

What Makes a Good Bug Report?

Nicolas Bettenburg*
nicbet@st.cs.uni-sb.de

Cathrin Weiss‡
weiss@ifi.uzh.ch

* Saarland University, Germany

‡ University of Victoria, BC, Canada

Sascha Just*
just@st.cs.uni-sb.de

Rahul Premraj*§
premrj@cs.uni-sb.de

‡ University of Zurich, Switzerland

+ University of Calgary, Alberta, Canada

Adrian Schröter¶
schadr@uvic.ca

Thomas Zimmermann+§
tz@acm.org

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 to design 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. In our experiments, CUEZILLA was able to predict the quality of 31–48% of bug reports accurately.

Categories and Subject Descriptors:

D.2.5 [Software Engineering]: Testing and Debugging; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms: 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 occasionally hint at the location of the fault in the code (in 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

§Contact authors are Rahul Premraj and Thomas Zimmermann.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-59593-995-1 ...\$5.00.

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 and 3).
2. *Rate the quality of bug reports* from very poor to very good on a five-point Likert scale [22]. We received a total of 1,186 votes for 289 randomly selected bug reports (Section 4).

In addition, we asked 1,354 reporters¹ 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**. To enable swift fixing of bugs, this mismatch should be bridged, for example with tool support for reporters to furnish information that developers want. We developed a prototype tool called CUEZILLA (see Figure 1), 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 5).
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 6).

¹Throughout 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.



Figure 1: Mockup of CUEZILLA’s user interface. It recommends improvements to the report (left image). To encourage the user to follow the advice, CUEZILLA provides facts that are mined from history (right image).

Table 1: Number of invitations sent to and responses by developers and reporters of the APACHE, ECLIPSE, and MOZILLA projects.

Project	Developers					Reporters				
	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. the CUEZILLA tool that measures the quality of bug reports and suggests how reporters could enhance their reports, so that their problems get fixed sooner.

We conclude this paper with threats to validity (Section 7), related work (Section 8), and future research directions (Section 9).

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 five minutes, which we also advertised in the invitation email (“we would much appreciate five 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 email, the number of bounces, and the number of responses and comments received. The response rate was highest for MOZILLA reporters at 32.7%. Our overall response rate of 23.2% is comparable to other Internet surveys in software engineering, which range from 14% to 20% [28].

2.2 The Questionnaire

Keeping the five minute rule in mind, we asked developers the following questions, which we grouped into three parts (see Figure 2):

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 sixteen items selected on the basis of Eli Goldberg’s bug writing guidelines [13]; or being standard fields in the BUGZILLA database.

Developers were free to check as many items 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 twenty-one problems that developers could select. Again, they were free to check as many items for the first question (D3), but at most three for the second question (D4).

For the reporters of bugs, we asked the following questions (again see Figure 2):

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

We listed the same sixteen items to reporters, which we have 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), 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 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).

²Did you know? In ECLIPSE, 205 bug reports were submitted for “Windows” but later re-assigned 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?			
	<input type="checkbox"/> product	<input type="checkbox"/> hardware	<input type="checkbox"/> observed behavior	<input type="checkbox"/> screenshots
	<input type="checkbox"/> component	<input type="checkbox"/> operating system	<input type="checkbox"/> expected behavior	<input type="checkbox"/> code examples
	<input type="checkbox"/> version	<input type="checkbox"/> summary	<input type="checkbox"/> steps to reproduce	<input type="checkbox"/> error reports
	<input type="checkbox"/> severity	<input type="checkbox"/> build information	<input type="checkbox"/> stack traces	<input type="checkbox"/> 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:
	<input type="checkbox"/> product name	<input type="checkbox"/> code examples	<input type="checkbox"/> bad grammar	<input type="checkbox"/> duplicates
	<input type="checkbox"/> component name	<input type="checkbox"/> steps to reproduce	<input type="checkbox"/> unstructured text	<input type="checkbox"/> spam
	<input type="checkbox"/> version number	<input type="checkbox"/> test cases	<input type="checkbox"/> prose text	<input type="checkbox"/> incomplete information
	<input type="checkbox"/> hardware	<input type="checkbox"/> stack traces	<input type="checkbox"/> too long text	<input type="checkbox"/> viruses/worms
	<input type="checkbox"/> operating system		<input type="checkbox"/> non-technical language	
	<input type="checkbox"/> observed behavior		<input type="checkbox"/> no spell check	
	<input type="checkbox"/> expected behavior			
Comments.	D5/R4: Please feel free to share any interesting thoughts or experiences.			

Figure 2: The questionnaire presented to APACHE, ECLIPSE, and MOZILLA developers (Dx) and reporters (Rx).

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 vs. 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 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%) failed the consistency check and were removed from our analysis. For reporters we received 310 and had to remove 95 inconsistent responses (30.6%). 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):

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

	All consistent responses for the project
	Number of times that <i>item</i> was selected in D1
	Number of times that <i>item</i> was selected in D1 and D2
	Number of times that <i>item</i> 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 denotes the count of responses for the item in both question D1 and D2. The larger the black bar is in proportion to the grey bar, the higher is the corresponding item’s importance 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 our technical report [5].

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 narrowing 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 relative 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.”

Table 2: Results from the survey among developers. (130 consistent responses by APACHE, ECLIPSE, and MOZILLA developers.)

Contents of bug reports (D1/D2).				<i>In parentheses: importance of item.</i>			
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).				<i>In parentheses: severeness of problem.</i>			
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%)	observed behavior (48%)	non-technical language (19%)	no spell check (0%)	expected behavior (27%)			

Table 3: Results from the survey among reporters. (215 consistent responses by APACHE, ECLIPSE, and MOZILLA reporters.)

Contents of bug reports (R1/R2).				<i>In parentheses: difficulty of item.</i>			
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).				<i>In parentheses: frequency of item in R3.</i>			
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%)

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 few users added *stack traces*, *code examples*, and *test cases* to their bug reports. An explanation might be the **difficulty to provide 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 being 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 helpful, but only few provide them because they are most difficult to provide. This suggests that capture/replay tools which record test cases [16, 25, 38] 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 Figure 3(a), 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. Figure 3(a) 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 Figure 3(b), 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, Figure 3(c) 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

⁴Spearman 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 [35].

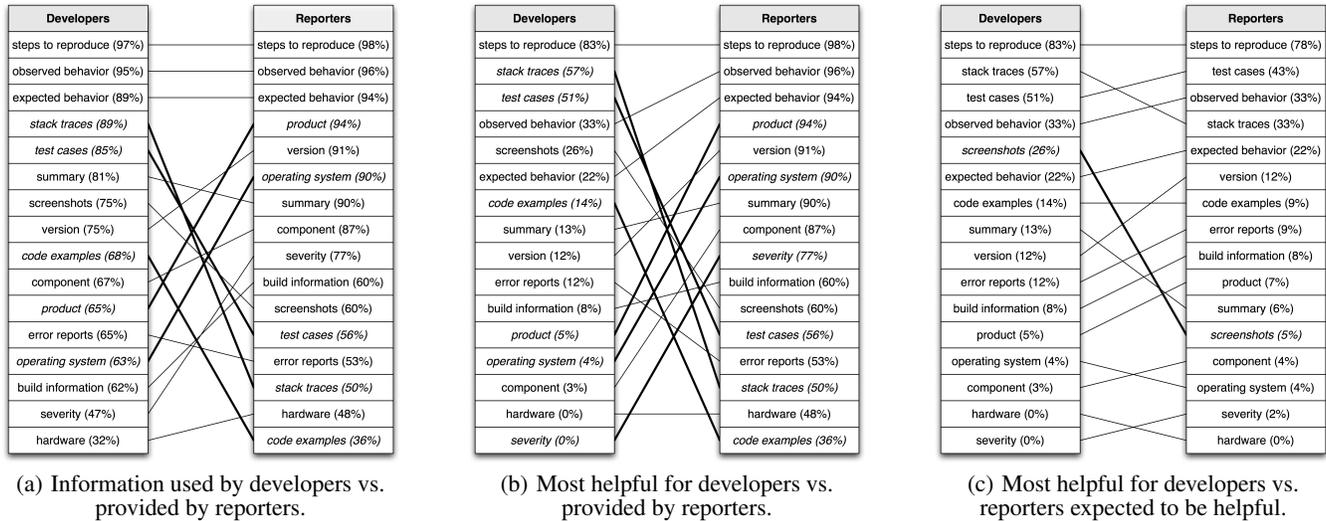


Figure 3: Mismatch between developers and reporters.

developers; the right column now shows what reporters expect to be most relevant (question R3). Overall there is a 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.

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 problem [11, 30, 34]). Possibly, developers can easily recognize *duplicates*, and sometimes even benefit by a different bug description. As commented by one developer:

“Duplicates are not really problems. They often add useful information. That this information were filed under a new report is not ideal thought.”

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. Lastly, 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.

3.5 Developer Comments

We received 48 developer comments in the survey responses. Most comments stressed the importance of clear, complete, and correct bug descriptions. However, some revealed additional problems:

Different knowledge levels. *“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 weird behavior that is difficult to reproduce.”*

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.”*

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.”* Another one: *“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.”*

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.”*

Also, some developers pointed out situations where bug reports get preferred treatment:

Human component. *“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.”*

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



Figure 4: Screenshot of interface for rating bug reports

Bug severity. “For me it amounts to a consideration of ‘how serious is this?’ vs ‘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.”

4. 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 email. 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.

4.1 Rating Infrastructure

The rating system was inspired by Internet sites such as RateMyFace [29] and HotOrNot [15]. 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 Figure 4 for a screenshot). 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 the current report and navigates to the next one. This feature seemed preferable to guesswork on 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 5).

4.2 Rating Results

The following number of developer votes for bug reports were received for the samples of 100 bugs from each project: 229 for APACHE, 397 for ECLIPSE, and 560 for MOZILLA. Figure 5 plots the distribution of the ratings, which is similar across all projects, with the most frequent ratings being 3 (average) and 4 (good).

Table 4 lists the bug reports that were rated highest and lowest by ECLIPSE developers. Some bug reports were found to be of

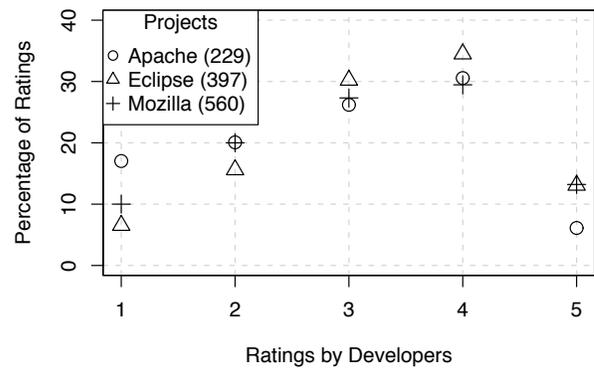


Figure 5: Distribution of ratings by developers

Table 4: 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

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 want 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 want to create a new plugin (user defined plugin) using CDT. After that I want to impliment it in my programe. If it possible?. Any one can help me please...

(ECLIPSE bug report #175222)

4.3 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 this 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%) 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.

5. MEASURING BUG REPORT QUALITY

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 Shepherd [27]. They examined how subjects revised their 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 towards assistance, we developed a prototype tool — CUEZILLA that measures the quality of bug reports; and provides suggestions to reporters on how to enhance the quality. For example, “*Have you thought about adding a screenshot to your bug report?*”

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 5.1). Then it uses supervised learning to train models (Section 5.2) that measure the quality of bug reports (Section 5.3 and 5.4). Our models can also quantify the increase in quality, when elements are added to bug reports (Section 5.5). In contrast to other quality measures for bug reports such as lifetime [14], we use the ratings that we received by developers.

5.1 Input Features

Our CUEZILLA tool measures 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 parenthesis or brackets or followed by a single punctuation character.

Keyword completeness. We reused the data set provided by Andy Ko et al. [20] 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% of bug reports. Next we categorized the words into the following groups:

- action items (e.g., open, select, click)
- expected and observed behavior (e.g., error, missing)
- steps to reproduce (e.g., steps, repro)
- build-related (e.g., build)
- user interface elements (e.g., toolbar, menu, 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 [24]. 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) [23].

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 [7], in which we showed that we can identify the above features with a 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 “analyses the surface characteristics of the writing style of a document” [10]. 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 [19].

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.⁵

⁵This paper has a 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.

Table 5: The results of the classification by CUEZILLA (using stepwise linear regression) compared to the developer rating.

Measured		Observed				
		very poor	poor	medium	good	very good
very poor	[< 1.8]	0	0	1	0	0
poor	[1.8, 2.6]	0	2	0	0	0
medium	[2.6, 3.4]	4	11	29	17	4
good	[3.4, 4.2]	0	1	6	12	5
very good	[> 4.2]	0	1	2	4	1

Table 6: Leave-one-out cross-validation within projects.

	APACHE	ECLIPSE	MOZILLA
Support vector machine	28% (82%)	48% (91%)	37% (82%)
Generalized linear regression	28% (82%)	40% (87%)	29% (80%)
Stepwise linear regression	31% (86%)	44% (87%)	34% (85%)

5.2 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 three models: support vector machines (SVM), generalized linear regression (GLR), and stepwise linear regression [35].

Each model used the scores from the features described in Section 5.1 as input variables and predicted the average developer rating as output variable. We evaluated CUEZILLA using two setups:

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

Across projects. We also tested if 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 5 shows the results for ECLIPSE bug reports and stepwise linear regression using leave-one-out cross-validation. The column names in the table indicate the average rating of the bug report by developers (Observed); the row names denote the quality measured by CUEZILLA (Measured). The ranges within the square brackets next to the row names indicate the equidistant mapping of predicted values to the Likert scale.

The counts in the diagonal cells, with a dark gray background, indicate the number of bug reports for which there was complete agreement between CUEZILLA and developers on their quality. In Table 5, this is true for 44% of the bug reports. In addition, we also look at predictions that are off by one from the developer ratings. These are the cells in the tables that are one row, either to the left or right of the diagonal. Using perfect and off-by-one agreements, the accuracy increases to 87%, in other words only 1 in 10 recommendations off by more than one scale.

5.3 Evaluation within Projects

Results from predicting the quality of bug reports using other bug reports from the same project (with leave-one-out cross-validation) are presented in Table 6. The first number is the percentage of bug reports with perfect agreement on the quality between CUEZILLA and the developers, while the number in the parentheses indicates the percentage for off-by-one accuracy.

Table 7: Validation across projects.

		Testing on			
		APACHE	ECLIPSE	MOZILLA	
Training	APACHE				SVM GLR Stepwise
	ECLIPSE				SVM GLR Stepwise
	MOZILLA				SVM GLR Stepwise

Of the three models used, support vector machines appear to provide more number of perfect agreements than other techniques. In case of off-by-one agreements, stepwise linear regression outperforms the two other models. But on the whole, all three models seem to perform comparably across the projects. The figures also show that a higher proportion of perfect agreements were made for ECLIPSE bug reports than for APACHE and MOZILLA.

5.4 Evaluation across Projects

In Table 7, we present the results from the across projects evaluation setup. The bars in the table can be interpreted in a similar fashion as before in Tables 2 and 3. Here, the bars have the following meanings.

	Number of unique bugs rated by developers
	Number of perfect agreements
	Number of off-by-one agreements

The accuracy of CUEZILLA is represented by the black bar (■) and the off-by-one accuracy by the overall shaded part (▒). In order to facilitate comparison, Table 7 also contains the results from the within project evaluation (for which the bars have a thinner border).

The results in Table 7 show that models trained from one project can be transferred to other projects without much loss in predictive power. However, we can observe more variability in prediction accuracy for stepwise and generalized linear regression. It is interesting to note that models using data from APACHE and MOZILLA are both good at predicting quality of ECLIPSE 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.

5.5 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 on 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 to add certain types of information; likely to improve bug report quality.

The left panel of Figure 1 illustrates the concept. The text in the panel is determined by investigating the current contents of the report, and then determining that would be best, for instance, adding 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 enhance the tool and conduct experiments to show its usefulness. We briefly discuss our plans in Section 9.

6. INCENTIVE FOR REPORTERS

If CUEZILLA tips reporters on how to enhance 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 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 5.1 for each of the 150,000 bugs. To recall, the scores for some of the items are continuous values, while others are binary.

6.1 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—that 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 above, duplicates may provide more information about the bug to developers.

For binary valued features, we performed Chi-Square tests [31] ($p < 0.05$) 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 case of features with continuous valued scores, we performed a Kruskal-Wallis test [31] ($p < 0.05$) on the distribution of scores across the three resolution groups to check whether the distribution significantly differ 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.

6.2 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 [31] ($p < 0.05$) 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 vs. bugs resolved slowly. We then compared the distribution of the scores across the two categories using the Kruskal-Wallis test [31] ($p < 0.05$) 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.

6.3 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 the right part of Figure 1). A sample of our key findings are 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 [14] 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 [26].

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 to either get fixed at all, or get 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.

7. 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 towards 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*: the participation in the survey was voluntary. As a consequence, results might be skewed towards people that 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” [32].

In order to take as little time as possible of participants, 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 con-

clusions because any process depends on a *large number of context variables* [3]. 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 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 others is challenged (“bug duplicates considered harmful”).

8. RELATED WORK

So far, mostly anecdotal evidence has been reported on what makes a good bug report. For instance, Joel Spolsky described how to achieve painless bug tracking [33] and numerous articles and guidelines on effective bug reporting float around the Internet (e.g., [13]). 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 [14]. 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 (1,186 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 [14] 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 [14,26], can encourage users to submit better reports as discussed in Section 6.

In a workshop paper, we presented preliminary results from the developer survey on the ECLIPSE project using a handcrafted prediction model [4]. In other work, we quantified the amount of additional information in bug duplicates [6] and gave recommendations on how to improve existing bug tracking systems [17].

Several studies used bug reports to automatically assign developers to bug reports [2,9], assign locations to bug reports [8], track features over time [12], recognize bug duplicates [11,30], and predict effort for bug reports [37]. All these approaches should benefit by our quality measure for bug reports since training only with high-quality bug reports will likely improve their predictions.

In 2004, Antoniol et al. [1] 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 [18] allows to attach a task context to bug reports so that changes can be tracked on a very fine-grained level.

In order to inform the design of new bug reporting tools, Ko et al. [20] 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 user-side and replay it on developer-side. While there exist several capture/replay techniques (such as [16,25,38]), their user-orientation and scalability can still be improved.

Not all bug reports are generated by humans. Some bug-finding tools can report violations of safety-policies and annotate them with back-traces 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” [36].

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 sent back crash information, e.g., Apple CrashReporter, Windows Error Reporting, Gnome Bug-Buddy, Mozilla Talkback. Liblit et al. introduced statistical debugging [21]. 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 non-crashing bugs, there will always be the need for manual bug reporting.

9. 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 tool, CUEZILLA that measures the quality of bug reports. This tool can rate up to 41% bug reports in complete agreement with developers. 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.

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.

Impact on other research. In Section 8, 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?

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. To learn more about our work in mining software archives, visit

<http://www.softevo.org/>

Acknowledgments. Many thanks to Avi Bernstein, Harald Gall, Christian Lindig, Stephan Neuhaus, Andreas Zeller, and the FSE reviewers for valuable and helpful suggestions on earlier revisions of this paper. A special thanks to everyone who responded to our 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”.

10. REFERENCES

- [1] G. Antoniol, H. Gall, M. D. Penta, and M. Pinzger. Mozilla: Closing the circle. Technical Report TUV-1841-2004-05, Technical University of Vienna, 2004.
- [2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.
- [3] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Trans. Software Eng.*, 25(4):456–473, 1999.
- [4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. Quality of bug reports in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 21–25, October 2007.
- [5] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? Version 1.1. Technical report, Saarland University, Software Engineering Chair, March 2008. The technical report is an extended version of this paper. <http://www.st.cs.uni-sb.de/publications/details/bettenburg-tr-2008/>.
- [6] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *ICSM '08: Proceedings of the 24th IEEE International Conference on Software Maintenance*, September 2008. To appear.
- [7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of the Fifth International Working Conference on Mining Software Repositories*, May 2008.
- [8] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *MSR '06: Proceedings of the International Workshop on Mining Software Repositories*, pages 105–111, 2006.
- [9] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1767–1772, 2006.
- [10] L. Cherry and W. Vesterman. Writing tools - the STYLE and DICTION programs. Technical report, AT&T Laboratories, 1980.
- [11] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.
- [12] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 90–101, 2003.
- [13] E. Goldberg. Bug writing guidelines. <https://bugs.eclipse.org/bugs/bugwritinghelp.html>. Last accessed 2007-08-04.
- [14] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 34–43, 2007.
- [15] HOT or NOT. <http://www.hotornot.com/>. Last accessed 2007-09-11.
- [16] S. Joshi and A. Orso. SCARPE: A Technique and Tool for Selective Record and Replay of Program Executions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, Paris, France, October 2007.
- [17] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *VL/HCC '08: Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, September 2008. To appear.
- [18] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*, pages 1–11, 2006.
- [19] 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, Millington, TN: Naval Technical Training, U. S. Naval Air Station, Memphis, TN, 1975.
- [20] A. J. Ko, B. A. Myers, and D. H. Chau. A linguistic analysis of how people describe software problems. In *Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006)*, pages 127–134, 2006.
- [21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, 2005.
- [22] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:1–55, 1932.
- [23] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with GNU Diff and Patch*. Network Theory Ltd., 2003.
- [24] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*, pages 13–22, 2001.
- [25] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with JINSI. In *Proc. of Fifth International Workshop on Dynamic Analysis (WODA 2007)*, May 2006.
- [26] L. D. Panjer. Predicting Eclipse bug lifetimes. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007. MSR Challenge Contribution.
- [27] U. Passing and M. J. Shepperd. An experiment on software project size and effort estimation. In *Proc. of International Symposium on Empirical Software Engineering (ISESE '03)*, pages 120–131, 2003.
- [28] T. Punter, M. Ciolkowski, B. Freimut, and I. John. Conducting on-line surveys in software engineering. In *Proc. of International Symposium on Empirical Software Engineering (ISESE '03)*, pages 80–88, 2003.
- [29] Ratemylface.com. <http://www.ratemylface.com/>. Last accessed 2007-09-11.
- [30] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 499–510, 2007.
- [31] S. Siegel and N. J. Castellan, Jr. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill, second edition, 1988.
- [32] J. Singer and N. G. Vinson. Ethical issues in empirical studies of software engineering. *IEEE Trans. Software Eng.*, 28(12):1171–1180, 2002.
- [33] J. Spolsky. *Joel on Software*. APress, US, 2004.
- [34] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, May 2008.
- [35] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, second edition, 2004.
- [36] W. Weimer. Patches as better bug reports. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190, 2006.
- [37] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.
- [38] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC-FSE '07: Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 85–94, 2007.