Don’t Program on Fridays!
How to Locate Fix-Inducing Changes

Jacek Śliwerski Thomas Zimmermann Andreas Zeller
Saarland University
Department of Computer Science
Saarbrücken, Germany
{sliwers,zimmerth,zeller}@st.cs.uni-sb.de

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 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 ECLIPSE history, it turns out that fix-inducing changes are most frequent on Fridays.

1 What are Fix-Inducing Changes?

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.

Recent research investigated changes that *fixed* problems. In contrast, we identified those changes that *caused* problems [4]. The basic idea is as follows:

1. We start with the changes from the version archive, that are associated with a fix. This gives us the *locations* of the fix, i.e., the affected lines.

2. We determine the *earlier changes* at these locations that were applied before the bug was reported.

These earlier changes are the ones that *caused* the later fix. We call such changes *fix-inducing*.

2 The Technique in a Nutshell

Our approach consists of two steps: identify fixes and locate changes that induced these fixes. Both steps are performed automatically.

Identify Fixes

We identify fixes based on the log messages that are supplied with a change. There are two approaches for this step: looking for *keywords* as introduced by Mockus and Votta [3] and looking for *references to bug databases* as introduced by Fischer et al. [2] as well as Čubranić and Murphy [1].

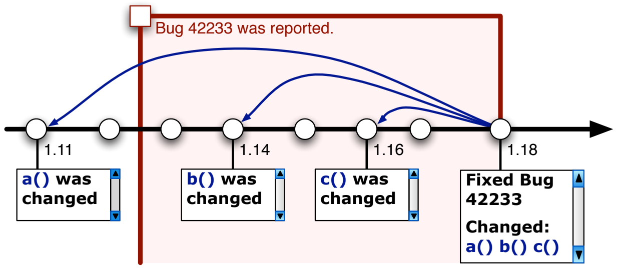

Figure 1: Locating fix-inducing changes for bug 42233

For instance, in Figure 1 both approaches would recognize revision 1.18 as a fix, because of the keywords “Fixed” and “Bug” and because of the reference to the bug database “42233”.

However, our approach relies on the connection to a bug database because we get additional information about fixes. For instance, we use the bug report date to distinguish between several fixes within the same change or to recognize false positives.

Locate Fix-Inducing Changes

Once we know that revision 1.18 is a fix, we annotate each line of the preceding revision 1.17 with the most recent author and revision that touched this line. Figure 2 shows the output of such an annotation. Right now, we use the CVS *annotate* command, but it is straightforward to implement a similar feature for other version control systems.

By computing the differences between revision 1.17 and 1.18, we get the lines that have been changed by the fix. For these lines, we use the annotations for 1.17 to find *candidates* for fix-inducing changes. In our example, we assume that lines 20, 40, and 60 have been changed. Thus our candidates are 1.11, 1.14, and 1.16.

Finally, we use the bug database to rule out changes that cannot be fix-inducing because they have been made after the bug was reported, i.e., they are no real causes for the bug. In our example, revisions 1.14 and 1.16 are not fix-inducing. Without the connection to the bug database, they would be false positives.

Thus, revision 1.11 is the only fix-inducing change in our example. Frequently, but not always, such a fix-inducing change introduced the bug that has been fixed. However, they always are unstable and thus risky changes.

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

Figure 2: CVS annotations for Foo.java

	Day of Week							
in %	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
$P(\mathit{bug} \mid \mathit{fix})$	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 1: Distribution of fixes and fix-inducing changes across day of week in ECLIPSE

3 Don't Program on Fridays

In a first case study, we have broken down fixes and fix-inducing changes by the day of the week when they were applied. Table 1 presents the results for ECLIPSE. The amount of fixes on a day is indicated by $P(\mathit{fix})$. It turns out that most fixes are performed on Friday; Saturday and Sunday are the days of week with the lowest amount of fixes. The amount of fix-inducing changes is indicated by $P(\mathit{bug})$; it is highest on Friday.

Table 1 shows that for ECLIPSE, the average risk of introducing a fix-inducing change is almost three times higher for fixes, indicated by the conditional likelihood $P(\mathit{bug} \mid \mathit{fix})$, than for regular changes, indicated by $P(\mathit{bug} \mid \neg\mathit{fix})$.

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.

4 Perspectives

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

Which properties may lead to problems? 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 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 [5], we would like to avoid fix-inducing changes in our suggestions.

For ongoing information on the project, see

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

References

[1] 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.

[2] 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.

[3] 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.

[4] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? On Fridays. In *Proc. International Workshop on Mining Software Repositories (MSR)*, Saint Louis, Missouri, USA, May 2005.

[5] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering (ICSE)*, pages 563–572, Edinburgh, Scotland, May 2004.