What Makes a Good Bug Report?

Thomas Zimmermann, *Member*, *IEEE*, Rahul Premraj, Nicolas Bettenburg, *Member*, *IEEE*, Sascha Just, *Member*, *IEEE*, Adrian Schröter, *Member*, *IEEE*, and Cathrin Weiss

Abstract—In software development, bug reports provide crucial information to developers. However, these reports widely differ in their quality. We conducted a survey among developers and users of APACHE, ECLIPSE, and MOZILLA to find out what makes a good bug report. The analysis of the 466 responses revealed an information mismatch between what developers need and what users supply. Most developers consider steps to reproduce, stack traces, and test cases as helpful, which are, at the same time, most difficult to provide for users. Such insight is helpful for designing new bug tracking tools that guide users at collecting and providing more helpful information. Our CUEZILLA prototype is such a tool and measures the quality of new bug reports; it also recommends which elements should be added to improve the quality. We trained CUEZILLA on a sample of 289 bug reports, rated by developers as part of the survey. The participants of our survey also provided 175 comments on hurdles in reporting and resolving bugs. Based on these comments, we discuss several recommendations for better bug tracking systems, which should focus on engaging bug reporters, better tool support, and improved handling of bug duplicates.

Index Terms—Testing and debugging, distribution, maintenance, and enhancement, human factors, management, measurement.

1 Introduction

Bug reports are vital for any software development. They allow users to inform developers of the problems encountered while using a software. Bug reports typically contain a detailed description of a failure and they occasionally hint at the location of the fault in the code (in the form of patches or stack traces). However, bug reports vary in their quality of content; they often provide inadequate or incorrect information. Thus, developers sometimes have to face bugs with descriptions such as "Sem Web" (APACHE bug COCOON-1254), "wqqwqw" (ECLIPSE bug #145133), or just "GUI" with comment "The page is too clumsy" (MOZILLA bug #109242). It is no surprise that developers are slowed down by poorly written bug reports because identifying the problem from such reports takes more time.

In this paper, we investigate the **quality of bug reports** from the perspective of developers. We expected several factors to impact the quality of bug reports, such as the length of descriptions, formatting, and presence of stack traces and attachments (such as screenshots). To find out which matter most, we asked 872 developers from the APACHE, ECLIPSE, and MOZILLA projects to:

1. *Complete a survey* on important information in bug reports and the problems they faced with them. We received a total of 156 responses to our survey (Section 2).
2. *Rate the quality of bug reports* from very poor to very good on a five-point Likert scale [42]. We received a total of 1,244 votes for 289 randomly selected bug reports (Section 6).

In addition, we asked 1,354 reporters[1] from the same projects to complete a similar survey, out of which 310 responded. The results of both surveys suggest that there is a **mismatch between what developers consider most helpful and what users provide** and revealed several hurdles in bug reporting (Sections 3, 4, and 5).

To enable swift fixing of bugs, the mismatch between developers and reporters should be bridged, for example, with tool support for reporters to furnish information that developers want. We developed a prototype tool called CUEZILLA, which gauges the quality of bug reports and suggests to reporters what should be added to make a bug report better.

1. *CUEZILLA measures the quality of bug reports.* We trained and evaluated CUEZILLA on the 289 bug reports rated by the developers (Section 7).
2. *CUEZILLA provides incentives to reporters.* We automatically mined the bug databases for encouraging facts such as "Bug reports with stack traces are fixed sooner" (Section 8).

1. Throughout this paper, *reporter* refers to the people who create bug reports and are not assigned to any. Mostly, reporters are end-users, but in many cases, they are also experienced developers.

- *T. Zimmermann is with Microsoft Research, One Microsoft Way, Redmond, WA 98052. E-mail: tzimmer@microsoft.com.*
- *R. Premraj is with the Computer Science Department, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. E-mail: rpremraj@cs.vu.nl.*
- *N. Bettenburg is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, 156 Barrie St., Kingston, ON K7L 3N6, Canada. E-mail: nicbet@cs.queensu.ca.*
- *S. Just is with Saarland University—Computer Science, Campus E1 1, 66123 Saarbruecken, Germany. E-mail: sascha.just@ieee.org.*
- *A. Schröter is with the University of Victoria, Engineering/Computer Science Building, Room 504, PO Box 3055 STN CSC, Victoria, BC V8W3P6, Canada. E-mail: schadr@uvic.ca.*
- *C. Weiss is with the Department of Informatics, University of Zurich, Binzmühlestrasse 14, CH-8050, Zürich, Switzerland. E-mail: weiss@ifi.uzh.ch.*

Manuscript received 21 June 2009; revised 9 Dec. 2009; accepted 18 Apr. 2010; published online 10 June 2010.
Recommended for acceptance by G. Murphy and W. Schäfer.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2009-06-0162.
Digital Object Identifier no. 10.1109/TSE.2010.63.

TABLE 1
Number of Invitations Sent to and Responses by Developers and Reporters of the APACHE, ECLIPSE, and MOZILLA Projects

	Developers					Reporters				
Project	Contacted	Bounces	Reached	Responses (Rate)	Comments	Contacted	Bounces	Reached	Responses (Rate)	Comments
APACHE	194	5	189	34 (18.0%)	12	165	17	148	37 (25.0%)	10
ECLIPSE	365	29	336	50 (14.9%)	15	378	8	370	50 (13.5%)	20
MOZILLA	313	29	284	72 (25.4%)	21	811	130	681	223 (32.7%)	97
Total	872	63	809	156 (19.3%)	48	1354	155	1199	310 (25.9%)	127

To summarize, this paper makes the following contributions:

1. a survey on how bug reports are used among 2,226 developers and reporters, out of which 466 responded;
2. empirical evidence for a mismatch between what developers expect and what reporters provide;
3. recommendations on how to improve bug tracking systems;
4. empirical evidence that bug duplicates contain extra information that can be helpful for fixing bugs;
5. the CUEZILLA prototype that measures the quality of bug reports and suggests how reporters could enhance their reports so that their problems get fixed sooner; and
6. our data set and R scripts to allow replication and extension of this research (see Appendix A).

We conclude this paper with threats to validity (Section 9), related work (Section 10), and future research directions (Section 11).

2 Survey Design

To collect facts on how developers use the information in bug reports and what problems they face, we conducted an online survey among the developers of APACHE, ECLIPSE, and MOZILLA. In addition, we contacted bug reporters to find out what information they provide and which is most difficult to provide.

For any survey, the response rate is crucial to draw generalizations from a population. Keeping a questionnaire short is one key to a high response rate. In our case, we aimed for a total completion time of 5 minutes, which we also advertised in the invitation e-mail ("we would much appreciate 5 minutes of your time").

2.1 Selection of Participants

Each examined projects' bug database contains several hundred developers that are assigned to bug reports. Of these, we selected only *experienced developers* for our survey since they have a better knowledge of fixing bugs. We defined experienced developers as those assigned to at least 50 bug reports in their respective projects. Similarly, we contacted only *experienced reporters*, which we defined as having submitted at least 25 bug reports (= a user) while at the same time being assigned to zero bugs (= not a developer) in the respective projects. Several responders in the reporter group pointed out that they had some development experience, though mostly in other software projects.

Table 1 presents for each project the number of developers and reporters contacted via personalized e-mail, the number of bounces, and the number of responses and comments received. The response rate was the highest for MOZILLA reporters at 32.7 percent. Our overall response rate of 23.2 percent is comparable to other Internet surveys in software engineering, which range from 14 to 20 percent [52].

2.2 The Questionnaire

Keeping the 5 minute rule in mind, we asked developers the following questions, which we grouped as follows (see Fig. 1):

Contents of bug reports. *Which items have developers previously used when fixing bugs? Which three items helped the most?*

Such insight aids in guiding reporters to provide or even focus on information in bug reports that is most important to developers. We provided 16 items selected on the basis of Eli Goldberg's bug writing guidelines [27] or being standard fields in the BUGZILLA database.

Developers were free to check as many items as they wished for the first question (D1), but at most three for the second question (D2), thus indicating the importance of items.

Problems with bug reports. *Which problems have developers encountered when fixing bugs? Which three problems caused most delay in fixing bugs?*

Our motivation for this question was to find prominent obstacles that can be tackled in the future by more cautious, and perhaps even automated, reporting of bugs.

Typical problems are when reporters accidentally provide incorrect information, for example, an incorrect operating system.[2] Other problems in bug reports include poor use of language (ambiguity), bug duplicates, and incomplete information. Spam recently has become a problem, especially for the TRAC issue tracking system. We decided not to include the problem of incorrect assignments to developers because bug reporters have little influence on the triaging of bugs.

In total, we provided 21 problems that developers could select. Again, they were free to check as many items for the first question (D3) as they wished, but at most three for the second question (D4).

For the reporters of bugs, we asked the following questions (again see Fig. 1):

Contents of bug reports. *Which items have reporters previously provided? Which three items were most difficult to provide?*

2. Did you know? In ECLIPSE, 205 bug reports were submitted for "Windows" but later reassigned to "Linux."

Contents of bug reports.

D1: Which of the following items have you previously used when fixing bugs?
D2: Which three items helped you the most?

R1: Which of the following items have you previously provided when reporting bugs?
R2: Which three items were the most difficult to provide?

R3: In your opinion, which three items are most relevant for developers when fixing bugs?

❑ product	❑ hardware	❑ observed behavior	❑ screenshots
❑ component	❑ operating system	❑ expected behavior	❑ code examples
❑ version	❑ summary	❑ steps to reproduce	❑ error reports
❑ severity	❑ build information	❑ stack traces	❑ test cases

Problems with bug reports.

D3: Which of the following problems have you encountered when fixing bugs?
D4: Which three problems caused you most delay in fixing bugs?

You were given wrong:	There were errors in:	The reporter used:	Others:
❑ product name	❑ code examples	❑ bad grammar	❑ duplicates
❑ component name	❑ steps to reproduce	❑ unstructured text	❑ spam
❑ version number	❑ test cases	❑ prose text	❑ incomplete information
❑ hardware	❑ stack traces	❑ too long text	❑ viruses/worms
❑ operating system		❑ non-technical language	
❑ observed behavior		❑ no spell check	
❑ expected behavior			

Comments.

D5/R4: Please feel free to share any interesting thoughts or experiences.

Fig. 1. The questionnaire presented to APACHE, ECLIPSE, and MOZILLA developers (Dx) and reporters (Rx).

We listed the same 16 items to reporters as we had listed to developers before. This allowed us to check whether the information provided by reporters is in line with what developers frequently use or consider to be important (by comparing the results for R1 with D1 and D2). The second question helped us to identify items, which are difficult to collect and for which better tools might support reporters in this task.

Reporters were free to check as many items for the first question (R1) as they wished, but at most three for the second question (R2).

Contents considered to be relevant. *Which three items do reporters consider to be most relevant for developers?*

Again, we listed the same items to see how much reporters agree with developers (comparing R3 with D2).

For this question (R3), reporters were free to check at most three items, but could choose any item, regardless of whether they selected it for question R1.

Additionally, we asked both developers and reporters about their thoughts and experiences with respect to bug reports (D5/R4).

2.3 Parallelism between Questions

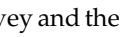
In the first two parts of the developer survey and the first part of the reporter survey, questions share the same items but have different limitations (select as many as you wish versus the three most important). We will briefly explain the advantages of this parallelism using D1 and D2 as examples.

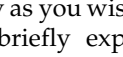
1. *Consistency check.* When fixing bugs, all items that helped a developer the most (selected in D2) *must* have been used previously (selected in D1). If this is not the case, i.e., an item is selected in D2 but not in D1, the entire response is regarded as inconsistent and discarded.
2. *Importance of items.* We can additionally infer the importance of individual items. For instance, for item i, let $N_{D1}(i)$ be the number of responses in which it was selected in question D1. Similarly, $N_{D1,D2}(i)$ is the number of responses in which the item was selected in both questions D1 and D2.[3] Then, the importance of item i corresponds to the conditional likelihood that item i is selected in D2 when selected in D1:

$$Importance(i) = \frac{N_{D1,D2}(i)}{N_{D1}(i)}.$$

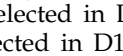
Other parallel questions were D3 and D4, as well as R1 and R2.

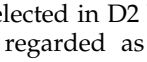
3 Survey Results

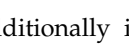
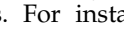
In this section, we discuss our findings from the survey responses. For developers, we received a total of 156 responses, out of which 26 (or 16.7 percent) failed the consistency check and were removed from our analysis. For reporters, we received 310 and had to remove 95 inconsistent responses (30.6 percent). The results of our survey are summarized in Table 2 (for developers) and Table 3 (for reporters). In the tables, responses for each item are annotated as bars (), which can be broken down into their constituents and interpreted as below (again, explained with D1 and D2 as examples):


All consistent responses for the project

Number of times that *item* was selected in D1


Number of times that *item* was selected in D1 and D2


Number of times that *item* was selected in D1 but not D2

The colored part () denotes the count of responses for an item in question D1 and the black part () of the bar

3. When *all* responses are consistent, $N_{D1,D2}(i) = N_{D2}(i)$ holds.

TABLE 2
Results from the Survey among Developers

Contents of bug reports (D1/D2). *In parentheses: importance of item.*

product (5%)	hardware (0%)	observed behavior (33%)	screenshots (26%)
component (3%)	operating system (4%)	expected behavior (22%)	code examples (14%)
version (12%)	summary (13%)	steps to reproduce (83%)	error reports (12%)
severity (0%)	build information (8%)	stack traces (57%)	test cases (51%)

Problems with bug reports (D3/D4). *In parentheses: severeness of problem.*

You were given wrong:	There were errors in:	The reporter used:	Others:
product name (7%)	code examples (15%)	bad grammar (16%)	duplicates (10%)
component name (15%)	steps to reproduce (79%)	unstructured text (34%)	spam (0%)
version number (22%)	test cases (38%)	prose text (18%)	incomplete information (74%)
hardware (8%)	stack traces (25%)	too long text (26%)	viruses/worms (0%)
operating system (20%)		non-technical language (19%)	
observed behavior (48%)		no spell check (0%)	
expected behavior (27%)			

130 consistent responses by APACHE, ECLIPSE, and MOZILLA developers.

TABLE 3
Results from the Survey among Reporters

Contents of bug reports (R1/R2). *In parentheses: difficulty of item.*

product (0%)	hardware (1%)	observed behavior (2%)	screenshots (8%)
component (22%)	operating system (1%)	expected behavior (3%)	code examples (43%)
version (1%)	summary (4%)	steps to reproduce (51%)	error reports (2%)
severity (5%)	build information (3%)	stack traces (24%)	test cases (75%)

Contents considered to be relevant for developers (R3). *In parentheses: frequency of item in R3.*

product (7%)	hardware (0%)	observed behavior (33%)	screenshots (5%)
component (4%)	operating system (4%)	expected behavior (22%)	code examples (9%)
version (12%)	summary (6%)	steps to reproduce (78%)	error reports (9%)
severity (2%)	build information (8%)	stack traces (33%)	test cases (43%)

215 consistent responses by APACHE, ECLIPSE, and MOZILLA reporters.

denotes the count of responses for the item in both question D1 and D2. The larger the black bar is in proportion to the gray bar, the higher the corresponding item's importance is in the developers' perspective. The importance of every item is listed in parentheses.

Tables 2 and 3 present the results for all three projects combined. For project-specific tables, we refer to Appendix B.

3.1 Contents of Bug Reports (Developers)

Table 2 shows that the **most widely used items** across projects are *steps to reproduce*, *observed* and *expected behavior*, *stack traces*, and *test cases*. Information rarely used by developers is *hardware* and *severity*. ECLIPSE and MOZILLA developers favorably used *screenshots*, while APACHE and ECLIPSE developers more often used *code examples* and *stack traces*.

For the **importance of items**, *steps to reproduce* stand out clearly. Next in line are *stack traces* and *test cases*, both of which help to narrow down the search space for defects. *Observed behavior*, albeit weakly, mimics steps to reproduce the bug, which is why it may be rated important. *Screenshots* were rated as high, but often are helpful only for a subset of bugs, e.g., GUI errors.

Smaller surprises in the results are the relatively low importance of items such as *expected behavior*, *code examples*, *summary*, and mandatory fields such as *version*, *operating system*, *product*, and *hardware*. As pointed out by a MOZILLA developer, not all projects need the information that is provided by mandatory fields:

> *That's why product and usually even component information is irrelevant to me and that hardware and to some degree [OS] fields are rarely needed as most our bugs are usually found in all platforms.*

In any case, we advise caution when interpreting these results: Items with low importance in our survey are not totally irrelevant because they still might be needed to understand, reproduce, or triage bugs.

3.2 Contents of Bug Reports (Reporters)

The **items provided by most reporters** are listed in the first part of Table 3. As expected *observed* and *expected behavior* and *steps to reproduce* rank highest. Only a few users added *stack traces*, *code examples*, and *test cases* to their bug reports. An explanation might be the **difficulty in providing these items**, which is reported in parentheses. All three items rank among the more difficult items, with *test cases* being the most difficult item. Surprisingly, *steps to reproduce* and *component* are considered to be difficult as well. For the latter, reporters revealed in their comments that often it is impossible for them to locate the component in which a bug occurs.

Among the items **considered to be most helpful to developers**, reporters ranked *steps to reproduce* and *test cases* highest. Comparing the results for test cases among all three questions reveals that most reporters consider them to be

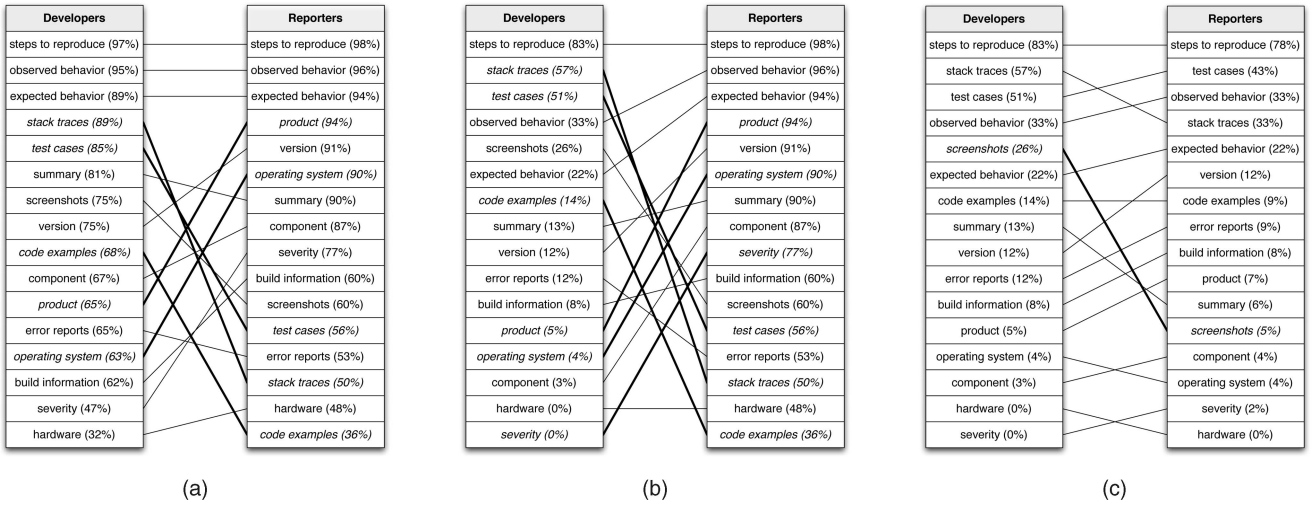


Fig. 2. Mismatch between developers and reporters. (a) Information used by developers versus provided by reporters. (b) Most helpful for developers versus provided by reporters. (c) Most helpful for developers versus reporters expected to be helpful.

helpful, but only a few provide them because they are most difficult to provide. This suggests that capture/replay tools which record test cases [34], [47], [65] should be integrated into bug tracking systems. A similar but weaker observation can be made for *stack traces*, which are often hidden in log files and difficult to find. On the other hand, both developers and reporters consider *components* only as marginally important; however, as discussed above, they are rather difficult to provide.

3.3 Evidence for Information Mismatch

We compared the results from the developer and reporter surveys to find out whether they agree on what is important in bug reports.

First, we compared which information developers use to resolve bugs (question *D1*) and which information reporters provide (*R1*). In Fig. 2a, items in the left column are sorted decreasingly by the percentage of developers who have used them, while items in the right column are sorted decreasingly by the percentage of reporters who have provided them. Lines connect same items across columns and indicate the agreement (or disagreement) between developers and reporters on that particular item. Fig. 2a shows that the results match only for the top three items and the last one. In between there are many disagreements, the most notable ones for *stack traces*, *test cases*, *code examples*, *product*, and *operating system*. Overall, the Spearman correlation between what developers use and what reporters provide was 0.321, far from being ideal.[4]

Next, we checked whether reporters provide the information that is most important for developers. In Fig. 2b, the left column corresponds to the importance of an item for developers (measured by questions *D2* and *D1*), and the right column to the percentage of reporters who provided an item (*R1*). Developers and reporters still agree on the first and last item; however, overall the disagreement increased. The Spearman correlation of -0.035 between what developers consider as important and what reporters provide shows a huge gap. In particular, it indicates that reporters do not focus on the information important for developers.

Interestingly, Fig. 2c shows that most reporters know which information developers need. In other words, ignorance of reporters is *not* a reason for the aforementioned information mismatch. As before, the left column corresponds to the importance of items for developers; the right column now shows what reporters expect to be most relevant (question *R3*). Overall, there is strong agreement; the only notable disagreement is for *screenshots*. This is confirmed by the Spearman correlation of 0.839, indicating a very strong relation between what developers and reporters consider as important. A breakdown of project-specific mismatch is presented in Appendix B.3.

As a consequence, to improve bug reporting systems, one could tell users *while* they are reporting a bug what information is important (e.g., screenshots). At the same time, one should provide better tools to collect important information, because often this information is difficult to obtain for users (see Section 3.2).

3.4 Problems with Bug Reports

Among the **problems experienced by developers**, *incomplete information* was, by far, most commonly encountered. Other common problems include errors in *steps to reproduce* and *test cases*, *bug duplicates*, and incorrect *version numbers*, *observed* and *expected behavior*. Another issue that developers often seemed challenged by is the fluency in language of the reporter. Most of these problems are likely to lead developers astray when fixing bugs.

The **most severe problems** were errors in *steps to reproduce* and *incomplete information*. In fact, in question D5, many developers commented on being plagued by bug reports with incomplete information:

> *The biggest causes of delay are not wrong information, but absent information.*

Other major problems included errors in *test cases* and *observed behavior*. A very interesting observation is that developers do not suffer too much from bug duplicates, although earlier research considered this to be a serious

4. Spearman correlation computes agreement between two rankings: Two rankings can be opposite (value -1), unrelated (value 0), or perfectly matched (value 1). We refer to textbooks for details [61].

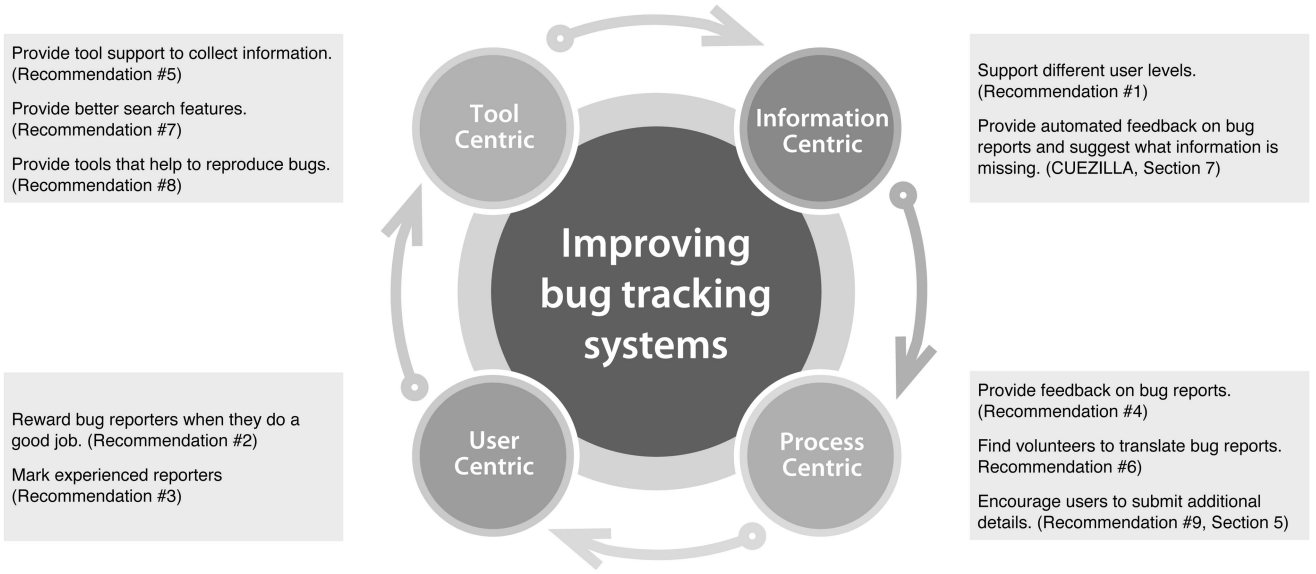


Fig. 3. Four areas to improve bug tracking systems.

problem [22], [54], [60]. Possibly developers can easily recognize *duplicates* and sometimes even benefit from a different bug description (see Section 5 for a discussion of the value of bug duplicates).

The low occurrence of *spam* is not surprising: In BUGZILLA and JIRA, reporters have to register before they can submit bug reports; this registration successfully prevents spam. Last, errors in *stack traces* are highly unlikely because they are copy-pasted into bug reports, but when an error happens, it can be a severe problem.

4 Analysis of Comments

Recall that in addition to responses to the specific questions asked in our survey, we also received 175 comments from the survey participants (see Table 1). In the comments field, the participants were free to elaborate upon any experiences or issues encountered while using bug tracking systems that were not covered by our questions.

To understand which other hurdles developers and reporters face with today's bug tracking, we analyzed the comments with an open card sort. Card sorting is an inexpensive and user-centered sorting technique that is widely used in information architecture to create mental models and derive taxonomies from input (see Barker [9]). We first manually examined and split comments into more atomic parts (statements). We then sorted the resulting 237 statements to identify common themes and problems that were established while doing the card sort. Based on these insights [35], we make recommendations to improve current bug tracking systems, which fall into the following four areas.

1. *Tool-centric improvements* are made to the features provided by bug tracking systems. They can help to reduce the burden of information collection and provision. An example of tool-centric enhancements is capture/replay tools, which can provide steps to reproduce automatically [7].
2. *Information-centric improvements* focus directly on the information being provided. As an example, we discuss in Section 7 the CUEZILLA tool, which provides real-time feedback on the quality of a bug report and what information can be added to increase value.
3. *Process-centric improvements* to bug tracking systems focus on administration of activities related to bug fixing. For example, *bug triaging*, i.e., deciding which bugs get fixed and determining which developer should resolve the bug, can be automated [4], [14].
4. *User-centric improvements* include both reporters and developers. Reporters can be educated on what information to provide and how to collect it. Developers too can benefit from similar training on what information to expect in bug reports and how this information can be used to resolve bugs.

The four areas are summarized in Fig. 3; we subsequently discuss specific comments and recommendations according to the numbers above. This section focuses on recommendations on how to engage users and build better tool support. In Section 5, we propose an improved handling of bug duplicates. In Section 7, we present a prototype tool CUEZILLA that helps reporters to submit better quality bug reports.

4.1 Engage Reporters

User levels. Often users of a software have different levels of knowledge, as pointed out by one developer:

> *In OSS, there is a big gap with the knowledge level of bug reporters. Some will include exact locations in the code to fix, while others just report a wierd behavior that is difficult to reproduce. In Eclipse, experienced users know that the Error log exists, so they can provide stack traces and errors* [...] **(comment 98)**

Guidance through bug reporting is desirable, especially for less experienced users, e.g., by helping to collect certain information or by splitting up bug reporting across multiple pages. However, experienced bug reporters prefer to have one page where they can provide everything.

In addition to different levels of knowledge, not all users know what information is important for developers:

> *It's easily forgotten that many peoples' brains just aren't wired the same as ours, and they just don't understand what we're asking for in bug reports, and why!* **(comment 97)**

Recommendation #1. *Support different levels of users (novice, expert) and provide different user interfaces for each level. Give cues to inexperienced users what information they should provide and how they can collect it.*

Reward reporters. Good quality reports are worthy of rewards: The "Mozilla Security Bug Bounty Program" [46] awards US $500 and a Mozilla T-shirt for every critical security bug reported. In 2006, the Eclipse Foundation had a "Callisto Simultaneous Release Bug Finding Contest" [25]. In this contest, any developer who saw a great bug report marked that bug with the "greatbug" keyword. Both the reporter and triager of this bug then received an "I Helped Callisto" shirt and participated in a random drawing of prizes such as iPods and mountain bikes.

Recommendation #2. *Do not just fix bugs, also reward reporters when they do a good job.*

Reporter reputation. Several developers pointed out that reporters who are well known, either personally or through well-written past bug reports, will get more attention.

> *Another import thing is that devs know you (because you have filed bug reports before, you discussed with them on IRC, conferences, ...) this is the human component of the system which is often forgotten.* **(comment 102)**

> *Well known reporters usually get more consideration than unknown reporters, assuming the reporter has a pretty good history in bug reporting. So even if a "well-known" reporter reports a bug which is pretty vague, he will get more attention than another reporter, and the time spent trying to reproduce the problem will also be larger.* **(comment 55)**

An improvement to bug tracking systems would be to introduce *reputation* into user profiles. This would help developers to quickly identify the experience of a reporter, even when they do not know him personally. Hooimeijer and Weimer measured the reputation of reporters as the success rate, i.e., the percentage of submitted bug reports that were fixed [30]. Ideally, reputation would have two components: a project-independent one that tells the experience with bug reporting in general and a project-specific one that tells the experience for a given project.

Recommendation #3. *Integrate reputation into user profiles to mark experienced reporters.*

Frustration of users. Seven reporters complained about not getting feedback on bug reports that they have submitted.

> *"From a bug submitter perspective, it would be nice to see that submitted bugs are being looked at (at least being read by someone). It can be frustrating to submit a bug and not know what the status is or when somebody last looked at it."* **[comment 42]**

A related comment received from a developer is:

> *Also, I don't think the general user comprehends the sheer volume of bugs that we must address, and thus, there is some misunderstanding when we can't address all of them.* **(comment 30)**

The reason why developers do not always provide feedback is that they are simply swamped with bug reports. An improved bug tracking system could take over to send some feedback, e.g., when developers first looked at a bug report, when they start working on it. Other feedback could be just counting the number of page views for a bug report.

Recommendation #4. *Always provide feedback on bug reports to keep users motivated.*

4.2 Better Tool Support

Collect information. Several reporters pointed out the need for tools that help them to collect information that they need to file bug reports. Ideally, such tools would be integrated in the software itself or in its bug reporting system.

> *Sometimes I wish for a special UI-tracker, which tells me what I have done to get into this.* **(comment 64)**

> *[M]aybe somehow it could be made possible to report bugs more like recording a macro.* **(comment 65)**

Such support is likely to lead to better bug reports. One example is ECLIPSE bug 113206, which was awarded "Best of Bugzilla" by Cunningham [23]. In this bug report, the reporter used a flash movie to demonstrate the rather complicated steps to reproduce. He stopped the video at important points where he added annotations to draw the attention of the developers to the crucial parts.

Recommendation #5. *Provide tool support for users to collect and prepare information that developers need.*

Internationalization. Bug reports that are not written in English are often closed immediately, although the software is internationalized.

> *Another frustrating issue with bug reporting sites is insensitivity to language issues. I've seen bugs immediately closed because they weren't filed in English, without even asking or waiting for someone to translate it into English.* **(comment 56)**

Recommendation #6. *Find volunteers to translate bug reports filed in foreign languages. Ideally, the bug tracking system should provide support for this.*

Better search facility. Most reporters are aware that they should check first as to whether a bug has already been filed. However, nine reporters commented on the limited search functionality in bug tracking systems and requested features such as regular expressions and a Google-like search.

> *It's very hard to find possible duplicates, when filing bugs. The "search" tool in Bugzilla is very poor—it seems that a search for "quick brown fox" will return results for "quick OR brown OR fox,",without prioritizing "quick AND brown AND fox". Furthermore, there's no way to assign or search keywords for a bug* [...]. **(comment 71)**

> *let me at a minimum enter ebay style search strings for finding relevant bugs* [,,,] *I'd expect regular expressions.* **(comment 72)**

Traditional bug tracking systems provide mostly keyword search. However, bug reports are more than just keywords; they consist of structured information such as stack traces, patches, and source code. Recently, Ashok et al. presented the DebugAdvisor tool [8], which is an important step toward better search in bug tracking: DebugAdvisor

allows programmers to more effectively search for related and duplicated bug reports using *fat queries*, which can be kilobytes of structured and unstructured data.

Recommendation #7. *Provide a powerful, yet simple and easy-to-use feature to search bug reports.*

Complicated steps to reproduce. This problem was pointed out by several developers:

> *If the repro steps are so complex that they'll require more than an hour or so (max) just to set up would have to be quite serious before they'll get attention.*

> *This is one of the greatest reasons that I postpone investigating a bug ... if I have to install software that I don't normally run in order to see the bug.*

Recommendation #8. *Provide tools that help to reproduce bugs, e.g., set up a test workspace automatically.*

4.3 Other Interesting Comments

In addition to the above, we also received the following interesting comments from developers:

Violating netiquette. *"Another aspect is politeness and respect. If people open rude or sarcastic bugs, it doesn't help their chances of getting their issues addressed."*

Misuse of bug tracking system. *"Bugs are often used to debate the relative importance of various issues. This debate tends to spam the bugs with various use cases and discussions* [...] *making it harder to locate the technical arguments often necessary for fixing the bugs. Some long-lived high-visibility bugs are especially prone to this."*

Keen bug reporters. A developer wrote about reporters who go the extra mile to identify offending code: *"I feel that I should at least put in the amount of effort that they did; it encourages this behavior."*

Bug severity. *"For me it amounts to a consideration of 'how serious is this?' versus 'how long will it take me to find/fix it?'. Serious defects get prompt attention but less important or more obscure defects get attention based on the defect clarity."*

5 Value of Duplicate Bug Reports

A common argument against duplicate bug reports is that they strain bug tracking systems and demand more effort from quality assurance teams—effort that could instead be utilized elsewhere to improve the product. In this section, we provide empirical evidence for the contrary by demonstrating that duplicate bug reports may actually contain additional information that may be useful to resolve bugs.

When a bug report is identified as a duplicate, a common practice is to simply close and discard the information, which in the long term discourages users from submitting bug reports. They become reluctant to provide additional information once they see that a bug report has already been filed.

> *Typically bugs I have reported are already reported but by much less savvy people who make horrible reports that lack important details. It is frustrating to have spent lots of time making an exceptionally detailed bug report to only have it marked as a duplicate* [...]." **[comment 18a]**

In our survey, developers also suggested that bug duplicates are not always bad, they often add important details.

> *Duplicates are not really problems. They often add useful information. That this information was filed under a new report is not ideal though.* **(comment 19)**

> *It would be better to somehow mend the reports instead of just writing off the good report simply because it was posted after the bad report. This would probably help software engineers much better.* **(comment 18b)**

Page makes a similar argument and summarizes three reasons why "worrying about [duplicates] is bad" [48], [49].

1. Often there are negative consequences for users who enter duplicates. As a result, *they might err on the side of not entering a bug*, even though it is not filed yet.
2. *Triagers are more skilled in detecting duplicates* than users and they also know the system better. While a user will need a considerable amount of time to browse through similar bugs, triagers can often decide within minutes whether a bug report is a duplicate.
3. Bug *duplicates can provide valuable information* that helps diagnose the actual problem.

These comments about duplicate bug reports motivated us to conduct an in-depth empirical study to investigate the value of information that bug duplicates contain (the third of Page's reasons [48]).

We use the term *master report* to refer to a original report that has associated duplicate bug reports in the bug tracking system and *extended reports* to refer to the original bug report and it's duplicates combined.

5.1 How Much Information Duplicates Add

In order to detect and quantify information items from bug reports for comparison, we used a tool called infoZilla [15]. infoZilla can reliably detect and extract information such as *patches*, *screenshots*, and *stack traces* from bug reports. Using the information from infoZilla and bug reports' predefined fields, we investigate whether duplicate bug reports contribute any additional information that may help resolve bugs. Table 4 summarizes our findings for ECLIPSE and MOZILLA using the following columns:

- The first column presents all information items that we extracted using infoZilla. The items fall in four categories: *predefined fields* such as product and component, *patches*, *screenshots*, and *stack traces*. Patches and stack traces are often found in the description field or as separate attachments.
- The second column "Master" lists the average count of each information item in the original master reports, i.e., when bug duplicates are ignored. This count corresponds to a practice that is found in many projects: Once duplicates are detected, they are simply closed and all information that they provided is discarded.
- The third column "Extended" lists the average count of each information item in the *extended* bug reports. This count would correspond to a practice where duplicates are merged with master reports and all information is retained.

TABLE 4
Average Amount of Information Added by Duplicates per Master Report

ECLIPSE[a]	Average per master report		
Information item	Master	Extended	Change[b]
PREDEFINED FIELDS			
– product	1.000	1.127	+0.127
– component	1.000	1.287	+0.287
– operating system	1.000	1.631	+0.631
– reported platform	1.000	1.241	+0.241
– version	0.927	1.413	+0.486
– reporter	1.000	2.412	+1.412
– priority	1.000	1.291	+0.291
– target milestone	0.654	0.794	+0.140
PATCHES			
– total	1.828	1.942	+0.113
– unique: patched files	1.061	1.124	+0.062
SCREENSHOTS			
– total	0.139	0.285	+0.145
– unique: filename, filesize	0.138	0.281	+0.143
STACKTRACES			
– total	0.504	1.422	+0.918
– unique: exception	0.195	0.314	+0.118
– unique: exception, top frame	0.223	0.431	+0.207
– unique: exception, top 2 frames	0.229	0.458	+0.229
– unique: exception, top 3 frames	0.234	0.483	+0.248
– unique: exception, top 4 frames	0.239	0.504	+0.265
– unique: exception, top 5 frames	0.244	0.525	+0.281

[a] Dataset from the MSR Mining Challenge 2008 [37].
[b] For all information items the increase is significant at $p < .001$.

MOZILLA[a]	Average per master report		
Information item	Master	Extended	Change[b]
PREDEFINED FIELDS			
– product	1.000	1.400	+0.400
– component	1.000	1.953	+0.953
– operating system	1.000	2.102	+1.102
– reported platform	1.000	1.544	+0.544
– version	0.814	0.979	+0.165
– reporter	1.000	3.705	+2.705
– priority	0.377	0.499	+0.122
– target milestone	0.433	0.558	+0.125
PATCHES			
– total	5.038	5.184	+0.146
– unique: patched files	2.003	2.067	+0.064
SCREENSHOTS			
– total	0.200	0.391	+0.191
– unique: filename, filesize	0.197	0.385	+0.187
STACKTRACES			
– total	0.100	0.185	+0.085
– unique: exception	0.033	0.047	+0.014
– unique: exception, top frame	0.069	0.130	+0.061
– unique: exception, top 2 frames	0.072	0.136	+0.064
– unique: exception, top 3 frames	0.073	0.139	+0.066
– unique: exception, top 4 frames	0.074	0.141	+0.067
– unique: exception, top 5 frames	0.075	0.143	+0.068

[a] Dataset contains all bug reports submitted until April 21, 2008.
[b] For all information items the increase is significant at $p < .001$.

- The fourth column "Change" is the difference between "Extended" and "Master" and represents the average number of information items that bug duplicates would add per master report.

In order to quantify the amount of additional *"new"* data for master reports, we counted *unique* items whenever possible. For predefined fields, we counted the unique values; for patches, the unique files that were patched; for screenshots, the number of unique filenames and file sizes. For stack traces, we counted the number of unique exceptions and unique top n stack frames ($n = 1, \ldots, 5$). To test for the statistical significance of our results, we conducted a one-sided paired t-test [56], [61]. For all information items, the increase in information items caused by duplicates was significant at $p < 0.001$.

Coming back to Table 4 and the results for ECLIPSE, every master report contains exactly one operating system (as indicated by the 1.000 in "Master"). When merged with their duplicates, the average number of unique operating systems in extended bug reports increases to 1.631 ("Extended"). This means that duplicates could add, on average, 0.631 operating systems to existing bugs as long as duplicates are not just discarded. The numbers are similar for MOZILLA, where bug duplicates could add 1.102 operating systems on average.

Most duplicates are filed by users who are different from the ones who filed the original master report,[5] which explains the large increase in the number of unique reporters in Table 4. A reporter's reputation can go a long way in influencing the future course of a bug report (Recommendation #3, Section 4.1). This suggests that a master report may get more attention if a duplicate filed by a known reporter gets noticed.

Besides reporters, duplicates also provide substantial additional information for operating system, version, priority, and component. We also found that duplicates add, on average, 0.113-0.146 patches and 0.062-0.064 patched files per master report. This is a rather small relative increase of less than 10 percent and suggests that most patches are filed against the master report. Bug duplicates add, on average, 0.143-0.187 screenshots (relative increase of roughly 100 percent).

We compared stack traces by considering the exception and the first five stack frames. For ECLIPSE, on average, 0.918 additional stack traces were found in the duplicate bug reports (increase of about 180 percent). Within these, we found, on average, 0.118 occurrences of additional exceptions in duplicates and 0.281 stack traces that contained code locations (in the top five frames) that have not been reported before.

For MOZILLA, stack traces in bug reports are less frequent (only in every 10th bug report), but the relative increases are similar to ECLIPSE. The reason for the low amount of stack traces in MOZILLA bug reports is that stack traces for crashes are submitted to a separate repository, called TALKBACK, where they can be displayed. Reporters often provide a link to the TALKBACK ID instead of pasting the complete stack trace.

Our findings show that duplicates are likely to provide different perspectives and additional pointers to the origin of bugs, and thus can help developers to correct bugs. For example, having more stack traces reveals more active methods during a crash, which helps to narrow down the suspects. Overall, these findings suggest that duplicate bug

5. However, this is not always the case, as discussed in Section 5.2.

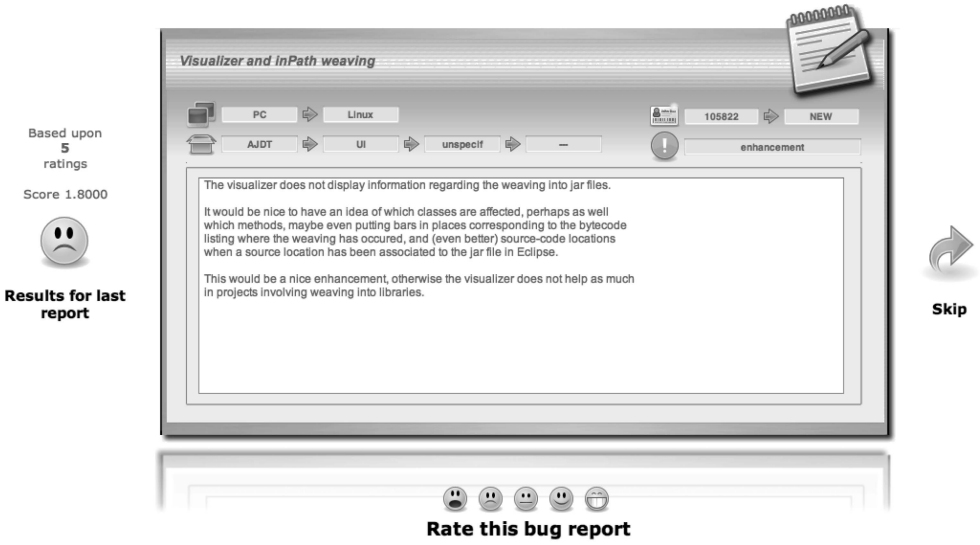


Fig. 4. Screenshot of interface for rating bug reports.

reports provide developers with information that was not present in the original report and make a case for reevaluation of the treatment and presentation of duplicates in bug tracking systems.

Recommendation #9. *Encourage users to submit additional details, ideally to an already existing bug report. Provide tool support for merging bugs.*

5.2 Reasons for Bug Duplicates

We performed a more detailed analysis on why duplicates are submitted in the first place. Here are the reasons that we could identify:

- *Lazy and inexperienced users.* Some users simply are not willing to spend time searching for duplicates. Others are not yet experienced enough with bug tracking.
- *Poor search feature.* Many survey participants recommended improvements for the search feature of BUGZILLA. One example of a bug where the search feature likely failed is bug #24448 from ECLIPSE "Ant causing Out of Memory." It was reported by 33 different users over a period of almost 900 days.
- *Multiple failures, one defect.* Sometimes, it is not obvious that two failures (error in the program execution) belong to the same defect (error in the program code). For example, for bug #3361, a total of 39 duplicates have been reported by the same user, likely a tester. In the case of #3361, the reports have been created automatically because it took the tester only 1 minute to submit 40 bug reports.
- *Intentional resubmission.* Some users intentionally resubmit a bug, often out of frustration that it has not been fixed so far. For example, the duplicate #54603 was submitted more than eight months after the creation of the corresponding master bug #39064. The duplicate started with "I raised this issue several times, but it is still a problem."
- *Accidental resubmission.* A substantial number of duplicate reports are created by users who accidentally clicked the submit button multiple times. For example, bugs #28538 and #96565 each have four confirmed duplicates, which have been submitted by the same users at exactly the same time. For ECLIPSE, we found 521 confirmed duplicates that were likely caused by an accidental resubmission, i.e., duplicate reports which had the same title, same description, same product, and same component as a bug report, which was submitted less than 10 minutes before by the same user.

Some of the above reasons for bug duplicates could be easily addressed by better tool support, e.g., by an improved search feature for bugs (Recommendation #7, DebugAdvisor [8]) and a warning when a user is submitting the same bug again.

6 Rating Bug Reports

After completing the questionnaire, participants were asked to continue with a voluntary part of our survey. We presented randomly selected bug reports from their projects and asked them to rate the quality of these reports. Being voluntary, we did not mention this part in the invitation e-mail. While we asked both developers and reporters to rate bug reports, *we will use only the ratings by developers in this paper*, as they are more qualified to judge what is a good bug report.

6.1 Rating Infrastructure

The rating system was inspired by Internet sites such as RateMyFace [53] and HotOrNot [31]. We drew a random sample of 100 bugs from the projects' bug database, which were presented one by one to the participants in a random order. They were required to read through the bug report and rate it on a five-point Likert scale ranging from very poor (1) to very good (5) (see Fig. 4 for a screenshot). We did not show whether a bug report is a duplicate because we wanted participants to focus exclusively on the bug report's quality. Once they rated a bug report, the screen showed the next random report and the average quality rating of the previously rated report on the left. On the right, we provided a *skip* button, which as the name suggests, skips

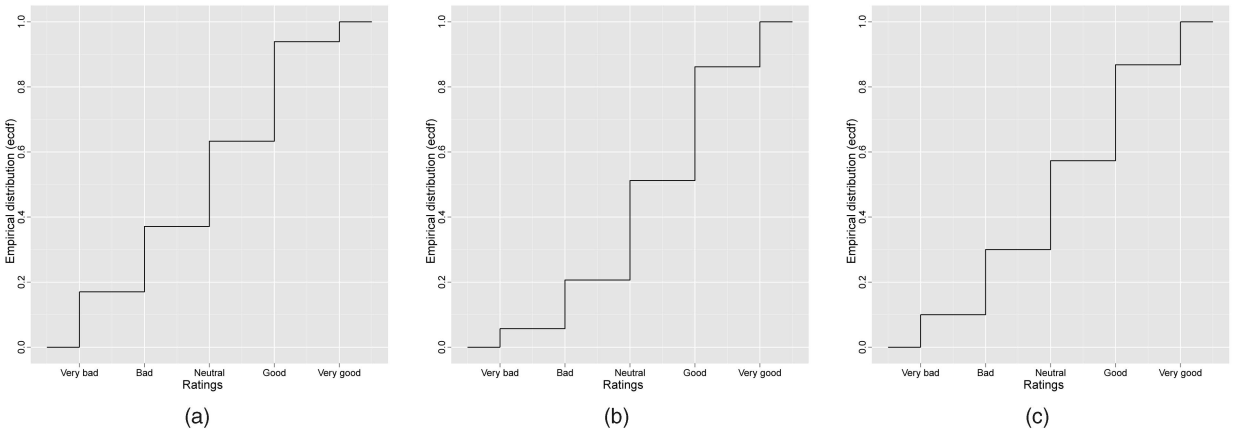


Fig. 5. Empirical cumulative distribution plots of the votes by developers. (a) Apache (229 votes). (b) Eclipse (455 votes). (c) Mozilla (560 votes).

the current report, and navigates to the next one. This feature seemed preferable to guesswork on the part of the participants in cases where they lacked the knowledge to rate a report. Participants could stop the session at any time or choose to continue until all 100 bugs had been rated.

These quality ratings by developers served two purposes:

1. They allowed us to verify the results of the questionnaire on concrete examples, i.e., whether reports with highly desired elements are rated higher for their quality and vice versa.
2. These scores were later used to evaluate our CUEZILLA tool that measures bug report quality (Section 7).

6.2 Rating Results

The following number of developer votes for bug reports was received for the samples of 100 bugs from each project: 229 for APACHE, 455 for ECLIPSE, and 560 for MOZILLA.

Fig. 5 plots the empirical cumulative distribution (ECDF) of the votes by developers. In an ECDF plot, the x-axis is the data under investigation; in our case, the ratings by developers on a Likert scale from very bad (numeric value of 1) to very good (5). The y-axis is the percentage of data points for which the rating is less than or equal to the rating on the x-axis. Thus, any coordinate (x, y) on the ECDF plot shows percentage of data (y percent) that lies within the range $(-\infty, x]$.

In Fig. 5, we see that very few developers rated bugs as very bad across all three projects. A slightly higher proportion of bugs were rated as bad by APACHE developers in comparison to the other projects. However, neutral and good were the most common ratings by developers from all three projects. Last, very few bug reports from the APACHE project were rated very good, while a larger percentage of bugs were rated the same by ECLIPSE and MOZILLA developers.

Table 5 lists the bug reports that were rated highest and lowest by ECLIPSE developers. Some bug reports were found to be of exceptional quality, such as bug report #31021 for which all three responders awarded a score of *very good* (5). This report presents a code example and adequately guides the developer on its usage and observed behavior.

I20030205
Run the following example. Double click on a tree item and notice that it does not expand.
Comment out the Selection listener and now double click on any tree item and notice that it expands.
public static void main(String[] args) {
Display display = new Display();
Shell shell = new Shell(display);
[...] (21 lines of code removed)
display.dispose();
}

(ECLIPSE bug report #31021)

On the other hand, bug report #175222 with an average score of 1.57 is of fairly poor quality. Actually, this is simply not a bug report and has been incorrectly filed in the bug database. Still, misfiled bug reports take away valuable time from developers.

I wand to create a new plugin in Eclipse using CDT. Shall it possible. I had made a R&D in eclipse documentation. I had get an idea about create a plugin using Java. But i wand to create a new plugin (user defined plugin) using CDT. After that I wand to impliment it in my programe. If it possible?. Any one can help me please ...

(ECLIPSE bug report #175222)

TABLE 5
Developers Rated the Quality of ECLIPSE Bug Reports

Bug Report	Votes	Rating
Tree - Selection listener stops default expansion (#31021)	3	5.00
JControlModel "eats up" exceptions (#38087)	5	4.8
Search - Type names are lost [search] (#42481)	4	4.50
150M1 withincode type pattern exception (#83875)	5	4.40
ToolItem leaks Images (#28361)	6	4.33
...	...	...
Selection count not updated (#95279)	4	2.25
Outline view should [...] show all project symbols (#108759)	2	2.00
Pref Page [...] Restore Defaults button does nothing (#51558)	6	1.83
[...]<Incorrect /missing screen capture> (#99885)	4	1.75
Create a new plugin using CDT. (#175222)	7	1.57

6.3 Concordance between Developers

We also investigated the concordance between developers on their evaluation of the quality of bug reports. It seems reasonable to assume that developers with comparable experiences have compatible views on the quality of bug reports. However, there may be exceptions to our belief or it may simply be untrue. We statistically verified concordance between developers by examining the standard deviations of quality ratings by developers (σ_{rating}) for the bug reports. Larger values of σ_{rating} indicate higher differences between developers' view of quality for a bug report. Of the 289 bug reports rated across all three projects, only 23 (which corresponds to 8 percent) had $\sigma_{\text{rating}} > 1.5$.

These results show that developers generally agree on the quality of bug reports. Thus, it is feasible to use their ratings to build a tool that learns from bug reports to measure the quality of new bug reports. We present a prototype of such a tool in the next section.

6.4 Preferred Information Items

The responses from the survey indicated which information items are desired by developers and reporters (Section 3). In this section, we now validate if the bug reports ratings were coherent with the responses from the survey. This analysis was conducted on the top five *bug reports with the highest average ratings* by developers and reporters separately for each project. After identifying these reports, we examined them to check for any patterns in their information contents.

We found that **developers** favored bug reports with code snippets and/or stack traces. In addition, they also preferred reports that provide a description of the environment (or the context) when the error occurred. Surprisingly reports with requests for features or documentation updates were not rated very high. However, this does not imply that developers do not welcome requests for new features; it may very well be an artifact of our sampling.

Reporters, on the other hand, appeared to only mildly favor bug reports with code snippets and demonstration of domain knowledge. Instead, they were more drawn toward reports with clear steps to reproduce, and observed and expected behavior. Also, bug reports that were either very long, haphazardly written, or missing information were less preferred by the reporters. Some bug reports appeared to have harsh tones; these were rated low by reporters but the developers seemed to be more tolerant to them.

6.5 Disagreements over Bug Report Quality

Developers and reporters may not share the same opinion about the quality of a bug report because of their different perspectives and levels of expertise. We performed a qualitative analysis to investigate disagreements in bug rating between developers and reporters. The degree of disagreement for a bug report was computed as the *absolute differences between the average ratings* from developers and reporters. Note that for this analysis, we only considered those reports that received at least two votes from each group.

We performed this analysis for each project separately because of a few project-specific preferences of information by developers and reporters (see Appendix B.3). From each project, we investigated the top-five bugs with the largest disagreements.

APACHE. Of the five bugs identified from the APACHE project, four were rated higher by the developers. The disagreement over the reports ranged from 1 to 1.75 points on the Likert scale.

Overall, all five bug reports exhibited good familiarity with the underlying system because of the use of domain-specific terminology such as *"Now when I try to iterate over the list of primary investigators in a JSP with the logic-tag"* in bug STR-2564 (favored by reporters) or "[...] *in this test case Xalan throws a NullPointerException I don't think it should throw. libslt can process this case."* in bug XALANJ-1837 (favored by developers).

One common pattern we observed in the bug reports favored by developers is that the reports provide additional analysis (e.g., of the problematic file in bug #13359) or provide solutions (e.g., the patch in bug WICKET-95). Although the bug report favored by the reporters includes code snippets as well, it merely provides input data to exhibit an exception.

ECLIPSE. All five bug reports from ECLIPSE were favored by the developers. The disagreement over the reports ranged from 1.13 to 1.5 points on the Likert scale.

Each report focused on specific and narrow issues, for example, *"ui.views.navigator.ResourceSorter still uses java.util.Collator"* in bug #158156. Such precise information was often coupled with domain-specific language such as *"test case using Sleak"* in bug #28361. This indicates that the bugs have been submitted by people with an understanding of ECLIPSE (actually, three of the five reports were submitted by IBM employees).

MOZILLA. Four of the top five bugs with disagreements were favored by developers. The disagreements ranged from 1.3 to 1.62 points on the Likert scale.

Again, developers showed a preference toward bug reports that used domain-specific language indicating familiarity with the project. One example is bug #243723 with *"Some methods on this interface (e.g., GetWidgetForView()) should really not be using COM-like signatures."* The remainder of the report describes the desired solution: *"it would be great if [the] lxr identifier searchers also worked for symbols which were #defined."*

The report favored by reporters gives a good description of the problem that a certain plug-in was not ported to the next version of the browser. However, porting the plug-in is not the responsibility of the MOZILLA developers, which may explain why the developers did not rate the bug high.

Generally, developers tend to favor reports that exhibit an understanding of the problem and the underlying program. The extent of this knowledge may vary from one project to another. In the cases of APACHE and ECLIPSE, whose users are more technically inclined, expectations may be higher in comparison to MOZILLA, which has a wider audience of users.

7 Measuring Bug Report Quality with CUEZILLA

In Section 3.3, we showed that the majority of reporters know what is important in bug reports. However, we also found evidence for an information mismatch: The importance of

some items, e.g., screenshots, is not recognized by users. There are also black sheep among users who do not know yet how to write good bug reports. In general, humans can benefit from cues while undertaking tasks, which was demonstrated in software engineering by Passing and Shepperd [51]. They examined how subjects revised their initial cost estimates of projects upon being presented checklists relevant to estimation.

Our conjecture is that bug reporters can provide better reports with similar assistance. As a first step toward assistance, we developed a prototype tool called CUEZILLA that measures the quality of bug reports. CUEZILLA also provides suggestions on how to enhance the quality of a bug report, for example, *"Have you thought about adding a screenshot to your bug report?"* To encourage reporters to actually provide additional information, CUEZILLA can show *did you know* facts mined from bug databases; for example, *"Bug reports with stack traces are fixed N-times faster."* Possible usage scenarios for CUEZILLA are to provide immediate feedback while new bug reports are entered, to solicit information for bug reports that are already in the bug database, or to prioritize bug reports during bug triage.

This section presents details on how CUEZILLA works and reports results of its evaluation at measuring quality of bug reports. To create recommendations, CUEZILLA first represents each bug report as a feature vector (Section 7.1). Then, it uses supervised learning to train models (Section 7.3) that measure the quality of bug reports (Sections 7.4 and 7.6). Our models can also quantify the increase in quality when elements are added to bug reports (Section 7.7). In contrast to other quality measures for bug reports such as lifetime [30], we use the ratings that we received by developers. Last, we describe the process of mining *did you know* facts from bug databases (Section 8).

7.1 Input Features

Our CUEZILLA tool measures the quality of bug reports on the basis of their contents. From the survey, we know the most desired features in bug reports by developers. Endowed with this knowledge, CUEZILLA first detects the features listed below. For each feature, a score is awarded to the bug report, which is either binary (e.g., attachment present or not) or continuous (e.g., readability).

Itemizations. In order to recognize itemizations in bug reports, we checked whether several subsequent lines started with an itemization character (such as —, *, or +). To recognize enumerations, we searched for lines starting with numbers or single characters that were enclosed by parentheses or brackets or followed by a single punctuation character.

Keyword completeness. We reused the data set provided by Ko et al. [40] to define a quality score of bug reports based on their content. In a first step, we removed stop words, reduced the words to their stem, and selected words occurring in at least 1 percent of bug reports. Next, we categorized the words into the following groups:

- action items (e.g., open, select, and click),
- expected and observed behavior (e.g., error and missing),
- steps to reproduce (e.g., steps and repro),
- build-related (e.g., build), and
- user interface elements (e.g., toolbar, menu, and dialog).

In order to assess the *completeness* of a bug report, we computed for each group a score based on the keywords present in the bug report. The maximum score of 1 for a group is reached when a keyword is found.

In order to obtain the final score (which is between 0 and 1), we averaged the scores of the individual groups.

In addition to the description of the bug report, we analyze the attachments that were submitted by the reporter within 15 minutes after the creation of the bug report. In the initial description and attachments, we recognize the following features:

Code samples. We identify C++ and JAVA code examples using techniques from island parsing [45]. Currently, our tools can recognize declarations (for classes, methods, functions, and variables), comments, conditional statements (such as `if` and `switch`), and loops (such as `for` and `while`).

Stack traces. We currently can recognize JAVA stack traces, GDB stack traces, and MOZILLA talkback data. Stack traces are easy to recognize with regular expressions: They consist of a start line (that sometimes also contains the top of the stack) and trace lines.

Patches. In order to identify patches in bug reports and attachments, we again used regular expressions. They consist of several start lines (which file to patch) and blocks (which are the changes to make) [43].

Screenshots. We identify the type of an attachment using the *file* tool in UNIX. If an attachment is an image, we recognize it as a *screenshot*. If the file is recognized as text, we process the file and search for code examples, stack traces, and patches (see above).

For more details about extraction of structural elements from bug reports, we refer to our previous work [16] in which we showed that we can identify the above features with close to perfect precision.

After cleaning the description of a bug report from source code, stack traces, and patches, we compute its readability.

Readability. To compute readability, we use the *style* tool, which "analyzes the surface characteristics of the writing style of a document" [21]. It is important to not confuse readability with grammatical correctness. The readability of a text is measured by the number of syllables per word and the length of sentences. Readability measures are used by Amazon.com to inform customers about the difficulty of books and by the US Navy to ensure readability of technical documents [38].

In general, the higher a readability score, the more complex a text is to read. Several readability measures return values that correspond to school grades. These grades tell how many years of education a reader should have before reading the text without difficulties. For our experiments, we used the following seven readability measures: Kincaid, Automated Readability Index (ARI), Coleman-Liau, Flesh, Fog, Lix, and SMOG Grade.[6]

6. This paper has an SMOG-Grade of 13, which requires the reader to have some college education. Publications with a similar SMOG-grade are often found in the *New York Times*.

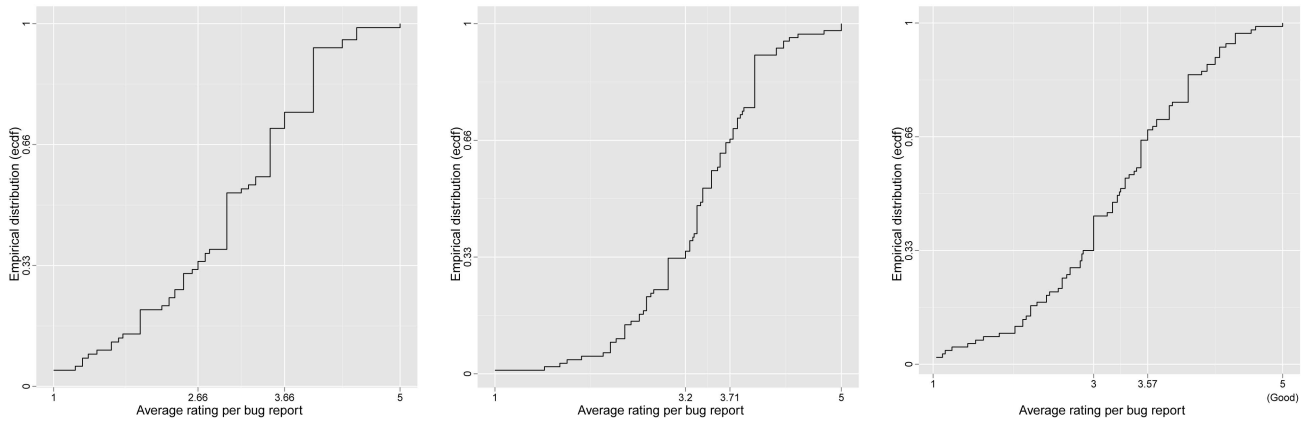


Fig. 6. Empirical cumulative distribution plots of the average quality of bug reports. (a) Apache. (b) Eclipse. (c) Mozilla.

We did not use the information whether a bug report is a duplicate as input feature because this was not available to participants when they rated bug reports. Furthermore, bug duplicates and bug report quality are different concepts, e.g., there are well-written bug duplicates that actually do add new information (see Section 5).

7.2 Output Feature

The output feature to train our supervised learning models is the average quality (awarded by developers) for each rated bug report. Recall that developers could rate bug reports on a quality scale from level 1 (very bad) to level 5 (very good). However, very few reports were classified as very bad or very good, resulting in an imbalance in their distribution across the quality levels. Supervised learning models can be very sensitive to such imbalances that may cause them to underperform. In order to avoid this, we balanced the data: We computed the ECDFs of the average ratings for each project and distributed the bug reports evenly across three different quality levels—bad, neutral, and good.

Fig. 6 shows the ECDF plots of the average bug report ratings from each project. Recall that any coordinate (x, y) on the ECDF plot shows the percentage of data (y percent) that lies within the range from $-(\infty, x]$. We used the ECDF plots to label the bug reports with the lowest 33 percent average ratings as *bad* bug reports. The next 33 percent of bug reports were labeled as *neutral* bug reports, while the remaining bug reports were rated as *good* bug reports. These three classes—bad, neutral, and good bug reports—were then used as the output feature to train our models. The intervals used to balance the data and the resulting class sizes are listed below the ECDF plots in Fig. 6. Note that a perfect balance is not always possible because ECDFs are stepwise functions. For example, the APACHE project has a big step at $x = 3.50$ from $y = 0.58$ to 0.71 (12 bug reports), which we labeled as *neutral*; as a result we have more neutral bug reports in APACHE.

7.3 Evaluation Setup

Out of the 300 bug reports in the sample, developers rated 289 bug reports at least once. These reports were used to train and evaluate CUEZILLA by building supervised learning models. We used the following models: support vector machines (SVMs), decision trees, random forests, linear regression, and stepwise linear regression [61], [64]

Each model used the scores from the features described in Section 7.1 as input variables and predicted the quality level as described in Section 7.2. For classification models, we predicted the quality level directly; for regression models, we first predicted a number, which was then mapped to one of the three classes poor, neutral, or good. We evaluated CUEZILLA using two setups:

Within-project. To test how well models predict within a project, we used the *leave-one-out* validation setup—for a given project, the quality of each bug report is predicted by learning a model from all other bug reports. Since we have limited data, we chose this setup to maximize the training data to build the prediction models.

Cross project. We also tested whether trained models from one project can be transferred to others. To exemplify, we built a model from all rated bug reports of project A and applied it to predict the quality of all rated bugs in project B.

TABLE 6
The Results of the Classification by CUEZILLA (Using Support Vector Machines Classification, SVMC, with Linear Kernel) Compared to the Developer Ratings for ECLIPSE Bug Reports

	Observed (Developers)		
Predicted	Bad	Neutral	Good
Bad	20	15	6
Neutral	9	10	13
Good	4	8	15

Correctly assessed bug reports are in the cells with a dark background.

TABLE 7
Percentage Accuracy of CUEZILLA Using the Within-Project Evaluation Setup

Model	APACHE (%)	ECLIPSE (%)	MOZILLA (%)
SVMC:* linear kernel	41	**45**	41
SVMC: polynomial kernel	39	37	27
SVMC: radial kernel	39	42	**43**
SVMC: sigmoid kernel	48	36	42
Decision trees	28	**45**	34
Random forest	44	43	32
SVMR:** linear kernel	**50**	37	37
SVMR: polynomial kernel	37	32	30
SVMR: radial kernel	47	36	26
SVMR: sigmoid kernel	40	30	27
Linear regression	46	38	35
Stepwise linear regression	38	35	33

* SVMC = Support vector machines classification
** SVMR = Support vector machines regression

7.4 Within-Project Evaluation

Table 6 illustrates the results from evaluating CUEZILLA on ECLIPSE bug reports using SVM classification. The column names in the table indicate the average rating of the bug report by developers (Observed). The row names denote the quality as classified by CUEZILLA (Predicted). The counts in the diagonal cells of the table indicate the number of bug reports for which there was complete agreement between CUEZILLA and developers. Forty-five of the 100 ECLIPSE bug reports were classified by CUEZILLA in agreement with the developers, which gives us an accuracy of 45 percent.

Similarly, the prediction accuracy of other models was computed likewise and the results are presented in Table 7. The numbers in the columns represent the accuracy of each model. The performance of each of the 12 models differs between projects, with none emerging as a clear winner. In the case of APACHE, SVM regression with a linear kernel delivered the top results with 50 percent accuracy. For MOZILLA, SVM classification with a radial kernel worked best with 43 percent accuracy. For ECLIPSE, decision trees classified 45 percent of bug reports correctly and tied with SVM classification on a linear kernel.

On the whole, classification-based models performed comparably well with regression-based models. These results suggest that CUEZILLA can, to some extent, mimic the reasoning of developers to rate the quality of bug reports; however, we must further investigate means and methods to arrive at higher accuracy. To encourage further research in this area, we make our data set and R scripts publicly available (see Appendix A).

7.5 Importance of Input Features

Next, we investigated which input features are most important for each project. For this analysis, we computed C4.5 decision trees with the Weka implementation J48 [64] on all bug reports of a project. Rather than for prediction, we use the decision trees in this section to describe our data and important input features.

```
smog <= 0.285235: BAD (30.0/11.0)
smog > 0.285235
|   steps_hasitemize = FALSE
|   |   kincaid <= 0.085161: NEUTRAL (5.0)
|   |   kincaid > 0.085161
|   |   |   flesch <= 0.42246: NEUTRAL (21.0/7.0)
|   |   |   flesch > 0.42246
|   |   |   |   code = FALSE
|   |   |   |   |   smog <= 0.35906: GOOD (15.0/7.0)
|   |   |   |   |   smog > 0.35906: BAD (8.0/2.0)
|   |   |   |   code = TRUE: GOOD (6.0)
|   steps_hasitemize = TRUE: GOOD (5.0/2.0)
```

(a)

```
keywords_score <= 0.025: BAD (8.0)
keywords_score > 0.025
|   steps_hasitemize = FALSE
|   |   kincaid <= 0.206226
|   |   |   fog <= 0.225: BAD (9.0/3.0)
|   |   |   fog > 0.225: NEUTRAL (11.0/2.0)
|   |   kincaid > 0.206226
|   |   |   code = FALSE
|   |   |   |   coleman_liau <= 0.393665: GOOD (15.0/6.0)
|   |   |   |   coleman_liau > 0.393665
|   |   |   |   |   coleman_liau <= 0.565611
|   |   |   |   |   |   lix <= 0.435556: BAD (7.0)
|   |   |   |   |   |   lix > 0.435556: NEUTRAL (10.0/3.0)
|   |   |   |   |   coleman_liau > 0.565611
|   |   |   |   |   |   stacktrace = FALSE: GOOD (10.0/4.0)
|   |   |   |   |   |   stacktrace = TRUE: BAD (5.0/3.0)
|   |   |   code = TRUE: GOOD (6.0/1.0)
|   steps_hasitemize = TRUE
|   |   fog <= 0.232143: NEUTRAL (5.0/2.0)
|   |   fog > 0.232143: GOOD (14.0/3.0)
```

(b)

```
steps_hasitemize = FALSE
|   coleman_liau <= 0.670051
|   |   keywords_score <= 0.5: BAD (38.0/13.0)
|   |   keywords_score > 0.5
|   |   |   coleman_liau <= 0.467005: BAD (5.0/2.0)
|   |   |   coleman_liau > 0.467005: GOOD (7.0/2.0)
|   coleman_liau > 0.670051: NEUTRAL (7.0/2.0)
steps_hasitemize = TRUE
|   flesch <= 0.637947: BAD (8.0/2.0)
|   flesch > 0.637947
|   |   smog <= 0.536585: GOOD (28.0/11.0)
|   |   smog > 0.536585: NEUTRAL (6.0/2.0)
```

(c)

Fig. 7. Decision trees from the projects. (a) Apache. (b) Eclipse. (c) Mozilla.

TABLE 8
Percentage Accuracy of CUEZILLA Using the Cross-Project Evaluation Setup

Training on	Model	Cross-project: Testing on APACHE	ECLIPSE	MOZILLA	Within-project (leave-one-out)
APACHE	Linear regression	-	38	37	46
	Random forest		45	32	44
	SVMC:* linear kernel		49	36	41
ECLIPSE	Linear regression	31	-	34	38
	Random forest	30		42	43
	SVMC: linear kernel	33		42	45
MOZILLA	Linear regression	33	34	-	35
	Random forest	39	40		32
	SVMC: linear kernel	35	34		41

* SVMC = Support vector machines classification

Fig. 7 shows the decision trees for each project; the most important features are placed at the top levels. *Decision nodes* are inner nodes in the tree and are aligned with vertical lines; for example, in APACHE, the topmost decision node splits the data into two separate paths based on whether the (normalized) *Smog* score is greater than 0.285235. *Leaf nodes* mark the end of a path in the tree and are classified into our quality levels, BAD, NEUTRAL, or GOOD. Leaf nodes are followed by two numbers enclosed in parentheses. The first is the total number of bug reports summarized by this leaf (path), while the second is the number of reports misclassified (omitted if zero).

In the case of APACHE, the most important variable is the readability score *Smog*, followed by whether a bug report has any *itemizations*, and the *Kincaid* score. The three most important features in ECLIPSE are the measure of *keyword completeness*, whether a bug report contains any *itemization*, and the *Kincaid* score. Other readability scores such as *Fog*, *Coleman-Liau*, and *Lix* also appear to play an important role in the quality ratings of ECLIPSE bug reports. Last, in the case of MOZILLA, the presence of *itemizations* and the *Coleman-Liau* and *Flesch* scores are the most important features.

It is noteworthy that in all projects, the top three important features comprise at least one readability score. Further itemizations also belonged to the top features in all three projects; we suspect that they too contribute toward good organization of information. These results further emphasize the importance of well-written bug reports that are easy to understand.

7.6 Cross-Project Evaluation

In Table 8, we present the prediction accuracies from the cross-project evaluation setup. For the sake of brevity, we only present the results from three models in the table: linear regression, random forests, and SVM classification (linear kernel). In the table, the training projects for the models are presented as rows, while testing projects are presented as columns. For easier comparison, we provide in the last column the results from the within-project evaluation, which was discussed earlier in this section.

Overall, the prediction performance from the cross-project evaluation setup was lower than the within-project evaluation setup. However, it is noteworthy that models trained on data from APACHE are good at classifying ECLIPSE reports, and ECLIPSE models can reasonably well classify MOZILLA bug reports. One can infer from these results that CUEZILLA's models are largely portable across projects to predict quality, but they are best applied within projects.

7.7 Recommendations by CUEZILLA

The core motivation behind CUEZILLA is to help reporters file better quality bug reports. For this, its ability to detect the presence of information features can be exploited to tip reporters obout what information to add. This can be achieved simply by recommending additions from the set of absent information, starting with the feature that contributes to the quality further by the largest margin. These recommendations are intended to serve as cues or reminders to reporters of the possibility of adding certain types of information likely to improve bug report quality.

Fig. 8a illustrates the concept. The text in the panel is determined by investigating the current contents of the report, and then determining that it would be best, for instance, to add a code sample to the report. As and when new information is added to the bug report, the quality meter revises its score.

Our evaluation of CUEZILLA shows much potential for incorporating such a tool in bug tracking systems. CUEZILLA is able to measure quality of bug reports within reasonable accuracy. However, the presented version of CUEZILLA is an early prototype and we plan to further

Fig. 8. Mockup of CUEZILLA's user interface. (a) It recommends improvements to the report. (b) To encourage the user to follow the advice, CUEZILLA provides facts that are mined from history.

enhance the accuracy before we will conduct user studies to show CUEZILLA's usefulness. We briefly discuss our plans in Section 11.

8 Incentive for Reporters

If CUEZILLA tips reporters on how to enhance the quality of their bug reports, one question comes to mind: *"What are the incentives for reporters to do so?"* Of course, well-described bug reports help in comprehending the problem better, consequently increasing the likelihood of the bug getting fixed. But to explicitly show evidence of the same to reporters, CUEZILLA randomly presents relevant facts that are statistically mined from bug databases. In this section, we elaborate upon how this is executed, and close with some facts found in the bug databases of the three projects.

To reduce the complexity of mining the several thousand bug reports filed in bug databases, we sampled 50,000 bugs from each project. These bugs had various resolutions, such as FIXED, DUPLICATE, MOVED, WONTFIX, and WORKSFORME. Then, we computed the scores for all items listed in Section 7.1 for each of the 150,000 bugs. To recall, the scores for some of the items are continuous values, while others are binary.

8.1 Relation to Resolution of Bug Reports

A bug being fixed is a mark of success for both developers and reporters. But what items in bug reports increase the chances of the bug getting fixed? We investigate this on the sample of bugs described above for each project.

First, we grouped bug reports by their resolutions as: FIXED, DUPLICATE, and OTHERS. The FIXED resolution is most desired and the OTHERS resolution—which includes MOVED, WONTFIX, and the likes—are largely undesired. We chose to examine DUPLICATE as a separate group because this may potentially reveal certain traits of such bug reports. Additionally, as pointed out above, duplicates may provide more information about the bug to developers.

For binary-valued features, we performed Chi-Square tests [56] ($p < 0.05$) on the contingency tables of the three resolution groups and the individual features for each project separately. The tests' results indicate whether the presence of the features in bug reports significantly determine the resolution category of the bug. For example, the presence of stack traces significantly increases the likelihood of a FIXED desirable resolution.

In the case of features with continuous-valued scores, we performed a Kruskal-Wallis test [56] ($p < 0.05$) on the distribution of scores across the three resolution groups to check whether the distribution significantly differs from one group to another. For example, bug reports with FIXED resolutions have significantly lower SMOG-grades than reports with OTHERS resolutions, indicating that reports are best written using simple language constructs.

8.2 Relation to Lifetime of Bug Reports

Another motivation for reporters is to see what items in bug reports help making the bugs' lifetimes shorter. Such motivations are likely to incline reporters to furnish more helpful information. We mined for such patterns on a subset of the above 150,000 bugs with resolution *FIXED* only.

For items with binary scores, we grouped bug reports by their binary scores, for example, bugs containing stack traces and bugs not containing stack traces. We compared the distribution of the lifetimes of the bugs and, again, performed a Kruskal-Wallis test [56] ($p < 0.05$) to check for statistically significant differences in distributions. This information would help encourage reporters to include items that can reduce lifetimes of the bugs.

In the case of items with continuous-valued scores, we first dichotomized the lifetime into two categories: bugs resolved quickly versus bugs resolved slowly. We then compared the distribution of the scores across the two categories using the Kruskal-Wallis test [56] ($p < 0.05$) to reveal statistically significant patterns. Differences in distributions could again be used to motivate users to aim at achieving scores for their reports that are likely to have lower lifetimes. In our experiments, we used one hour, one day, and one week as boundaries for dichotomization.

8.3 Results

This section lists some of the key statistically significant patterns found in the sample of 150,000 bug reports. These findings can be presented in interfaces of bug tracking systems (see Fig. 8b). A sample of our key findings is listed below:

- Bug reports containing stack traces get fixed sooner. (APACHE/ECLIPSE/MOZILLA)
- Bug reports that are easier to read have lower lifetimes. (APACHE/ECLIPSE/MOZILLA)
- Including code samples in your bug report increases the chances of it getting fixed. (MOZILLA)

We are not the first to find factors that influence the lifetime of bug reports. Independently from us, Hooimeijer and Weimer [30] observed for FIREFOX that bug reports with attachments get fixed later, while bug reports with many comments get fixed sooner. They also confirmed our results that easy-to-read reports are fixed faster. Panjer observed for ECLIPSE that comment count and activity, as well as severity, affect the lifetime the most [50].

In contrast, our findings are for factors that can be determined while a user is reporting a bug. Each finding suggests a way to increase the likelihood of their bugs either getting fixed at all or getting fixed faster. Keen users are likely to pick up on such cues since this can lessen the amount of time they have to deal with the bug.

9 Threats to Validity

For our survey, we identified the following threats to validity.

Our *selection of developers* was constrained to only experienced developers; in our context, developers who had at least 50 bugs assigned to them. While this skews our results toward developers who frequently fix bugs, they are also the ones who will benefit most by an improved quality of bug reports. The same discussion applies to the *selection of reporters*.

A related threat is that to some extent, our survey operated on a *self-selection principle:* Participation in the survey was voluntary. As a consequence, results might be skewed toward people who are likely to answer the survey, such as developers and users with extra spare time—or who care about the quality of bug reports.

Avoiding the self-selection principle is almost impossible in an open-source context. While a sponsorship from the Foundations of APACHE, ECLIPSE, and MOZILLA might have reduced the amount of self-selection, it would not have eliminated skew. As pointed out by Singer and Vinson, the decision of responders to participate "could be unduly influenced by the perception of possible benefits or reprisals ensuing from the decision" [57].

In order to take as little of the participants' time as possible, we constrained the *selection of items* in our survey. While we tried to achieve completeness, we were aware that our selection was not exhaustive of all information used and problems faced by developers. Therefore, we encouraged participants to provide us with additional comments, to which we received 175 responses. We could not include the comments into the statistical analysis; however, we studied and discussed the comments by developers in Section 3.

As with any empirical study, it is difficult to draw general conclusions because any process depends on a *large number of context variables* [10]. In our case, we contacted developers and users of three large open-source initiatives, namely, APACHE, ECLIPSE, and MOZILLA. We are confident that our findings also apply to smaller open-source projects. However, we do not contend that they are transferable to closed-software projects (which have no patches and rarely stack traces). In future work, we will search for evidence for this hypothesis and point out the differences in the quality of bug reports between open-source and closed-source development.

A common misinterpretation of empirical studies is that nothing new is learned ("I already knew this result"). Unfortunately, some readers miss the fact that this wisdom has rarely been shown to be true and is often quoted without scientific evidence. This paper provides such evidence: Most common wisdom is confirmed (e.g., "steps to reproduce are important"), while other is challenged ("bug duplicates considered harmful").

10 Related Work

So far, mostly anecdotal evidence has been reported on what makes a good bug report. For instance, Spolsky described how to achieve painless bug tracking [58] and numerous articles and guidelines on effective bug reporting float around the Internet (e.g., [27]). Still, the results from our survey suggest that bug reporting is far from being painless.

The work closest to ours is by Hooimeijer and Weimer who built a descriptive model for the lifetime of a bug report [30]. They assumed that the "time until resolved" is a good indicator for the quality of a bug report. In contrast, our notion of quality is based on feedback from developers (1,244 votes). When we compared the ratings of the bug reports with lifetime, the Spearman correlation values were between 0.002 and 0.068, indicating that lifetime as measured by Hooimeijer and Weimer [30] and quality are independent measures. Often a bug report that gets addressed quicker can be of poor quality, but describes an urgent problem. Also, a well-written bug report can be complicated to deal with and take more time to resolve. Still, knowing what contributes to the lifetime of bug reports [1], [30], [50] can encourage users to submit better reports, as discussed in Section 8.

Bird et al. investigated the effects of bias in bug data sets for open-source projects [17]. Their work focuses on the quality of an entire data set for defect prediction. In contrast, our work concentrates on the quality of individual bug reports, mostly to support developers in fixing bugs and not to develop prediction techniques.

More recently, Aranda and Venolia examined the communication between developers about bug reports at Microsoft [6]. They observed that many bugs are discussed even before a bug report is created and that not all information is recorded in bug tracking systems. We believe that this observation is specific to Microsoft or industry in general (where developers share a common office). In open-source software, most bugs are discussed in bug tracking systems (or special mailing lists) to ensure transparency and accommodate developers who are geographically distributed.

In a workshop paper, we presented preliminary results from the developer survey on the ECLIPSE project using a handcrafted prediction model [12]. In other work, we quantified the amount of additional information in bug duplicates [14] and gave recommendations on how to improve existing bug tracking systems [35], [66]. This paper is an extended version of this previous work [13].

Several studies used bug reports to automatically assign developers to bug reports [4], [5], [20], assign locations to bug reports [19], track features over time [26], recognize bug duplicates [22], [29], [32], [54], [60], assess the severity [44], predict effort for bug reports [63], identify bug tossing [33], and characterize fixed bug reports [28]. All of these approaches should benefit from our quality measure for bug reports since training only with high-quality bug reports will likely improve their predictions.

In a separate study, Schröter et al. [55] showed the value of stack traces for developers when fixing bugs. Breu et al. [18] identified information needs in bug reports. Several researchers studied the social and collaborative aspects of bug tracking systems: Bertram et al. [11] studied small, collocated teams, van Liere [59] analyzed how information provided by open-source community members influences the repair time of software defects, and Ko and Chilana [39] studied how power users help and hinder bug reporting in the Mozilla project.

In 2004, Antoniol et al. [3] pointed out the lack of integration between version archives and bug databases, which make it hard to locate the most faulty methods in a system. In the meantime, things have changed: The Mylyn tool by Kersten and Murphy [36] allows attaching a task context to bug reports so that changes can be tracked on a very fine-grained level. Antoniol et al. [2] also pointed out that not all bug reports are related to software problems; in some cases, bug reports correspond to feature requests (which is indicated by a special property).

In order to inform the design of new bug reporting tools, Ko et al. [40] conducted a linguistic analysis of the titles of bug reports. They observed a large degree of regularity and a substantial number of references to visible software entities, physical devices, or user actions. Their results suggest that future bug tracking systems should collect data in a more structured way.

According to the results of our survey, errors in steps to reproduce are one of the biggest problems faced by

developers. This demonstrates the need for tools that can capture the execution of a program on the user side and replay it on the developer side. While there exist several capture/replay techniques (such as [34], [47], [65]), their user orientation and scalability can still be improved. Dit and Marcus suggested an approach to improve the readability of discussions in bug reports which is based on making connections between comments explicit [24].

Not all bug reports are generated by humans. Some bug-finding tools can report violations of safety policies and annotate them with backtraces or counterexamples. Weimer presented an algorithm to construct such patches automatically. He also found that automatically generated "reports also accompanied by patches were three times as likely to be addressed as standard bug reports" [62].

Furthermore, users can help developers to fix bugs without filing bug reports. For example, many products ship with automated crash reporting tools that collect and send back crash information, e.g., Apple CrashReporter, Windows Error Reporting, Gnome BugBuddy, and Mozilla Talkback. Liblit et al. introduced statistical debugging [41]. They distribute specially modified versions of software which monitor their own behavior while they run and report back how they work. This information is then used to isolate bugs using statistical techniques. Still, since it is unclear how to extract sufficient information for rarely occurring and noncrashing bugs, there will always be the need for manual bug reporting.

11 Conclusion and Consequences

Well-written bug reports are likely to get more attention among developers than poorly written ones. We conducted a survey among developers and users of APACHE, ECLIPSE, and MOZILLA to find out what makes a good bug report. The results suggest that across all three projects, steps to reproduce and stack traces are most useful in bug reports. The most severe problems encountered by developers are errors in steps to reproduce, incomplete information, and wrong observed behavior. Surprisingly, bug duplicates are encountered often but not considered as harmful by developers. In addition, we found evidence for a mismatch between what information developers consider as important and what users provide. To a large extent, lacking tool support causes this mismatch.

We also asked developers to rate the quality of bug reports on a scale from one (poor quality) to five (excellent quality). Based on these ratings, we developed a prototype called CUEZILLA that measures the quality of bug reports. Additionally, it recommends what additions can be made to bug reports to make their quality better. To provide incentive for doing so, CUEZILLA automatically mines patterns that are relevant to fixing bugs and presents them to users. In the long term, an automatic measure of bug report quality in bug tracking systems can ensure that new bug reports meet a certain quality level. Our future work is as follows.

Problematic contents in reports. Currently, we award scores for the presence of desired contents, such as itemizations and stack traces. We plan to extend CUEZILLA to identify problematic contents such as errors in steps to reproduce and code samples in order to warn the reporter in these situations.

Impact on other research. In Section 10, we discussed several approaches that rely on bug reports as input to support developers in various tasks such as bug triaging, bug localization, and effort estimation. Do these approaches improve when trained only with high-quality bug reports? To facilitate further research in this area, we made our data and scripts publicly available (see Appendix A).

Usability studies for new bug reporting tools. We listed several comments by developers about problems with existing bug reporting tools in Section 3. To address these problems, we plan to develop prototypes for new, improved reporting tools which we will test with usability studies.

Additionally, aiding reporters in providing better bug reports can go a long way in structuring bug reports. Such structured text may also be beneficial to researchers who use them for experiments. In effect, in the short to medium term, data quality in bug databases would generally increase, in turn providing more reliable and consistent data to work with and feedback to practitioners. We encourage readers to replicate and extend our research; our data set and R scripts accompany this paper and are explained in Appendix A.

To learn more about our work in mining software archives, visit http://www.softevo.org/.

Appendix A
How to Extend This Study

To encourage readers to replicate and extend our study, we share our data set and R scripts. In this section, we present two small case studies to demonstrate potential extensions of our work.

- *How does the length of a bug report (in characters) affect its quality?* Longer bug reports will likely contain more information, and thus be perceived as better by developers; however, some bug reports could be too long and contain excessive details.
- *Do reporters with more experience and higher reputation submit better bug reports?* Reporters who submitted many bugs will likely know how to write good reports. Hooimeijer and Weimer [30] observed that bugs submitted by reporters with high reputation are fixed sooner; do they also write better bug reports?

A.1 The Data Set

The data set and R scripts that we used for this paper are attached as Supplemental Material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2010.63.

We provide two files: *bugreports.zip* contains the raw bug reports (with attachments) which were rated by developers and reporters as part of our survey (Section 6). From these reports, we built the data that we used in our experiments (Section 7). This data and the corresponding R scripts are contained in *experiments.zip*. In most cases, for example, to replicate the experiments in this paper and section, you will simply need *experiments.zip*.

To save typing the R commands in this section, load file *appendix.R*. The file contains the commands for APACHE, ECLIPSE, and MOZILLA; however, for brevity, we discuss only ECLIPSE in this section.

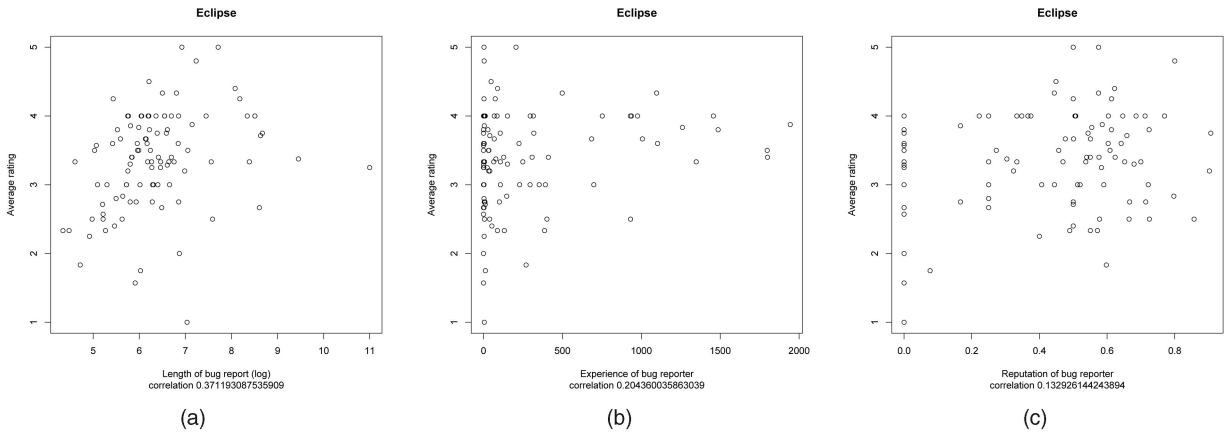


Fig. 9. The influence of various factors on bug report quality. (a) Length of bug report. (b) Experience of bug reporter. (c) Reputation of bug reporter.

A.2 Initialize the Workspace

First, we need to prepare the workspace for the experiments. Load the original data set (*dataset.eclipse.csv*) and the new data, in our case, the length of bug reports (*length.eclipse.csv*) and the experience and reputation of bug submitters (*reputation.eclipse.csv*). Finally, combine the data sets with the *merge()* function (similar to a SQL join) and check that the data loaded properly with the *summary()* function.

```
data.eclipse <- read.csv
 ("dataset.eclipse.csv")

bug.length.eclipse <- read.csv
 ("length.eclipse.csv")
reputation.eclipse <- read.csv
 ("reputation.eclipse.csv")

data.eclipse <- merge(data.eclipse,
 bug.length.eclipse, all.x=T, by.x = "bug_id",
 by.y = "bug_id")
data.eclipse <- merge(data.eclipse,
 reputation.eclipse, all.x=T, by.x = "bug_id",
 by.y = "bug_id")

summary(data.eclipse)
```

A.3 Check Hypothesis: Length of a Bug Report

We first check whether the length of a bug report (in characters) affects its quality. We create a scatter plot between the length and average rating of bug reports with the *plot()* function. In addition, we compute the Spearman correlation with the function *cor()* and add it as a subtitle to the plot with the function *title()*.

```
plot(log(data.eclipse$length),
     data.eclipse$votes_mean,
     xlab="Length of bug report (log)",
     ylab="Average rating")
title(main="Eclipse")

cor.value <- cor(data.eclipse$length,
  data.eclipse$votes_mean,
  method="spearman")
title(sub=paste("correlation", cor.value))
```

Fig. 9a shows the results of the above R commands. In the plot, we observe that the quality of bug reports seems to be increasing with their length. This observation is supported by the positive and moderate Spearman correlation of 0.371. This finding is no surprise because longer bug reports are more likely to have more information, such as stack traces and patches, and thus are perceived to be of higher quality by developers.

A.4 Check Hypothesis: Experience and Reputation

Next, we check the influence of experience and reputation of the reporter on bug reports. We define them as follows:

1. *Experience* is the number of bug reports that a reporter opened before the current bug report.
2. *Reputation* is based on Hooimeijer and Weimer [30] and corresponds roughly to the percentage of successfully fixed bug reports opened by the reporter before the current bug report; in addition, it includes a correction factor of $+1$ to prevent division by zero and increase the reputation of frequent bug reporters:

$$\text{Reputation} = \frac{|S \cap R|}{|S| + 1}.$$

In the equation, S is the set of all reports opened by the reporter before the current bug report and R is the set of all bug reports that were marked as FIXED in the bug database.

The R commands for this experiment are similar to ones in the previous section. Again, we use the functions *plot()* and *cor()* to produce a scatter plot and compute the Spearman correlation. Instead of the length of bug reports, we now analyze the variables *opened* and *reputation*.

```
plot(data.eclipse$ opened,
     data.eclipse$votes_mean,
     xlab="Experience of bug reporter",
     ylab="Average rating")
title(main="Eclipse")
```

```
cor.value <- cor(data.eclipse$opened,
     data.eclipse$votes_mean,
     method="spearman")
title(sub=paste("correlation", cor.value))

plot(data.eclipse$ reputation,
     data.eclipse$votes_mean,
     xlab="Reputation of bug reporter",
     ylab="Average rating")
title(main="Eclipse")

cor.value <- cor(data.eclipse$reputation,
     data.eclipse$votes_mean,
     method="spearman")
title(sub=paste("correlation", cor.value))
```

Fig. 9b shows the results for the *experience* of bug reporters. In general, reporters who opened many bugs in the past are more likely to submit better bug reports, which suggests the presence of a learning effect. This observation is supported by the Spearman correlation of 0.204, which is weak but positive. Reporters with no past experience ($opened = 0$) submit very poor but also very good bug reports, which means that newcomers to bug tracking start at different levels.

Fig. 9c shows the results for the *reputation* of bug reporters. Here, it is impossible to identify a trend: The range of average ratings for bug reports is about the same for reporters with low reputation and reporters with high reputation; in addition, the Spearman correlation of 0.132 is very low. This suggests that the reputation of a reporter, i.e., the success rate of getting bugs fixed in the past, has little impact on the quality of newly submitted bug reports. Instead, the experience of the reporter, as measured by the number of bug reports opened in the past, seems to be a better indicator. (Note that these findings only hold for ECLIPSE.)

A.5 Run the Prediction Experiments

Finally, we check whether the length of bug reports and the experience and reputation of reporters can improve the modeling of bug report quality. We define a function *evaluate.rpart()*, which takes a formula and a data set as inputs and runs leave-one-out validation for a decision tree model. For each bug report i in *thedata*, the function builds a *model* using the function *rpart()*, which takes all bug reports except i as training set, i.e., *thedata[-i,]*. The model is then used to rate the bug report i, i.e., *thedata[i,]*. Finally, a classification table is assembled with function *table()* and returned.

```
library(rpart)

evaluate.rpart <- function(formula, thedata){
    yhat <- rep(factor(levels=levels
       (thedata$votes_mean_classes)),
       nrow(thedata))

    for (i in 1:nrow(thedata)){
      model <- rpart(as.formula(formula),
       data=thedata[-i,])
      yhat[i] <- predict(model,
       newdata=thedata[i,], type="class")
    }
    t <- table(yhat,
      thedata$votes_mean_classes)
    return(t)
}
```

Formulas in R are written in the format *lhs~rhs*, where *lhs* is the output variable, in our case, *votes_mean_classes*, and *rhs* a list of input features, typically separated by plus signs. We call function *evaluate.rpart()* with two formulas: *formula.old* is the one that was used in this paper to rate bug reports and takes the features defined in Section 7.1 as input, *formula.new* adds the variables *length*, *opened*, and *reputation* as additional input features (the *paste()* function is string concatenation).

```
formula.old <- "votes_mean_classes ~ code
  + stacktrace + att_other + att_screenshot
  + kincaid + ari + coleman_liau + fog + smog
  + flesch + lix + keywords_score
  + steps_hasitemize"
formula.new <- paste(formula.old,
  "+ length + opened + reputation")

evaluate.rpart(formula.old, data.eclipse)
evaluate.rpart(formula.new, data.eclipse)
```

The output of the evaluation is below. For decision trees and *formula.old*, 45 bug reports were classified correctly as indicated by the sum of the downward sloping diagonal; for *formula.new*, this number improves to 49.

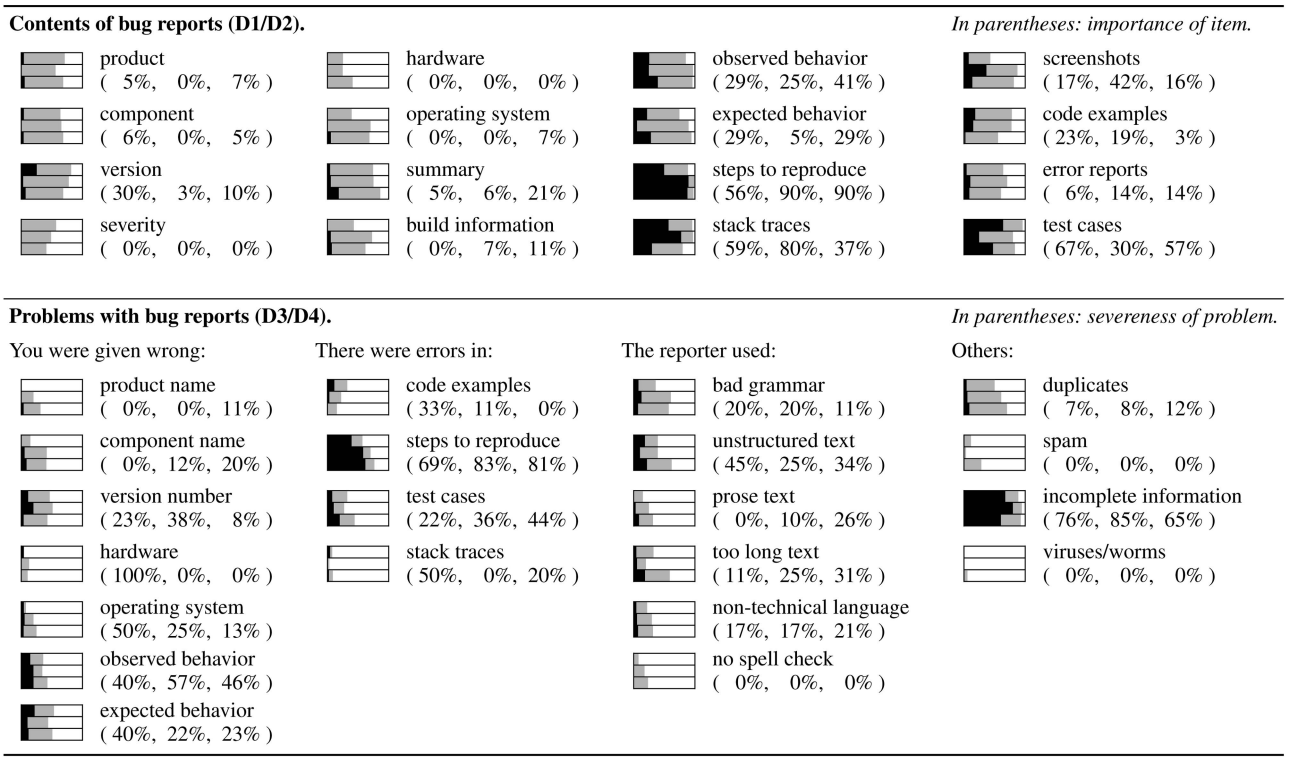
```
> evaluate.rpart (formula.old , data.eclipse)
  yhat       BAD     GOOD     NEUTRAL
    BAD       17        8           8
    GOOD       9       18          15
    NEUTRAL    7        8          10
> evaluate.rpart (formula.new , data.eclipse)
  yhat       BAD     GOOD     NEUTRAL
    BAD       16        2           9
    GOOD       4       19          10
    NEUTRAL   13       13          14
```

However, the number of correctly classified bug reports does not increase across all models when adding *length*, *opened*, and *reputation*, as the following example for support vector machines shows. Here, the number of correctly classified bug reports decreases from 42 to 35.

```
> evaluate.svm (formula.old , data.eclipse)
   yhat        BAD    GOOD     NEUTRAL
    BAD         14       5          12
    GOOD         6      18          11
    NEUTRAL     13      11          10
> evaluate.svm (formula.new , data.eclipse)
   yhat        BAD    GOOD     NEUTRAL
    BAD         16       8          16
    GOOD         7      15          13
    NEUTRAL     10      11           4
```

To run the experiments in this paper, we used slightly more complex evaluation functions because we had to test many models and data sets in different settings (within project, across project). Our functions are in file *experiments.R*

TABLE 9
Results from the Survey among Developers



The topmost bar refers to the 28 responses by APACHE developers, the middle bar to the 41 responses by ECLIPSE developers, and the lowest bar to the 61 responses by MOZILLA developers.

and we encourage you to extend our code and try your own experiments. Good luck.

APPENDIX B

SURVEY RESULTS BY PROJECT

B.1 Comparison of Responses by Developers Across Projects

In Table 9, we compare the responses to the survey from developers grouping them by their respective projects. The format of the table is similar to that of Table 2. To recap, the colored part (▬+▭) denotes the count of responses for an item in question D1 and the black part (▬) of the bar denotes the count of responses for the item in both question D1 and D2. The larger the black bar is in proportion to the gray bar, the higher is the corresponding item's importance in the developers' perspective. The importance of every item is listed in parentheses. The topmost bar refers to the 28 responses by APACHE developers, the middle bar to the 41 responses by ECLIPSE developers, and the lowest bar to the 61 responses by MOZILLA developers.

Overall, the responses between the ECLIPSE and MOZILLA project appear to be more similar to each other in comparison to APACHE. Nearly all participants from the former two projects have previously used *steps to reproduce* and *observed behavior* to resolve bugs, while relatively fewer developers from APACHE have used the same. In contrast, many APACHE developers have used *test cases*, which is not ranked very high for the other projects. Another marked dissimilarity is the usage of *screenshots*. They have been used by many ECLIPSE and MOZILLA developers, but only by a few from the APACHE project. Another example is the use of *code examples*, which is very low in MOZILLA.

There also seem to be some notable differences in the most desirable information between developers across the projects. *Steps to reproduce* are high on the list of ECLIPSE and MOZILLA developers, but few developers from APACHE consider it important. Both APACHE and MOZILLA developers favor *test cases* more than ECLIPSE developers. On the contrary, a greater number of ECLIPSE developers prefer *screenshots*, while this is true for only a few APACHE and MOZILLA developers. Other items with marked dissimilarities include *code examples* (very low in MOZILLA), *version* (high in APACHE), *summary* (preferred in MOZILLA), and *expected behavior* (low in ECLIPSE).

These comparisons reflect that there are a few differences in the preferences of developers for information that helps them resolve bugs. However, there is also consensus on many types of information that developers prefer across all projects.

In the case of problems experienced in bug reports, the experiences of developers across the three projects are more comparable except for a few differences. Similar percentages of developers from the projects have experienced *errors in observed* and *expected behavior*, *steps to reproduce*, *incomplete information*, and *issues with language in the descriptions*. Of these, *errors in steps to reproduce* and *incomplete information* are considered to be the most serious of all problems.

TABLE 10
Results from the Survey among Reporters

Contents of bug reports (R1/R2). *In parentheses: difficulty of item.*

product (0%, 0%, 0%)	hardware (0%, 0%, 1%)	observed behavior (0%, 3%, 3%)	screenshots (0%, 16%, 7%)
component (4%, 30%, 23%)	operating system (0%, 0%, 1%)	expected behavior (4%, 6%, 3%)	code examples (52%, 37%, 41%)
version (0%, 3%, 1%)	summary (4%, 0%, 4%)	steps to reproduce (35%, 57%, 52%)	error reports (0%, 0%, 2%)
severity (10%, 0%, 6%)	build information (23%, 0%, 1%)	stack traces (9%, 10%, 35%)	test cases (64%, 33%, 82%)

Contents considered to be relevant for developers (R3). *In parentheses: frequency of item in R3.*

product (0%, 14%, 7%)	hardware (0%, 0%, 0%)	observed behavior (22%, 34%, 35%)	screenshots (0%, 11%, 4%)
component (7%, 3%, 4%)	operating system (7%, 0%, 5%)	expected behavior (22%, 31%, 20%)	code examples (19%, 9%, 8%)
version (11%, 14%, 11%)	summary (4%, 0%, 7%)	steps to reproduce (59%, 91%, 78%)	error reports (0%, 3%, 12%)
severity (4%, 0%, 2%)	build information (7%, 6%, 8%)	stack traces (33%, 46%, 30%)	test cases (56%, 14%, 48%)

The topmost bar refers to the 27 responses by APACHE reporters, the middle bar to the 35 responses by ECLIPSE reporters, and the lowest bar to the 153 responses by MOZILLA reporters.

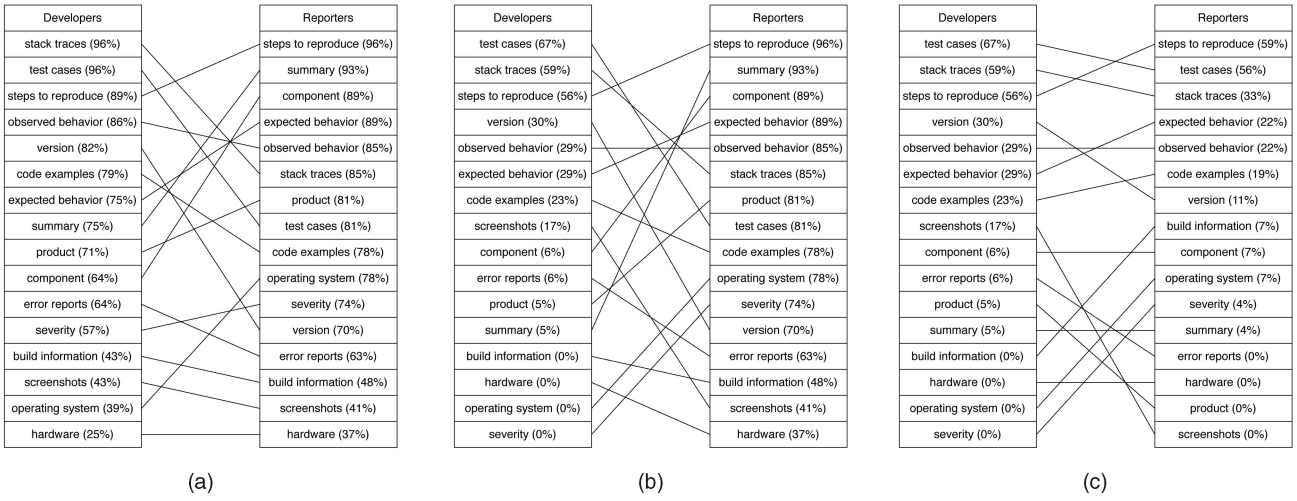


Fig. 10. Mismatch between APACHE developers and reporters. (a) Information used by developers versus provided by reporters. (b) Most helpful for developers versus provided by reporters. (c) Most helpful for developers versus reporters expected to be helpful.

B.2 Comparison of Responses by Reporters across Projects

We now compare the responses by reporters across the different projects in Table 10. The reporters' responses are relatively more uniform across the board as compared to developers. Nearly all reporters have previously submitted information such as *steps to reproduce*, *observed* and *expected behavior*, *product*, and *operating system*. But a higher proportion of reporters from the APACHE project submitted *stack traces*, *code examples*, and *test cases*. A higher proportion of APACHE reporters considered submitting the same information as difficult to provide. Marked differences are observed for *version*, which is submitted by a lower fraction of APACHE reporters as compared to the other projects, although *version* is one of the most desirable information by APACHE developers (Table 9). Most noteworthy is that a fraction of reporters from each project have never submitted a *summary* in a bug report.

Next, we compare the ranks of information items considered most helpful by reporters for the purpose of resolving bugs. Reporters across all three projects consider *steps to reproduce* to be most helpful. There also appears to be agreement on the importance of other items such as *stack traces*, *observed*, and *expected behavior*. *Test cases* are rated high by both APACHE and MOZILLA developers, but not many ECLIPSE reporters rate it as important. But ECLIPSE reporters stand out by placing *product* relatively high on the list; the same is true for *error reports* in the case of MOZILLA reporters.

In general, similarly to the developers, there is a general consensus between reporters on which items are most important to fix. Unfortunately, as shown earlier in our paper, although there is an overall agreement between developers and reporters on which information is important,

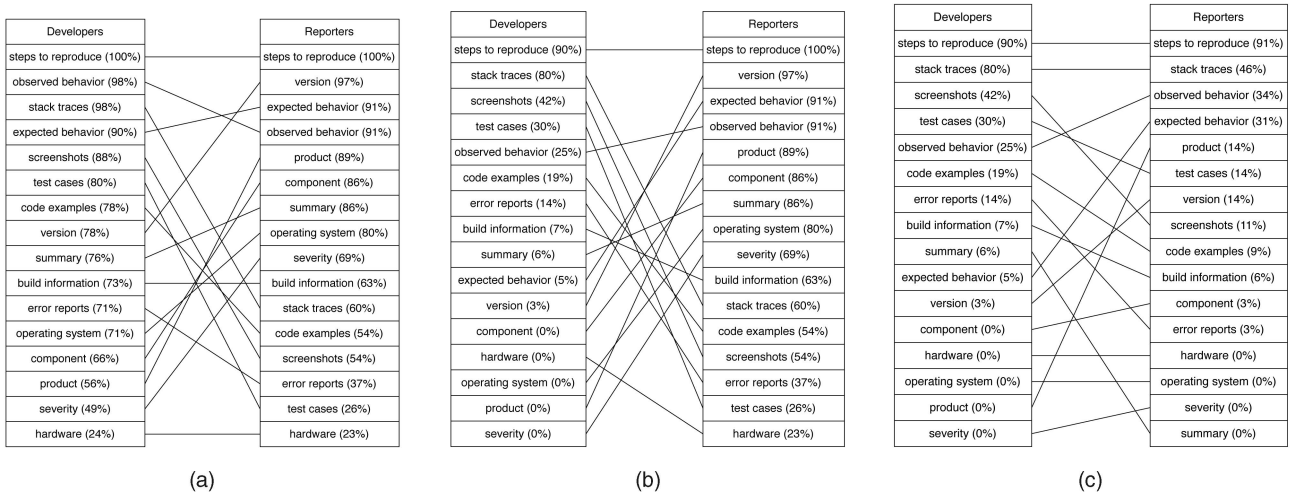


Fig. 11. Mismatch between ECLIPSE developers and reporters. (a) Information used by developers versus provided by reporters. (b) Most helpful for developers versus provided by reporters. (c) Most helpful for developers versus reporters expected to be helpful.

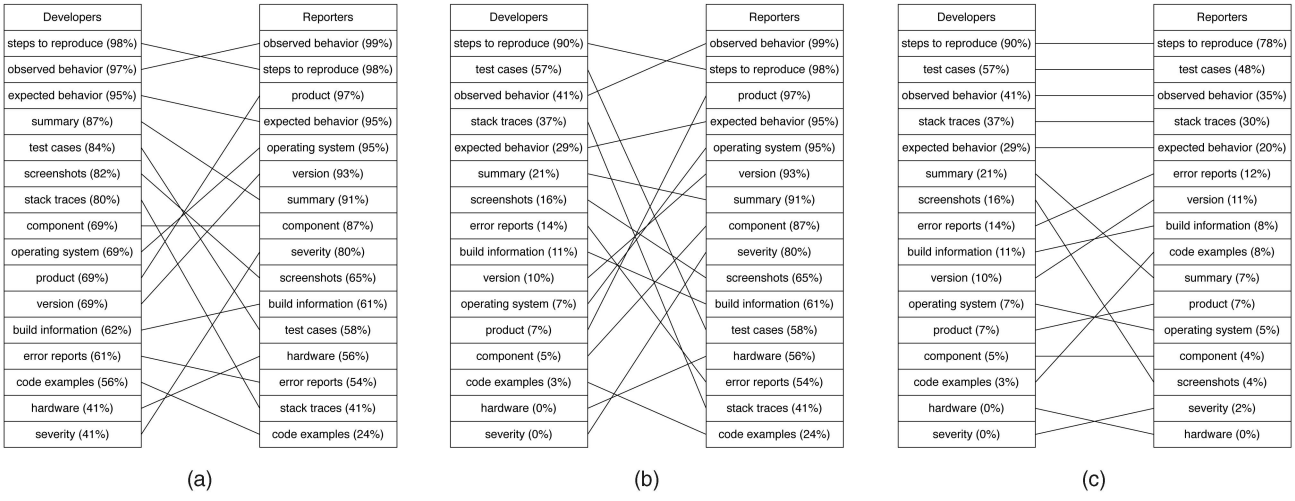


Fig. 12. Mismatch between MOZILLA developers and reporters. (a) Information used by developers versus provided by reporters. (b) Most helpful for developers versus provided by reporters. (c) Most helpful for developers versus reporters expected to be helpful.

reporters do not provide the information as often. Thus, it is important to address this gap by aiding reporters in providing information that may help in the quick resolution of bugs.

B.3 Mismatch between Developers and Reporters by Project

Previously, in Section 3.3, we showed that reporters have the knowledge of which information is most useful to developers, but they still do not provide this information in the bug reports. These findings were arrived at by combining the data across all three projects.

We now present results by performing the same comparison on project-specific data. The results from the APACHE, ECLIPSE, and MOZILLA projects are presented in Figs. 10, 11, and 12, respectively. These plots are in the same format as Fig. 2.

The mismatch in the project-specific responses appears to be very similar to the mismatch observed earlier for all projects combined in Fig. 2. In each case, we observe that reporters have some knowledge about which information developers most desire. In fact, in the case of MOZILLA, reporters have very good knowledge about which information is most helpful to the developer. However, for all three projects, we observe marked differences between what information is most commonly considered important and what information is most commonly reported.

ACKNOWLEDGMENTS

The authors thank Avi Bernstein, Harald Gall, Christian Lindig, Stephan Neuhaus, Andreas Zeller, and the FSE and *TSE* reviewers for valuable and helpful suggestions on earlier revisions of this paper. A special thanks to everyone who responded to their survey. When this research was carried out, all authors were with Saarland University; Thomas Zimmermann was funded by the DFG Research Training Group "Performance Guarantees for Computer Systems."

REFERENCES

[1] P. Anbalagan and M. Vouk, "On Predicting the Time Taken to Correct Bugs in Open Source Projects (Short Paper)," *Proc. 25th IEEE Int'l Conf. Software Maintenance*, Sept. 2009.

[2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is It a Bug or An Enhancement?: A Text-Based Approach to Classify Change Requests," *Proc. 2008 Conf. Center for Advanced Studies on Collaborative Research,* pp. 304-318, 2008.

[3] G. Antoniol, H. Gall, M.D. Penta, and M. Pinzger, "Mozilla: Closing the Circle," Technical Report TUV-1841-2004-05, Technical Univ. of Vienna, 2004.

[4] J. Anvik, L. Hiew, and G.C. Murphy, "Who Should Fix This Bug?" *Proc. 28th Int'l Conf. Software Eng.,* pp. 361-370, 2006.

[5] J. Anvik and G. Murphy, "Reducing the Effort of Bug Report Triage: Recommenders for Development-Oriented Decisions," *ACM Trans. Software Eng. and Methodology,* to appear.

[6] J. Aranda and G. Venolia, "The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories," *Proc. 31st Int'l Conf. Software Eng.,* 2009.

[7] S. Artzi, S. Kim, and M.D. Ernst, "Recrash: Making Software Failures Reproducible by Preserving Object States," *Proc. 22nd European Object-Oriented Programming Conf.,* pp. 542-565, 2008.

[8] B. Ashok, J. Joy, H. Liang, S.K. Rajamani, G. Srinivasa, and V. Vangala, "DebugAdvisor: A Recommender System for Debugging," *Proc. Seventh Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng. on European Software Eng. Conf. and Foundations of Software Eng. Symp.,* pp. 373-382, 2009.

[9] I. Barker, "What Is Information Architecture? KM Column," http://www.steptwo.com.au, 2010.

[10] V.R. Basili, F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Trans. Software Eng.,* vol. 25, no. 4, pp. 456-473, July 1999.

[11] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, Collaboration, and Bugs: The Social Nature of Issue Tracking in Small, Collocated Teams," *Proc. 2010 ACM Conf. Computer Supported Cooperative Work,* pp. 291-300, 2010.

[12] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "Quality of Bug Reports in Eclipse," *Proc. 2007 OOPSLA Workshop Eclipse Technology eXchange,* pp. 21-25, Oct. 2007.

[13] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What Makes a Good Bug Report?" *Proc. 16th Int'l Symp. Foundations of Software Eng.,* Nov. 2008.

[14] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate Bug Reports Considered Harmful... Really?" *Proc. 24th IEEE Int'l Conf. Software Maintenance,* pp. 337-345, Sept. 2008.

[15] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting Structural Information from Bug Reports," *Proc. Fifth Working Conf. Mining Software Repositories,* 2008.

[16] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting Structural Information from Bug Reports," *Proc. Fifth Int'l Working Conf. Mining Software Repositories,* May 2008.

[17] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced? Bias in Bug-Fix Data Sets," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* 2009.

[18] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information Needs in Bug Reports: Improving Cooperation between Developers and Users," *Proc. 2010 ACM Conf. Computer Supported Cooperative Work,* pp. 301-310, 2010.

[19] G. Canfora and L. Cerulo, "Fine Grained Indexing of Software Repositories to Support Impact Analysis," *Proc. Int'l Workshop Mining Software Repositories,* pp. 105-111, 2006.

[20] G. Canfora and L. Cerulo, "Supporting Change Request Assignment in Open Source Development," *Proc. 2006 ACM Symp. Applied Computing,* pp. 1767-1772, 2006.

[21] L. Cherry and W. Vesterman, "Writing Tools—The STYLE and DICTION Programs," technical report, AT&T Laboratories, 1980.

[22] D. Cubranic and G.C. Murphy, "Automatic Bug Triage Using Text Categorization," *Proc. 16th Int'l Conf. Software Eng. and Knowledge Eng.,* pp. 92-97, 2004.

[23] W. Cunningham, "Best of Bugzilla," http://eclipse-projects.blogspot.com/2005/12/best-of-bugzilla.html, 2005.

[24] B. Dit and A. Marcus, "Improving the Readability of Defect Reports," *Proc. 2008 Int'l Workshop Recommendation Systems for Software Eng.,* pp. 47-49, 2008.

[25] Eclipse Foundation, Callisto Simultaneous Release Bug Finding Contest, http://www.eclipse.org/projects/callisto-files/callisto-bug-contest.php, 2010.

[26] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and Relating Bug Report Data for Feature Tracking," *Proc. 10th Working Conf. Reverse Eng.,* pp. 90-101, 2003.

[27] E. Goldberg, "Bug Writing Guidelines," https://bugs.eclipse.org/bugs/bugwritinghelp.html, 2010.

[28] P.J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows," *Proc. 32nd Int'l Conf. Software Eng.,* May 2010.

[29] L. Hiew, "Assisted Detection of Duplicate Bug Reports," master's thesis, The Univ. of British Columbia, 2006.

[30] P. Hooimeijer and W. Weimer, "Modeling Bug Report Quality," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 34-43, 2007.

[31] HOT or NOT, http://www.hotornot.com/, 2010.

[32] N. Jalbert and W. Weimer, "Automated Duplicate Detection for Bug Tracking Systems," *Proc. Conf. Dependable Systems and Networks,* pp. 52-61, June 2008.

[33] G. Jeong, S. Kim, and T. Zimmermann, "Improving Bug Triage with Bug Tossing Graphs," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* 2009.

[34] S. Joshi and A. Orso, "SCARPE: A Technique and Tool for Selective Record and Replay of Program Executions," *Proc. 23rd IEEE Int'l Conf. Software Maintenance,* Oct. 2007.

[35] S. Just, R. Premraj, and T. Zimmermann, "Towards the Next Generation of Bug Tracking Systems," *Proc. 2008 IEEE Symp. Visual Languages and Human-Centric Computing,* pp. 82-85, Sept. 2008.

[36] M. Kersten and G.C. Murphy, "Using Task Context to Improve Programmer Productivity," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 1-11, 2006.

[37] S. Kim, MSR Mining Challenge 2008, http://msr.uwaterloo.ca/msr2008/challenge/, 2008.

[38] J.P. Kincaid, R.P. Fishburne, Jr., R.L. Rogers, and B.S. Chissom, "Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel," technical report, Research Branch Report 8-75, Naval Technical Training, US Naval Air Station, 1975.

[39] A.J. Ko and P.K. Chilana, "How Power Users Help and Hinder Open Bug Reporting," *Proc. ACM Conf. Human Factors in Computing Systems,* 2010.

[40] A.J. Ko, B.A. Myers, and D.H. Chau, "A Linguistic Analysis of How People Describe Software Problems," *Proc. 2006 IEEE Symp. Visual Languages and Human-Centric Computing,* pp. 127-134, 2006.

[41] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan, "Scalable Statistical Bug Isolation," *Proc. 2005 ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 15-26, 2005.

[42] R. Likert, "A Technique for the Measurement of Attitudes," *Archives of Psychology,* vol. 140, pp. 1-55, 1932.

[43] D. MacKenzie, P. Eggert, and R. Stallman, *Comparing and Merging Files with GNU Diff and Patch.* Network Theory, Ltd., 2003.

[44] T. Menzies and A. Marcus, "Automated Severity Assessment of Software Defect Reports," *Proc. 24th IEEE Int'l Conf. Software Maintenance,* pp. 346-355, Sept. 2008.

[45] L. Moonen, "Generating Robust Parsers Using Island Grammars," *Proc. Eighth Working Conf. Reverse Eng.,* pp. 13-22, 2001.

[46] Mozilla.org. Mozilla Security Bug Bounty Program, http://www.mozilla.org/security/bug-bounty.html, 2010.

[47] A. Orso, S. Joshi, M. Burger, and A. Zeller, "Isolating Relevant Component Interactions with JINSI," *Proc. Fifth Int'l Workshop Dynamic Analysis,* May 2006.

[48] A. Page, "Duplicate Bugs," http://blogs.msdn.com/alanpa/archive/2007/08/01/duplicate-bugs.aspx, Apr. 2008.

[49] A. Page, K. Johnston, and B. Rollison, *How We Test Software at Microsoft.* Microsoft Press, 2008.

[50] L.D. Panjer, "Predicting Eclipse Bug Lifetimes," *Proc. Fourth Int'l Workshop Mining Software Repositories,* 2007.

[51] U. Passing and M.J. Shepperd, "An Experiment on Software Project Size and Effort Estimation," *Proc. Int'l Symp. Empirical Software Eng.,* pp. 120-131, 2003.

[52] T. Punter, M. Ciolkowski, B. Freimut, and I. John, "Conducting Online Surveys in Software Engineering," *Proc. Int'l Symp. Empirical Software Eng.,* pp. 80-88, 2003.

[53] Ratemyface.com., http://www.ratemyface.com/, 2010.

[54] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," *Proc. 29th Int'l Conf. Software Eng.*, pp. 499-510, 2007.
[55] A. Schröter, N. Bettenburg, and R. Premraj, "Do Stacktraces Help Developers Fix Bugs?" *Proc. 2010 Int'l Working Conf. Mining Software Repositories*, 2010.
[56] S. Siegel and N.J. Castellan Jr., *Nonparametric Statistics for the Behavioral Sciences*, second ed. McGraw-Hill, 1988.
[57] J. Singer and N.G. Vinson, "Ethical Issues in Empirical Studies of Software Engineering," *IEEE Trans. Software Eng.*, vol. 28, no. 12, pp. 1171-1180, Dec. 2002.
[58] J. Spolsky, *Joel on Software*. APress, 2004.
[59] D.W. van Liere, "How Shallow Is a Bug? Open Source Communities as Information Repositories and Solving Software Defects," *SSRN eLibrary*, http://ssrn.com/paper=1507233, 2009.
[60] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information," *Proc. 30th Int'l Conf. Software Eng.*, May 2008.
[61] L. Wasserman, *All of Statistics: A Concise Course in Statistical Inference*, second ed. Springer, 2004.
[62] W. Weimer, "Patches as Better Bug Reports," *Proc. Fifth Int'l Conf. Generative Programming and Component Eng.*, pp. 181-190, 2006.
[63] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How Long Will It Take to Fix This Bug?" *Proc. Fourth Int'l Workshop Mining Software Repositories*, 2007.
[64] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, second ed. Morgan Kaufmann, 2005.
[65] G. Xu, A. Rountev, Y. Tang, and F. Qin, "Efficient Checkpointing of Java Software Using Context-Sensitive Capture and Replay," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 85-94, 2007.
[66] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving Bug Tracking Systems," *Proc. Companion 31st Int'l Conf. Software Eng.*, pp. 247-250, 2009.

Thomas Zimmermann received the Diploma degree in computer science from the University of Passau in 2004, and the PhD degree from Saarland University, Germany. He is a researcher in the Software Reliability Research Group at Microsoft Research, and an adjunct assistant professor in the Department of Computer Science at the University of Calgary. His research interests include empirical software engineering, mining software repositories, software reliability, and development tools. He is a member of the ACM, the IEEE, and the IEEE Computer Society.

Rahul Premraj received the master's degree in information systems in 2002 from Robert Gordon University, and the PhD degree from Bournemouth University in 2007. He is an assistant professor in the Computer Science Department at VU University Amsterdam. His research interests include empirical software engineering, mining software archives, software quality assurance, distributed software development, and software process improvement.

Nicolas Bettenburg received the BS and MS degrees in computer science from Saarland University in 2006 and 2008, respectively. He is currently working toward the PhD degree in computer science at Queen's University under Ahmed E. Hassan. His research interests are in mining unstructured information from software repositories with a focus on enriching source code with information from developer communication. He is a member of the IEEE.

Sascha Just received the BS degree in computer science in 2009 from Saarland University, Saarbrücken, Germany, where he is currently working toward the MS degree. His research interests are in software mining, software engineering issues, development tools and interfaces, and software security. He is a member of the IEEE and the ACM.

Adrian Schröter received the MSc degree in computer science from Saarland University under the supervision of Dr. Andres Zeller and Dr. Thomas Zimmermann. He is currently working toward the PhD degree in the Computer Science Department, University of Victoria. His research interests cover software development in distributed teams, software quality, and mining software repositories. He is a member of the IEEE.

Cathrin Weiss received the BS and MS degrees in computer science from Saarland University, Germany, in 2007. Since November 2007, she has been working toward the PhD degree at the University of Zurich, Switzerland. In November 2009, she joined the Condor Project at the University of Wisconsin-Madison as a systems programmer. Her interests include software engineering, database technology for graph-based data, and distributed computing.