences have similar topics. As such, just as how cosine-sim measures not *if*, but *how much*, two sentences share a topic, here we also propose to quantify *the strength* of the evaluation link between two sentences. Furthermore, sentences in comment i can only be evaluated by sentences from comments posted after comment i.

We measure the strength of the evaluation link from s_i to s_j as the strength of the ℓ_{tp} link between the two sentences if s_i is an evaluative sentence or is a sentence within a comment that contains an evaluative sentence. We can now redefine ℓ_{ev} as:

$$\ell_{\text{ev}}(s_i, s_j) = \begin{cases} \ell_{\text{tp}}(s_i, s_j)^{\frac{\left(\mathrm{p}_{\mathcal{S}}(s_i)+\mathrm{p}_{\mathcal{C}}(s_i)\right)}{2}} & \text{if } \mathrm{c}(s_i) > \mathrm{c}(s_j), \\ 0 & \text{otherwise,} \end{cases}$$

where $\mathrm{p}_{\mathcal{S}}(s)$ returns 1 if s has polarity and 0 otherwise, $\mathrm{p}_{\mathcal{C}}(s)$ returns 1 if s is contained in a comment that contains a sentence that has polarity and zero otherwise, and $\mathrm{c}(s)$ returns the index of the comment that contains sentence s.

4.2.3 Chunking Comments into Sentences

Our extractive summarization approach works with sentences as the minimal chunks of text to be extracted into a summary. Bug reports, however, are not segmented into sentences. In fact, segmenting comments into sentences is a challenging problem itself, since text in bug reports is very informal and often contains source code, stack traces, logs, and enumerations. Using a traditional sentence chunker that uses a '.' character to identify sentence boundaries does not produce good results. While Bettunburg et al. [12] parses bug report comments to collect structured information, such as stack traces and patches, they do not work on sentence chunking, but acknowledge that it is a non-trivial problem.

Based on our previous experience, we chunk comments into sentences using the following heuristics: (i) each item from an enumeration is a sentence; (ii) each line in a stack trace or source code snippet is a sentence; (iii) we use '.', ';', '?', '!' as sentence delimiters, except when '.' is used in version strings, urls, code snippets, and abbreviations; (iv) if a line of text has a line break before 80 characters we consider the line break ends a sentence.

4.3 Evaluation

4.3.1 Methodology

Our evaluation has two parts. We first test if our hypotheses are valid, by assessing the quality of the generated summaries. We then ask developers to evaluate the quality and usefulness of summaries for bug reports they had previously worked on.

Hypothesis Tests

To test our hypotheses, we implement three different summarizers, one for each link function ℓ_{tp}, ℓ_{ev}, and ℓ_{df}. We will consider our hypotheses to be valid if our summarizers produce summaries that have competitive or improved evaluation measures compared to the summaries created by the email summarizer [52], which we have implemented to the best of our knowledge. We also test a summarizer using ℓ_{all} to assess if the combination of the links yields improved summaries.

The corpus we use for this evaluation is the corpus created by Rastkar et al. [52], which consists of 36 bug reports, each with three reference *golden summaries* created by humans. We use these reference summaries to compare against the generated summaries. For this evaluation, and as was done by Rastkar et al., we generate summaries by selecting sentences until the summary reaches a predefined percentage of the original bug report's length, in number of words. For completeness, we evaluate the performance of the summarizers when generating summaries of different lengths, from 25% to 70% of the length of original bug report in number of words.

To assess our hypotheses, we use the following established metrics for evaluating summaries:

Precision and Recall We measure precision and recall for the summaries, considering a *master golden summary* G^* composed of the sentences that are present in the majority of the golden summaries. For the corpus created by Rastkar, the master golden summary is composed of the sentences that are present in at least two golden summaries.

$$\text{precision}(S) = \frac{|\{S \cap G^*\}|}{|S|} \qquad \text{recall}(S) = \frac{|\{S \cap G^*\}|}{|G^*|} \tag{4.7}$$

$$\text{f-score}(S) = \frac{2 * \text{precision}(S) * \text{recall}(S)}{\text{precision}(S) + \text{recall}(S)} \tag{4.8}$$

Precision then measures the percentage of sentences in a summary that is also present in G^*, while recall measures the percentage of sentences in G^* that are present in the summary being evaluated. The f-score is the geometric mean of the precision and recall rates, given by (4.8), which, differently from the arithmetic mean, will give a higher weight to the lower values.

Precision and recall, however, measure the quality of a summary against a master golden summary which is artificially composed by the sentences present in at least half of the golden summaries. Thus, it does not mean that such a master golden summary is necessarily a good one, since different golden summaries can provide all the relevant information with different extracted sentences [50].

Pyramid Score To circumvent the issues of recall and precision, Nenkova et al. [50] propose the *pyramid score*, an evaluation metric that should better measure the quality of an extractive summary based on a set of golden summaries created by several annotators. When evaluating a summary composed of n sentences, pyramid score is the sum of the number of golden summaries that contain each of the n sentences from the evaluated summary, divided by the sum of the number of golden summaries that contain the n sentences that are most frequently present in golden summaries. Pyramid score is, therefore, a recall-related evaluation metric for a summary, that measures the quality of a summary against the best summary of the same length.

The formula for the calculation of pyramid score is shown below, where S is a summary, $s \in S$ are the sentences in summary S, $\mathcal{G}$ is the set of all golden summaries G, and $\mathcal{G}^{\text{top}}_{|S|}$ is the set of size $|S|$ of sentences that are most frequently present in golden summaries:

$$\text{pyramid}(S) = \frac{\sum_{s \in S} \left|\{G \in \mathcal{G} : s \in G\}\right|}{\sum_{s \in \mathcal{G}^{\text{top}}_{|S|}} \left|\{G \in \mathcal{G} : s \in G\}\right|}$$

Nenkova also defines pyramid precision and pyramid recall. Pyramid precision calculates the percentage of sentences in a summary that are present in at least one golden summary. Pyramid recall calculates the percentage of sentences present in any one of the golden summaries that are present in the summary being evaluated. In effect, these are the precision and recall as defined in (4.7), with G^* being composed of sentences that are present in any of the golden summaries.

Developer Evaluation

For the second part of our evaluation, we use the ℓ_{all} summarizer to generate summaries of length 25% of the original bug report for a random set of bug reports from the Debian, Launchpad, Mozilla, and Chrome bug tracking systems and invite the developers who contributed to these bug reports to assess the quality and usefulness of the summaries. We ask the developers to: (i) assess the quality of the summary by indicating the mistakes made by the summarizer: sentences that should have been extracted but weren't and the sentences that were extracted but are not so relevant; (ii) explain what are the most important types of information that a summary should contain; and (iii) indicate, using a Likert scale, what are the most important use cases for such summaries.

We also investigate how to best present a summary so that contributors can easily navigate and read through it. We present the developers with summaries in two formats: *condensed* and *interlaced*. The condensed format shows only the extracted sentences. The interlaced format presents the complete bug report content, with the extracted sentences shown highlighted out from the other sentences. Figure A.1 shows a sample of the interlaced view for a summary for our running example and Figure A.2 shows a sample of the condensed view. We ask developers to indicate if they preferred the condensed or interlaced format for reading a bug report summary, along with an explanation of their rational.

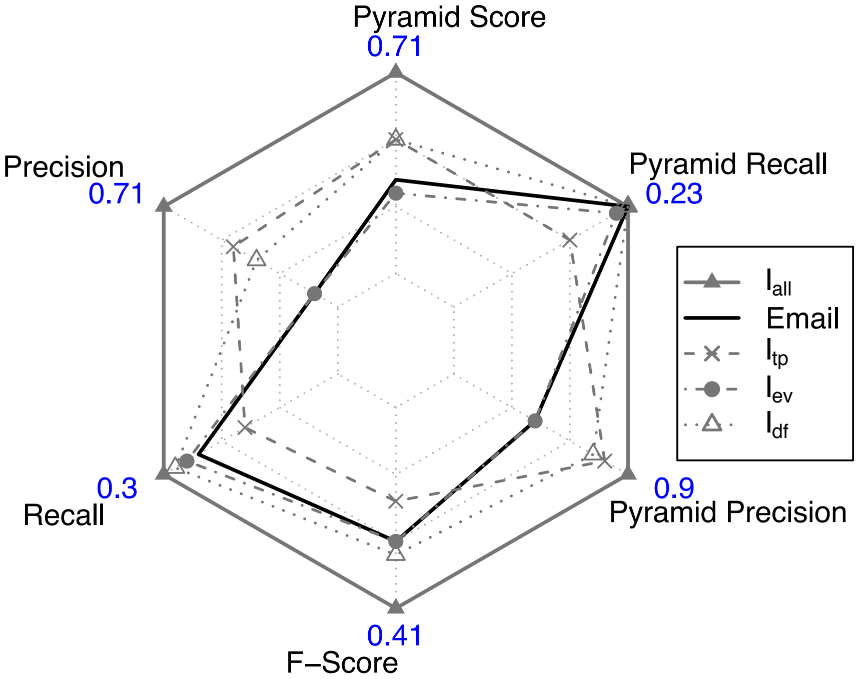


Figure 4.3: Comparison of evaluation measures for $\ell_{\rm tp}$, $\ell_{\rm ev}$, $\ell_{\rm df}$, $\ell_{\rm all}$, and email summarizers for summaries of length 25%.

4.3.2 Results of Hypothesis Tests

To test our hypotheses, we generate summaries for the bug reports in the Rastkar corpus and compare it quality with the summaries generated by the email summarizer. Like Rastkar, we start by generating summaries of length 25% of the original bug report. Figure 4.3 presents, for each evaluation metric, the averages—weighed by the number of sentences in a bug report—for each of the 5 summarizers we evaluate: $\ell_{\rm tp}$ summarizer, $\ell_{\rm ev}$ summarizer, $\ell_{\rm df}$ summarizer, $\ell_{\rm all}$ summarizer, and email summarizer. The maximum value in each of the evaluation metrics' axes is the maximum for that metric, for all of the summarizers and each tick in the axis corresponds to a difference of 0.05. The chart shows, for example, that the pyramid score for $\ell_{\rm all}$ is 0.71, while for email summarizer it is around 0.63, and for $\ell_{\rm tp}$ it is 0.66.

The results show that $\ell_{\rm tp}$ summarizer has slightly more precision and pyramid precision than the email summarizer, but has less recall and pyramid recall—the non-parametric Mann-Whitney U test supports that these distributions are indeed different, with p-

value < 0.05 for precision and recall and p-value < 0.01 for pyramid precision and recall. The chart also shows that the ℓ_{ev} summarizer has similar evaluation results compared to the email summarizer, for summaries of length 25%. For ℓ_{df}, the chart shows that it has slightly better values for all evaluation metrics compared to the email summarizer, except for pyramid recall. The Mann-Whitney U test, however, supports that, for ℓ_{df}, only the pyramid precision measure is better than email summarizer, with p-value < 0.01.

We can also use Figure 4.3 to compare each of our three individual summarizers—ℓ_{tp}, ℓ_{ev}, and ℓ_{df}—amongst themselves. This comparison shows that that ℓ_{tp} has significantly less recall than the other summarizers, while ℓ_{ev} has significantly less precision than the others. We find that ℓ_{tp} has such low recall because ℓ_{tp} prefers longer sentences than the other summarizers. Thus, since we take sentences until we reach 25% of the number of words in the original bug report, it extracts less sentences than the other summarizers.

The results we have presented show that all of our individual summarizers are at least competitive with the email summarizer. Furthermore, their combination as ℓ_{all}, has an improvement of 12% in precision, 8% in pyramid precision, and 8% in pyramid score, confirmed by the Mann-Whitney U test with p-value < 0.01. These results indicate that Hypotheses 1, 2, and 3 have a high likelihood to be valid: relevant sentences for a bug report summary are those that discuss topics that are frequently discussed; those that are evaluated by other sentences; and those that do not deviate from the problems as described in the bug report title and description. Thus, important information in bug reports is information that is frequently discussed; due to the uncertainty of bugs and the ad-hoc nature of bug resolution, the evaluation of previous sentences are very important and readers need to follow the constant evolution of these claims.

It is important to note that all of the summarizers we evaluate here have reasonable precision but quite low recall. One of the main causes of the low recall is that most of expert summaries for the Rastkar corpus are larger than 25% of the original bug report length. We will discuss how the summarizers perform for different summary lengths next and elaborate on how to aid the user in deciding on a target summary length in Section 4.4.

Varying Summary Target Length

For most summarization approaches, the length of the resulting summaries are decided beforehand, by the user, usually through an input parameter, either in number or percentage of characters, words, sentences, or paragraphs [42]. Rastkar [52], for example, ranks sentences by their scores and includes the highest scoring sentences until the summary reaches 25% the original bug report length in words.

We have shown in the previous section that ℓ_{tp}, ℓ_{ev}, ℓ_{df}, and ℓ_{all} produce good quality summaries of length 25% of the original bug report and that ℓ_{all} produces significant improvements over the email summarization approach. For a more complete evaluation, we now present the accuracy of these summarizers when generating summaries of different lengths.

Figure 4.4 presents, similar to Figure 4.3, the precision, recall, f-score, pyramid precision, pyramid recall, and pyramid score for ℓ_{tp}, ℓ_{ev}, ℓ_{df}, ℓ_{all}, and the email summarizer, for lengths varying from 10% to 70% of the original bug reports. Similar to Figure 4.3, for all metrics, except recall and pyramid recall, the ℓ_{all} summarizer has clearly better values than all other summarizers, for all summary lengths. For recall and pyramid recall, however, the figure shows that ℓ_{all} has similar values to the other summarizers for shorter summary lengths, but the values decrease, compared to the other summarizers—excluding ℓ_{tp}—as the summary length increases. The reason why ℓ_{all} and ℓ_{tp} have lower recall even while having higher precision over the other summarizers is because these two summarizers select longer sentences than the other summarizers. Since we stop including sentences when we reach a predetermined word count, a summarizer that prefers longer sentences will have included fewer sentences than summarizers that prefer shorter sentences.

Figure 4.4 also shows that, overall, summary quality does not decay as we include more sentences. Although precision and pyramid precision does decrease consistently, the pyramid score remains almost constant and the f-score increases, indicating that the recall and pyramid recall increases at a higher rate than the decrease in precision.

4.3.3 Results of Evaluation with Developers

The evaluation with the developers from the Debian, Launchpad, Mozilla, and Chrome bug tracking systems was very insightful. From the 250 invitations we sent out, we received a response from 58 developers, each one evaluating a different bug report: 22 from Debian, 14 from Mozilla, 13 from Ubuntu, and 9 from Chrome.

From the passion of the responses we received from developers, they seemed genuinely interested in bug report summaries. This feeling is corroborated by the results of our survey, in which more than 80% of developers stated that bug report summaries would be at least *very* useful—out of a scale of not useful, somewhat useful, useful, very useful, and extremely useful—when (i) looking for a solution or workaround for a bug; (ii) searching for similar or duplicate bugs; (iii) trying to understand the status of the bug and its open issues; and (iv) when consulting bugs for prioritization, triaging, or closing out old bugs.

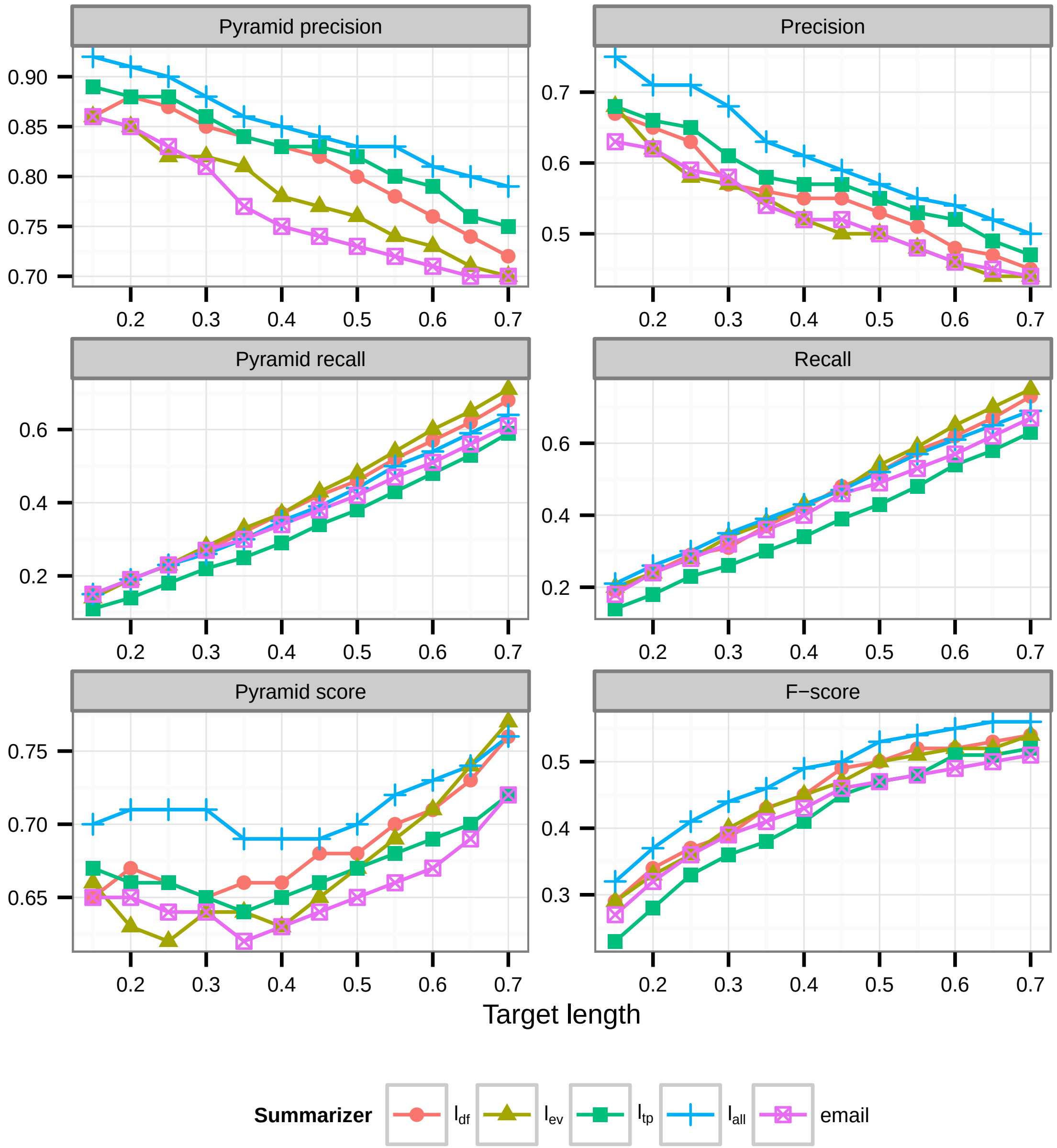


Figure 4.4: Evaluation measures for ℓ_{tp}, ℓ_{ev}, ℓ_{df}, ℓ_{all}, and the email summarizer for summary lengths varying from 15% to 70% of original bug report word count.

When asked about the most important kind of information that needs to be present in a bug report summary, developers repeatedly affirmed that summaries should focus on showing information (i) about the current status and the reason for such state; (ii) about solutions or workarounds to the bug and the environments each remedy is applicable to; and (iii) about the consensus on diagnostic information, such as the agreed steps and environment settings to reproduce the bug. Developers also stressed the importance of being able to recognize the different types of structured information in bug reports: stack traces, code snippets, commands and their results, and enumerations such as steps to reproduce. A qualitative analysis of the mistakes made by summarizer, pointed out by the developers, indicates that sentences, as chunked by the procedure explained in Section 4.2.3, might not be the most appropriate way to chunk content in bug reports, since the resulting summaries often contain, for example, only some of the items of an enumeration and the result of a command but not the command itself.

Table 4.1: Precision and recall for developer evaluation.

	Debian		Mozilla		Launchpad		Chrome		All	
	ℓ_{all}	email	ℓ_{all}	email	ℓ_{all}	email	ℓ_{all}	email	ℓ_{all}	email
Precision	0.49	0.41	0.77	0.58	0.56	0.62	0.69	0.62	0.59	0.52
Recall	0.45	0.40	0.60	0.34	0.47	0.46	0.61	0.53	0.51	0.42
F-score	0.44	0.34	0.62	0.39	0.50	0.51	0.59	0.50	0.52	0.41

When evaluating the quality of the summaries presented to the developers, we asked them to mark the mistakes made by the summarizer: sentences that were included in the summary but shouldn't have been (false positives) and sentences that weren't included in the summary but should have been (false negatives). Such marking made by the developers allows us to quantify the quality of the summaries sent to the developers by calculating precision and recall—since these bug reports only have one reference golden summary, pyramid score is not applicable and the precision and recall and pyramid precision and recall will have the same values.

The results for the 58 bug reports assessed by developers, shown in Table 4.1, indicate that, in average, the summaries include half of the relevant information, with 60% of the sentences in the summaries being relevant ones. These results are promising, since they are substantially better than the results for the Rastkar corpus, showing an improvement 20% in recall with a decrease in precision of only 10%. This indicates that the ℓ_{tp}, ℓ_{ev}, ℓ_{df}, and

ℓ_{all} summarizers have not been tailored to a particular subset of bug reports and are pretty general, achieving our objective of creating an unsupervised bug report summarization approach that is readily applicable to any bug tracking system.

When as asked if developers preferred the condensed or interlaced summary formats, responses were mixed: 56% preferred condensed, while 46% preferred interlaced, a non-significant difference. We did, however, find a consensus on the advantages of each one, which can be summarized by the two following responses:

> "I would never trust an automated summary, and would always need to refer to the original. By highlighting the important sentences, it allows me to 'speed read' a long report with many comments and status updates."

> "Interlaced works when there aren't pages of irrelevant data. For a bug with *lots* of comments, a condensed view would help. Personally, I use a greasemonkey script that highlights comments from people that are likely to be providing useful information."

Users were, thus, generally skeptical that an automated system would generate a perfect summary, and would need to be able to refer to the non-relevant sentences when needed. An optimal user interface for speed-reading a bug report would be one that would allow users to easily skip irrelevant comments but at the same time be able to quickly scan them looking for relevant sentences that the summarizer missed.

4.4 What length should my summary be?

Our evaluation has shown how precision and recall vary for different target summary lengths. In practice, however, this and most summarization mechanisms require the user to select the desired target summary length as an input parameter [42]. Unfortunately, it might not always be clear what a good target summary length should be for particular bug report. Should one consider that 25% of the length of the original bug report is a good summary length for all bugs?

The summaries created by experts for the bug reports in Rastkar's corpus, for example, have a wide variety of lengths, ranging from 18% to 69%, with a mean and median of approximately 47%. It seems, therefore, safe to assume that different bugs, with different informational content, density and different flow of threads and topics will probably have different optimal summary lengths. The question we now pose is: *how can we assist the user in choosing the length of the target summary when using our summarizers?*

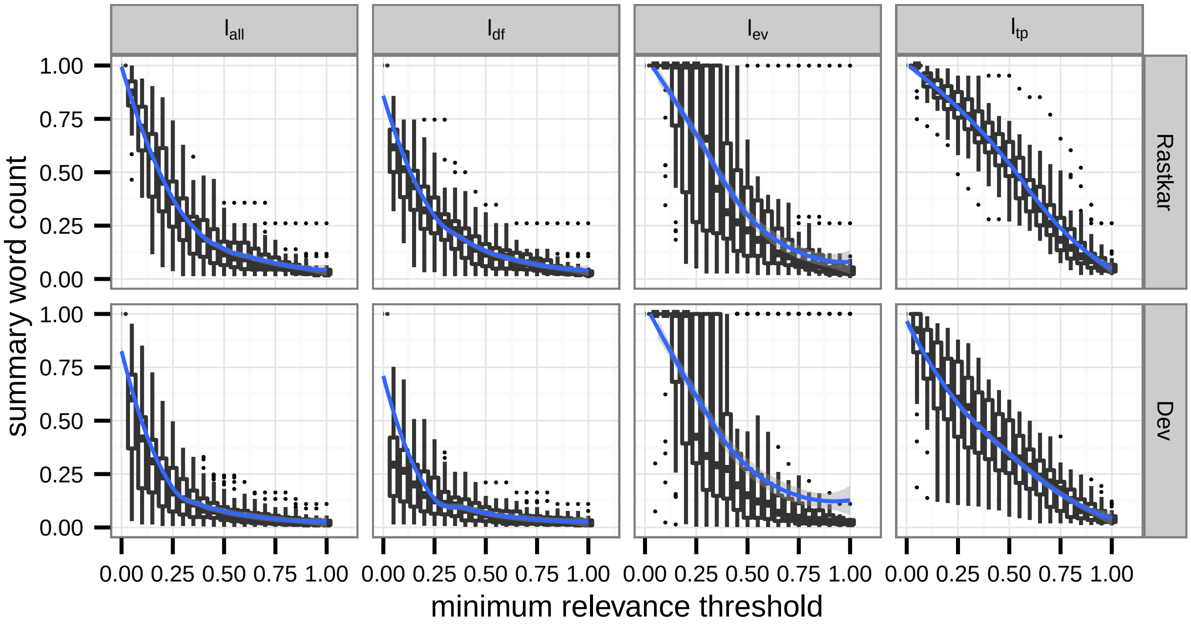


Figure 4.5: Variation of summary length for different minimum relevance threshold values.

For this part of the work, we focus mainly on the ℓ_{all} summarizer, as we did in our developer evaluation, since this is the best summarizer of all, as our evaluation has shown. We will present, nevertheless, details about summary lengths for the remaining summarizers, mostly for the purpose of understanding how each part of the ℓ_{all} summarizer works, and how each different heuristic contributes to its final result. We hope this will help others in improving each of the heuristics for a new, better summarizer.

4.4.1 Sentence Relevance Threshold vs. Maximum Length

One way to facilitate the use of the summarizers is to substitute the target summary length parameter with an alternative parameter that would be easier for users to decide on a value for. An alternative parameter to maximum summary length is the *minimum relevance threshold*. Instead of including the top ranking sentences until a certain summary length is achieved, one could include into the summary only the sentences that cross a certain relevance threshold. The length of the summary would be, hence, determined solely by how many sentences are found to be relevant, thus allowing different bug reports to have different lengths for the same value of minimum relevance threshold.

Creating a summary with only the sentences above a certain relevance threshold is straightforward. Since the sum of the sentence probabilities resulting from PageRank have sum of 1, we normalize the probabilities by dividing the probability of each sentence by the maximum sentence probability in the bug report. Such normalization would transform all sentence relevance scores into values ranging from 0 to 1, now allowing us to specify that we want to include in the summary only sentences with relevance at least τ% of the maximum relevance.

When using such scheme, increasing the threshold τ will produce shorter summaries, while decreasing the threshold will produce longer summaries. Figure 4.5 shows how the distribution of resulting summary lengths per bug report changes as we vary the minimum relevance threshold, for each of the different summarizers for the Rastkar bug corpus as well as the developer bug corpus. What is notable from the figure is that, while there is a large variation in summary lengths for the same relevance threshold, each summarization approach has a characteristic curve for how summary length varies as we increase the relevance threshold. For the ℓ_{df} and ℓ_{all} summarizers, the curve shows a steep decrease in summary lengths for low values of the threshold, whereas the ℓ_{tp} summarizer has a quasi-linear decay in summary length as the threshold increases. This means that the ℓ_{tp} summarizer produces something close to a uniform probability distribution, while the ℓ_{df} and ℓ_{all} summarizers produce a distribution where there are only a few high relevance sentences and a majority of low relevance sentences—it seems likely that the minority of high relevance sentences are the sentences in the bug description and sentences with high similarity to them.

Figure 4.6 presents the receiver operating characteristic (ROC) curve for both maximum length and minimum relevance selectors, presenting the false positive rates (1-recall) in the x-axis and the true positive rate (precision) on the y-axis. Since the ROC curve effectively shows the trade-off in increasing recall at the expense of decreasing precision, the curve is often used to compare different information retrieval approaches by measuring the Area Under the Curve (AUC). The curve generally starts of at (1,1), where there are few positives, recall is very low and precision high, and progresses to (0,0) where there are many positives, recall is very high but precision is low. As a result, the closer the curve comes to (0,1)—high recall and precision—the higher the AUC. Thus, Figure 4.6 shows that, first of all, the AUC for minimum relevance is slightly greater than the AUC for maximum length, for all summarizers. More importantly, particularly for high relevance scores, the minimum relevance selector produces significantly higher precision than maximum length selector for the ℓ_{all} summarizer. Thus, although the user would not be able to predetermine the length

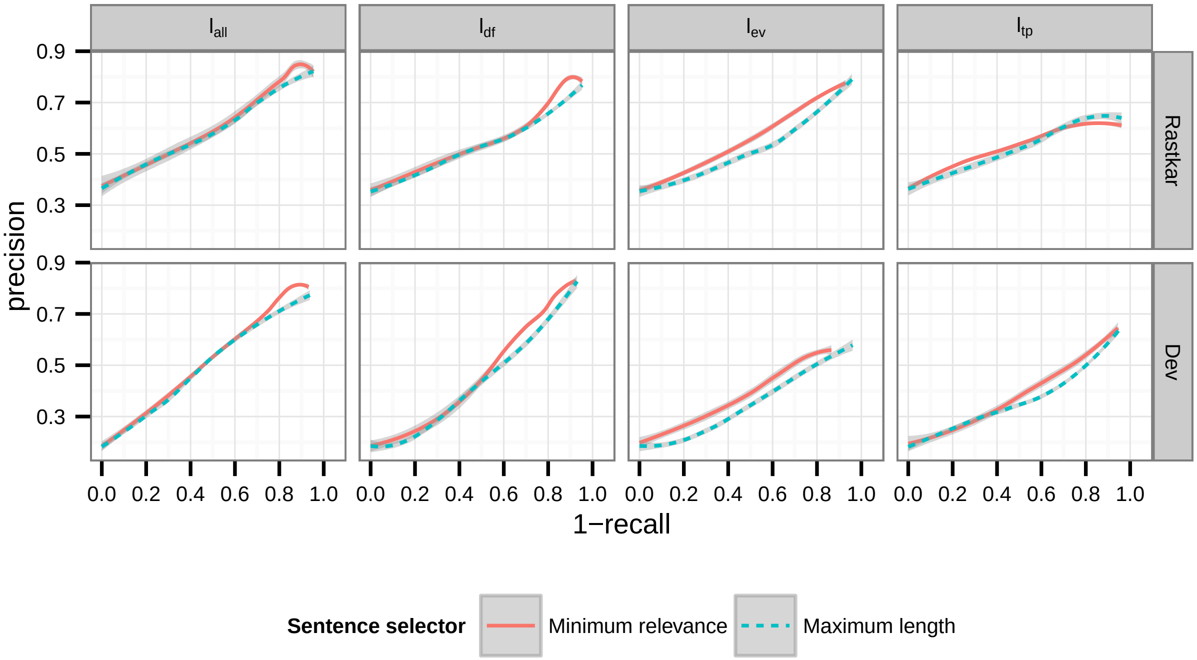


Figure 4.6: ROC curves for ℓ_{all} summarizer for Rastkar and developer bug corpus relevance threshold and maximum length selectors.

of the resulting summaries, it effectively allows the user to better control the *precision* of the resulting summary.

Although this technique eliminates the need for users to input the desired summary length, it does require users to input the desired minimum sentence relevance. This is beneficial when the user does not know the desired summary length but has a low tolerance for false-positives—irrelevant sentences included in the summary—and only wants the most relevant sentences. Thus, if users have a priori knowledge or constraints on the summary lengths, they should use the maximum summary length parameter; otherwise, if they have constraints on the false-positive rates for the summary, the minimum relevance threshold parameter would be more appropriate.

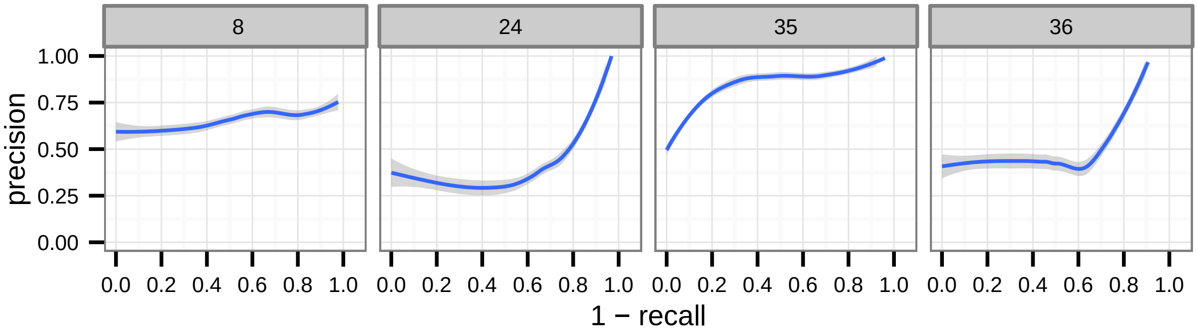


Figure 4.7: ROC curve for 5 bugs in the Rastkar corpus for the ℓ_{all} summarizer.

4.4.2 Suggesting Summary Length that Optimizes Quality

While we have presented an alternative to the target summary length parameter, we still have not presented a tool to aid the choice of neither parameters. Without such aid, using a target summary length that is shorter than the expert summary length would produce a summary with low recall and using a target summary length that is larger than the expert summary length would produce a summary with low precision, regardless of how well a summarizer performs.

Unfortunately, there are few studies investigating the optimal length for summaries [42]. We could easily suggest small summary target lengths, 25% of the original size, for example, for all bug reports. This would lead to summaries with high precision, but low recall. On the other hand, we could suggest large summary target lengths, say 70% of the original size, leading to lower precision but higher recall rates, or medium-sized summary lengths for a balanced precision and recall. Or, optimally, be able to suggest an appropriately-sized summary that balances well precision and recall to the needs of the user.

We can use the ROC curve, introduced previously, to aid the choice of input parameters for information retrieval task such as this one. Since recall and precision are higher the closer a point is to (0,1), the input parameters responsible for the points in the curve that are closest to (0,1) would be good choices. Figure 4.7 shows the ROC curve for 4 bug reports in the Rastkar corpus for the ℓ_{all} summarizer, while varying the target length parameter. The curve starts close to (1,1), where low target lengths produce low recall, but generally high precision summaries. The curve then progresses towards (0,0), where high target lengths produce high recall, but low precision summaries.

The figure shows, however, that the path from one extreme to the other is different for each of the bug reports. The curves for bugs 24 and 36, for instance, present a sharp decrease in precision while recall varies from 0 to 30%; for bugs 8 and 35, on the other hand, precision remains a constant as the recall increases up to 75%. It seems, therefore, for bugs 8 and 35, which have high AUC, we could recommend longer summaries, since the result would be high recall with still relatively high precision; for bugs 24 and 36, which have lower AUC, on the other hand, it seems it would be best to recommend shorter summaries so that precision is not diminished at the expense of higher recall. One could, in fact, for bugs 24 and 36, present to the user the choice of a longer summary with high recall and low precision or a shorter summary with low precision but high recall.

It seems, therefore, that we would be able to suggest good summary lengths if we were able to predict the AUC for each bug report—the difficulty the ℓ_{all} summarizer had when summarizing each bug report. Nenkova and Louis [49] find that the difficulty of summarizing a text, for humans or machines, can be predicted by the text's Shannon entropy (4.9), word count, and vocabulary size. Particularly, they suggest that low entropy values for a text generally indicates that the text is more cohesive and, thus, easier to summarize.

$$\text{entropy}(B) = -\sum_{x \in \text{vocab}(B)} P_B(x) * \log(P_B(x)) \tag{4.9}$$

$$P_B(x) = \frac{\#\text{ occurrences of } x \text{ in } B}{\text{wordcount}(B)} \tag{4.10}$$

With the insight from Nenkova, we use the Rastkar corpus as a training set and find that ϕ (4.11), a relation between the three measurable properties of texts identified by Nenkova—entropy, word count, and vocabulary size—has a small, but significant correlation with the AUC for each bug in the Rastkar corpus—Spearman rank correlation of 0.20 and p-value < 0.03.

$$\phi(B) = \text{entropy}(B) \div \text{wordcount}(B) \div |\text{vocab}(B)| \tag{4.11}$$

To evaluate if ϕ can be used to suggest appropriate summary lengths for bug reports when using the ℓ_{all} summarizer, we use $\bar{\phi}$, the mean of ϕ, as a threshold to decide between two predefined summary lengths: a short summary if $\phi > \bar{\phi}$ or a long summary otherwise. We then measure the recall / precision ratio for the summaries for which the predictor decided for a longer summary, effectively measuring how much recall we can gain by every afforded 1% precision. Finally, we compare the recall / precision ratio resulting from using

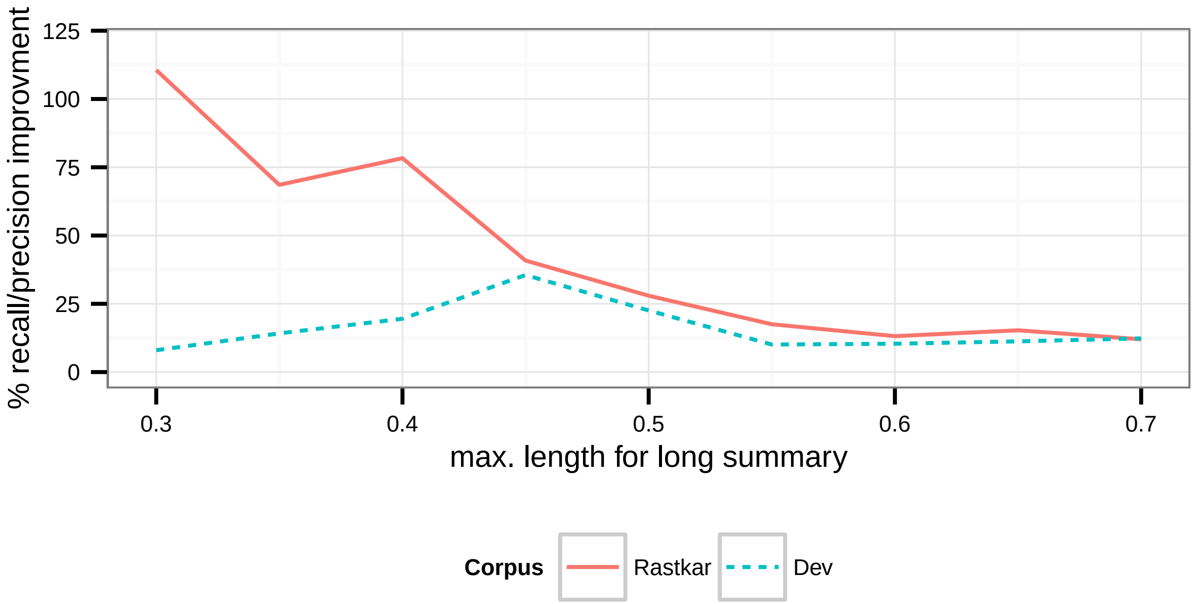


Figure 4.8: Precision/recall improvement over random

ϕ as a predictor for summary lengths to the recall / precision rates resulting from using a random predictor—we repeat the random experiment 100 times and take the average precision and recall.

Figure 4.8 shows the difference between the recall / precision ratio in percentage points, when using 25% as the short summary length and 30%, 35%, 40%, 45%, 50%, 55%, 60%, 65%, and 70% as the long summary lengths. The figure shows that, for the Rastkar corpus, when choosing between a 25% and a 30% long summary, the ϕ predictor creates summaries with 110% better recall / precision ratio than when randomly choosing between a 25% or 30% long summary. When deciding between a 25% and a 50% long summary, the ϕ predictor creates summaries with 25% improvement over the random predictor. As can be seen, although the improvements for the developer corpus are not as large as for the Rastkar corpus when the long summary varies from 30% to 40%, it does produce improvements of at least 8% over the random predictor, effectively showing that the ϕ predictor, which was originally tailored to the Rastkar corpus, is more general and should be applicable to other corpuses when using the ℓ_{all} summarizer.

4.5 Threats

4.5.1 Construct Validity

We measure precision and recall and pyramic precision and pyramid recall to evaluate the quality of bug reports. As we have discussed, these might not be such objective measures the quality of a summary. Nevertheless, these are the most used evaluation metrics for this purpose, including pyramid precision and recall which were created specifically for evaluating summaries.

4.5.2 Internal Validity

We use a comparison of the evaluation metrics with the summaries created by the email summarizer to test our hypotheses. We consider this is a reasonable first test, since Rastkar et al. claim the email summarizer produces good-quality summaries. To mitigate this threat, we also ask expert developers to assess our summaries and find that they consider our summaries useful. Another threat comes from our evaluation with developers. Since only 25% of invited developers agreed to participate, this might be a subset of developers that is already biased about the utility of bug report summaries.

PageRank has been used for text summarization before. TextRank [46] and Lexrank [23] also calculate sentence probability using textual similarity. Lexrank adds a post-processing after sentence ranking to avoid redundancy by excluding sentences that *subsume* other sentences. We are the first, however, to propose the use of PageRank for bug report summarization and to adapt it to the bug report domain, considering the importance of evaluation links and title and description similarity.

4.5.3 External Validity

We claim our summarizer should be widely applicable to any bug tracking system. We consider this is a reasonable claim, since we pose our hypotheses from an analysis of a set of bug tracking systems, but test our hypotheses on a different set of bug tracking systems, the ones from the corpus created by Rastkar et al. Furthermore, the risk of our summarization approach being *over-fitted* is small, since we do not use machine learning, but heuristics that should be valid in most bug tracking systems.

4.6 Summary

In this chapter, we have presented a novel, general bug report summarization approach that can be effectively used to reduce the amount of unreliable and irrelevant contributions in bug reports. In essence, our approach identifies relevant content as content that is repeatedly mentioned, that is evaluated by others, and that discusses the problem as described by the reporter. The summarizer we propose produces summaries with as much as 12% better precision than previous approaches. We have also tested the approach by asking open-source developers to evaluate the summary's quality and usefulness. Our finds show that developers appreciate the summaries and would indeed like to have a summarization tool at their disposition. Our work also shows that it is important to create a good interface to present the summaries, so that contributors can easily access related portions of the bug report or portions that were erroneously found to be irrelevant.

Chapter 5

Shifting Contributor Attention from Conversational to Informational Content to Increase Contribution Quality and Better Organize Bug Resolution Knowledge

This chapter proposes a new communication interface for bug reports that changes how data is collected and organized. It replaces the traditional forum-like user interface of today's bug tracking systems with an interface that encourages users to focus on the most relevant data for bug reports and create informational content instead of conversational content. Shifting the focus from conversational content to informational content should, thus, increase the amount of useful information and provide an interface better suited to organize bug resolution knowledge. We present a qualitative and quantitative evaluation of this novel bug tracking system on a team of nine software developers throughout a period of six months.

5.1 Increasing Archival Value and Facilitating Reasoning

As we have shown in Chapter 4, contributors value being able to consult bug reports and find relevant information quickly. Evermore, bug reports continue to be consulted much time after their submission date. A random sample of 600 bugs from the Mozilla, Chrome, and Debian bug tracking systems shows that 50% of bugs take at least 18, 31, and 71 days, respectively, to be closed, and 50% of bugs still receive comments 7, 12, and 27 days after being closed. This indicates that the attention that bug reports receive spans large periods of time.

Users should, therefore, be able to easily understand and locate relevant information the first time they consult a bug report or after much time since last consulting it. Users should be able to easily check if a failure they have experienced has already been acknowledged by the development community, check how the resolution of a particular bug is progressing, look for workarounds to similar issues, and gather analytics on previous bugs and their causes.

Contributors also need to be able to reason about information in bug reports so that they can form new hypotheses and move forward in the bug resolution process. In particular, since we have found from our bug report summarization study (Chapter 4) that developers consider that diagnostic information and solutions are the most important information, in general, contributors should be able to find, understand, and reason about diagnostic information and solutions. Developers should be able to reason about the execution results of test cases for a series of software revisions, for example, to conclude that a bug is a regression. Once the bug has been understood, to select the most appropriate solution, developers also need to compare the proposed solutions and reason about their differences.

Thus, we argue that a productive bug tracking system should have bug reports with high *archival value*—are easy to consult and understand for first-time readers—and its data should be organized to *facilitate reasoning* about diagnostics and about solutions.

5.2 Related Work in Mailing Lists, Forums, Q&A Sites

As presented in Section 2.2.6 and Section 2.2.7, organizing and collecting data as a linear sequence of comments sorted by time hinders contributors' ability to locate, understand, reason about, and analyze data since the linear comments do not represent well the complex, non-linear bug resolution process and the different of perspectives of its many contributors.

Mailing lists and community forums are other collaborative channels that face similar issues, since collaboration also develops as a conversation. Similar to bug tracking systems, content is the result of the communication that took place in order to address, solve, or discuss an issue. In fact, previous work studying communication in mailing lists and forums have found that such conversational content exchanged by numerous contributors spur frequent changes of topics in the discussion, making useful content hard to locate and digest [40, 43].

Mailing lists and forums have used threaded comments to try to better handle changes of topic. In threaded comments, comments are not sorted only by time. They are first grouped by some discussion topic. It is argued, however, that threaded comments are not successful in better organizing conversations since the topic of a thread is often not clear, and each topic thread often diverges into another number of topics.[1]

Q&A sites improve on mailing lists and forums by organizing information differently: they categorize content as being either a question or answer and allow localized comments to discuss each question or answer individually. Q&A sites, therefore, aim to reduce the importance of comments and to promote informational content. Our investigation of peer-reviewing in Stack Overflow (Section 3.5.2), for example, shows that comments are often used to suggest improvements to questions and answers and that contributors in fact often update their questions and answers with these suggestions.

Harper [31] argues, however, that many Q&A sites suffer much of the issues found in mailing lists and forums, since they allow conversational types of questions, which also have low archival value. Informational questions, however, have much higher archival value. Mamykina et al. [43] investigate Stack Overflow, one of the most successful Q&A sites, and find that the quality of its content comes from valuing information over conversation. This success is achieved, however, not only through a careful design of its interface and its game mechanisms, but also by its designers participating in the community and effectively discouraging conversations and promoting information.

5.3 Bug Tracking Systems, Reloaded

The approach we take to overhaul bug tracking systems builds upon the previous successes of Q&A sites, particularly of Stack Overflow, of discouraging conversational content and

[1] http://www.joelonsoftware.com/articles/BuildingCommunitieswithSo.html

promoting informational content; and from our study about bug report summarization, presented in Chapter 4, in which open source developers stated what information they consider most relevant in bug reports.

Here, we propose much more significant changes to bug tracking systems that the previous approaches presented. The use of game mechanisms and automatic summarization do not significantly change the bug report model, how its information is collected and organized. Game mechanisms are simply a means to motivate desirable contributor behavior and higher engagement, while summarization is a post-processing step applied to existing bug report content. We believe, however, that to significantly improve the productivity in bug tracking systems, communication and data storage must follow a different paradigm, one that is closer to the actual collaborative bug resolution process as described in Chapter 2.

5.3.1 Diagnostics and Solutions

To discourage conversational content, we eliminate comments as the main focus of bug reports, and ask contributors to post informational content relevant to bug resolution.

This and previous works find that information about bug diagnosis and solutions are the information most valued by developers in bug reports. Developers from our bug report summarization study—presented in Section 4.3.3—affirmed that information about diagnosis and solutions are the most relevant information in bug reports. Bettenburg et al. [10] and Breu et al. [15] also find that these are the information types most sought by developers. Bettenburg et al. [10] find that the fields in a bug report that are most valued by developers are about diagnostic information: *steps to reproduce*, *expected behavior* and *actual behavior*.

We, thus, allow only two main types of informational content as input from the user: either *diagnostic* or *solution* posts. Diagnostic information characterizes the bug and differentiates it from other bugs and is fundamental for developers to reproduce the bug. Such information includes the software version, steps to reproduce, external dependencies, and execution results (stack traces, screen shots, textual descriptions, etc). As for solutions, these are generally abstract or concrete proposals for how to resolve the issue, such as ideas or source code patches. By having contributors focus on the information most valued by developers, this should, thus, improve the cognitive fit between the mental representation of the bug and the bug report.

In contrast to how data is collected and stored in current bug reports, diagnostic and solution data should be presented appropriately so that it can be easily analyzed [28].

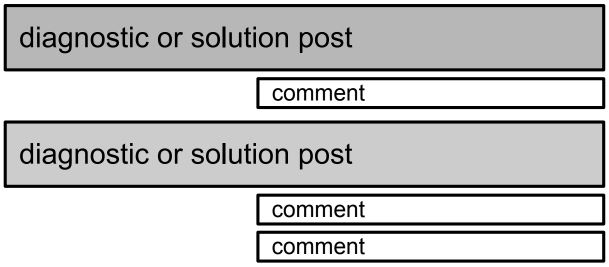


Figure 5.1: Layout for diagnostic and solution posts and nested comments.

Diagnostic and solution posts will be listed following the bug description as the main components of the bug report. Additionally, we will group and present first all diagnostic posts and then all solution posts.

5.3.2 Contextual Comments

Although conversations are difficult to read and digest, we believe that they are necessary for coordinating bug resolution and content creation, in general. Wikis are a prime example of a collaborative ecosystem for creating content with high archival value. To this end, wikis provide a 'background' channel for collaborators to discuss the content of wiki pages.

To support the discussion of diagnostic information and of solutions, which are crucial for the resolution of bugs [15], we provide the ability for users to post comments that are specific for each diagnostic or solution post. Such comments will be nested underneath its respective post and will be presented with secondary importance, as illustrated in Figure 5.1, which shows two diagnostic or solution posts (grey boxes), each followed by their own comments (white boxes).

5.3.3 Editing and Improving Content

Although we allow users to discuss diagnostic and solution posts, we consider that users consulting bug reports should not have to read the comments for diagnostics or solutions to find the information they seek, just as a user consulting a wiki page does not have to read its discussion page. To allow diagnostic and solution posts to be updated with the main resolutions and findings of the discussions from comments and continuously improved, we allow users to edit their own posts.

5.4 Evaluation

5.4.1 Methodology

To evaluate these ideas, we implement BugBot, a bug tracking system that implements the ideas from the previous section: diagnostic and solution posts, contextual comments, and editing and improving content.

We set BugBot as the bug tracking system for Clafer, a lightweight modeling language in active development, written in Haskell, that currently has 13k LOC. The bug tracking system has been used actively by the Clafer team since Feb 2012 by a total of 9 users. For this evaluation we will consider its usage from February to August 2012, at which it had a total of 138 bugs, of which 85 (62%) have been resolved and 48 (35%) are feature requests. Of the 9 users, 4 of them are Clafer developers and five are Clafer users. All 9 BugBot users have software development experience. To train users in using BugBot, we simply presented them a document outlining its philosophy and intended usage.

To understand the advantages and disadvantages of BugBot we use grounded theory, as proposed by Strauss and Corbin [58], triangulating interviews, an on-line questionnaire, and a qualitative and quantitative analysis of BugBot's usage (included in Appendix 6.2). Since the number of bug reports is not too large, we have read and analyzed all bug reports.

5.4.2 Findings

We now present our findings for each of BugBot's features.

Diagnostics and Solutions

Users found that having informational content about diagnostics and solutions greatly facilitated locating relevant information and consulting bugs since, differently from conversational comments, each post does not assume a predefined order and is highly independent of other posts.

> *The main benefits of BugBot is the ability to quickly skim through a Bug to see the diagnostic and evaluate whether such a bug is relevant or not to the question at issue.*

For the 138 bug reports, there was a total of 206 diagnostic and solution posts, of which 37% were diagnostic and 63% were solutions. Excluding 32 bugs that received zero posts, the mean number of posts per bug is 1.9 and median 2; 26 bugs received three or more posts.

Solution posts were mostly as expected: ideas and proposals for resolving the bug. Users took advantage of the ability to post several different solutions to discuss the benefits and drawbacks of each one. Diagnostic posts were mostly used to present execution results and a comparison with the expected results. Execution results were mostly either error messages, logs, or textual output. Apart from software revision, environment settings were hardly ever mentioned, most likely since Clafer's code is highly portable across operating systems. For feature requests, diagnostic posts were used to motivate the need for the requested improvement. During the first two weeks of usage, however, users were unclear about what information diagnostic posts should contain.

Although users appreciated the benefits of diagnostic and solution posts to improve archival value and facilitate reasoning, it was clear that users needed some time to get familiarized with the new approach. The biggest problem faced by users was deciding if their contribution should be a diagnostic or a solution post, since there are some content for which such distinction is unclear. A workaround to a bug, for example, might also be perceived as diagnostic information. Take the following post: "The software crashes when I invoke with the `--debug` flag. It doesn't crash without the flag." One might see that this should be posted as a solution, others will think that this is not a solution, but diagnostic information. We also found that many posts contained both diagnostic information and solutions and could have been split into two diagnostic and solution posts.

Users also found that some content they wished to post were *neither* diagnostic nor a solution, such as discussing who should fix the bug, or what the bug's priority is. For these cases, users used the comments section of the bug description to discuss priority and the comments section of the solution to discuss who will implement the solution. In one particular case, a user posted a solution proposing that the failure the bug report describes is not really a bug. The comments for this solution were then used to discuss this merit.

Contextual Comments

Contextual comments are clearly the most appreciated change over traditional bug tracking systems. They are effective in BugBot because each comment thread belongs to a specific diagnostic or solution post and thus has a clear topic. The following comments from two users summarize its advantages:

> *My favourite feature is the nesting. It makes reading bug reports simpler since all the discussion and replies are grouped together.*
>
> *It is especially important if there are multiple solutions and we need to discuss each of them. [...] It is easier to distinguish more important threads and to disregard those that are either incorrect or insignificant.*

Most posts received at least one comment. Of the 206 posts, 44% received no comments, 33% received one comment, 10% received two comments, and 13% received three or more comments. The maximum number of comments for a post was 12, for a contentious bug that required much explanation of why it was not a bug. Solution posts, in average, receive 50% more comments than diagnostic posts: of the total of 266 comments, 76 were for diagnostic posts, 190 for solution posts.

Comments for diagnostic posts were mostly used to verify or dispute the validity of the diagnostic. Comments for solutions were used mostly to get feedback about the solution: if it is a good idea, if it has been proven to resolve the issue or not, its execution results, and ideas for improving it. Comments for solutions were also often used to disclose the status of solutions: in development, committed, released, etc. A user wanting to understand if a diagnostic post was valid or if a solution has been proven to work, need only read the comments for the post. Finally, comments for solutions were also used for coordination, such as deciding who will implement the solution.

Comments, in general, also allowed users to digress, which is what commonly occurs in typical bug tracking systems. Common examples of this is users discussing interesting, but not crucial, details about solutions, and posting references to external sources of information.

Just as users were sometimes unsure if their contribution should be posted as a diagnostic or solution, users were also unsure if they should contribute with a comment or with a new diagnostic or solution. A user wanting to suggest an improvement to a solution, for example, could either suggest it through a comment or post a new solution that extends the original solution with improvements.

Editing and Improving Content

In traditional bug tracking systems, content improvement can only be done by posting additional comments, adding more noise to the bug report, as a result. BugBot users claimed that being able to edit posts increases the archival value of bugs since it does not require additional comments. And most edits were substantial: 50% of them having a *Levenshtein edit distance* of 100 characters or more. 15 posts were edited three or more times.

We found, however, that this feature was not used as much as it could have been: only 35% of diagnostics posts and 27% of solutions were edited. Users are aware of this problem:

> *Our users do not have enough self discipline and motivation to write the solutions that actually were implemented. Sometimes, the solutions are what was proposed, something slightly different was implemented, but the solutions were not updated.*

This could be explained by two findings. First is that users stated that editing existing posts is time consuming, since they have to revise an existing content. Users were also not clear on the benefits of editing content for *all bugs*. They consider it is most beneficial for bugs with higher activity and importance. Second is that users were again unsure, similar to the decision of posting a comment or a new post, if they should either improve a diagnostic or solution by editing it, by posting a new one, or by adding a comment.

Effects on Archival Value and Reasoning

Users were convinced that BugBot at least marginally improved archival value of bugs, compared to traditional bug tracking systems, with three users affirming that it significantly improved archival value. The features that mostly improved archival value, in the perception of users, were contextual comments and being able to edit posts.

Users were also convinced that BugBot improved reasoning, compared to traditional bug tracking systems; only one user said that it was only marginally better. Users found that diagnostics and solutions and contextual comments were the features that most contributed to facilitate reasoning, since it allows bug report information to be structured hierarchically (via contextual comments) and topically (via diagnostics and solutions) which "*... is helpful for building a mental model of the bug report*".

5.4.3 Discussion

This evaluation finds that users were much satisfied with BugBot. Users clearly perceived that classifying posts as diagnostic or solution, having contextual comments, and editing posts allowed significant improvements to archival value and to facilitating reasoning. All but one user, who was unsure about his preference, affirmed that they would choose BugBot over traditional bug tracking systems and three users affirmed that it is significantly better than traditional bug tracking systems. In fact, some users asked for further installations of BugBot for other projects.

Nevertheless, BugBot can be improved. The first issue comes from leaving the responsibility of organization to users: they must decide if they should contribute by commenting or editing an existing post or by posting a new diagnostic or solution. We acknowledge that there is no rule of thumb for when to post something as a diagnostic, solution, comment, or edit. Users will have to decide on this depending on the situation, and there will certainly be disagreements. Given the advantages that it brings, however, we consider that many communities will appreciate the afforded power and learn to deal with the learning curve.

Users might also find BugBot to be too restrictive, since it only allows users to post diagnostics or solutions. From our evaluation, however, users managed to discuss everything that they needed to discuss. We consider that feeling restricted is a result of the learning curve that BugBot presents to new users, which is natural, since BugBot breaks several paradigms.

> *The structured entries [...] and certain philosophy we could follow allowed us to improve the quality of the bug/new feature reports. [...] We could, with some discipline [...] replicate the same in a traditional bug tracker. It wouldn't be that nice and convenient, however.*

Finally, users find editing posts to be time consuming. Since they acknowledge the advantages of keeping posts updated, we argue that this could be enforced or encouraged through moderation. A promising approach for encouragement, as we have shown in Chapter 3, is Stack Overflow's game mechanisms, which encourages desirable behavior by rewarding users with administrative privileges and reputation points [43].

Overall, BugBot motivates future work on creating ecosystems optimized in helping large groups of users solve problems as a collective, primarily by organizing the enormous amounts of input they should receive. Bug tracking systems are just one case in which problem-solving is performed collaboratively. The open-source movement, which is expanding to areas other than software development, is a prime driver of this need.

5.5 Threats

5.5.1 Internal Validity

The main threats of this study relates to the group of users who evaluated BugBot and the project it was used for. Clafer team and the first author of this paper work physically close to each other in the same research laboratory, increasing the potential for biased responses. The Clafer developer group, however, is an independent and professional team with complete authority to decide which bug tracking system they will use. They would not have kept using BugBot if they found it unuseful or unproductive. They are still using it, plan to continue using it indefinitely, and have asked for further installations for other projects.

The Hawthorne effect might also affect our results. Users might have behaved in a different fashion just because they were being evaluated. Since this evaluation lasted for 6 months, we consider that users did not try to keep up good behavior for such long periods.

5.5.2 External Validity

Clafer might not be representative of all types of software due to its particular domain and small size. In addition, we received responses from only 7 users which might not be representative of the all the population of bug tracking system users.

5.6 Summary

In this chapter, we have presented BugBot, a bug tracking system that changes the bug report communication interface so that contributors can focus on posting structured information instead of mixing data with conversations within comments. BugBot removes comments as the main elements in bug reports and replaces them with posts with information on either diagnostics of solutions. Each diagnostic or solution allows its own set of contextualized comments. Furthermore, BugBot allows contributors to edit exiting diagnostic and solution posts. Our evaluation over 6 months of usage shows that users value the shift of focus from conversational content to informational content about bug diagnosis and solutions. Categorizing information as diagnostic or solutions and contextualized comments facilitates search and locating relevant information, thus reducing noise; and content editing allows users to improve content as the information about the bug is improved over time. Overall, users found that such features improve productivity by making it easier for contributors to reason about data in bug reports and to create bug reports with higher archival value. Although this new paradigm for collecting and organizing information has a learning curve, users found that BugBot offers a better interface to collect and organize bug resolution knowledge, compared to traditional bug tracking systems, even considering the paradigm shift.

Chapter 6

Conclusion

This dissertation has shown that, although bug tracking systems are fundamental to support virtually any software development process, current bug tracking systems are suboptimal to support the needs and complexities of large communities. The way that current bug tracking systems collect and present data invites much noise into bug reports, creating an ecosystem in which it is difficult for contributors to locate, understand, and reason about relevant information for bug resolution.

To make bug tracking systems more productive for large communities we have shown that bug tracking systems should use a combination of game mechanisms, to create an active community where contributors strive to post high quality content; and of automated and manual facilities to better structure and organize information in bug reports.

In Chapter 2, we have presented the issues brought by noisy bug tracking systems and the extent to which noise affects contributor productivity. We have also analyzed the factors that increase noise in bug tracking systems and argued that the way current bug tracking systems collect and present data as a linear sequence of comments is a prominent source for noise. The linear sequence of comments invites overlapping conversations and do not fit well with the complex problem-solving nature of bug resolution. We have also shown that the large number of contributors further increases noise in bug tracking systems since this brings larger numbers of inexperienced users contributing low-quality content and spurring more conflicts.

To reduce the number of low-quality contributions and conflicts, we have presented, in Chapter 3, how game mechanisms can be used to create a healthy and engaged online software development community that will strive to post better and better contributions. Our findings show that game mechanisms are able to push contributors to new heights of achievements. As a result, users contribute more frequently, contribute higher quality content, and review and suggest improvements to their peers' contributions. We also show that a well-crafted reputation and rewards system, that creates a population-pyramid of privileges by awarding moderation privileges to users, creates a dependable and agile moderation system that should be capable of quickly detecting and resolving conflicts. We also find that the more time a user has invested in the ecosystem, the less likely the user is to abandon the community and the more the user will play a role at reviewing and suggesting improvements to existing content.

Adding game mechanisms to bug tracking systems is simple. In our proposal, bug tracking systems should allow users to recognize and down-vote bug reports and comments. As such, we would expect that important, useful, bug reports should be recognized, whereas irrelevant bug reports should receive down-votes to discourage further noise. Comments with useful information should also receive recognition-votes from contributors, while comments with irrelevant content should receive down-votes. In addition to offering the ability of recognizing and down-voting bug reports and comments, bug tracking systems also need a rewards and reputation system, similar to Stack Overflow's, in which every recognition vote increases a contributor's reputation score. And, as users reach certain reputation levels, users are awarded with already existing bug tracking systems privileges, such as closing a bug report, marking as a duplicate, changing bug report status, and moderation.

Our analysis also studies the conditions for game mechanisms to be successful in increasing contribution quality. We find that a well crafted and tailored rewards system is essential for such success. The rewards system must reward users with privileges that they appreciate and will fight for. Most interestingly, we find that the rewards that give contributors higher levels of moderation privileges—in essence, more *power*—are the rewards that are able to most motivate users.

We have also shown that game mechanisms can be used to reduce noise: content that has been recognized is likely to be important and useful, while content that has not been recognized or has received down-votes are likely to be irrelevant. In light of this, we have presented the effectiveness of current recognition mechanisms in bug tracking systems to detect useful bug reports. We find, however, that current recognition mechanisms are not so effective at identifying bug reports that will be fixed. We thus suggest and show that considering duplicate bug reports as a form of recognition of the original bug report significantly increases the accuracy for prioritizing bug reports.

While we show that manual recognition-votes can be used to highlight useful information, we have also presented an automated method for identifying useful information in bug reports, in Chapter 4. This approach models the 'hurried' bug report reading process using a Markov chain, where nodes are sentences in a bug report and weighted edges between sentences represent the probability that a reader will read another sentence next. We find that, after reading a sentence, the most likely next sentence to be read should be one that discusses similar topics to the previous sentence, that has been evaluated by the previous sentence, and that discusses the same problem as described in the bug report description. Our evaluation, first of all, shows that this summarizer produces higher quality summaries than previous approaches and has virtually zero setup or configuration cost. Furthermore, open-source developers presented with summaries for bug reports that they had worked on, affirmed that the summaries are of reasonable quality and that they would appreciate the summaries to help them work with bug reports. We also presented two alternative views for the summaries, a condensed view and a interlaced view. Developers found that both views have their advantages and disadvantages and that it would be useful to switch back-and-forth between the views when consulting a bug report.

While game mechanisms and summarization should go a long way in improving productivity in bug tracking systems, they only re-mediate the issues brought by how current bug reports use comments to collect and present information. Comments will still not be an optimal way to collect and organize information. Thus, we present, in Chapter 5, a different approach for collecting and organizing information. This approach reduces the importance of conversational comments and enhances the importance of informational content, in particular, information about bug diagnosis and solutions, which our and previous studies find are the most important information in bug reports. We implement BugBot, a novel bug tracking system that asks users to post either informational posts about diagnostics or solutions. Each diagnostic post or solution has its own set of comments that should be used to discuss the diagnostic or solution post, to ask for clarification, and to suggest improvements. To make these improvements, BugBot allows users to edit existing diagnostic and solution posts. Our evaluation shows that users find BugBot to be significantly better than traditional bug tracking systems since it allows users to create bug reports with content that is easier to locate and understand—have higher archival value—and that BugBot presents information in a way that facilitates reasoning.

Each of the three approaches to improve bug tracking systems play a different role in improving bug tracking systems. Game mechanisms use a behavioral approach to motivate users to post higher quality contributions and peer-review and moderate contributions; bug report summarization uses heuristics to identify and filter high-quality contributions; and BugBot asks for users to post only specific relevant informational content to increase contribution quality and isolates conversational comments to reduce noise.

The three approaches, however, are quite orthogonal and can be used together, each one benefiting from the other. We will present or vision for how this could play out in

Section 6.2. First, we will discuss the limitations of our work.

6.1 Limitations

Although we have presented solid empirical evaluations for each part of our work, our evaluations are inevitably limited.

6.1.1 Construct Validity

We have set out to improve contribution quality and reduce noise in bug tracking systems so that they become more productive ecosystems. None of our evaluations, however, attempt to objectively check how much the approaches increase bug tracking system productivity. First of all, productivity is difficult to measure objectively [4]. Furthermore, bug tracking systems are used for a variety of different tasks, such as simply consulting a bug report, looking for duplicate bug reports, looking for a bug report to fix, and fixing a bug. A complete evaluation would, therefore, require us to evaluate increased productivity in all of these tasks. Instead of evaluating productivity increases, this work assumes that an increase in contribution quality and a reduction in noise will lead to higher productivity rates for consulting, understanding, reasoning about, and thus, fixing bug reports.

6.1.2 External Validity

Our investigations of the use of game mechanisms in bug tracking systems and of BugBot have notable limitations. We consider that an optimal evaluation of the effects of game mechanisms in bug tracking systems would be an empirical investigation of a real, large software development community using a bug tracking system with game mechanisms. Such an investigation, however, is considerably challenging, starting with the requirement of finding a real, large community willing to test such a bug tracking system. We consider that testing game mechanisms with a small team will not lead to any significant findings since the game mechanisms will likely not be compelling to small teams, as we have argued in Section 3.6.2. For these reasons, we decided that the scope of our study would not support such a large empirical study and we, thus, settled for studying Stack Overflow and inferring that the conditions present in Stack Overflow for the game mechanisms to create its benefits are also present in bug tracking systems.

Our evaluation of BugBot has similar limitations. We evaluated BugBot on a small team since we considered that a small team would benefit from and appreciate the advantages of better organized bug reports. However, since we evaluated it with only one software project and a small team, the generalizability of our findings is limited.

6.1.3 Internal Validity

Our evaluation of BugBot is also mostly a subjective evaluation. We've asked its users how they *feel* that BugBot improves the archival value of bugs and facilitates reasoning, compared to traditional bug tracking systems. A better evaluation should objectively compare archival value and reasoning in BugBot and traditional bug tracking systems. Similarly, our work on bug report summarization asks developers how they feel bug report summaries should help them with common tasks in bug tracking systems. While we have objective measures of the quality of the summaries, it is still not clear how much such summaries would effectively increase productivity. Nevertheless, the response from developers affirming that they value bug report summaries suggest they have significant discomfort working with bug reports.

6.2 Future Directions

We now present our vision of how the contributions of our work can be used to create the next generation of bug tracking systems for large communities.

Improving Summarization with Game Mechanisms

Our work has shown how contributors value tools that help them identify and locate relevant information in bug reports. Our work on bug report summarization has shown that it is important to be able to identify sentences that evaluate other sentences, since these sentences are more likely to be relevant. The approach we present to identify evaluation sentences, however, is trained from a corpus of messages from Twitter, which contains content completely unrelated to software development. Thus, a promising line of investigation to improve the identification of evaluation sentences would be to use a corpus of sentences from bug reports, studying the characteristics of communication in bug reports, beyond emoticons, that indicate different types of evaluations.

Nevertheless, it is important to note that evaluation sentences are not much different from the concept of a recognition vote or down-vote in game mechanisms. Most evaluation sentences agree or disagree with a claim in a previous comment and presents the rational for the agreement or disagreement. For the running bug report example shown in Figure 4.1, for example, sentence $s_{2,1}$ could, thus, be replaced by a down-vote for sentences $s_{0,0}$ and $s_{1,0}$ together with the rational for the disagreement.

Thus, adding game mechanisms to bug tracking systems, as we have proposed, and allowing contributors to recognize or down-vote comments or portions of comments, could be used to improve the detection of evaluation sentences and, therefore, further improve our bug report summarization approach.

Game mechanisms can also be added to BugBot: contributors could be able to recognize or down-vote bug reports, diagnostic or solution posts, or comments. While our summarization approach does not seem to be applicable to BugBot, BugBot would still benefit from such recognition-votes for filtering useful information: diagnostics and solution posts could be sorted not by submission time but rather by the number of recognition-votes, as done in Stack Overflow.

Game Mechanisms Motivate Lightweight Modeling

Allowing contributors to recognize or down-vote useful or irrelevant data in bug reports requires users to chunk bug report content into smaller information pieces. We suppose users might decide, in some cases, that the data granularity for an evaluation is an entire comment while, in other cases, it might be one or two sentences containing a pertinent observation within a comment with other irrelevant sentences.

Once contributors are breaking bug report data into pieces, categorizing such data should be significantly simple and relevant. Our evaluation of bug report summarization with developers and our evaluation of BugBot, for example, has shown that diagnostic information and solutions are the most important in bug reports. For the running example shown in Figure 4.1, for example, it would be relevant to categorize sentences $s_{0,0}$, $s_{1,0}$, $s_{2,1}$, $s_{4,2}$, and $s_{5,2}$ as diagnostic information. It would also be important for future work to investigate further relevant data categories. It might be relevant, for some use-cases, to have sub-categories for diagnostic information, such as operating system, RAM, steps to reproduce, or others.

Data chunking and categorization is, in essence, a form of lightweight modeling [36] for bug reports. The objective of light-weight modeling is to add some formalism to data such that the additional meta-data will enable some useful automation, such as verification, simulation, or organization, at a low-cost to users. In the case of bug reports, data chunking and categorization is an attempt to formalize its informal content. The addition of game mechanisms to data categorization is another step at formalization. Data that has received many recognition-votes can likely be considered to be valid—formal—while data that has received down-votes should likely be considered as invalid.

Bug tracking systems allowing contributors to annotate information could allow contributors to either categorize information with no restrictions to the chosen categories or they could allow contributors to only select categories from a predefined list. To enable the former option, however, an ontology of bug report information would have to be built. Such an ontology would add another level of formalism to the annotations.

The benefits of such lightweight modeling for bug reports are many. Zimmermann et al. [68] also suggest many benefits that an improved data semantic in bug reports should lead to. The first important possible benefit of better structured bug reports is duplicate bug report detection. Previous work has shown that simply textual similarity in bug report descriptions are not enough to achieve good recall and precision for duplicate bug report identification [32, 37, 54, 60, 64], but a more precise, formal, bug description might provide important features for disambiguation. Furthermore, once data in bug reports is focused on information and not on conversation, such as proposed in BugBot, merging the information about duplicate bug reports should be simply a matter of putting all the information into the same bag: since existing conversations are contextual and belong to their own informational posts, they can, thus, be moved around together with their posts. Additionally, along the lines of the work of Bettenburg et al. [10], an ontology of bug report information together with a history of annotated bug reports could be used to determine the relevant information for each type of bug. A bug tracking systems could then ask bug reporters for missing information even before they complete a bug report submission, and avoid developers having to ask for such information and delaying bug resolution, as occurs today [15, 35].

Our work suggests that light-weight modeling could also be useful for improving bug summarization. For our bug report summarization approach, we have shown that it is important to be able to measure how much two sentences talk about similar topics. This work, however, uses simple lexical similarity metrics to approximate such measure. Once bug report content has been categorized, such meta-data can be used infer the topics being discussed and, thus, improve topic similarity detection. Additionally, once topics are identified, it might be relevant to give them different weights: the topic of solutions to a bug, for example, could be considered more relevant than the topic of who is fixing the bug, for example.

Although lightweight modeling for bug reports have many promises, the main challenge to enable its advantages is of creating a user interface that allows and encourages lightweight modeling bug does not overwhelm users by its complexity and formalism, and that, most importantly, users see an advantage in using. For users to feel the advantage of using the lightweight modeling features, they must clearly appreciate that the meta-data they input allows them benefits that were not possible without the meta-data. It is also likely that game mechanisms could be used to further motivate user engagement on lightweight modeling by rewarding users that add useful annotations to information.

Reorganizing Data based on Lightweight Modeling

Our work on bug report summarization in Chapter 4 has shown how developers find that different views of bug report summaries have their own advantages and disadvantages. Designing optimal interfaces to facilitate bug report navigation is, thus, another important direction of work.

BugBot presents an alternative, yet simple, way to present information. As we have hinted, BugBot is an instance of a bug tracking systems with lightweight modeling. In BugBot, information is categorized either as diagnostic or solution. BugBot takes advantage of such meta-data and uses it to cluster all diagnostic posts together, followed by all the solution posts. However, once information in bug reports is annotated with categories from a rich ontology containing more than simply diagnostic or solution categories, such annotations might invite different forms of presenting and organizing information in bug reports. Such organization could be tailored to a users preferences and objective. A trivial example is a user looking for a workaround to resolve a bug. In such case, an optimal bug report should highlight the posts about solutions and workarounds, leaving posts about diagnostics in the background. A more envolved case is a contributor comparing two bug reports to decide if one is a duplicate of another. In such case, an optimal view would be a line-up of the two bugs, highlighting information that seems common between the two bug reports.

Reducing the Learning Curve for BugBot

Although our work on BugBot has shown that categorizing information posts as either diagnostic or solution has many advantages, our evaluation shows that there is a significant learning-curve for using BugBot comfortably. In particular, users found that it is not always clear how some posts should be categorized. A future direction to minimize this problem is to attempt to aid users in submitting their contributions by using NLP and machine learning to automatically classify a user's contribution.

Our and Mamykina's [43] findings from Stack Overflow also suggest that a willing community *can be taught* how to use such a system in the correct way. Game mechanisms and moderation provide extrinsic motivation, which can turn into intrinsic motivation once users see the benefit of such organized information.

Crafting Attractive Reputation and Rewards Systems for Software Developers

As we have seen, the most important condition for game mechanisms to produce the benefits we expect is that the community accepts and is motivated by the formal merit-based reputation system and its rewards. Since different communities have different interests, some communities might not accept such a merit-based reputation system. Android, for instance, as seen in Section 3.6.3, shows very little response to votes compared to Chrome, Launchpad, and Mozilla, suggesting that current contributors are not so open to input from outside contributors.

Creating an attractive merit-based ecosystem requires an understanding of the unique motivations of different open-source contributors—payed, non-payed, enthusiast, beginner, developer, non-developer—in different projects, of different sizes, and guidelines on applying and tailoring a reputation and reward system according to contributor's profiles. Core developers, for instance, will probably not be so interested in privileges for themselves but they might be interested in motivating users to use it, since it should bring valuable crowd-sourcing effort to help developers.

References

[1] B. Thomas Adler and Luca de Alfaro. “A content-driven reputation system for the wikipedia”. In: *Proceedings of the 16th International Conference on World Wide Web*. ACM, 2007.

[2] B. Thomas Adler, Krishnendu Chatterjee, Luca de Alfaro, Marco Faella, Ian Pye, and Vishwanath Raman. “Assigning trust to Wikipedia content”. In: *Proceedings of the 4th International Symposium on Wikis*. ACM, 2008.

[3] Anupriya Ankolekar, Katia Sycara, James Herbsleb, Robert Kraut, and Chris Welty. “Supporting online problem-solving communities with the semantic web”. In: *Proceedings of the 15th international conference on World Wide Web*. ACM, 2006.

[4] Donald Anselmo and Henry Ledgard. “Measuring productivity in the software industry”. In: *Communications of the ACM* 46.11 (2003).

[5] John Anvik, Lyndon Hiew, and Gail C. Murphy. “Coping with an open bug repository”. In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. ACM, 2005.

[6] Jorge Aranda and Gina Venolia. “The secret life of bugs: Going past the errors and omissions in software repositories”. In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[7] Andrea Arcuri and Lionel Briand. “A practical guide for using statistical tests to assess randomized algorithms in software engineering”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011.

[8] Philip Beineke, Trevor Hastie, Christopher Manning, and Shivakumar Vaithyanathan. “Exploring sentiment summarization”. In: *AAAI Spring Symposium on Exploring Attitude and Affect in Text: Theories and Applications (AAAI tech report SS-04-07)*. 2004.

[9] Magnus Bergquist and Jan Ljungberg. "The power of gifts: organizing social relationships in open source communities". In: *Information Systems Journal* 11.4 (2001).

[10] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. "What makes a good bug report?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2008.

[11] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. "Duplicate Bug Reports Considered Harmful... Really?" In: *Proceedings of the 24th IEEE International Conference on Software Maintenance.* 2008.

[12] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. "Extracting structural information from bug reports". In: *Proceedings of the 2008 international working conference on Mining software repositories.* ACM, 2008.

[13] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation". In: *Journal of Machine Learning Research* 3 (2003).

[14] Barry Boehm and Victor R. Basili. "Software Defect Reduction Top 10 List". In: *Computer* 34.1 (2001).

[15] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. "Information needs in bug reports: improving cooperation between developers and users". In: *Proceedings of the 2010 ACM conference on Computer Supported Cooperative Work.* ACM, 2010.

[16] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual Web search engine". In: *Computer Networks and ISDN Systems* 30.1-7 (1998).

[17] Stefan Büttcher, Charles Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* The MIT Press, 2010.

[18] Peter H. Carstensen, Carsten Sørensen, and Tuomo Tuikka. "Let's talk about bugs!" In: *Scandinavian Journal of Information Systems* 7.1 (1995).

[19] D. Cohen. "On Holy Wars and a Plea for Peace". In: *Computer* 14.10 (1981).

[20] Chrysanthos Dellarocas, Ming Fan, and Charles Wood. "Self-interest, reciprocity, and participation in online reputation systems". In: *MIT Sloan Working Papers* (2004).

[21] Bogdan Dit and Andrian Marcus. "Improving the readability of defect reports". In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering.* ACM, 2008.

[22] Nicolas Ducheneaut, Nicholas Yee, Eric Nickell, and Robert J. Moore. “Alone together?: exploring the social dynamics of massively multiplayer online games”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2006.

[23] Günes Erkan and Dragomir R. Radev. “LexRank: graph-based lexical centrality as salience in text summarization”. In: *Journal Artificial Intelligence Research* 22.1 (2004).

[24] Roy T. Fielding. “Shared leadership in the Apache project”. In: *Communications of the ACM* 42.4 (1999).

[25] Karl Fogel. *Producing Open Source Software – How to Run a Successfull Free Software Project*. O’Reilly Media, 2010.

[26] Les Gasser and Gabriel Ripoche. “Distributed collective practices and free/open-source software problem management: perspectives and methods”. In: *2003 Conference on Cooperation, Innovation & Technologie*. Citeseer. 2003.

[27] Alec Go, Richa Bhayani, and Lei Huang. “Twitter sentiment classification using distant supervision”. In: *CS224N Project Report, Stanford* (2009).

[28] Thomas R. G. Green and Marian Petre. “Usability Analysis of Visual Programming Environments: A ’Cognitive Dimensions’ Framework”. In: *Journal of visual languages and computing* 7.2 (1996).

[29] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. “Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows”. In: *Proceedings of the 32nd International Conference on Software Engineering*. ACM, 2010.

[30] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. “Not my bug! and other reasons for software bug report reassignments”. In: *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011.

[31] F. Maxwell Harper, Daniel Moy, and Joseph A. Konstan. “Facts or friends?: distinguishing informational and conversational questions in social Q&A sites”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009.

[32] Lyndon Hiew. “Assisted Detection of Duplicate Bug Reports”. MA thesis. The University of British Columbia, 2006.

[33] Starr Roxanne Hiltz, Kenneth Johnson, and Murray Turoff. "Experiments in Group Decision Making Communication Process and Outcome in Face-to-Face Versus Computerized Conferences". In: *Human communication research* 13.2 (1986).

[34] Thomas Hofmann. "Probabilistic latent semantic indexing". In: *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1999.

[35] Pieter Hooimeijer and Westley Weimer. "Modeling bug report quality". In: *Proceedings of the 22nd international conference on Automated Software Engineering*. ACM, 2007.

[36] Daniel Jackson. "Lightweight Formal Methods". In: *FME 2001: Formal Methods for Increasing Software Productivity*. Springer, 2001.

[37] Nicholas Jalbert and Westley Weimer. "Automated duplicate detection for bug tracking systems". In: *The 38th Annual International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*. IEEE Computer Society, 2008.

[38] Andrew J. Ko and Parmit K. Chilana. "Design, discussion, and dissent in open bug reports". In: *Proceedings of the 2011 iConference*. ACM, 2011.

[39] Andrew J. Ko and Parmit K. Chilana. "How power users help and hinder open bug reporting". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010.

[40] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. "GroupLens: applying collaborative filtering to Usenet news". In: *Communications of the ACM* 40.3 (1997).

[41] Karim R Lakhani and Eric von Hippel. "How open source software works: free user-to-user assistance". In: *Research Policy* 32.6 (2003).

[42] Elena Lloret and Manuel Palomar. "Text summarisation in progress: a literature review". In: *Artificial Intelligence Review* 37.1 (2012).

[43] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. "Design lessons from the fastest Q&A site in the west". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2011.

[44] Abraham H. Maslow. "A theory of human motivation". In: *Psychological review* (1943).

[45] Jane McGonigal. *Reality Is Broken: Why Games Make Us Better and How They Can Change the World*. Penguin Press, 2011.

[46] Rada Mihalcea and Paul Tarau. “TextRank: Bringing order into texts”. In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (2004).

[47] Audris Mockus, Roy T. Fielding, and James Herbsleb. “A case study of open source software development: the Apache server”. In: *Software Engineering, 2000. Proceedings of the 2000 International Conference on.* 2000.

[48] Gabriel Murray and Giuseppe Carenini. “Summarizing spoken and written conversations”. In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2008.

[49] Ani Nenkova and Annie Louis. “Can you summarize this? Identifying correlates of input difficulty for generic multi-document summarization”. In: *Penn Engineering Departmental Papers* (2008).

[50] Ani Nenkova, Rebecca Passonneau, and Kathleen McKeown. “The Pyramid Method: Incorporating human content selection variation in summarization evaluation”. In: *ACM Trans. Speech Lang. Process.* 4.2 (2007).

[51] Martin F Porter et al. *An algorithm for suffix stripping.* 1980.

[52] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. “Summarizing software artifacts: a case study of bug reports”. In: *Proceedings of the 32nd International Conference on Software Engineering.* ACM, 2010.

[53] Eric S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary.* O’Reilly & Associates, Inc., 2001.

[54] P. Runeson, M. Alexandersson, and O. Nyholm. “Detection of Duplicate Defect Reports Using Natural Language Processing”. In: *Proceedings of the 29th International Conference on Software Engineering.* 2007.

[55] Walt Scacchi. “Free/open source software development: Recent research results and methods”. In: *Advances in Computers* 69 (2007).

[56] Vibha Singhal Sinha, Senthil Mani, and Saurabh Sinha. “Entering the circle of trust: developer initiation as committers in open-source projects”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories.* ACM, 2011.

[57] Qinbao Song, Martin Shepperd, Michelle Cartwright, and Carolyn Mair. “Software Defect Association Mining and Defect Correction Effort Prediction”. In: *IEEE Trans. Softw. Eng.* 32.2 (2006).

[58] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications, 2008.

[59] Bingjun Sun, Prasenjit Mitra, C. Lee Giles, John Yen, and Hongyuan Zha. “Topic segmentation with shared topic detection and alignment of multiple documents”. In: *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval.* ACM, 2007.

[60] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. “A discriminative model approach for accurate duplicate bug report retrieval”. In: *Proceedings of the 32nd International Conference on Software Engineering.* ACM, 2010.

[61] Jian Sun. “Why are Bug Reports Invalid?” In: *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation.* 2011.

[62] Huifeng Tang, Songbo Tan, and Xueqi Cheng. “A survey on sentiment detection of reviews”. In: *Expert Systems with Applications* (2009).

[63] Davor Čubranić and Gail C. Murphy. “Hipikat: recommending pertinent software development artifacts”. In: *Proceedings of the 25th International Conference on Software Engineering.* IEEE Computer Society, 2003.

[64] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. “An approach to detecting duplicate bug reports using natural language and execution information”. In: *Proceedings of the 30th International Conference on Software Engineering.* ACM, 2008.

[65] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. “How Long Will It Take to Fix This Bug?” In: *Proceedings of the Fourth International Workshop on Mining Software Repositories.* IEEE Computer Society, 2007.

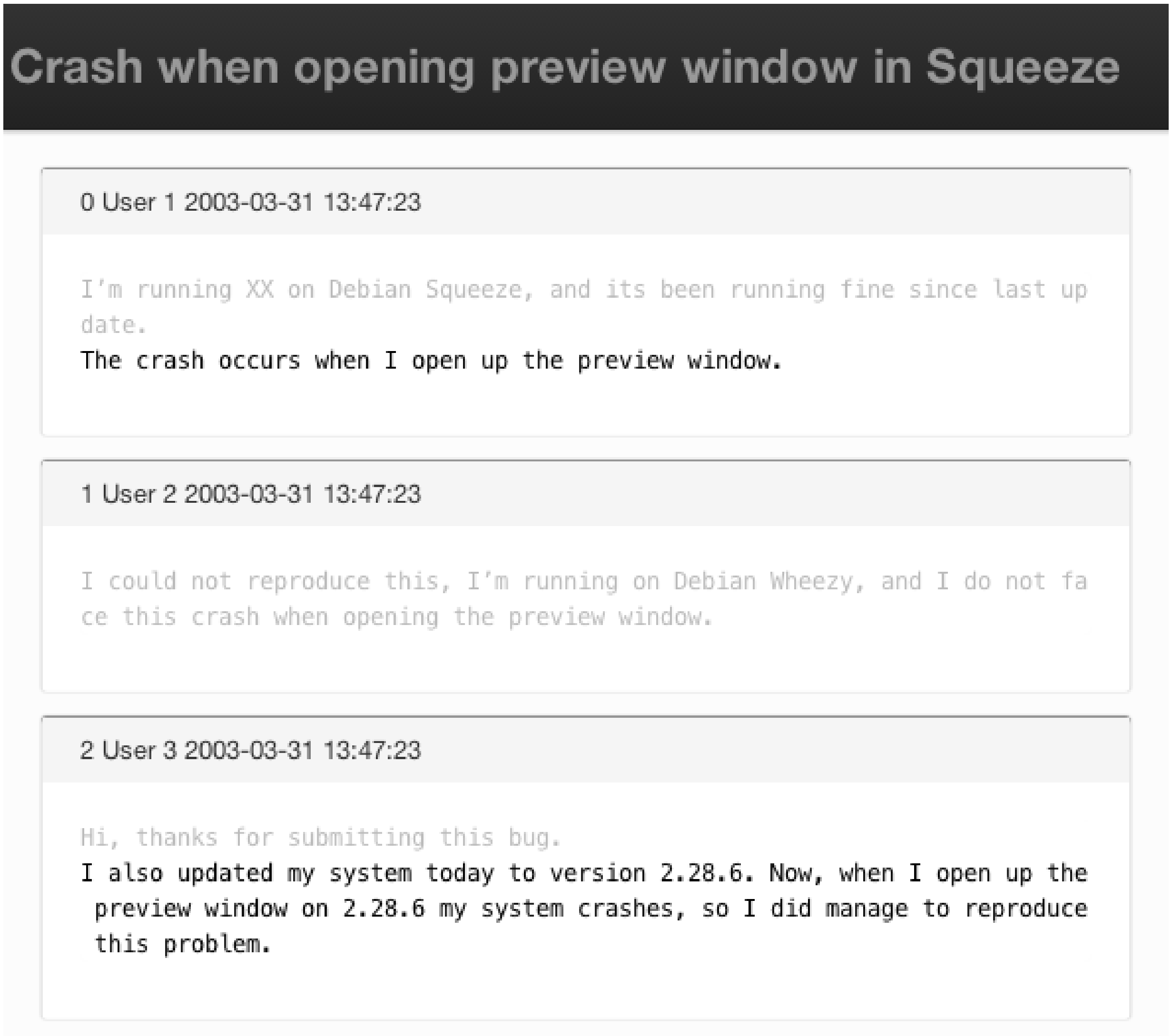
[66] J. Christopher Westland. “The cost behavior of software defects”. In: *Decis. Support Syst.* 37.2 (2004).

[67] C Wu, J Gerlach, and C Young. “An empirical analysis of open source software developers motivations and continuance intentions”. In: *Information & Management* (2007).

[68] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. “Improving Bug Tracking Systems”. In: *Companion to the 31th International Conference on Software Engineering.* 2009.

Appendix A

Interlaced and Condensed Summary Views



Crash when opening preview window in Squeeze

0 User 1 2003-03-31 13:47:23

I'm running XX on Debian Squeeze, and its been running fine since last up date.
The crash occurs when I open up the preview window.

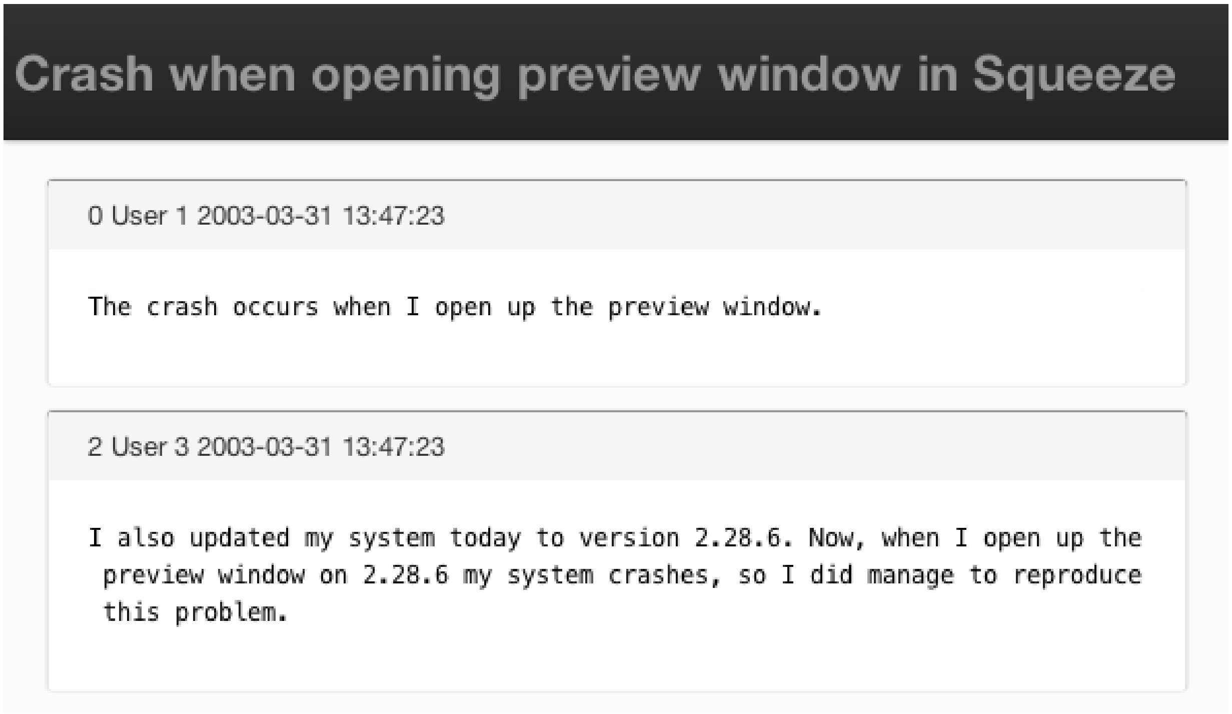
1 User 2 2003-03-31 13:47:23

I could not reproduce this, I'm running on Debian Wheezy, and I do not fa ce this crash when opening the preview window.

2 User 3 2003-03-31 13:47:23

Hi, thanks for submitting this bug.
I also updated my system today to version 2.28.6. Now, when I open up the preview window on 2.28.6 my system crashes, so I did manage to reproduce this problem.

Figure A.1: Interlaced view for bug report summary.



Crash when opening preview window in Squeeze

0 User 1 2003-03-31 13:47:23

The crash occurs when I open up the preview window.

2 User 3 2003-03-31 13:47:23

I also updated my system today to version 2.28.6. Now, when I open up the preview window on 2.28.6 my system crashes, so I did manage to reproduce this problem.

Figure A.2: Condensed view for bug report summary.

Appendix B

BugBot Questionnaire

User	How do you perceive the way BugBot organizes information improves the archival value of bugs, in comparison to traditional bug tracking systems? [worse(1) - better(7)]
User 1	7
User 2	5
User 3	7
User 4	5
User 5	5
User 6	7
User 7	5

User	**How to you perceive the way BugBot organizes information can help users and developers to reason about bugs, in comparison to traditional bug tracking systems? [worse(1) - better(7)]**
User 1	7
User 2	6
User 3	7
User 4	6
User 5	6
User 6	7
User 7	5

User	Please elaborate on how you perceive the overall benefits or disadvantages of BugBot, in comparison to traditional bug tracking systems.
User 1	The only other bug tracker I was using before was GitHub's built in bug tracker. After we switched to BugBot, we were much more satisfied. Given the structured entries we could create and certain philosophy we could follow, it allowed us to improve the quality of the bug/new feature reports. Now, however, that we got used to consciously distinguishing between diagnostics and solutions, we could, with some discipline by adding certain tags, etc. replicate the same way in a traditional bug tracker. It wouldn't be that nice and convenient, however.
User 2	I find it a bit more useful to organize the report into sections, but not sure of how practical it is to have its cost (extra time and effort)
User 3	There are several things that I like about bugbot. 1. Separation of solutions. When looking at a bugbot report, you can immediately distinguish proposed solutions from all the other information. While different types of information count, solutions are what matters for the implementer. 2. The tree structure. It is very helpful in building a mental model of the report and organizing information. I know how two user inputs (e.g., solution, comment) related to each other. 3. Voting is cool. If there are multiple and distinct solutions, we can vote on the one that is most adequate. I believe that all the above matters for longer bug reports. If bug reports are short (several user inputs), then bugbot and traditional bug tracking systems do equally well.
User 4	

Continued on next page ...

User 5	To an untrained user, BugBot is both harder to use and read properly. A conventional bug report is almost exactly like a forum thread. You read it top-down, just like any other written work, and you post your comments at the end. Seeing the conversation evolve chronologically can shed some insight into the decision-making process. However, if there are side-discussions, it can cause confusion. With BugBot, reading a bug report takes some adjusting to, since the posts are grouped by category instead of arranged chronologically. Additionally, it is difficult for a new user to know what to post, since they have to choose from a diagnostic, solution, or a comment on a previous post. Having an easily accessible web page which outlines the proper use of each feature would make it much easier for a new user. Some features, especially the ability to edit posts, do need to be explained so that they can be used properly.
User 6	My favourite feature is the nesting. It makes reading bug reports simpler since all the discussion and replies are grouped together. Not crazy about the diagnostic/solution. Many times, I feel like the distinction between them are not very clear.
User 7	The main benefits of BugBot is the ability to quickly skim through a Bug to see the diagnostic and evaluate whether such a bug is relevant or not to the question at issue. The main disadvantage is that it forces to make everything a diagnostic or solutions, whereas some input might be discussion about whether the bug should be considered a bug or not, the priority of the bug, the related bugs, and discussion about solutions or ways to avoid the bug.

User	**There is a cost to creating bug reports that are well structured, organized, and easy to consult. Users are required to think more about what they are contributing, possibly editing and revising previous contributions. How much do you consider this cost is acceptable for the benefits of having bug reports that are easy to consult? [not at all(1) - significantly(5)]**
User 1	3
User 2	2
User 3	5
User 4	3
User 5	5
User 6	4
User 7	4

User	**Please elaborate on why you consider bug report archival value to be important or not.**
User 1	They describe the ”as is” state of the system and we link to bugs from the commit messages that fix/implement them. Having the record of which issues were fixed and how gives us better confidence in the code base. Also, it is the only form of system documentation we currently have. However, the bug report often are out of synch with the actual system implementation. Our users do not have enough self discipline and motivation to write the solutions that actually were implemented. Sometimes, the solutions are what was proposed, something slightly different was implemented, but the solutions were not updated. So the value for bug resolution/discussion/collaboration actually was much higher than for archival purposes.
User 2	1. I think that people wouldn't spend that much time organizing and reading when reporting a bug. They just post it and whoever has an answer would reply. It would certainly be much more helpful to have organized bug reports, but not sure of how practical it is to consider the cost.
User 3	I think it's important because good and well-organized bug reports allow to fix bugs and add new features more quickly. It is especially important if there are multiple solutions and we need to discuss each of them. When the discussion is threaded, it is easier to distinguish more important threads and to disregard those that are either incorrect or insignificant. Bug reports (if they are structured) help to keep track of required features and improvements.

Continued on next page ...

User 4	I think that the best improvement BugBot introduces is the fact that it distinguishes important points (Bug itself, diagnostics, solutions) from comments on those points. Myself, as a user, would read the bug and go through solutions and diagnosis and I would easily get up to speed what is going on. The fact that it has diagnosis and solutions is not bringing too much of a difference to me. Probably I would go straight to the solution when looking for a solution to the bug, but otherwise there is not a huge difference. This is probably caused by the fact that I have never created a diagnosis and have no knowledge of what that exactly means and how should it help (it should I guess explain how to reproduce the bug or what are the symptoms but I have never used it for such purpose). With more experience I would probably be able to give better feedback on that part.
User 5	There are many reasons for bug reports to have archival value. If a user is looking for a solution, having a well-organized bug report makes that easy. As a developer, if I come across a problem, searching similar bug reports can provide a solution. Having well-organized bug reports makes this easier.
User 6	For Clafer, the bug reports have low view counts. Sometimes feel like lots of effort to write good reports, and only a few (1-4) people actually read them. I've rarely needed to look up old bug reports to consult them.
User 7	It is important to help in getting an overview of maturity of the tool and to help resolve new bugs that could be related to previous ones. However to improve archival value it is important to categorize the bugs by type based on the domain of the software (e.g in clafer it could be semantics, parameters, parsing, output generation, etc) and resolution (e.g won't fix, fixed, accepted as bug but not fixed, etc). I don't think bugbot provides any of this important points, but despite that the solution/diagnostic that speeds up the process of skimming through a bug report to see if it is relevant to me.

User	**How important do you consider ease of reasoning to be for resolving bug reports? [not at all(1) - signficantly(5)]**
User 1	5
User 2	4
User 3	4
User 4	4
User 5	5
User 6	5
User 7	2

User	Please elaborate on why you consider ease of reasoning about bug reports important or not.
User 1	the partition into different categories is crucial - one can get a very quick overview of what's going on. The traditional list of comments requires one to spend much more time.
User 2	Ease of reasoning about bug reports is extremely helpful for the person reading the solution so that he would get a better understanding about the bug, but it requires a lot of work from the developer posting the solution.
User 3	It is very important to clarify things that should be implemented. It is equally important to see which solutions are correct and which are out of whack
User 4	It is important for the reasons already mentioned - if one wants to see if it is a duplicate, or relevant at all. Also, it is important for users if they want to speed up understanding if the particular bug is the same bug they are experiencing.
User 5	Being able to quickly reason about a bug report is essential to resolving it in a timely manner. Being able to identify the causes of the problem, as well as possible solutions, means that you can implement the best solution available.
User 6	Easier to reason means easier to solve bug reports. Shortens the time from open to close.
User 7	It is important to decide whether a bug is actually a bug, whether it will be fixed, and what solution to choose. It is important to have this choices said and justified, as it will then help in understanding the bug report. However I think just having structured choices (e.g combobox) isn't enough as we need justification and rationale for each one of this choices and this means having clear writing.

User	**How do you perceive each of BugBot's following features improves or worsens reasoning, in comparison to traditional bug tracking systems? [Categorizing posts as diagnostics and solutions]**
User 1	Improves
User 2	Improves
User 3	Improves
User 4	Improves
User 5	Improves
User 6	No difference
User 7	Slightly improves

User	**How do you perceive each of BugBot's following features improves or worsens reasoning, in comparison to traditional bug tracking systems? [Comments nested under each diagnostic or solution]**
User 1	Improves
User 2	No difference
User 3	Improves
User 4	Improves
User 5	No difference
User 6	Improves
User 7	Slightly improves

User	**How do you perceive each of BugBot's following features improves or worsens reasoning, in comparison to traditional bug tracking systems? [Listing all diagnostics first then solutions]**
User 1	No difference
User 2	Slightly worsens
User 3	Slightly improves
User 4	Slightly improves
User 5	Slightly improves
User 6	No difference
User 7	No difference

User	**How do you perceive each of BugBot's following features improves or worsens reasoning, in comparison to traditional bug tracking systems? [Allowing diagnostics, solutions, and comments to be edited]**
User 1	Improves
User 2	No difference
User 3	Improves
User 4	No difference
User 5	Slightly improves
User 6	No difference
User 7	Slightly improves

User	**How do you perceive each of BugBot's following features affects archival value, in comparison to traditional bug tracking systems? [Categorizing posts as diagnostics and solutions]**
User 1	Better
User 2	Slightly better
User 3	Better
User 4	Slightly better
User 5	Slightly better
User 6	No difference
User 7	Slightly better

User	**How do you perceive each of BugBot's following features affects archival value, in comparison to traditional bug tracking systems? [Comments nested under each diagnostic or solution]**
User 1	Better
User 2	Better
User 3	Better
User 4	Better
User 5	No difference
User 6	Better
User 7	Better

User	**How do you perceive each of BugBot's following features affects archival value, in comparison to traditional bug tracking systems? [Listing all diagnostics first then solutions]**
User 1	Slightly better
User 2	Slightly worse
User 3	No difference
User 4	Slightly better
User 5	Slightly better
User 6	No difference
User 7	No difference

User	**How do you perceive each of BugBot's following features affects archival value, in comparison to traditional bug tracking systems? [Allowing diagnostics, solutions, and comments to be edited]**
User 1	Better
User 2	Slightly better
User 3	Better
User 4	No difference
User 5	Better
User 6	Slightly better
User 7	Slightly better

User	**Overall, how to you compare Bugbot to traditional bug tracking systems? [worse(1) - better(5)]**
User 1	7
User 2	5
User 3	7
User 4	6
User 5	5
User 6	7
User 7	5

User	**There is a cost to creating bug reports that are well structured, organized, and easy to reason about. Users are required to think more about what they are contributing, possibly editing and revising previous contributions. How much do you consider this cost is acceptable for the benefits of having bug reports that are easy to reason about? [worse(1) - better(7)]**
User 1	
User 2	3
User 3	5
User 4	4
User 5	4
User 6	5
User 7	3