The Missing Links: Bugs and Bug-fix Commits

Adrian Bachmann[1], Christian Bird[2], Foyzur Rahman[2], Premkumar Devanbu[2] and Abraham Bernstein[1]

[1]Department of Informatics, University of Zurich, Switzerland
[2]Computer Science Department, University of California, Davis, USA

{bachmann,bernstein}@ifi.uzh.ch
{cabird,mfrahman,ptdevanbu}@ucdavis.edu

ABSTRACT

Empirical studies of software defects rely on links between bug databases and program code repositories. This *linkage* is typically based on bug-fixes identified in developer-entered commit logs. Unfortunately, developers do not always report which commits perform bug-fixes. Prior work suggests that such links can be a biased sample of the entire population of fixed bugs. The validity of statistical hypotheses-testing based on linked data could well be affected by bias. Given the wide use of linked defect data, it is vital to gauge the nature and extent of the bias, and try to develop testable theories and models of the bias. To do this, *we must establish ground truth*: manually analyze a complete version history corpus, and nail down those commits that fix defects, and those that do not. This is a difficult task, requiring an expert to compare versions, analyze changes, find related bugs in the bug database, reverse-engineer missing links, and finally record their work for use later. This effort must be repeated for hundreds of commits to obtain a useful sample of reported and unreported bug-fix commits. We make several contributions. First, we present LINKSTER, a tool to facilitate link reverse-engineering. Second, we evaluate this tool, engaging a core developer of the APACHE HTTP WEB SERVER project to exhaustively annotate 493 commits that occurred during a six week period. Finally, we analyze this comprehensive data set, showing that there are serious and consequential problems in the data.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product Metrics, Process Metrics*

General Terms

Experimentation; Measurement; Verification

Keywords

case study; apache; bias; tool; manual annotation

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.
FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...$10.00.

1. INTRODUCTION

Software process data, especially *bug reports* and *commit logs*, are widely used in software engineering research. The *integration* of these two provides valuable information on the history and evolution of a software project. It is used, *e.g.,* to predict the number and locale of bugs in future software releases (e.g., [27, 31, 17, 6]). The two data sources are normally integrated by scanning through the version control log messages for potential bug report numbers; conscientious developers enter this information when they check-in bug fixes (e.g., see [14]). We used similar techniques in our previous work, and, in fact, improved current practice by adding heuristics to check the results [3, 4]. Even so, the links (between program code commits and bug reports) thus extracted cannot be guaranteed to be correct, as they are reliant on voluntary developer annotations in commit logs.

In prior work, we have shown that such data sets are plagued by quality issues [4]; furthermore, these issues (e.g., incompleteness, bias, etc.) adversely affect applications and algorithms which rely on such data [10]. We defined two types of bias: *bug-feature bias*, where only the fixes of certain types of defects are linked, and *commit-feature bias*, where only the certain kinds of fixes, or fixes to certain kinds of files, are linked. In addition to these data quality issues, many researchers make questionable process assumptions: for instance they assume that all the relevant bugs of a software product are actually reported in the bug tracking database of the project. To truly understand defect-reporting bias and verify such assumptions, we must uncover the *ground truth*: we must analyze completely (at least a time-window of) the commit version history of a project, and precisely identify *all* the commits that are defect fixes, and those that are not.

To get at ground truth requires skill, knowledge and effort: one must compare successive versions, understand the changes, identify any relevant reported bugs in the repo, and establish a link when possible. This process must be repeated until we have a large enough sample for statistical analysis. This is costly, difficult, and time-consuming.

Linkster is a convenient, interactive tool, integrating multiple queryable, browseable, time-series views of version control history and bug report history. LINKSTER enables an expert to quickly find and examine relevant changes, and annotate them as desired; specifically, LINKSTER makes it easy to find defect-fix commits. We engaged an expert APACHE core developer, Dr. Justin Erenkrantz, to use LINKSTER to manually annotate 6 full weeks (including 493 commit messages) of the APACHE history. This case study helped us to

improve the tool, and yielded a trove of data to examine three research questions.

Traditionally, researchers have made several assumptions about the bug fixing, reporting, and linking phenomena. The first two research questions reflect general internal validity concerns that arise when using linked bug data for software engineering research.

RQ 1: *Do the bug reporting and fixing practices of developers correspond to the assumptions commonly made by researchers?*

Second, researchers have tended to gloss over the issue of whether automated tools that find links between commits and bug reports have false-positives or false-negatives.

RQ 2: *How well does the automated approach of finding links between commits and bug reports work?*

Finally, the linked set of bug-fixing commits are a sample of the full set of bug-fix commits. We can check and see if this sample is biased in any detectible way.

RQ 3: *Is there any evidence of systematic bias in the linking of bug-fix commits to bug reports [10]?*

To our knowledge, the only published study on this question is by Aranda and Venolia [1]: they analyzed the completeness and degree of truth in software engineering datasets and provided a partial answer to RQ 1 (see Sub-Section 2.2). Most studies do not even address data quality issues [23].

In addition, we were able to qualitatively explore how the Apache project actually uses software engineering tools such as bug tracker and version control systems, yielding some rather surprising observations.

We begin with a discussion of related work (Section 2), followed by an overview of the tools and processes (Section 3) used in Apache HTTP web server project. We then present (Section 4) a description of Linkster, and details of the case study procedure evolving an Apache core developer (Section 5). In Sections 6 and 7 we present our findings, which we summarize briefly below:

Finding 1: A so-called "bug" is not always a bug; neither is a "commit" always a commit. In other words: in Apache, the most important bugs are not handled in the bug tracker but mentioned in the mailing list system; and only a fraction of commits actually pertain to program changes (RQ 1).

Finding 2: We compared the manual annotations with data produced by automated linking (*viz.,* for false-positives or false-negatives); the automated approach finds virtually all the commit log messages which contain a link to the bug tracking database (RQ 2). Sadly, however, many defect-fix commits are un-identified in the commit logs, and thus are invisible to automated approaches.

Finding 3: In the manually annotated sample, we find strong statistical evidence that different bug-fixers vary in their linking behavior. Investigating further, we find anecdotal evidence suggesting that factors such as experience, ownership and the size (number of files) of the commit affect linking behaviour. We also find that reporting bias affects the performance of a bug prediction algorithm (BugCache). Given the small size of the manually annotated sample, the evidence here is mostly suggestive rather than statistically significant; however, it points out the strong need for further studies—for if this type of reporting bias is confirmed as a widespread problem, this is of serious, fundamental concern to all empirical research that uses this type of linked bug-fix data.

2. RELATED WORK

Areas closely related to this research include data extraction and integration, data quality in software engineering, data verification in software repositories, and our own previous work on data quality effects on empirical software engineering.

2.1 Data Extraction and Integration

Software engineering process data such as bug reports and version control log files are widely used in empirical software engineering. Therefore, the extraction and integration of this data is critical.

Fischer *et al.* [14] presented a Release History Database (RHDB) which contains the version control log and the bug report information. To link the change log and the bug tracking database, Fischer *et al.* searched for change log messages which match to a given regular expression. Later, they improved the linking algorithm and built in a file-module verification [13]. A similar approach to link the change log with the bug tracking database was chosen by other researchers. All of them used regular expressions to find bug report link candidates in the change log file (e.g., [32, 31, 30, 33, 34, 30]).

In [3], we presented a step-by-step approach to retrieve, parse, convert and link the data sources. We improved the well-established prior art, enhancing both the quality and quantity of links extracted.

2.2 Data Quality in Software Engineering

As discussed in [10], empirical software engineering researchers have considered data quality issues. Space limitations inhibit a full survey, we present a few representative papers.

Koru and Tian [21] surveyed members of 52 different medium to large size Open Source projects with regards to defect handling practices. They found that defect-handling processes varied among projects. Some projects are disciplined and require recording of all bugs found; others are more lax. Some projects explicitly mark whether a bug is pre-release or post-release. Some record defects only in source code; others also record defects in documents. This variation in bug datasets requires a cautious approach to their use in empirical work. Liebchen *et al.* [22] examined noise, a distinct, equally important issue.

Liebchen and Shepperd [23] surveyed hundreds of empirical software engineering papers to assess how studies manage data quality issues. They found only 23 that explicitly referenced data quality. Four of the 23 suggested that data quality might impact analysis, but made no suggestion of how to deal with it. They conclude that there is very little work to assess the quality of data sets and point to the extreme challenge of knowing the "true" values and populations. They suggest that simulation-based approaches might help.

Bettenburg *et al.* [7, 8, 9] provided first analysis of bug report quality. They investigated the attributes of a good bug report surveying developers and used it to develop a computational model of a bug report quality. The resulting model allowed to display the current quality of a defect report whilst typing. Hooimeijer *et al.* [16] also analyzed the

quality of defect reports and tried to predict whether the defect report will be closed within a given amount of time.

Chen *et al.* [12] studied the change logs of three Open Source projects and analyzed the quality of these log files.

In [4] we surveyed five Open Source and one Closed Source project in order to provide a deeper insight into the quality and characteristics of these often-used process data. Specifically, we defined quality and characteristics measures, computed them and discussed the issues arose from these observation. We showed that there are vast differences between the projects, particularly with respect to the quality of the link rate between bugs and commits.

Aranda and Venolia [1] provided a field study of coordination activities around bug fixing, based on a survey of software professionals at Microsoft. Specifically, they studied 10 bugs in detail and showed that (i) electronic repositories often hold incomplete or incorrect data, and (ii) the histories of even simple bugs are strongly dependent on social, organizational, and technical knowledge that cannot be solely extracted through the automated analysis of software repositories. They report that software repositories show an incomplete picture of the social processes in a project. While they studied 10 bugs in detail, we focus on commit history: we employed an expert, supported by a specially-designed tool to fully annotate a sample of 493 commits. This data helped us uncover a) some of the weaknesses of software repositories as well as b) anecdotal evidence of systematic bias in bug-fix reporting.

2.3 Studying Bias

Papers in empirical software engineering rarely tackle data quality issues directly (see discussion earlier in this section); our earlier work is an exception. In [2] and [10] we investigated historical data from several software projects, and found strong evidence of systematic bias. We then investigated potential effects of "unfair, imbalanced" datasets on the performance of prediction techniques.

Ideally, all bug-fixing commits are linked to bug reports; then empirical research would consider all type of fixed bug reports. However only some of the fixed bugs have links to the bug-fixing commits. This raises the possibility of two types of bias: *bug feature bias*, where only certain types of bugs are linked, or *commit feature bias*, whereby only certain types bug-fixing repairs are linked. Either type of bias is highly undesirable. With access to all the fixed bugs, and the linked bugs, we could check for bug feature bias. Our study [10] suggested that bug feature bias does exist, and also that it affects the performance of the award-winning BugCache defect prediction algorithm [19]. In this work, we have a fully annotated list of commits for the first time, thus achieving "ground truth" for a subset of the Apache dataset, and thus we can analyze the data for commit feature bias.

In summary: a few studies explicitly consider the quality of systematic bias in the data. This study, in contrast, explores the implications of this behavior by attempting to unearth the ground truth by enlisting a core developer to annotate all commits, and thus seek out quality and bias issues.

3. CASE STUDY: APACHE

The Apache HTTP web server is an Open Source software system developed under the auspices of the Apache Software Foundation. Apache is the most popular web server on the Internet, serving over 55% of all websites [26]. Apache is also one of the most popular Open Source projects among researchers. It is widely used in current empirical software engineering research (e.g., [25, 28, 20, 8, 18]), and thus a good subject for an in-depth examination of data quality.

3.1 Project Tools

Like many other Open Source projects, Apache uses the BugZilla[1] bug tracker and the SVN[2] version control system. In addition, the Apache Software Foundation provides officially maintained git[3] mirrors for all projects. The Apache project allows free access to the contents of all these tools. Apache also maintains a public mailing list for developers and Apache users to discuss issues of concern.

3.2 Data Gathering and Integration

We retrieved, processed and linked the Apache HTTP web server process data as presented in [3]. Basically, we downloaded all BugZilla bug reports and SVN version control log files. Then, we scanned each commit log message for indications of fixing a bug using a set of heuristics; typically we look for bug report numbers in log messages. This leads to a set of automatically extracted links between program code commits and bug reports. This set of links is validated using another set of heuristics (*op cit*).

3.3 Apache Dataset

With our own (rather modest) resources, we could only completely evaluate and manually verify a subset of the original Apache dataset. Therefore, we had to sample the original dataset. There were two choices: random sampling or temporal sampling.

Random sampling requires some rationale for selecting a sample—*e.g.,* prior knowledge of the distribution of the relevant co-variates to the study, so that a sample representative of the population could be chosen. It is difficult to decide *a priori* what such co-variates might be, let alone their distribution. So, we chose to perform *temporal sampling.*

Table 1: Apache Datasets: Details

Dataset	Original Dataset	Evaluation Sample
Considered time period	2004-06-18 – 2008-04-25	2005-09-23 – 2005-11-18
#Bug reports	2,409 (100%)	103 (100%)
#Fixed bug reports[4]	559 (23.20%)	23 (22.33%)
#Linked bug reports	256 (10.63%)	10 (9.71%)
#Duplicate bug reports	364 (15.11%)	8 (7.77%)
#Invalid bug reports	766 (31.80%)	38 (36.89%)
#Different bug reporters	1,827	98
#Commit messages (transactions)	8,580 (100%)	493 (100%)
#Empty commit messages	0 (0.00%)	0 (0.00%)
#Linked commit messages	472 (5.50%)	29 (5.88%)
#Different developers	63	23

[1]See `http://www.bugzilla.org/`
[2]See `http://subversion.tigris.org/`
[3]See `http://git-scm.com/`
[4]We define "fixed" bug reports as bug reports that have at least one associated fixing activity (which means a status change to "fixed") within the considered time period.

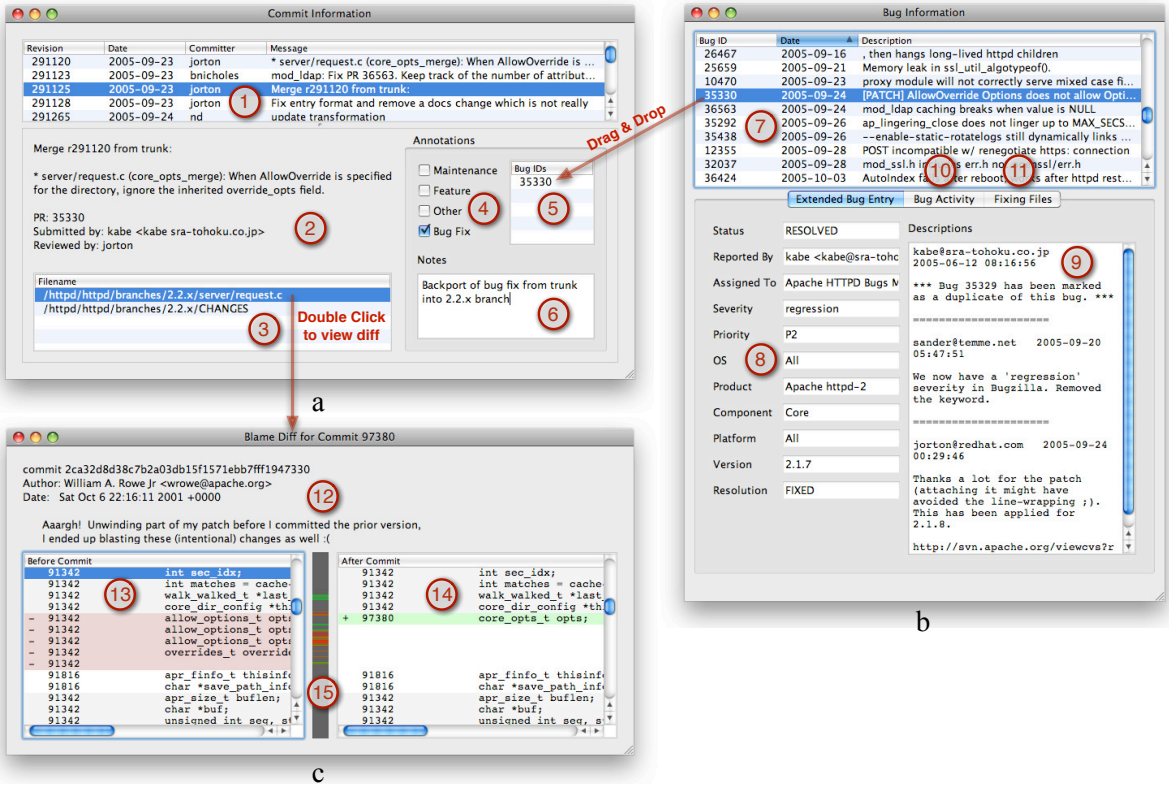

Figure 1: Linkster (Screenshot)

With this approach, we chose to verify *all the commits* in a given period. With complete results for that period, we can then revisit our earlier results and judge the quality against this limited but complete and accurate temporal sample. To find a "typical" period for our evaluation dataset we analyzed the whole original APACHE dataset based on week-long epochs. Then, we chose a period of 6 consecutive weeks that was as representative as possible to the overall original APACHE dataset in terms of its descriptive process statistics (e.g., similar proportions of bugs and commits). Table 1 lists some basic software process statistics for both—the original and the evaluation—APACHE datasets including the finally defined time-frames.

4. LINKSTER

The use of LINKSTER simplified our domain expert's task, greatly accelerating an otherwise tedious, repetitive and inconvenient sequence of invocations of multiple tools.

Figure 1 shows a screenshot of LINKSTER, showing windows containing three kinds of information: commit transactions including all the changed files (a), bug reports (b), and diff & blame information for all of the lines in a file before and after a particular commit (c).

LINKSTER requires access to a version control system for file content and a database (local or remote), containing the raw mined repository and bug tracking information. We use git as our backend repository format, given its increasing popularity [11], and ready availability of tools supporting conversion from competitors such as CVS, SVN, *etc.* However, for convenience, LINKSTER displays the revision IDs from the original repository. All notes, links, and annotations (explained below) made by the user are also recorded in the database to facilitate use and analysis thereof after annotation. LINKSTER efficiently displays, integrates, and allows inspection and annotation of information from all data sources. LINKSTER is written in Python, using the PyQt widget toolset and has been written with portability in mind. We have successfully run it on Linux, OS X, and Windows.

To our knowledge, no other tool provides integrated project information in combination with functionality to annotate / link commits. Hipikat [32], which was developed at UBC, is similar in that it creates links between different types of software artifacts. However, these links are based purely on heuristics and Hipikat functions as a recommender system rather than a browsing and annotation system.

Other tools such as EvoLens, softChange, or Shrimp provide only part of the functionality, but all existing tools have goals other than expert commit annotation.

SoftChange [15] is a tool to aid software engineering research by visualizing data. Similar to LINKSTER, SoftChange integrates data from multiple sources such as version control systems, releases, and bug databases. However, softChange uses visualizations (usually plots) to answer questions,(*e.g.,* how many bugs are closed in each time period?) and does not allow annotation of data as LINKSTER does.

EvoLens [29] helps developers to understand the evolution of a piece of software by visualizing the software as well as metrics of the software over time. The visual nature across time facilitates identifying design erosion and hot spots of activity. LINKSTER does not leverage advanced visualization techniques and integrates multiple types of data rather than just source code information.

Shrimp [24] integrates and visualizes source code, docu-

mentation (Javadoc), and architectural information to aid source code exploration. LINKSTER is more concerned with *process* related artifacts, (*e.g.,* changes, discussions, bug reports, and fixes) than understanding the source code itself.

4.1 Commit Information

Figure 1-a shows the *Commit Information Window* of LINKSTER. The top (**1**) contains a list of commits that satisfy some query, *e.g.,* commits within a time window or changes made by a particular author. Each line shows the revision identifier (as used in the original repository), commit time, author, and the first line of the commit message. The entire commit message is shown in a tooltip when the mouse hovers over an entry.

When a commit entry in the list is selected, the metadata is updated in the bottom half (**2**). The list of files modified in the commit (**3**) is also displayed. Double clicking a file brings up the *Blame & Diff Information* for the file allowing the user to examine the exact changes that were made. For annotation purposes, the user may select the reason(s) for the commit by checking boxes (**4**) or drag and drop (or remove) a bug record from the *Bug Information Window* into the list of *bug IDs* (**5**), which is populated with the set of automatically identified links between the commit and bug records. Finally, the user may enter free form notes for the commit (**6**).

4.2 Bug Information

Figure 1-b contains the *Bug Information Window*. The top portion (**7**) is a scrollable list of bugs from the bug database. Each entry contains the bug ID, the date of creation, and a one line summary of the bug. Hovering over an entry shows the bug severity in a tooltip. Any of these entries may be dragged to the *bug IDs* list (**5**) in the commit information window to indicate a commit that is associated with the bug.

Selecting a bug entry populates the bottom half of the window with detailed information. The left side (**8**) contains short attributes of the bug, while the right side (**9**) displays the full bug description followed by all of the comments in chronological order with author and date. Clicking on the *Bug Activity* tab (**10**) displays a list (not shown) of all changes to the bug record, such as assigning the bug to a developer or marking a bug as closed. Each entry indicates when the change was made and who made it along with old and new values for the changed field as appropriate. Finally, clicking on the *Fixing Files* tab (**11**) presents a list (not shown) of all of the commits to files that are associated with the fix of the bug. This list is comprised of files automatically or manually linked to the bug. Double clicking on any file in this list will bring up a blame & diff window for the commit.

4.3 Blame & Diff Information

Figure 1-c shows the *Blame & Diff Information Window* for the changes to a file in a particular commit. The left view (**13**) shows the content of the file prior to the change, and the right view (**14**) shows the content after the change. Removed lines are prefixed with "−" and are highlighted red, and added lines are in green with a "+" prefix. Each line is also prefixed with revision identifier of the commit that introduced the line. Selecting a line highlights all other lines introduced in the same commit, and also updates the metadata area (**12**) with information about that commit. This can help the user learn why, when, and by whom, the line was originally added. If additional information is desired, double clicking a line will bring up a new *Blame & Diff* window for the commit which introduced the line (if, for example, one desires to see why a line that was removed in one revision was originally added in a prior revision). An annotator can, thus, gradually step back through version history.

The views are synchronized such that scrolling up, down, left, or right in one view causes the other to change accordingly. The thumbnail view (**15**) graphically shows the differences for the entire file with red indicating removed lines and green, added lines. Clicking on a location in the thumbnail view will cause the pre and post views to jump to that location, making it easier to identify and examine changes in larger files.

5. APACHE DATA EVALUATION

To address our research questions, we began our evaluation with the creation of an evaluation dataset, as defined in Section 3.3. Armed with LINKSTER to facilitate browsing and annotation, we engaged the services of an informant: an experienced APACHE developer, Dr. Justin Erenkrantz, to manually annotate a temporal sample of commits using LINKSTER. Clearly, the quality of this completely annotated evaluation dataset is predicated on the expertise of the annotator. Justin is a core developer of the APACHE HTTP WEB SERVER project (since January 2001), the President of the Apache Foundation and serves on the Foundation's Board of Directors. He also develops for Apache Portable Runtime, Apache flood and Subversion[5].

Using LINKSTER, Justin annotated each commit, to flag it as a *bug fix*, an implemented *feature request*, a *maintenance* task or *other*. With this information, we obtain fully annotated commit data, providing a complete picture of all the changes during the given period and how/why/by whom these changes were made. This data can be used to verify our automated linking approach (which includes mainly bug fixes and some feature requests). Indeed, annotating program code commits dating back months or years in the past is a challenge, even for an experienced core developer like Justin. LINKSTER was very helpful, providing an integrated view of all the relevant information. Based on the log message, the changed files and the file diffs of the changed files, Justin was able to annotate all commits, and, in most cases, provided additional information about the commits.

Justin's familiarity with the APACHE project gives us confidence that the results of our evaluation can be trusted. In addition, detailed discussions and interviews with him revealed facts about the tools and processes used in the APACHE HTTP WEB SERVER project, and also ideas for improving LINKSTER.

6. RESULTS

All 493 commits in our selected temporal sample were annotated. In addition to the annotation into the four categories above: *bug fix, feature request, maintenance/refactoring,* and *other,* our informant helped us further sub-classify the commits. Table 2 summarizes the annotation results including the sub-classification. Note, a single commit can have many annotations, *e.g.,* a commit may be annotated as both a "bug fix" and a "feature request".

[5]See `http://www.erenkrantz.com/` for more details.

Table 2: Linkster Commit Categorization (non-exclusive)

Category	Sub-Category	#Commits
Bug fix	–	82
Bug fix	Bug report	32
Bug fix	Bug report (merge)	7
Bug fix	Mailing list	13
Bug fix	Backport	13
Bug fix	Other	17
Feature request	–	54
Feature request	Documentation	7
Feature request	Backport	14
Feature request	Other	33
Maintenance	–	49
Maintenance	Documentation	5
Maintenance	Backport	5
Maintenance	Other	39
Other	–	356
Other	Documentation	156
Other	Backport	49
Other	Non-functional	30
Other	Release	44
Other	Voting	26
Other	Other	51

Based on Justin's insights into the APACHE development process, we developed a second, orthogonal categorization that was more consistent with the procedures within the project (Table 3). In contrast to our categorization, this one assigns each commit exclusively to one of its process-specific categories: *backport/forward port, security fix, bug fix, documentation, voting, release,* or *other.*

Table 3: Process Specific Commit Categorization (exclusive)

Category	#Commits
Backport / Forward port	79
Security fix	7
Bug fix	69
Documentation	158
Voting	26
Release	44
Other	110

In the following sub-sections, we present our findings relative to the research questions presented in Section 1. We also present additional findings based on interviews with Justin.

6.1 Bugs Incognito

Contrary to conventional wisdom, participants of the APACHE project *do not report all the bugs solely through BugZilla.* We found that developers and professional users also make use of the APACHE mailing list to report bugs and provide bug fixes (sometimes at the same time) without reporting them in the bug tracker.

FINDING 1. *Not all fixed bugs are mentioned in the bug tracking database. Some are discussed (only) on the mailing list.*

As shown in Table 2, we have 82 bug fix related commits in our evaluation dataset. 32 of them (bug report) are directly related to the bug tracking database. 7 other commits contain a bug-fix, but are not the initial bug fix commit rather than a merge of versions which contain bug fixes indirectly (bug report (merge)). This means, that only 47.6% of bug fix related commits ($\frac{32+7}{82}$) are documented in the bug tracking database. For 13 other commits (16% of total) identified by Justin as bug fixes, there are related discussions in the APACHE mailing list. This leads to the discouraging observation that many bugs never appear in the bug tracking database, but rather are *only* discussed on the mailing list. Such a discussion often includes the bug fix provided by a non APACHE core developer. According to Justin, these bugs are often the very important bugs especially because of the high attention by APACHE developers and the core community on the mailing list. Note also that reporting some types of bugs (e.g., security related ones) on the mailing list is a practice explicitly requested by the APACHE Foundation[6].

Unfortunately, even knowing about the mailing list bugs, it is hard to i) identify and ii) automatically mine them or extract information similar to a bug report stored in the bug tracking database (such as status changes, priority, severity, etc.). APACHE SVN revision #291558 (see Figure 2), for instance, is related to a bug discussed on the mailing list[7]. If one were to inspect the mailing list message, one would find almost no evidence that this was a bug fix.

Finally, Justin found 17 other bug-fixing commits (21%) which have neither an associated bug report or mailing list message. This phenomenon, of under-reporting of bugs, is a big problem. If important bugs are excluded from experimental data (i.e., many bugs are left out) then the effectiveness of defect prediction models and the validity of statistical studies (which rely on them being in the bug tracking database) may be threatened. This leads to the conclusion, that not all fixed bugs are reported as bugs in the bug tracking database, or in other words: bugs go "incognito".

6.2 Backport Incognito

In the APACHE HTTP WEB SERVER project only a few developers are allowed to commit to an APACHE release version: thus a bug-fix on one release may actually have to be committed by someone else to an older or different release. Typically, this process works as follows. First, a developer fixes a given bug and commits the new version to the current version under active development (also known as the "trunk"). Ideally s/he also refers to the related bug report in the commit log. Next, at least two other developers review the changed code, verify the changes and vote either for or against the fix (this step is related to the voting commits as shown in Table 2 and 3). Finally, if the votes are positive, the fix is committed (or merged) to APACHE release versions, which is called a backport. As a result of this process, we might find several different commits in the version history, that fix the same bug.

FINDING 2. *To fix a bug in an* APACHE *release, multiple similar commits by different developers are needed.*

Unfortunately, backport commits are not that easy to identify by existing linking algorithms and heuristics; frequently, while the log message for original commit to the trunk refers to the bug report, the backport commit log does not. To worsen matters, after the bug is actually closed,

[6]See `http://httpd.apache.org/security_report.html`

[7]See `http://mail-archives.apache.org/mod_mbox/httpd-docs/200509.mbox/%3c200509260627.33737@news.perlig.de%3e`

there is a rigorous review, verification and voting process before the backport is accepted and committed. Therefore, the time difference between the backport commit and the status change (to fixed) on the bug report may rise to several days, which again, makes it difficult to link the bug with the commit. As a result, automated linking algorithms will largely ignore backport fixes. Arguably, these are fixes are very important: often they are involved in post-release failures. They should not be ignored by researchers engaged in hypothesis testing or defect prediction work. Alas, finding them may require extensive, high-expertise combing through commit histories.

```
------------------------------------------------------------------------
r291558 | pquerna | 2005-09-26 06:44:16 +0200 (Mon, 26 Sep 2005) | 2 lines
Changed paths:
   M /httpd/httpd/trunk/Makefile.in

As recommended by nd, build docs for all languages.

------------------------------------------------------------------------
```

Figure 2: Commit message of Apache HTTP web server revision #291558

6.3 Impact-of-Defect vs. Cause-of-Defect

This is a thorny issue: a defect in one project's code base might actually manifest as a failure in a different project. Thus, some of the reported bugs in APACHE HTTP WEB SERVER have their root-cause outside of the APACHE program code. APACHE uses external libraries, as well as Apache Commons modules. Therefore, failures in the APACHE HTTP WEB SERVER, even if duly reported in the APACHE bug tracking database, may actually have to be fixed elsewhere. The reverse is also possible.

The mod-python[8] sub-project maintains its own version control system repository and an APACHE project's main bug tracker independent Jira issue tracker[9]. Mod-python issue 83[10], for instance, was reported in the Jira issue tracker but fixed in the APACHE program code.

FINDING 3. *Developers sometimes fix bugs that are only reported in some other projects' bug tracker, rather than in their own; and vice-versa.*

Ideally, we have a complete, integrated source of all the bugs in the bug repository, and all the fixes in the version control system. Our findings, and indeed, the widespread prevalence of cross-project module reuse, we can expect that this type of separation between causes and effects of defects is quite common. Given this, it would be helpful if a report of a bug *impacting* one system would be transferred to the bug repository of the *causing* system, and linked to fix in the version control of that system. However, given the poor linking behaviour when the cause and effect are in the *same* system, we might expect that this type of cross-system linking is pretty unlikely to occur.

6.4 Commits Incognito

In earlier work [4], we encountered the problem of unexplained commits, *e.g.,* due to empty commit log messages. Sadly, even an experienced developer would find it difficult to retrospectively reconstruct the explanation of an unexplained commit.

[8] http://www.modpython.org/
[9] http://www.atlassian.com/software/jira/ and https://issues.apache.org/jira/browse/MODPYTHON
[10] https://issues.apache.org/jira/browse/MODPYTHON-83

FINDING 4. *Even if we annotate all commits, the cause of a commit still remains unspecified in some cases.*

Table 2 and 3 show the annotation, sub-classification and process-oriented classification of all the commits in our evaluation dataset. Based on the values in Table 3, for 110 commits (22.3%) we have a process specific annotation of other. The reason for these commits, therefore, is not justified by one of the APACHE software engineering core tasks.

In addition, most of the commits are not justified by a bug fix or feature request rather than for documentation (32%), voting (5.3%) or releases (8.9%). Only 37.1% of all commits have a functional impact on the software product (feature requests and bug fixes including all backport), which leads us to the conclusion that not all commits are commits that actually change the software.

For additional information to the quality and characteristics of the version control data, we refer to our previous work presented in [4].

6.5 Performance of the Linking Algorithm

In earlier work [3, 4, 10, 5], we reported a linking algorithm whose performance was found to be best-in-class. The fully annotated data provided the first known oracle to evaluate linking algorithms, and so we evaluated ours.

FINDING 5. *The algorithm (*op cit*) finds most of the commit log messages that the developers linked to bugs reported in the bug tracker, subject to the time constraints used by our algorithm.*

In the chosen temporal sample, our linking algorithm found 29 links between the commit messages and the bug tracking database. Justin also identified all these links; we thus found no false-positive links in our evaluation dataset. In addition to these, Justin found 10 additional links. Seven did not satisfy our heuristic for valid links (time constraint of ± 7 days between commit and status change on the bug report), and so our algorithm rejected them as invalid links. Hence, we found three false-negative links in our evaluation dataset. The seven invalid links resulted from backport commits (as explained earlier, Sub-Section 6.2). These backports corresponded to bug-fix links in the original trunk which in fact, were successfully discovered by our algorithm.

Unfortunately, as we elaborated before, even with a high linking rate between the commit messages and the bug tracker, only a subset of the fixed bugs are considered. Hence, bugs discussed on the mail discussion system are often left out by automated linking approaches.

6.6 Performance of LINKSTER

LINKSTER performed mostly as expected and Justin was able to annotate all the commits (493) of our evaluation sample dataset in one working day. In the discussions with Justin, we found some minor issues, which were promptly remedied. In addition, we found that the most important bugs are discussed in the mailing list system only. Therefore, LINKSTER has been extended to support browsing of messages from development mailing lists and also enables linking them to both bug reports and repository commits.

6.7 Threats to Validity

This sub-section discusses external and internal threats to validity that can affect the results reported in this section.

Threats to external validity. Can we generalize from the results based on the APACHE HTTP WEB SERVER dataset to other datasets? Software engineering tools and processes vary in different projects and, therefore, our findings based on APACHE may not generalize. However, our findings indicate that developers may use software process support tools for various goals not envisioned by its original developers (such as version control systems for voting or mailing list systems for bug reporting). It seems prudent to assume that the APACHE project is not a complete exception and that, therefore, the data used in studies of other projects may also lack important information. Another threat is the use of a single annotator (Justin). Getting the same data annotated by other developers, and checking agreement, would have been better; we hope to do this in future work.

Threats to internal validity. Did we choose our evaluation dataset well, and properly analyze it? We chose our time-frame carefully; however, it may not properly represent the original APACHE dataset. The annotation and classification were performed carefully by a very experienced APACHE core developer. Still, there may be errors. Nonetheless, according to Justin, the interesting practices of the APACHE developers are by no means exceptional to this time period.

7. COMMIT-FEATURE BIAS, REVISITED

The manual annotation effort indicates that many bug fixes are not identified in the commit logs, and thus are completely invisible to the automated linking tools used to extract bug-fix data. Thus the linked bug-fix commits are a *sample* of the entire group of commits. However, samples thus extracted have been central to many research efforts. The natural question is: is this sample representative, or biased? We seek to test for the two kinds of bias: *bug feature bias*, whereby only fixes to certain kinds of bugs are linked, and *commit feature bias* whereby only certain types of commits are linked [10]. Earlier, with access to the entire set of fixed bugs, and the subset of linked bugs, we could check for (and did find) bug feature bias; lacking access to a fully annotated set of commits that tells us which commits are bug fixes, we were previously unable to check for commit feature bias.

Now, with a fully annotated temporal sample of commits, we can indeed check for commit feature bias. Commit features are properties of the file and its revision history, such as size, complexity, authorship, *etc.*. These are critical properties that have been studied in dozens of papers that test theories of bug introductions; they are also the features used for bug prediction. So it is important to test for commit feature bias, and evaluate its impact. In this section, we describe some findings related to commit feature bias, and its effect on a well-known bug-prediction algorithm (BUGCACHE).

We remind the reader that our sample size (despite the time and effort required to gather even that much) is not big enough to realistically expect to find statistically significant support for answers to the questions discussed in this section. However, there are some takeaways: we do find statistical support for the answer to one question, and we do find some anecdotal answers for the other questions. Furthermore, actual bias along any of the lines discussed here would have a highly deleterious effect on the external validity of theories tested using only the linked data. Most importantly, *we hope to convince the reader that such studies are important and need to be repeated and conducted at larger scales.*

7.1 Sources and Extent of Commit Feature Bias

The first question arises naturally from the fact that there are different individual developers, who may have different attitudes towards linking. The simplest and most obvious question is as follows:

Do different developers show significantly different linking behaviour? The anonymized table of developers' linking behavior indicates that this is the case: ($p \simeq 0.002$).

Name	Linked	Not Linked	Name	Linked	Not Linked
a	0	6	b	10	5
c	1	1	d	11	8
e	0	3	f	0	1
g	0	3	h	0	5
i	2	7	j	0	3
k	0	2	l	0	1
m	0	2	n	0	1
o	0	1	p	1	0
q	4	0	Total	26	52

We now hypothesize several different specific possible motivational theories of linking behavior. In several cases, there was a visually apparent signal, in boxplots, albeit none that were statistically significant. The results are shown in Figure 3. We list them below, but we caution the reader to interpret all these findings as at best anecdotal. However, it is important to bear in mind that actual bias influenced by any of the processes hypothesize below would be *very damaging to the external validity of theories tested solely on the linked data.*

Does the experience of the author(s) whose code is being fixed influence linking behaviour? We hypothesized that the quest for greater reputation might incentivize people to link fixes when the code under repair belonged to an experienced (and thus more reputable) person. We measured the fixed code's "author reputation" as the geometric of the prior commit experience of everyone who contributed to the fixed code. The left most boxplot in Figure 3 is weakly suggestive that fixes made to code with more experienced authorship are more likely to be linked.

Does the number of files involved in the bug fix matter? If more files are repaired in a bug fix, perhaps the fix is more "impactful"; this might motivate the fixer to more carefully document the change. In fact, the boxplot (second from left in Figure 3) is suggestive that this might be the case, with all the unlinked fixes being single-file fixes.

Are more experienced bug fixers more likely to link? We might expect that more experienced developers behave more responsibly. We measure experience as the number of prior commits. The boxplot (second from right) suggests support for this theory, with a noticeably higher median for the linked case.

Are developers who "own" a file more likely to link bug-fixes in that file? One might expect that people fixing bugs in their own files are more likely to behave responsibly and link; on the other hand, there is a anti-social reputation-preserving instinct that suggests that they may be *less* likely to link. We measure ownership as the proportion of lines in the file authored by the bug fixer. Indeed, the boxplot visually supports the "anti-social" theory.

We created plots to evaluate two other theories: *Are bug*

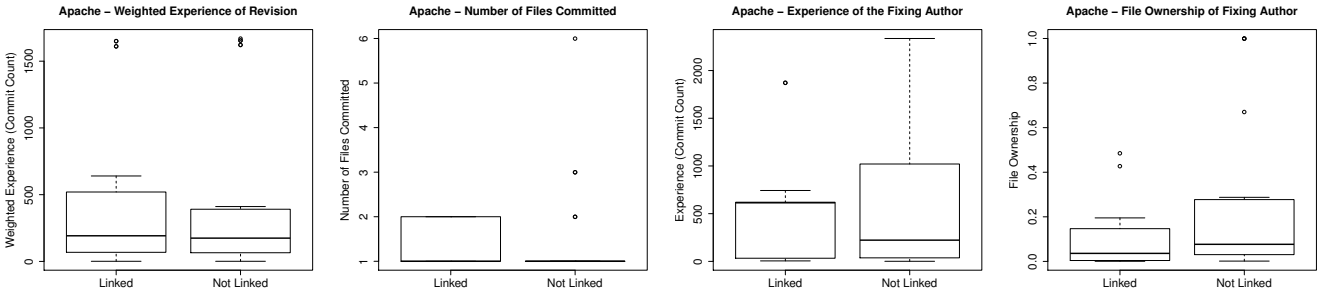

Figure 3: Commit feature bias (reading left to right) weighted experience of the original authors of the fix-inducing code; number of files changed in the bug fix; experience of the author committing the bug fix; proportion of fixed file owned by bug fix author at the time of the bug fix.

fixes to bigger files more likely to be linked? and *Does the prior experience of the file owner influence linking behaviour?* and found no informal visual evidence supportive of these theories.

7.2 Practical Effects: BugCache Revisited

The above analysis shows that the extent of bias in the data is significant and that the effort of finding the ground truth (*e.g.,* through manual annotation with LINKSTER) leads to important insights. *But do those insights translate to practical impact?* In this sub-section we investigate the impact of approaching ground truth in terms of changes in the accuracy of the award-winning BUGCACHE algorithm [19]. To that end, we repeated our experiment showing the impact of bias using APACHE data [10]. Specifically, we departed from two different datasets: The first dataset (called A below) contained all 1576 bugs introduced in the APACHE 2.0 branch. The second one contained the additional 65 bugs found by Justin (called J). Table 4 shows the resulting accuracies for training and predicting on each combination of these two datasets.

Consider training on the extracted data A and predicting on the same data. This provides a baseline accuracy of 0.875. If the prediction is, however, performed on the dataset representing ground truth for the period of manual annotation $A \cup J$ then the accuracy falls to 0.870. We accede that due to the limited manually annotated period the difference—like all the differences in the table—is not significant. But as the following shows we can recognize a tendency. Alternatively, consider adding the manually annotated bugs to the training set (i.e., training on $A \cup J$). In each possible prediction target (i.e., A, J, and $A \cup J$) we find that the availability of the additional information actually leads to an improvement in prediction accuracy. This is especially impressive where the prediction target is A as it shows that the manually annotated bugs actually contain information relevant to the automatically extracted ones helping BUGCACHE to find four additional bugs.

Table 4: BugCache Prediction Quality

Learning Set	Test Set	Accuracy	95% Confidence Interval
A	A	0.875	0.858, 0.890
A	$A \cup J$	0.870	0.852, 0.885
A	J	0.738	0.620, 0.830
$A \cup J$	A	0.878	0.860, 0.893
$A \cup J$	$A \cup J$	0.874	0.857, 0.889
$A \cup J$	J	0.785	0.670, 0.867

8. DISCUSSION AND CONCLUSIONS

In this paper, we analyzed three main research questions and tried to find "ground truth" in the commit annotations of a very popular software engineering dataset. We used temporal sampling to define an evaluation subset of the original APACHE dataset and manually annotated all commits, with the assistance of an APACHE core developer and the use of LINKSTER.

As presented in our previous work, bias in empirical software engineering datasets may affect results of applications which rely on such data [10]. Unfortunately, based on our data verification, we found that things are even worse: our findings cast doubt on some of the core assumptions made in empirical research. Specifically:

1. Bugs often go incognito as they are not always reported as a bug in the bug tracker but, *e.g.*, in mailing lists, and

2. commits not always clearly change the functionality of the program.

Specifically, we showed that not all fixed bugs are reported in the bug tracking database and most of the commits (62.9%) are not related to a bug fix or feature request (which would introduce a program change) rather than for documentation (32%), voting (5.3%), or releases (8.9%). In addition, we presented the curious case of backport commits and the challenging impact-of-defect vs. cause-of-defect problem. Both issues have an impact on software engineering datasets. Consequently, even though automated linkage tools are able to connect a remarkable number of commits to bugs reports, many bugs—sometimes the most critical ones—never show up in the bug tracker and are, therefore, not linked. This raises new issues concerning the validity of studies that rely on version control and bug report data only—beyond what we reported earlier [10]. We presented a detailed examination of the bias in automatically linked set, when compared to the manually linked set. Especially notable is the significant variation in linking behavior among developers, and the anecdotal evidence suggesting that bug-fixing experience and code ownership play a role in linking behaviour. We also showed that BUGCACHE has a strong tendency to miss predictions if it is not trained on ground truth.

Another implication of the work presented here is that empirical software engineering studies will need to take the whole software development social eco-system (revision control system, bug tracking database, mailing list systems, email discussions, discussion boards, chats, etc. as well as these data from other, related projects) into account in order

to elicit a more complete picture of the underlying development process. This would allow to capture the nature of some of the bugs and commits that our informant tediously collected manually.

Nonetheless, this study is only a first step towards quality-approved datasets and we acknowledge that we were only able to verify a small subset of the overall APACHE dataset. Therefore, we hope to influence the community to seek more ground truth for more software engineering datasets. Granted, such work would entail a significant manual labor, but, undoubtedly, the resulting valuable improvements in data fidelity will serve the community well in years to come. We seek mechanisms for fostering this community effort, and welcome suggestions from readers to this end.

Acknowledgment

Many thanks to Dr. Justin Erenkrantz for the time he spent in Zurich annotating commits and providing feedback to the APACHE dataset and LINKSTER. This work was supported by Zurich Cantonal Bank (Bachmann), U.S. NSF SoD-TEAM 0613949 and an IBM Faculty Fellowship (Bird, Rahman, and Devanbu), and Swiss National Science Foundation award number 200021-112330 (Bernstein).

9. REFERENCES

[1] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE'09*, pages 298–308, May 2009.

[2] E. Aune, A. Bachmann, A. Bernstein, C. Bird, and P. Devanbu. Looking back on prediction: A retrospective evaluation of bug-prediction techniques. Student Research Forum at SIGSOFT 2008/FSE 16, November 2008.

[3] A. Bachmann and A. Bernstein. Data retrieval, processing and linking for software process data analysis. Technical Report IFI-2009.0003, Department of Informatics, University of Zurich, May 2009.

[4] A. Bachmann and A. Bernstein. Software process data quality and characteristics - a historical view on open and closed source projects. In *IWPSE-Evol'09*, pages 119–128, Amsterdam, The Netherlands, August 2009.

[5] A. Bachmann and A. Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *MSR'10*, pages 62–71, Cape Town, South Africa, May 2010. IEEE Computer Society.

[6] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *IWPSE'07*, pages 11–18, Dubrovnik, Croatia, September 2007.

[7] N. Bettenburg, S. Just, A. Schroeter, C. Weiss, R. Premraj, and T. Zimmermann. Quality of bug reports in eclipse. In *eTX'07*, pages 21–25, Montreal, Canada, October 2007.

[8] N. Bettenburg, S. Just, A. Schroeter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? Technical report, Saarland University, Saarbrücken, Germany, September 2007.

[9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *ICSM'08*, pages 337–345, October 2008.

[10] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. In *ESEC/FSE'09*, pages 121–130, Amsterdam, The Netherlands, August 2009.

[11] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *MSR'09*, pages 1–10, Vancouver, Canada, May 2009. IEEE Computer Society.

[12] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller. Open-source change logs. *Emp. Softw. Eng.*, 9(3):197–210, 2004.

[13] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *WCRE'03*, pages 90–99, Victoria, B.C., Canada, November 2003.

[14] M. Fischer, M. Pinzger, and H. C. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM'03*, pages 23–32, Amsterdam, Netherlands, September 2003.

[15] D. M. German. Mining cvs repositories, the softchange experience. In *MSR'04*, pages 17–21, Edinburgh, Scotland, UK, May 2004. ACM.

[16] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE'07*, pages 34–43, Atlanta, Georgia, USA, November 2007.

[17] H. Joshi, C. Zhang, S. Ramaswamy, and C. Bayrak. Local and global recency weighting approach to bug prediction. In *MSR'07*, pages 33–34, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society.

[18] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *VL/HCC'08*, pages 82–85, September 2008.

[19] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498, Washington, DC, USA, 2007.

[20] A. J. Ko, B. A. Myers, and D. H. Chau. A linguistic analysis of how people describe software problems. In *VL/HCC'06*, pages 127–134, Washington, DC, USA, 2006.

[21] A. G. Koru and J. Tian. Defect handling in medium and large open source projects. *IEEE Softw.*, 21(4):54–61, 2004.

[22] G. Liebchen, B. Twala, M. Shepperd, M. Cartwright, and M. Stephens. Filtering, robust filtering, polishing: Techniques for addressing quality in software data. In *ESEM'07*, pages 99–106, Washington, DC, USA, 2007.

[23] G. A. Liebchen and M. Shepperd. Data sets and data quality in software engineering. In *PROMISE'08*, pages 39–44, New York, NY, USA, 2008.

[24] J. Michaud, M.-A. Storey, and H. Muller. Integrating information sources for visualizing java programs. In *ICSM'01*, pages 250–258, Florence, Italy, November 2001. IEEE Computer Society.

[25] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[26] Netcraft Ltd. May 2010 web server survey. `http://news.netcraft.com/archives/2010/05/14/may_2010_web_server_survey.html`, May 2010.

[27] M.-T. J. Ostrand, F.-E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[28] J. W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Trans. Softw. Eng.*, 30(4):246–256, 2004.

[29] J. Ratzinger, M. Fischer, and H. C. Gall. Evolens: Lens-view visualizations of evolution data. In *IWPSE'05*, pages 103–112, Lisbon, Portugal, September 2005.

[30] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... In *ICSE'06*, pages 18–20, Rio de Janeiro, Brazil, September 2006.

[31] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR'05*, pages 24–28, Saint Louis, Missouri, USA, May 2005. ACM.

[32] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *ICSE'03*, pages 408–418, Washington, DC, USA, 2003.

[33] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE'07*, pages 1–9, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society.

[34] T. Zimmermann and P. Weissgerber. Preprocessing cvs data for fine-grained analysis. In *MSR'04*, pages 2–6, Edinburgh, Scotland, UK, May 2004. ACM.