Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 55–59
www.elsevier.com/locate/entcs

Where Do Bugs Come From?

Andreas Zeller[1]

Saarland University
Saarbrücken, Germany

Abstract

The analysis of bug databases reveals that some software components are far more failure-prone than others. Yet it is hard to find properties that are universally shared by failure-prone components. 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, and may thus help to find features that predict how defect-prone a component will be.

Keywords: debugging, verification, program analysis, tracing

Both verification and debugging are concerned with finding *defects*—either defects that cause known failures (debugging), or defects that may cause yet unknown failures (verification). Both techniques have seen substantial advances in the past and can be widely automated—each day, it becomes easier to detect and fix defects. But how did these defects come to be in the first place?

Humans, of course. All defects are artifacts, and hence created by humans such as developers, designers, technical writers (or builders of advanced verification and debugging tools, for that manner). But what is it that makes these people err?

To address this question, we started with the software failures themselves. We have mined a number of actual *bug databases*—that is, archives which store problems as reported by users. By relating these problems to their respective fixes, and by relating the fixes to the locations they apply to, we could trace the problems back to the actual software components. This gave us the *defect density* of each component—simply by counting the applied fixes.

In each project, the defect density varied enormously across components. One may wonder whether these measures correlate with common software features, such as size or complexity. In a study at Microsoft [1], we found that for each project, we could find metrics that correlated with defect density—but none of the metrics we found would correlate across all projects.

[1] Email: zeller@acm.org

doi:10.1016/j.entcs.2006.12.029

Packages used	Failure probability
org.eclipse.jdt.internal.compiler.lookup.*	0.8629
org.eclipse.jdt.internal.compiler.*	0.8623
org.eclipse.jdt.internal.compiler.ast.*	0.8409
org.eclipse.jdt.internal.compiler.util.*	0.8175
org.eclipse.jdt.internal.ui.preferences.*	0.7619
org.eclipse.jdt.core.compiler.*	0.7169
org.eclipse.jdt.internal.ui.actions.*	0.6727
org.eclipse.jdt.internal.ui.viewsupport.*	0.6666
org.eclipse.swt.internal.photon.*	0.6600
org.eclipse.jdt.internal.corext.util.*	0.5982
org.eclipse.swt.internal.motif.*	0.5869
org.eclipse.jdt.internal.ui.dialogs.*	0.5666
...	
org.eclipse.ui.model.*	0.1797
org.eclipse.swt.custom.*	0.1760
org.eclipse.pde.internal.ui.*	0.1659
org.eclipse.jface.resource.*	0.1654
org.eclipse.pde.core.*	0.1608
org.eclipse.jface.wizard.*	0.1566
org.eclipse.ui.*	0.1488

Table 1
Good and bad imports (packages) in Eclipse 2.0 (from [3])

Since developers create bugs in the first place, are there developers which create more defect-prone code than others? The answer is yes—but again, it is hard to correlate defect density with specific features of developers. In particular, managers typically know about the strengths and weaknesses of each team member. Hence, it is the most experienced developers that get to do the most risky coding—and it is therefore not surprising that these create the most defects.

So far, we found the best predictor for defect density to be the *problem domain.* This is inspired by a simple observation: *Some problem domains are more failure-prone than others.* For instance, when working on the Eclipse code base, we find working on compiler internals to be much more difficult and error-prone than, say, building user interfaces. This observation holds regardless of developer, programming language, or the complexity of the resulting code.

The domain is implicitly described by *the components that are used.* When building an Eclipse plug-in that works on Java files, one has to import JDT classes; if the plug-in comes with a user interface, GUI classes are mandatory. Table 1 shows how the usage of specific packages in Eclipse impacts failure probability. A component which uses compiler internals has a 87% chance to have a defect that needs to be fixed in the first six months after release. However, a component using user interface packages has only a 15% defect chance.

Again, such numbers can be collected from existing bug and version databases. It also turns out that import relationships are very predictive—in fact, we found the domain, as expressed by imports, to be among the best predictors for software defects [3].

If it is the domain which makes software defect-prone, what is it that makes the

domain defect-prone? Right now, I do not know. I do have a hypothesis, though.

In general, software defects related to the usage of other components come to be by violating the *constraints* of these components. As an example, consider the internal data structures of a compiler, in particular the abstract syntax tree. This internal representation of the program has an enormous number of constraints, reflecting the syntactic and semantic properties of the programming language.

Some of these constraints are being made explicit via the declared structure and types of the abstract syntax tree. Most constraints are *implicit*, though—and thus easy to violate. If I introduce a cycle in an abstract syntax tree, it is no longer a tree, and passing it as an argument to a function that expects a tree will probably cause havoc. Yet, few languages allow me to express the properties of an abstract syntax tree explicitly, which is why such errors remain uncaught at compile time.

In contrast, a GUI offers far fewer chances for something to go wrong. It may be that some interface element is badly placed or plain invisible—but such errors are easily caught at the first test, and hardly escape into production. The constraints to be observed are fewer—and they are explicit, in particular in frequently used standard classes.

My hypothesis is that it is these *implicit constraints* that determine whether a component is defect-prone or not—the implicit constraints of the domain, as expressed by the imported components. The more such constraints we have, and the less explicit they are, the easier it is to violate them.

How to we determine implicit constraints? We must analyze the components for valid usages as well as their users for normal usages. We must find appropriate abstractions to tell valid from invalid usage, or normal from abnormal usage. We must find appropriate measures to tell which constraints are easy to violate, and which ones are not. And we must see whether such measures actually predict the defect-proneness of a module.

Note that I may be completely wrong, and that it is not implicit constraints at all which determine defect density. It may be that certain domains call for a specific complexity in control or data flow, for instance, and that such flow result in more defects. It may also be that specific domains attract specific programmers, which impose their own work standards. There may be other sources of errors that we have not thought about yet. And the most likely outcome is that defect-proneness is a *combination* of multiple factors, some listed above, and some not listed at all.

The good news, though, is that you can contribute finding out where bugs come from. We have made our Eclipse defect density data publicly available for download [2]. It tells for each component of Eclipse how many post-release defects occurred in that component (Figure 1). The challenge, now, is to find out which properties of these components are the best predictors for defect density. This is a challenge for people in several disciplines:

- in *program analysis* (for determining all the base measures),
- in *verification* (for figuring out what the constraints are, and what makes them complex),

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

Fig. 1. The Eclipse bug data set (excerpt).

- in *debugging* (for finding out how these defects came to be), and
- in *design* (for devising new ways to avoid these defects).

In the past, empirical research in software engineering has seen people with lots of ideas and people with lots of data—but while people have always been willing to share their ideas, it was hard to find people who would share their data. We hope that the public availability of data sets like ours will foster empirical research in debugging and verification, just like the public availability of open source programs brought forward research in program analysis. Eventually, we will not only learn where the bugs are, but also where they come from—and what we can do to avoid them in the first place.

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. This work has been made possible by Thomas Ball, Nachi Nagappan, Rahul Premraj, Adrian Schröter, and Tom Zimmermann. The discussions with them have helped to shape the present paper.

References

[1] Nagappan, N., T. Ball and A. Zeller, *Mining metrics to predict component failures*, in: *Proc. International Conference on Software Engineering*, Shanghai, China, 2006, pp. 452–461.

[2] Schröter, A., R. Premraj, T. Zimmermann and A. Zeller, *If your bug database could talk. . .*, in: *Proc. 5th International Symposium on Empirical Software Engineering (ISESE)*, Rio de Janeiro, Brazil, 2006.

[3] Schröter, A., T. Zimmermann and A. Zeller, *Predicting component failures at design time*, in: *Proc. 5th International Symposium on Empirical Software Engineering (ISESE)*, Rio de Janeiro, Brazil, 2006.