When Do Changes Induce Fixes?

(On Fridays.)

Jacek Śliwerski
International Max Planck Research School
Max Planck Institute for Computer Science
Saarbrücken, Germany
sliwers@mpi-sb.mpg.de

Thomas Zimmermann Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany
{tz, zeller}@acm.org

ABSTRACT

As a software system evolves, programmers make changes that sometimes cause problems. We analyze CVS archives for *fix-inducing changes*—changes that lead to problems, indicated by fixes. We show how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as BUGZILLA). In a first investigation of the MOZILLA and ECLIPSE history, it turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*corrections, version control*; D.2.8 [**Metrics**]: Complexity measures

General Terms

Management, Measurement

1. INTRODUCTION

When we mine software histories, we frequently do so in order to detect patterns that help us understanding the current state of the system. Unfortunately, not all changes in the past have been beneficial. Any bug database will show a significant fraction of problems that are reported some time after some change has been made.

In this work, we attempt to identify those *changes that caused problems.* The basic idea is as follows:

1. We start with a bug report in the bug database, indicating a *fixed problem.*
2. We extract the associated change from the version archive, thus giving us the *location* of the fix.
3. We determine the *earlier change* at this location that was applied before the bug was reported.

This earlier change is the one that *caused* the later fix. We call such a change *fix-inducing.*

What can one do with fix-inducing changes? Here are some potential applications:

Which change properties may lead to problems? We can investigate which properties of a change correlate with inducing fixes, for instance, changes made on a specific day or by a specific group of developers.

How error-prone is my product? We can assign a *metric* to the product—on average, how likely is it that a change induces a later fix?

How can I filter out problematic changes? When extracting the architecture via co-changes from a version archive, there is no need to consider fix-inducing changes, as they get undone later.

Can I improve guidance along related changes? When using co-changes to guide programmers along related changes, we would like to avoid fix-inducing changes in our suggestions.

This paper describes our first experiences with fix-inducing changes. We discuss how to extract data from version and bug archives (Section 2), and how we link bug reports to changes (Section 3). In Section 4, we describe how to identify and locate fix-inducing changes. Section 5 shows the results of our investigation of the MOZILLA and ECLIPSE: It turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied. Sections 6 and 7 close with related and future work.

2. WHAT'S IN OUR ARCHIVES?

For our analysis we need all changes and all fixes of a project. We get this data from *version archives* like CVS and *bug tracking systems* like BUGZILLA.

A CVS archive contains information about changes: Who changed what, when, why, and how? A *change* δ transforms a revision r_1 to a revision r_2 by inserting, deleting, or changing lines. We will later investigate changes on the line level. Several changes $\delta_1, \ldots, \delta_n$ form a *transaction* t if they were submitted to CVS by the same developer, at the same time, and with the same log message, i.e., they have been made with the same intention, e.g. to fix a bug or to introduce a new feature. As CVS records only individual changes to files, we group these to transactions with a *sliding time window* approach [12].

A CVS archive also lacks information about the *purpose* of a change: Did it introduce a new feature or did it fix a bug? Although it is possible to identify such reasons solely with log messages [7], we combine both CVS and BUGZILLA for this step because this increases the precision of our approach.

A BUGZILLA database collects bug reports that are submitted by a *reporter* with a *short description* and a *summary*. After a bug has been submitted, it is discussed by developers and users who provide additional *comments* and may create *attachments.* After the

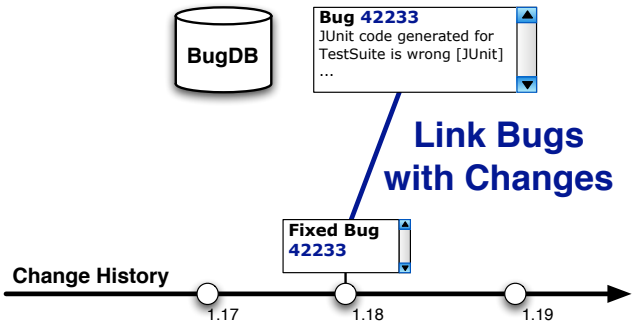

Figure 1: Link transactions to bug reports

bug has been confirmed, it is *assigned* to a developer who is responsible to fix the bug and finally commits her changes to the version control archive. BUGZILLA also captures the *status* of a bug, e.g., `UNCONFIRMED`, `NEW`, `ASSIGNED`, `RESOLVED`, or `CLOSED` and the *resolution*, e.g., `FIXED`, `DUPLICATE`, or `INVALID`. Details on the lifecycle of a bug can be found in the BUGZILLA documentation [10, Sections 6.3 and 6.4].

For our analysis, we mirror both CVS and BUGZILLA in a local database. Our mirroing techniques for CVS are described in [12]. To mirror a BUGZILLA database, we use its XML export feature. Additionally, we import attachments and activities directly from the web interface of BUGZILLA. Our local BUGZILLA database schema is similar to the one described in [2].

3. IDENTIFYING FIXES

In order to locate fix-inducing changes, we first need to know whether a change is a fix. A common practice among developers is to include a *bug report number* in the comment whenever they fix a defect associated with it. Čubranić and Murphy [4] as well as Fischer, Pinzger, and Gall [5, 6] exploited this practice to link changes with bugs. Figure 1 sketches the basic idea of this approach.

In our work, we refine these techniques by assigning every link (t, b) between a transaction t and a bug b two independent levels of confidence: a *syntactic* level, inferring links from a CVS log to a bug report, and a *semantic* level, validating a link via the bug report data. These levels are later used to decide which links shall be taken into account in our experiments.

3.1 Syntactic Analysis

In order to finds links to the bug database, we split every log message into a stream of tokens. A token is one of the following items:

- a *bug number*, if it matches one of the following regular expressions (given in FLEX syntax):
  - `bug[# \t]*[0-9]+`,
  - `pr[# \t]*[0-9]+`,
  - `show\_bug\.cgi\?id=[0-9]+`, or
  - `\[[0-9]+\]`
- a *plain number*, if it is a string of digits `[0-9]+`
- a *keyword*, if it matches the following regular expression: `fix(e[ds])?|bugs?|defects?|patch`
- a *word*, if it is a string of alphanumeric characters

Every number is a potential link to a bug. For each link, we initially assign a syntactic confidence *syn* of zero and raise the confidence by one for each of the following conditions that is met:

1. The number is a *bug number*.
2. The log message contains a *keyword*,
   or the log message contains only *plain* or *bug numbers*.

Thus the syntactic confidence *syn* is always an integer number between 0 and 2. As an example, consider the following log messages:

- `Fixed bug 53784: .class file missing from jar file export`
  The link to the bug number 53784 gets a syntactic confidence of 2 because it matches the regular expression for `bug` and contains the keyword `fixed`.
- `52264, 51529`
  The links to bugs 52264 and 51529 have syntactic confidence 1 because the log message contains only numbers.
- `Updated copyrights to 2004`
  The link to the bug number 2004 has a syntactic confidence of 0 because there is no syntactic evidence that this number refers to a bug.

3.2 Semantic Analysis

In the previous section, we inferred links that point from a transaction to a bug report. To validate a link (t, b) we take information about its transaction t and check it against information about its bug report b. Based on the outcome we assign the link a semantic level of confidence.

Initially, a link (t, b) has semantic confidence of 0 which is raised by 1 whenever one of the following conditions is met:

- The bug b has been resolved as `FIXED` at least once.[1]
- The short description of the bug report b is contained in the log message of the transaction t.
- The author of the transaction t has been assigned to the bug b.[2]
- One or more of the files affected by the transaction t have been attached to the bug b.

This list is not meant to be exhaustive. One could for example check whether a change has been committed to the repository within a small timeframe around the time when a bug has been closed.[3]

Consider the following examples from ECLIPSE, which all have low confidence levels:

- `Updated copyrights to 2004`
  The potential bug report number "2004" is marked as *invalid* and thus the semantic confidence of the link is zero.
- `Fixed bug mentioned in bug 64129, comment 6`
  The number "6" appears in the comment for a fix. The syntactic confidence is 1, but the semantic confidence is 0.
- `Support expression like (i)+= 3; and new int[] {1}[0] + syntax error improvement`
  "1" and "3" are (mistakenly) interpreted as bug report numbers here. Since the bug reports 1 and 3 have been fixed, the links both get a semantic confidence of 1.

[1]Notice that only 27% of all bugs in the MOZILLA project are `FIXED` (47% for ECLIPSE).

[2]For this check, we need a mapping between the CVS and BUGZILLA *user accounts* of a project. For ECLIPSE, we mapped the accounts of the most active developers manually; for MOZILLA, we derived a simple heuristic based on the observation that email addresses were used as logins for both CVS and BUGZILLA.

[3]Čubranić and Murphy already applied this as a standalone technique to relate bugs to transactions in their HIPIKAT tool [4].

- `Fixed bug 53784: .class file missing from jar file export.`
  The bug 53784 has not been closed, but resolved as LATER. Its short description is: "Different results when running under debugger" and author of the change has not been assigned this bug. Thus the semantic confidence of the link is 0.

  However, there exists a bug 53284 with the following short description: ".class file missing from jar file export". If the comment had contained a correct number, the link would be assigned the semantic confidence 3.

3.3 Results

We identified 25,317 links for ECLIPSE, connecting 47% of fixed bugs with 29% of transactions and 53,574 links for MOZILLA, connecting 55.30% of fixed bugs with 43.91% of transactions. Tables 1 and 2 summarize the distribution of links across different classes of syntactic and semantic levels for both projects.

Based on a manual inspection of several randomly chosen links (see Section 3.2 for some examples), we decided to use only those links whose syntactic and semantic levels of confidence satisfy the following condition:

$$sem > 1 \vee (sem = 1 \wedge syn > 0)$$

Notice that we disregard less than 10% of links for both projects.

Our heuristics can be ported to almost any project that contains in the log messages links to a bug database. In some cases it may be necessary to implement further or different conditions to raise the confidence levels. However, the quality of the linking will always depend on the investigated project.

4. LOCATING FIX-INDUCING CHANGES

A fix-inducing change is a change that later gets undone by a fix. In this section, we show how to automatically locate fix-inducing changes.

Suppose that a change $\delta \in t$, which is known to be a fix for bug b (thus a link (t, b) must exist), transforms the revision $r_1 = 1.17$ of `Foo.java` into $r_2 = 1.18$ (see Figure 2), i.e., δ introduces new lines to r_2 or changes and removes lines of r_1. First, we detect the lines L that have been touched by δ in r_1. These are the locations of the fix. To locate them, we use the CVS *diff* command. In our example, we assume that line 20 and 40 have been changed and line 60 has been deleted, thus the fix locations in r_1 are $L = \{20; 40; 60\}$.

Next, we call the CVS *annotate* command for revision $r_1 = 1.17$ because this was the last revision without the fix; in contrast, revision $r_2 = 1.18$ already contains the applied fix. The annotations prepend each line with the most recent revision that touched this line. Additionally, CVS includes the developer and the date in the output. We show an excerpt of the annotated file in Figure 3. The CVS *annotate* command is only reliable for text files, thus we ignore all files that are marked as binary in the repository.

We scan the output and take for each line $l \in L$ the revision r_0 that annotates line l. These revisions are candidates for fix-inducing changes. We add (r_0, r_2) to the candidate set S, which is in our example $S = \{(1.11, 1.18); (1.14, 1.18); (1.16, 1.18)\}$.

From this set, we remove pairs (r_a, r_b) for which it is not possible that r_a induced the fix r_b—for instance, because r_a was committed to CVS *after* the bug fixed by r_b has been reported. In particular, we say that such a pair (r_a, r_b) is a *suspect* if r_a was committed after the *latest* reported bug linked with the revision r_b. Suspect changes could not contribute to the failure observed in the bug report. In Figure 2 the pairs $(1.14, 1.18)$ and $(1.16, 1.18)$ are examples of suspects.

We investigate suspects further on:

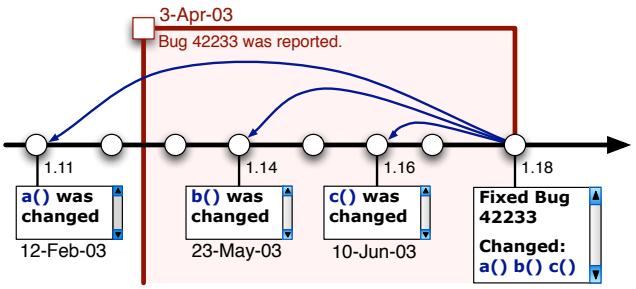

Figure 2: Locate fix-inducing changes for bug 42233

```
$ cvs annotate -r 1.17 Foo.java
...
19: 1.11 (john 12-Feb-03): public int a() {
20: 1.11 (john 12-Feb-03):     return i/0;
...
39: 1.10 (mary 12-Jan-03): public int b() {
40: 1.14 (kate 23-May-03):     return 42;
...
59: 1.10 (mary 17-Jan-03): public void c() {
60: 1.16 (mary 10-Jun-03):     int i=0;
...
```

Figure 3: CVS annotations for Foo.java

- We say that a suspect (r_a, r_b) is a *partial fix* if r_a is a fix.

  Some bugs are fixed more than once. It may happen that one of the previous attempts was fixed by a later one, or that the bug is fixed across several transactions.
- We say that a suspect (r_a, r_b) is a *weak suspect* if there exists a pair (r_a, r_c) which is not a suspect.

  A weak suspect indicates a revision for which there exists an alternative evidence of being fix-inducing, e.g, revision 1.14 may be a suspect for bug 42233 in Figure 2, but it still can be a strong candidate for another bug.
- We say that a suspect (r_a, r_b) is a *hard suspect* if it is neither a partial fix, nor a weak suspect.

  A hard suspect indicates a revision for which there is no real evidence of being fix-inducing.

We say that a revision r is *fix-inducing* if there exists a pair $(r, r_x) \in S$ which is not a hard suspect. We say that a transaction t is *fix-inducing* if one of its revisions is fix-inducing.

5. FIRST RESULTS

We extracted fix-inducing changes for two large open-source projects: ECLIPSE and MOZILLA. We considered all changes and bugs until January 20, 2005; our database contains 78,954 transactions for ECLIPSE and 109,658 transactions for MOZILLA. They account for 278,010 and 392,972 individual revisions for both projects, respectively.

5.1 Fix-Inducing Transactions are Large

In our first experiment, we examined if the span of the transaction (i.e. the number of files touched) correlates with the fact that the transaction is fix-inducing. Table 3 presents the average sizes of transactions for ECLIPSE. The transactions are split into four classes, depending on whether the transaction is a fix, fix-inducing, both, or none. For instance, the top-left cell means that

syn / sem	0	1	2	3	4	total
0	270 (1%)	1,287 (5%)	2,057 (8%)	1,439 (6%)	2 (0%)	5,055 (20%)
1	324 (1%)	4,152 (16%)	9,265 (37%)	1,581 (6%)	5 (0%)	15,327 (61%)
2	110 (0%)	1,922 (8%)	2,421 (10%)	482 (2%)	0 (0%)	4,935 (19%)
total	704 (3%)	7,361 (29%)	13,743 (54%)	3,502 (14%)	7 (0%)	25,317 (100%)

Table 1: Distribution of links accross different classes of syntactic and semantic confidence levels in ECLIPSE

syn / sem	0	1	2	3	4	total
0	560 (1%)	2,899 (5%)	4,281 (8%)	639 (1%)	8 (0%)	8,387 (16%)
1	1,211 (2%)	9,059 (17%)	16,336 (30%)	2,241 (4%)	22 (0%)	28,669 (54%)
2	478 (1%)	5,250 (10%)	9,133 (17%)	1,645 (3%)	12 (0%)	16,518 (31%)
total	2,249 (4%)	17,208 (32%)	29,750 (55%)	4,525 (8%)	42 (0%)	53,574 (100%)

Table 2: Distribution of links accross different classes of syntactic and semantic confidence levels in MOZILLA

	fix-inducing	¬fix-inducing	all
fix	3.82±26.32	2.08± 7.42	2.73± 7.87
¬ fix	11.30±63.02	2.77±14.94	3.81±26.32
all	7.49±44.37	2.61±13.66	3.52±22.81

Table 3: Average sizes of fix and fix-inducing transactions for ECLIPSE

	fix-inducing	¬fix-inducing	all
fix	5.79±37.37	2.12± 9.74	4.39±30.05
¬ fix	4.61±30.59	1.91±10.30	3.05±21.39
all	5.19±34.12	1.97±10.13	3.58±25.23

Table 4: Average sizes of fix and fix-inducing transactions for MOZILLA

the average size of transactions which are fixes *and* induce later on a fix is 3.82 (with a standard deviation "±" of 26.32).

Additionally, Table 3 shows that fix-inducing transactions are roughly three times larger than non fix-inducing transactions. Table 4 presents the same breakdown for MOZILLA which shows a similar trend.

Such data can be automatically retrieved from all projects that supply both a version archive and a bug database. It is especially worthy when deciding where to spend efforts in *quality assurance.* If we were in charge of the ECLIPSE project, for instance, we would take care that large extensions are well reviewed and tested, as these have a high potential for inducing later fixes.

5.2 Don't Program on Fridays

We broke down changes by the day of the week when they were applied. We distinguished between *bugs* as indicated by fix-inducing changes, and *fixes* as detected by links to the bug database. Bugs may be also fixes, we refer to such changes as *fix-inducing fixes*; they have been previously been used for visualization by Baker and Eick [1]. Finally, there are changes that are no bugs and no fixes.

$$P(\mathit{fix}) + P(\mathit{bug}) - P(\mathit{bug} \cap \mathit{fix}) + P(\neg \mathit{bug} \cap \neg \mathit{fix}) = 100\%$$

We measured the frequencies of the categories mentioned above. Table 5 presents the results for ECLIPSE. The likelihood $P(\mathit{bug})$ that a change will induce a fix is highest on Friday. The same holds for MOZILLA (see Table 6). Friday is the day where most ECLIPSE developers do fixes, for MOZILLA this is Sunday.

	Day of Week							
% of revisions	Mon	Tue	Wed	Thu	Fri	Sat	Sun	avg
$P(\mathit{fix})$	18.4	20.9	20.0	22.3	24.0	14.7	16.9	20.8
$P(\mathit{bug})$	11.3	10.4	11.1	12.1	12.2	11.7	11.6	11.4
$P(\mathit{bug} \cap \mathit{fix})$	4.6	4.8	4.6	5.2	5.6	4.5	4.5	4.9
$P(\neg \mathit{bug} \cap \neg \mathit{fix})$	74.9	73.5	73.5	70.8	63.4	78.1	76.0	72.7
$P(\mathit{bug} \mid \mathit{fix})$	25.1	22.9	23.3	23.5	23.2	30.3	26.4	23.7
$P(\mathit{bug} \mid \neg \mathit{fix})$	8.2	7.1	8.1	8.8	8.7	8.4	8.6	8.1

Table 5: Distribution of fixes and fix-inducing changes across day of week in ECLIPSE

	Day of Week							
% of revisions	Mon	Tue	Wed	Thu	Fri	Sat	Sun	avg
$P(\mathit{fix})$	42.5	46.5	49.7	45.9	48.4	50.2	61.1	48.5
$P(\mathit{bug})$	39.1	44.1	41.2	40.8	46.2	44.9	26.4	41.5
$P(\mathit{bug} \cap \mathit{fix})$	19.4	23.6	22.8	21.6	26.9	19.6	13.2	21.9
$P(\neg \mathit{bug} \cap \neg \mathit{fix})$	37.8	33.0	31.9	34.9	32.3	24.5	25.7	31.9
$P(\mathit{bug} \mid \mathit{fix})$	45.7	50.8	45.8	47.1	55.6	39.1	21.6	45.2
$P(\mathit{bug} \mid \neg \mathit{fix})$	34.1	38.3	36.7	35.5	37.3	50.6	33.9	38.1

Table 6: Distribution of fixes and fix-inducing changes across day of week in MOZILLA

We used fix-inducing fixes to investigate whether non-fixes or fixes are more likely to be fix-inducing. Table 5 shows that for ECLIPSE, the average likelihood of introducing a fix-inducing change is almost three times higher for fixes, indicated by $P(\mathit{bug} \mid \mathit{fix})$, than for regular changes, indicated by $P(\mathit{bug} \mid \neg \mathit{fix})$. This does not hold for MOZILLA (see Table 6). The risk that a fix will be later undone is highest for ECLIPSE on Saturdays, and for MOZILLA on Fridays.

Almost every second change in MOZILLA is a fix and two out of five changes are fix-inducing. In the future we will investigate MOZILLA to find out what makes MOZILLA risky.

Besides the day of week, one can easily determine further properties of a change that correlate with inducing fixes—such as the development group, or the involved modules. Again, all this data is automatically retrieved for arbitrary projects.

6. RELATED WORK

To our knowledge, this is the first work that shows how to locate fix-inducing changes in version archives. However, fix-inducing changes have been used previously under the name *dependencies* by Purushothaman and Perry [9] to measure the likelihood that small changes introduce errors. Baker and Eick proposed a similar concept of *fix-on-fix changes* [1]. Fix-on-fix changes are less general than fix-inducing changes because they require both changes to be fixes.

In order to locate fix-inducing changes, we need first to *identify fixes* in the version archive. Mockus and Votta developed a technique that identifies reasons for changes (e.g., fixes) in the log message of a transaction [7]. In our approach, we refine the techniques of Čubranić and Murphy [4] and of Fischer, Pinzger, and Gall [6, 5], who identified references to bug databases in log messages and used these references to infer links from CVS archives to BUGZILLA databases.

Čubranić and Murphy additionally inferred links in the other direction, from BUGZILLA databases to CVS archives, by relating bug activities to changes. This has the advantage to identify fixes that are not referenced in log messages. For more details about this approach, we refer to [3].

Rather than searching for fix-inducing changes, one can also directly determine *failure-inducing changes*, where the presence of the failure is determined by an automated test. This was explored by Zeller, applying Delta Debugging on multiple versions [11].

7. CONCLUSION

As soon as a project has a bug database as well as a version archive, we can link the two to identify those changes that caused a problem. Such fix-inducing changes have a wide range of applications. In this paper, we examined the properties of fix-inducing changes in the ECLIPSE and MOZILLA projects and found, among others, that the larger a change, the more likely it is to induce a fix; checking for other correlated properties is straight-forward. We also found that in the ECLIPSE project, fixes are three times as likely to induce a later change than ordinary enhancements. Such findings can be generated automatically for arbitrary projects.

Besides the applications listed in Section 1, our future work will focus on the following topics:

Which properties are correlated with inducing fixes? These can be properties of the change itself, but also properties or metrics of the object being changed. This is a wide area with several future applications.

How do we disambiguate earlier changes? If a fixed location has been changed multiple times in the past, which of these changes should we consider as inducing the fix? We are currently evaluating a number of disambiguation techniques.

How do we present the results? Simply knowing which changes are fix-inducing is one thing, but we also need to present our findings. We are currently exploring visualization techniques to help managers as well as programmers.

For ongoing information on the project, see

`http://www.st.cs.uni-sb.de/softevo/`

Acknowledgments.
This project is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Christian Lindig and the anonymous MSR reviewers provided valuable comments on earlier revisions of this paper.

8. REFERENCES

[1] M. J. Baker and S. G. Eick. Visualizing software systems. In *Proceedings of the 16th International Conference on Software Engineering*, pages 59–70. IEEE Computer Society Press, May 1994.

[2] N. Barnes. Bugzilla database schema. Technical report, Ravenbrook Limited, July 2004. http://www.ravenbrook.com/project/p4dti/master/design/bugzilla-schema/.

[3] D. Čubranić. *Project History as a Group Memory: Learning From the Past*. PhD thesis, University of British Columbia, Canada, Dec. 2004.

[4] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.

[5] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[7] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc. International Conference on Software Maintenance (ICSM 2000)*, pages 120–130, San Jose, California, USA, Oct. 2000. IEEE.

[8] *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, UK, May 2004.

[9] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes. In MSR 2004 [8], pages 90–94.

[10] The Bugzilla Team. *The Bugzilla Guide - 2.18 Release*, Jan. 2005. http://www.bugzilla.org/docs/2.18/html/.

[11] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, volume LNCS 1687. Springer Verlag, 1999.

[12] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In MSR 2004 [8], pages 2–6.