If Your Bug Database Could Talk. . .

Adrian Schröter · Thomas Zimmermann · Rahul Premraj · Andreas Zeller
Saarland University
Saarbrücken, Germany
{schroeter|zimmerth|premraj|zeller}@st.cs.uni-sb.de

ABSTRACT
We have mined the Eclipse bug and version databases to map failures to Eclipse components. The resulting data set lists the defect density of all Eclipse components. As we demonstrate in three simple experiments, the bug data set can be easily used to relate code, process, and developers to defects. The data set is publicly available for download.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Process metrics, Product metrics*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*

General Terms
Management, Measurement, Reliability

1. INTRODUCTION
Why is it that some programs are more failure-prone than others? This is one of the central questions of software engineering. To answer it, we must first know *which* programs are more failure-prone than others. With this knowledge, we can search for properties of the program or its development process that commonly correlate with defect density; in other words, once we can measure the effect, we can search for its causes.

One of the most abundant, widespread, and reliable sources for failure information is a *bug database,* listing all the problems that occurred during the software life time. Unfortunately, bug databases frequently do not directly record how, where, and by whom the problem in question was fixed. This information is hidden in the *version database,* recording all changes to the software source code. In recent years, a number of techniques have been developed to relate bug reports to fixes [6, 3, 2]. Since we thus can relate bugs to fixes, and fixes to the locations they apply to, we can easily determine the *defect density* of a component—simply by counting the applied fixes.

We have conducted such a work on the code base of the Eclipse programming environment. In particular, we have computed the mapping of classes to the number of defects that were reported in the first six months before and after release, respectively. We have made this *Eclipse bug data set* freely available, such that anyone can use it for research purposes.

Figure 1 shows an excerpt of the data set in XML format. The file *Plugin.java* had 5 failures (and thus defects) before release 3.0 ("pre"); it had one failure after release ("post"). The enclosing package *org.eclipse.core.runtime* contains 43 files ("points") and encountered 16 failures before and one failure after release 3.0; on average each file in this package had 0.609 failures before and 0.022 failures after release ("avg").[1]

What can one do with such data? In this paper, we illustrate how the data set can be used to address simple research questions:

- Can one predict failure-proneness from metrics like code complexity? (Section 3)
- What does a high number of bugs found during testing mean for the number of bugs found after release? (Section 4)
- Do some developers write more failure-prone code than others? (Section 5)

This paper does not attempt to give definitive answers on these questions, but merely highlights the potential of bug data when it comes to answer these questions. We hope that the public availability of data sets like ours will foster empirical research in software engineering, just like the public availability of open source programs fostered research in program analysis.

2. GETTING BUG DATA
How do we know which components failed and which did not? This data can be collected from *version archives* like CVS and *bug tracking systems* like BUGZILLA in two steps:

1. We identify corrections (or fixes) in version archives: Within the messages that describe changes, we search for references to bug reports such as "Fixed 42233" or "bug #23444". Basically every number is a potential reference to a bug report, however such references have a low trust at first. We increase the trust level when a message contains keywords such as "fixed" or "bug" or matches patterns like "# and a number". This approach was previously used in research [3, 2].

[1] Since one failure can affect several files in one package, the counts on package level cannot be aggregated from file level and therefore are provided separately.

```
<defects project="eclipse" release="3.0">
<package name="org.eclipse.core.runtime">
  <counts>
   <count id="pre" value="16" avg="0.609" points="43" max="5">
   <count id="post" value="1" avg="0.022" points="43" max="1">
  </counts>
  <compilationunit name="Plugin.java">
   <counts>
    <count id="pre" value="5">
    <count id="post" value="1">
   </counts>
  </compilationunit>
  <compilationunit name="Platform.java">
   <counts>
    <count id="pre" value"1">
    <count id="post" value="0">
   </counts>
  </compilationunit>
  ...
 </package>
  ...
</defects>
```

Figure 1: The Eclipse bug data set (excerpt).

2. We use the bug tracking system to map bug reports to releases. The bug database *version* field lists the release for which the bug was reported; however, since the field value may change during the life-cycle of a bug, we only use the first reported release. We distinguish two different kinds of failures: *pre-release failures* are observed during development and testing of a program, while *post-release failures* are observed after the program has been deployed to its users.

Since we know the location of every failure that has been fixed, it is easy to count the number of defects per location and release—resulting in the data set of Figure 1.

3. THE CODE FACTOR

So where do these bugs come from? One hypothesis is that some code is more failure-prone than other because it is more complex. *Complexity metrics* attempt to quantify this complexity, mapping code to metric values. In earlier work on mining Microsoft bug databases [4], we could not find a single metric that would correlate with bug density across multiple projects. Using the Eclipse bug data set, we can easily check this result by correlating, for each class, complexity metrics with the number of bugs.

Chidamber and Kemerer [1] proposed several code metrics that capture the complexity of a class. Table 1 lists the correlations of each of these metrics (gathered using the tool *ckjm* [7]) with pre-release and post-release failures. Albeit weak, the most strongly correlated features[2] to pre-release and post-release failures include RFC (Response for a Class), CBO (Coupling Between Object classes) and WMC (Weighted Methods per Class).

These results are in line with our previous research at Microsoft [4], thus suggesting that either new or a combination of existing metrics need to be explored to study the relationship between the complexity of code to the presence of bugs in a given class. One important predictor might be the *domain* of a component—in related work, we could predict the failure-proneness of an Eclipse package from its imports alone [5].

[2]For detailed explanations of these code metrics, the reader is requested to refer to [1].

Number of	Pre-release failures		Post-release failures	
	Pearson	Spearman	Pearson	Spearman
Pre-release failures	1.00	1.00	**0.26**	0.19
Post-release failures	0.26	0.19	1.00	1.00
WMC	**0.32**	**0.31**	0.16	0.11
DIT	0.07	0.11	0.00	0.01
NOC	0.00	0.04	0.00	0.02
CBO	**0.36**	**0.40**	**0.23**	0.12
RFC	**0.39**	**0.38**	**0.21**	0.11
LCOM	0.13	0.23	0.03	0.07
CA	0.09	0.05	0.02	0.04
NPM	0.20	0.18	0.11	0.09

Table 1: Correlation of pre-release and post-eelease failures with code metrics

Number of	Pre-release failures		Post-release failures	
	Pearson	Spearman	Pearson	Spearman
Pre-release failures	1.00	1.00	**0.30**	0.20
Post-release failures	0.30	0.20	1.00	1.00
Changes	0.34	0.44	0.14	0.15
Changes since 2.1	**0.47**	**0.56**	0.19	0.17
Authors	0.30	0.30	0.15	0.13
Authors since 2.1	**0.41**	**0.49**	0.21	0.17

Table 2: Correlation of process measurements with failures [Eclipse 3.0].

4. THE PROCESS FACTOR

Any problem that raises after product release indicates a defect not only in the product, but also in its *process:* Clearly, the defect should have been caught by quality assurance first. In practice, this may mean that the product was not tested enough. Therefore, we could turn to the *testing process* as a cause for the problem.

Failures during testing are recorded as *pre-release failures* in bug tracking systems. Other measures for the development process are the number of changes and authors of a file. Tables 2 shows how these measurements correlate with each other. For pre-release failures the correlation is highest for the number of changes (0.47) and authors (0.41) since release 2.1. This is not surprising, since every pre-release failure also resulted in at least one change (namely the fix). Post-release failures show almost now correlation with process measurements, except for pre-release failures where the correlation is 0.30. To summarize, it is difficult to predict post-release failures solely from process measurements.

5. THE HUMAN FACTOR

As a third and final example of using the Eclipse bug data set, let us turn to the ultimate cause of errors: humans. Unfortunately, data from one project alone is not enough to judge managerial decisions. However, we can turn to the *developers* and examine whether specific developers are more likely to produce bugs than others.

Tables 3 and 4 summarize pre-release and post-release bug patterns introduced by developers. In both tables, the first column lists the names of developers[3] and the second column lists the number of files owned by the developer. The latter was derived by attributing

[3]Names have been changed to maintain anonymity.

		Failure-densities	
Developer	No. of Files	PrRF / 1000 lines	Avg. PrRF / File
Frederick	320	16.42	2.81
Peter	97	14.70	1.96
Isaac	178	9.95	1.69
Mary	392	9.35	1.84
London	63	9.18	1.41
David	88	8.77	1.64
Harry	55	2.55	1.18
Tommy	92	2.20	0.35
King	162	2.18	0.36
Charles	63	1.82	0.43
Nellie	60	1.14	0.32
Robert	58	0.47	0.17

Table 3: Pre-release failures by developer

		Failure-densities	
Developer	No. of Files	PoRF / 1000 lines	Avg. PoRF / File
Jack	54	0.71	0.13
London	63	0.52	0.08
Queen	111	0.51	0.20
Edward	55	0.41	0.04
Samuel	67	0.39	0.12
Tommy	92	0.34	0.05
Alfred	152	0.03	0.01
Oliver	106	0.03	0.02
Frederick	320	0.02	0.00
King	162	0.00	0.00
Benjamin	119	0.00	0.00
George	52	0.00	0.00

Table 4: Post-release failures by developer

the file to the developer(s) that owned most number of lines of code in a file and only those developers that owned 50 or more files were included in the analysis. Columns 3 and 4 record the number of pre-release and post-release failures per 1000 lines of code and the average number of pre-release and post-release failures per file. For brevity, only the first and last six entries of each table are reported.

In Table 3, one observes substantial differences in pre-release failure densities in files (indicated by Columns 3 and 4) between different developers. However, such results should be carefully interpreted. We suspect that the results do not indicate developer competency but instead, reflect the complexity of code they are working on. Hence, developers with lesser pre-release or post-release failures are not necessarily better developers that the others. Our stance is further supported by there being no clear relation between the number of files owned by a developer and the corresponding failure densities observed since experienced and better programmers may own more files.

Likewise, Table 4 again indicates a high variance in failure density in files owned by different developers, although the densities are smaller in comparison to pre-releasure failures. It is noteworthy that developer Frederick lists in Table 3 as the owner of the files with highest pre-release failure density, while in Table 4, the same developer is the owner of nearly failure free post-release files. In contrast to Frederick, files owned by Tommy are less pre-release failure prone while the post-release failures are considerably higher.

Hence, different developers are likely to introduce different number of failures into the code for manifold possible reasons. We consider such information to be only the tip of the iceberg indicating directions for future investigations pertaining to the human factor in software development.

6. CONCLUSION AND CONSEQUENCES

Where do bugs come from? By mapping failures to components, the Eclipse bug data set offers the opportunity to research these questions. Our initial studies, as shown in this paper, do not give a definitive answer. However, they raise obvious follow-up questions and indicate the potential of future empirical research based on such bug data. To support this very research, we are happy to make the bug data set publicly available.

Overall, we would like this set to become both a *challenge* and a *benchmark:* Which factors in programs and processes are the ones that predict future bugs, and which approach gives the best prediction results? The more we learn about past mistakes, the better are our chances to avoid these mistakes in the future—and build better software at lower cost.

For access to the Eclipse bug data set, as well as for ongoing information on the project, see

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

Acknowledgments. Our work on mining software repositories is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Thomas Zimmermann is additionally funded by the DFG-Graduiertenkolleg “Leistungsgarantien für Rechnersysteme”.

7. REFERENCES

[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[2] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[3] 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.

[4] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering (ICSE 2006)*. ACM, May 2006.

[5] A. Schröter, T. Zimmermann, and A. Zeller. Predicting failure-prone components at design time. In *Proceedings of the 5th International Symposium on Empirical Software Engineering (ISESE 2006)*. ACM, Sept. 2006.

[6] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, U.S., May 2005.

[7] D. Spinellis. *Code Quality: The Open Source Perspective*. Addison Wesley, 2006.