# What Makes a Good Bug Report?

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

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

Adrian Schröter*
adrian@st.cs.uni-sb.de

Cathrin Weiß*
weiss@st.cs.uni-sb.de

Rahul Premraj*§
premraj@cs.uni-sb.de

Thomas Zimmermann+§
tz@acm.org

*Department of Computer Science
Saarland University, Saarbrücken, Germany

+ Department of Computer Science
University of Calgary, Calgary, Alberta, Canada

## ABSTRACT

The information in bug reports influences the speed at which bugs are fixed. However, bug reports differ in their quality of information. In order to determine the elements that developer widely use to fix bugs and the problems frequently encountered, we conducted a survey among APACHE, ECLIPSE, and MOZILLA developers.

The analysis of the 156 responses shows that steps to reproduce and stack traces are most sought after, while inaccurate steps to reproduce and incomplete information pose the largest hurdles. This insight is helpful to design new bug tracking tools that guide reporters at providing more helpful information.

Our CUEZILLA prototype is such a tool and provides reporters with feedback on the quality of new bug reports and recommends to add missing elements. We trained CUEZILLA on a sample of 289 bug reports, rated by developers as part of the survey. In our evaluation, CUEZILLA was able to predict the quality of 31–48% of bugs reports accurately.

## 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; often providing inadequate or incorrect information. Thus, developers sometimes have to face bugs with descriptions such as “Sem Web” (APACHE bug #12324563), “wqqwqw” (ECLIPSE bug #145133), or “nuff said” (MOZILLA bug #6069). 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 extract the notion of *quality of bug reports* from the perspective of developers. Several factors impact the quality of bug reports such as length of description, formatting, and presence of stack traces and attachments (such as screenshots). In order to find out which factors matter most, we asked 867 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 [17]. We received a total of 1,248 votes for 289 randomly selected bug reports (Section 3).

By knowing what developers desire in bug reports, it becomes possible to provide real-time tool support to reporters for furnishing such information. As expected, most developers regarded *steps to reproduce* as the most helpful element in bug reports. Other important elements included stack traces and test cases. As a first step towards better tool support, we developed a prototype CUEZILLA (see Figure 1) that gauges the quality of bug reports and suggests additions to bug reports to make them better.

1. *CUEZILLA measures the quality of bug reports.* We trained and evaluated CUEZILLA on the 289 bug reports rated by the developers (Section 4).
2. *CUEZILLA provides incentives to reporters.* We automatically mined the bug databases for facts such as “Bug reports with stack traces are fixed sooner than other bugs” (Section 5).

To summarize, this paper makes the following contributions:

1. a survey on the *quality of bug reports* among 156 developers,
2. the CUEZILLA tool that measures bug report quality, and
3. an automatic approach to mine facts from bug databases.

We conclude this paper with threats to validity (Section 6), related work (Section 7), and future research directions (Section 8).

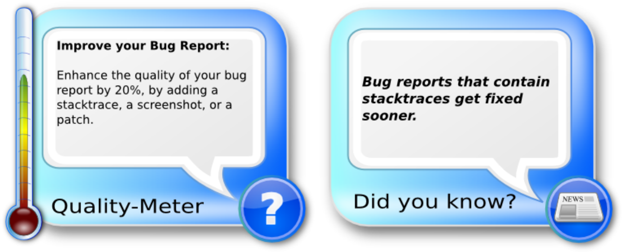

Improve your Bug Report:
Enhance the quality of your bug report by 20%, by adding a stacktrace, a screenshot, or a patch.
Quality-Meter
?
Bug reports that contain stacktraces get fixed sooner.
Did you know?
NEWS

**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).**

[§] Contact authors are Rahul Premraj and Thomas Zimmermann.

Technical Report. Saarland University, Saarbrücken, Germany.
Published online on September, 16, 2007.
http://www.st.cs.uni-sb.de/publications/details/bettenburg-tr-2007/

| | | | | |
|---|---|---|---|---|
| **Contents of bug reports.** | Q1: Which of the following items have you previously used when fixing bugs? | | | |
| | Q2: Which three items helped you the most? | | | |
| | ❑ 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.** | Q3: Which of the following problems have you encountered when fixing bugs? | | | |
| | Q4: 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 spellcheck | |
| | ❑ expected behavior | | | |
| **Comments.** | Q5: Please feel free to share any interesting thoughts or experiences. | | | |

**Figure 2: The questionnaire presented to APACHE, ECLIPSE, and MOZILLA developers.**

## 2. SURVEY ON BUG QUALITY

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.

### 2.1 Survey Design

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 time of five minutes, which we also advertised in the invitation email ("we would much appreciate five minutes of your time").

#### *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 fifty bug reports in their respective projects. Table 1 presents for each project the number of developers contacted via personalized email, the number of bounces, and the number of responses and comments received. The response rate was highest for MOZILLA at 25%. Our overall response rate of 19.3% is comparable to other Internet surveys in software engineering, which range from 14% to 20% [22].

#### *The Questionnaire.*

Keeping the five minute rule in mind, we chose the following questions that we grouped into three parts (see also 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 [10]; or being standard fields in the BUGZILLA database.

Responders were free to check as many items for the first question (Q1), but at most three for the second question (Q2), thus indicating the importance of items.

**Table 1: Number of developers, bounces, and responses.**

| Project | Developers | Bounces | Responses | Comments |
|---|---|---|---|---|
| APACHE | 194 | 5 | 34 (18.0%) | 12 |
| ECLIPSE | 365 | 29 | 51 (15.2%) | 15 |
| MOZILLA | 313 | 29 | 71 (25.0%) | 21 |
| Total | 872 | 63 | 156 (19.3%) | 48 |

**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.[1] 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, responders were free to check as many items for the first question (Q3), but at most three for the second question (Q4).

**Comments.** *What are the thoughts and experiences of developers with the quality of bug reports?*

#### *Parallelism between Questions.*

In the first two parts of the survey, constituent 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 Q1 and Q2 as examples.

[1] Did you know? In ECLIPSE, 205 bug reports were submitted for "Windows" but later re-assigned to "Linux".

**Table 2: Importance of items by project. The topmost bar refers to the 33 responses by APACHE developers, the middle bar to the 44 responses by ECLIPSE developers, and lowest bar to the 69 responses by MOZILLA developers.**

**Contents of bug reports.**

| Item | APACHE | ECLIPSE | MOZILLA |
|---|---|---|---|
| product | 4% | 0% | 8% |
| component | 5% | 0% | 6% |
| version | 28% | 2% | 8% |
| severity | 0% | 0% | 0% |
| hardware | 0% | 0% | 0% |
| operating system | 0% | 0% | 8% |
| summary | 4% | 6% | 20% |
| build information | 0% | 6% | 9% |
| observed behavior | 32% | 25% | 39% |
| expected behavior | 24% | 5% | 28% |
| steps to reproduce | 60% | 88% | 88% |
| stack traces | 61% | 75% | 36% |
| screenshots | 15% | 39% | 18% |
| code examples | 23% | 23% | 7% |
| error reports | 13% | 13% | 12% |
| test cases | 67% | 34% | 59% |

**Problems with bug reports.**

| Group | Item | APACHE | ECLIPSE | MOZILLA |
|---|---|---|---|---|
| You were given wrong: | product name | 0% | 0% | 10% |
| | component name | 0% | 11% | 18% |
| | version number | 26% | 33% | 17% |
| | hardware | 100% | 0% | 33% |
| | operating system | 50% | 12% | 17% |
| | observed behavior | 45% | 50% | 50% |
| | expected behavior | 35% | 11% | 21% |
| There were errors in: | code examples | 41% | 20% | 11% |
| | steps to reproduce | 78% | 86% | 82% |
| | test cases | 30% | 33% | 44% |
| | stack traces | 100% | 0% | 20% |
| The reporter used: | bad grammar | 18% | 19% | 12% |
| | unstructured text | 38% | 12% | 32% |
| | prose text | 0% | 0% | 25% |
| | too long text | 27% | 22% | 31% |
| | non-technical language | 14% | 16% | 20% |
| | no spellcheck | 0% | 0% | 0% |
| Others: | duplicates | 12% | 7% | 12% |
| | spam | 0% | 0% | 0% |
| | incomplete information | 72% | 90% | 70% |
| | viruses/worms | 0% | 0% | 0% |

1. *Consistency check.* When fixing bugs, all items that helped a developer the most (selected in Q2) *must* have been used previously (selected in Q1). If this is not the case, i.e., an item is selected in Q2 but not in Q1, the response is regarded as inconsistent. Responses from one APACHE, seven ECLIPSE, and three MOZILLA developers failed the consistency check and were removed from our analysis.

2. *Importance of items.* We can additionally infer the importance of individual items. For instance, for item *i*, let $Q1(i)$ be the number of times it was selected in question Q1; and $Q2(i)$ for question Q2. Then the importance of item *i* corresponds to the likelihood that item *i* is selected in Q2, when it is selected in Q1.

$$\mathit{Importance}(i) = \frac{Q2(i)}{Q1(i)}$$

## 2.2 Survey Results

In this section, we discuss our findings from the survey responses. To recall, we received a total of 156 responses, of which 11 were removed for being inconsistent. The results of our survey are summarized in Tables 2 and 3. In these tables, responses for each item are annotated as bars ( ■▨□ ), which can be broken down into their constituents and interpreted as below (again, explained with Q1 and Q2 as examples):

| | |
|---|---|
| ■▨□ | All consistent responses for the project |
| ■▨ | Number of times that *item* was selected in Q1 |
| ■ | Number of times that *item* was selected in Q1 and Q2 |
| ▨ | Number of times that *item* was selected in Q1 but not Q2 |

The colored part (■+▨) denotes the count of responses for an item in question Q1; and the black part (■) of the bar denotes the count of responses for the item in both question Q1 and Q2. The larger the black bar is in proportion to the grey bar, the higher is the corresponding item's importance in the developers' perspective.

In Table 2, three such bars are grouped together for each item. The topmost bar presents results for APACHE developers; the middle bar for ECLIPSE developers; and the lowest bar for MOZILLA developers. The width of each bar accounts for the number of consistent responses received for the individual project. In the table, the importance for all three projects is listed after every item in parentheses (left: APACHE; middle: ECLIPSE; right: MOZILLA).

Similarly, in Table 3, we present the results taking all responses from the three projects for each information item. Each bar is then interpreted in the same manner as described above.

### *Contents of Bug Reports.*

We see from Tables 2 and 3 that the **most widely used items** across projects are *steps to reproduce*, *observed* and *expected behavior*, *stack traces*, and *test cases*. *Screenshots* are used only by ECLIPSE and MOZILLA developers, while *code examples* and *error reports* are mostly used by APACHE and ECLIPSE developers. Rarely used contents include *version*, *summary*, and *operating system*, while *hardware* appears to be most seldom used across all projects.

For the **importance of items**, *steps to reproduce* stands out for all three projects. 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. To our surprise, *screenshots* were not rated high, possibly because they are helpful only for a subset of bugs, e.g., GUI errors.

**Table 3: Average importance of items across all projects (percentage values in parantheses)**

**Contents of bug reports.**

| | | | |
|---|---|---|---|
| product (4%) | hardware (0%) | observed behavior (32%) | screenshots (24%) |
| component (3%) | operating system (2%) | expected behavior (19%) | code examples (17%) |
| version (12%) | summary (10%) | steps to reproduce (78%) | error reports (12%) |
| severity (0%) | build information (5%) | stack traces (57%) | test cases (53%) |

**Problems with bug reports.**

| You were given wrong: | There were errors in: | The reporter used: | Others: |
|---|---|---|---|
| product name (3%) | code examples (24%) | bad grammar (16%) | duplicates (10%) |
| component name (9%) | steps to reproduce (82%) | unstructured text (27%) | spam (0%) |
| version number (25%) | test cases (35%) | prose text (8%) | incomplete information (77%) |
| hardware (44%) | stack traces (40%) | too long text (26%) | viruses/worms (0%) |
| operating system (26%) | | non-technical language (16%) | |
| observed behavior (48%) | | no spellcheck (0%) | |
| expected behavior (22%) | | | |

Interesting surprises in these 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 one of the MOZILLA developers, 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.

## *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 Q5 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 care too much about duplicates. 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, which successfully prevents spam. Lastly, errors in *stack traces* are highly unlikely because they are copy-pasted into bug reports, but this can be a severe problem.

## *Developer Comments.*

We received forty-eight 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."*

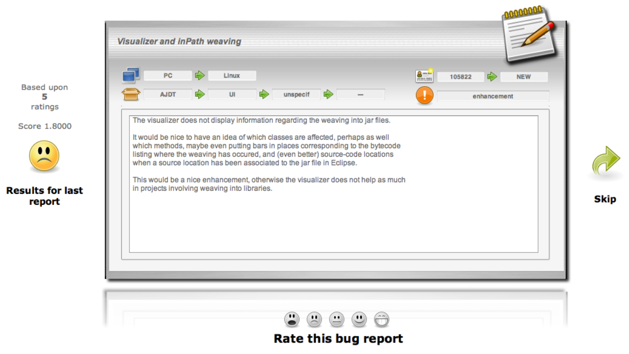
**Misuse as a debate system.** *"Bugs are often used to debate the relative importance of various issues. This debate tends to spam the bugs with various use cases and discussions therewithin, making it harder to locate the technical arguments often necessary for fixing the bugs. Some long-lived high-visibility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

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

**Human component.** *"Another import thing is that devs know you (because you have filed bug reports before, you discussed with them on IRC, conferences, ...)"*

Another interesting related comment: *"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."*



The visualizer does not display information regarding the weaving into jar files.

It would be nice to have an idea of which classes are affected, perhaps as well which methods, maybe even putting bars in places corresponding to the bytecode listing where the weaving has occured, and (even better) source-code locations when a source location has been associated to the jar file in Eclipse.

This would be a nice enhancement, otherwise the visualizer does not help as much in projects involving weaving into libraries.

**Figure 3: Screenshot of Rating Bugs' Quality Interface**

**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."*

## 3. RATING BUG REPORTS

After completing the questionnaire, developers were asked to participate in 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.

### *Rating Infrastructure.*

The rating system was inspired by Internet sites such as RateMyFace [23] and HotOrNot [11]. 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 3 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 developers, in cases where they lacked the knowledge to rate a report. Developers 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 allow 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 4).

### *Rating Results.*

The following number of 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 4 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 had the highest and lowest average ratings in the ECLIPSE sample. Some bugs 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.

| 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 |

**Table 4: Developers rated the quality of ECLIPSE bug reports.**

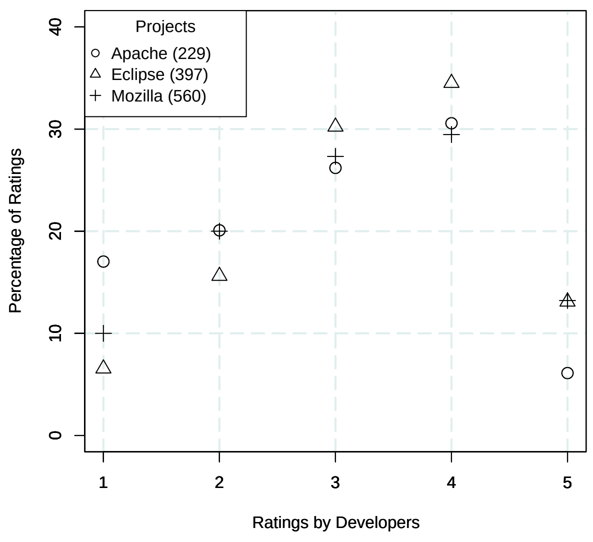


**Figure 4: Distribution of ratings by developers**

*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 data base.

*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)

*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 verify this by examining the standard deviations of quality ratings by developers ($\sigma_{\text{rating}}$) for the bugs reports. Larger values of $\sigma_{\text{rating}}$ indicate higher differences between developers' view of quality for a bug report. Of these, 289 bugs 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.

## 4. PREDICTING BUG REPORT QUALITY

Now that we know what is important in bug reports, we proceed to show how such knowledge can be applied. Humans can benefit from cues while undertaking tasks, which was demonstrated in software engineering by Passing and Shepperd [21]. 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 in real-time; 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 then, reports results of its evaluation at measuring quality of bugs reports. In order to create recommendations, CUEZILLA first represent each bug report as a feature vector (Section 4.1). Then it uses supervised learning to train models (Section 4.2) that measure the quality of bug reports (Section 4.3). These models can also quantify the increase in quality, when elements are added to bug reports (Section 4.4).

### 4.1 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 $-$, $\star$, 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. [15] 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.

For the following features, 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.

**Code Samples.** We identify C++ and JAVA code examples using technqiues from island parsing [19]. 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) [18].

**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 ASCII text, we process the file and search for code examples, stack traces, and patches (see above).

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

**Readability.** The *style* tool "analyses the surface characteristics of the writing style of a document" [7], which means it does not use grammatical correctness to assess how difficult a text is to read. In contrast it looks for 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 [14].

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.[2]

### 4.2 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:

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

– Generalized linear regression,

– Stepwise linear regression, and

– Support vector machines (SVM)

Each models uses the scores from the features described in (Section 4.1) as input variables; and tries to predict the averaged developer ratings as output variable. We evaluated CUEZILLA using the following 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.

**Across projects.** We also tested if models from one project can be transfered 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%.

## 4.3 Results of Experiments

### *Evaluation within Projects.*

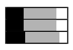
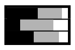
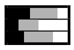
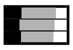
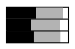
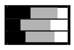
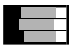
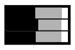
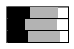
Results from predicting the quality of bug reports using other bug reports from the same project (with leave-one-out cross-validation) are presented in Table 6. The first number is the percentage of bug reports with perfect agreement on the quality between CUEZILLA and the developers, while the number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

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.

**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%) |

**Table 7: Validation across projects.**

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

### *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.

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 generalised 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.

## 4.4 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 8.

## 5. 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 data bases. In this section, we elaborate upon how this is executed, and close with some facts found in the bug data bases of the three projects.

### *Data Preparation.*

To reduce the complexity of mining the several thousand bug reports filed in bug data bases, 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 4.1 for each of the 150,000 bugs. To recall, the scores for some of the items are continuous values, while others are binary.

### *Relation between Contents and Resolutions.*

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 like—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.

Then, for binary valued features, we then performed chi-square tests ($p < .05$) 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.

On the other hand, for features with continuous valued scores, we performed an Kruskal-Wallis test ($p < .05$) 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.

### *Relation between Contents and Lifetimes.*

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 ($p < .05$) to check for statistically significant 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 discretized the lifetime of bugs into three pairs: [$<$ 1 hour , $>$ 1 hour]; [1 hour – 1 day, $>$ 1 day]; and [1 day – 1 week, $>$ 1 week], and then binned the bugs into the respective categories by lifetimes. We then compared the distribution of the item scores across all three pairs to using the Kruskal-Wallis test ($p < .05$) to reveal statistically significant patterns. Again, differences in distributions could be used to motivate users to aim at achieving scores for their reports that are likely to have lower lifetimes.

### *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 the interface of the bug tracking systems, as demonstrated in the right figure in Figure 1. Our key findings from the investigation are listed below:

**Did you know**

- 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)

Each of these findings suggest a way for reporters to increase the likelihood of their bugs to either get fixed at all, or get fixed faster. Keen reporters are likely to pick up on such cues since this can lessen the amount of time they have to deal with the bug.

## 6. 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.

A related threat is that to some extent our survey operated on a *self-selection principle:* the participation in the survey was voluntarily. As a consequence, the results might be skewed towards people that are likely to answer the survey, such as developers 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" [25].

In order to take as little time as possible off developers, 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 developers to provide us with additional comments, to which we received 48 responses. We could not include the comments into the statistical analysis; however, we studied and discussed them in Section 2.2.

As with any empirical study, it is difficult to draw general conclusions because any process depends on a *large number of context variables* [3]. In our case, we contacted developers of three large open-source initiatives 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.

## 7. RELATED WORK

To our knowledge, no other work has specifically studied the quality of bug reports or suggested a quality-meter tool for bug reports. So far, only anecdotical evidence has been reported, for instance, Joel Spolsky observed that funny bug report has higher chances of getting addressed [26].

In a workshop paper, we presented preliminary results for the ECLIPSE project using a handcrafted prediction model [4]. In this paper we improve our previous results as follows: (1) training of prediction models for bug quality, (2) fully-fledged evaluation of CUEZILLA's predictions within and across projects, and (3) automatic mining of facts that serve as incentives for bug reporters. The survey and all experiments were carried out for two additional projects: APACHE and MOZILLA.

Several studies used bug reports to automatically assign developers to bug reports [2, 6], assign locations to bug reports [5], track features over time [9], recognize bug duplicates [8, 24], and predict effort for bug reports [28]. All these approaches should benefit by our measure for the quality of bug reports since training only with high-quality bug reports will likely improve their predictions.

In order to inform the design of new bug reporting tools, Ko et al. [15] 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.

In 2004, Antoniol et al. [1] pointed out the lack of integration between version archives and bug databases. Providing such an integration allows queries to locate the most faulty methods in a system. While the lack of integration was problematic a few years ago, things have changed in the meantime: the Mylyn tool by Kersten and Murphy [13] allows to attach a task context to bug reports so that changes can be tracked on a very fine-grained level.

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/repay techniques (such as [20, 12, 29]), 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 suggested to additionally provide patches and presented an algorithm to construct such patches (in the presence of model-checking and safety-policy information). He found that automatically generated "reports also accompanied by patches were three times as likely to be addressed as standard bug reports" [27].

Furthermore, users can help developers to fix bugs without filing bug reports. Liblit et al. introduced statistical debugging [16]. 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 isolated bugs using statistical techniques. Currently, their approach works best for crashes, but one can imagine a "report back" button that send the state of a (failing) program. Still, it is unclear how to extract sufficient information for rarely occurring bugs.

## 8. CONCLUSION AND CONSEQUENCES

Well written bug reports are likely to get more attention among developers than poorly written ones. In order to get a notion of *bug report quality* from the developers' perspectives, we conducted a survey among APACHE, ECLIPSE, and MOZILLA developers. 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.

Additionally, the developers rated bug reports on their quality on a scale from one (poor quality) to five (excellent quality). Based on the results from the survey, we developed a tool, CUEZILLA that measures the quality of bug reports in real-time. This tool can rate up to 41% bug reports in complete agreement with developers. Additionally, it is also programmed to recommend what additions can be made to bug reports to make their quality better. As incentives for doing so, CUEZILLA automatically mines patterns that are relevant to fixing bugs and presents them to reporters. 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; and also serve as a data cleaning technique for research on bug reports. Our future work is thus 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.** In Section 2.2 we listed several comments by developers about problems with existing bug reporting tools. 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.** Several approaches used bug reports to automatically assign developers to bug reports [2, 6], assign locations to bug reports [5], recognize bug duplicates [8, 24], and predict effort for bug reports [28]. Does training only with high-quality bug reports improve their predictions?

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

`http://www.softevo.org/`

## 9. ACKNOWLEDGMENTS

Many thanks to Christian Lindig, Stephan Neuhaus, and Andreas Zeller for their valuable discussions and helpful suggestions on earlier revisions of this paper. A special thanks to all developers who responded to our survey. When this research was carried out, Thomas Zimmermann was with Saarland University and 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: Proceeding 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)*. ACM Press, October 2007. To appear.

[5] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 105–111, 2006.

[6] 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.

[7] L. Cherry and W. Vesterman. Writing tools - the STYLE and DICTION programs. Technical report, AT&T Laboratories, 1980.

[8] 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.

[9] 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 2003), 13-16 November 2003, Victoria, Canada*, pages 90–101, 2003.

[10] E. Goldberg. Bug writing guidelines. https://bugs.eclipse.org/bugs/bugwritinghelp.html. Last accessed 2007-08-04.

[11] HOT or NOT. http://www.hotornot.com/. Last accessed 2007-09-11.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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, New York, NY, USA, 2005. ACM Press.

[17] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:1–55, 1932.

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

[19] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*, pages 13–, 2001.

[20] 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.

[21] U. Passing and M. J. Shepperd. An experiment on software project size and effort estimation. In *ISESE*, pages 120–131. IEEE Computer Society, 2003.

[22] 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.

[23] Ratemyface.com. http://www.ratemyface.com/. Last accessed 2007-09-11.

[24] 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.

[25] J. Singer and N. G. Vinson. Ethical issues in empirical studies of software engineering. *IEEE Trans. Software Eng.*, 28(12):1171–1180, 2002.

[26] J. Spolsky. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those . . . Ill-Luck, Work with Them in Some Capacity*. APress,US, 2004.

[27] 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, New York, NY, USA, 2006. ACM Press.

[28] 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.

[29] 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 6th joint meeting of the european software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering*, pages 85–94, New York, NY, USA, 2007. ACM Press.