Learning from Defect Removals

Nathaniel Ayewah
Department of Computer Science
University of Maryland
College Park, USA
ayewah@cs.umd.edu

William Pugh
Department of Computer Science
University of Maryland
College Park, USA
pugh@cs.umd.edu

Abstract

Recent research has tried to identify changes in source code repositories that fix bugs by linking these changes to reports in issue tracking systems. These changes have been traced back to the point in time when they were previously modified as a way of identifying bug introducing changes. But we observe that not all changes linked to bug tracking systems are fixing bugs; some are enhancing the code. Furthermore, not all fixes are applied at the point in the code where the bug was originally introduced. We flesh out these observations with a manual review of several software projects, and use this opportunity to see how many defects are in the scope of static analysis tools.

1. Introduction

Static analysis tools are an attractive way to find violations of code quality and security requirements. They can analyze a program without running it, and when they operate soundly, they can find all instances in a class of defects [6]. But they also often produce false warnings or warnings that developers do not care about [3]. Many modern tools are not generally sound, but simply put some effort into finding patterns of problems that have been seen before. They use heuristics to minimize the number of false warnings and hone in on warnings that are likely to be interesting to developers. FindBugs is one such tool, with over 150 detectors written to find about 400 patterns of defective behavior.

In the FindBugs project, we generally discover new patterns by observing real (and usually simple) mistakes made by developers. We can find these mistakes by reviewing source code in software configuration management (SCM) systems. But we may not want to examine every revision or commit; just those that are simple, those that fix bugs and those that introduce bugs. *Bug fixing* commits may be identified by creating links between SCM systems and issue tracking databases such as Bugzilla[1], Jira[2] and Trac[3] [4], [8]. *Bug introducing* commits may be associated with unit test regressions, or may be found by tracking lines changed in bug fixing commits to the point in time when they were previously modified [9], [5], [10]. These previous modifications are sometimes called *fix-inducing* commits.

One problem with linking between SCM systems and issue tracking databases is that not all issues in the databases are associated with fixing bugs; some are requests for new features while others remind developers of needed tasks [1], [7]. Even when a report requests a bug fix, the associated source code changes may include other activities such as adding test cases or refactoring code, and so cannot always be regarded as bug-fix changes. And even when the source code changes are fixing bugs, the fixes may not have been directly induced by a previous modification of the same lines, but may instead have been caused by an API change or induced by a separate bug fix. In other words, not all modifications that directly precede bug-fix commits should be regarded as bug introducing or fix-inducing.

In this study, we illustrate these problems with examples found by reviewing bug reports and associated source code from three different projects. Our review suggests that applications and algorithms identifying fix-inducing commits may need to be more selective in how they choose bug-fixing commits because sometimes it is unclear whether the fix could have been prevented by more careful programming of the changes that previously modified the affected lines. While reviewing the three projects, we also look for problems that could have been detected by static analysis and evaluate real static analysis warnings that are removed to determine if they were fixed directly or removed incidentally. We observe unsurprisingly that only a few changes are in the scope of static analysis and many warnings were removed because their supporting context changed.

2. Reviewing Defects

We manually reviewed bug reports and source code changes from three projects, including two written primarily in Java, and one written in Python[4]. This manual review

1. http://www.bugzilla.org
2. http://www.jira.com
3. http://trac.edgewall.org
4. http://python.org

is subjective, but it enables us to find examples and make qualitative observations. We reviewed reports and associated code changes together, categorizing them as *defects*, *enhancements* or *refactorings*. We reserved the defects category for cases involving bug-fix commits, where developers are fixing problems not just responding to new requirements. In some cases, the issue tracking system contained labels set by users such as "bug" or "improvement". Since our review is conducted for the purpose of identifying bug-fix commits, our designation sometimes differs from the user's; we highlight these cases in our results.

Most of our reviews were on two medium size plugins from different releases of the Eclipse project[5]. We reviewed the Text Editor plugin and the Launching plugin which allows users to compile and run applications on their local JRE. The source code was in a CVS repository and issues were tracked using a Bugzilla database. To link issues to their associated source code changes, we rely on the work of Schröter et al. who mined Eclipse for this information by searching commit messages for bug report numbers [8]. Reports were restricted to those that occurred 6 months before or after a major release, with most occurring before the release.

We also reviewed some source code changes from Groovy[6], a dynamic language implemented in Java, and from CherryPy[7], a framework for building web applications using Python. Both are much smaller code bases, hosted in SVN repositories. Developers in both projects follow established conventions for including issue tracking numbers in commit messages, making it easy to link between the SCM and the issue tracker. In addition, both projects used issue trackers (Jira and Trac respectively) that allowed users to label each report (see Tables 2 and 3 in the results). Finally, we ran static analysis tools (FindBugs and Pylint[8] respectively) over several revisions of each project and noted the warnings that were removed.

3. Results and Observations

Table 1. Categorizing Reports and Commits in Eclipse

	Launching Plugin		Text Editor Plugin	
	Reports	Commits	Reports	Commits
Defects	33	80	38	48
Enhancements	21	37	11	11
Refactorings	0	0	3	4

Table 1 summarizes our reviews for the Eclipse plugins. Most reports focused on fixing defects but about 33% of reports were enhancements or refactorings. Tables 2 and 3 summarize our classifications for Groovy and CherryPy reports respectively. Unlike the Eclipse results, we only reviewed one commit per report. Our reviews generally matched the labels provided by users, but some reports labeled as defects by users were categorized as enhancements or refactorings. For example, CherryPy issue #600 calls for splitting some functionality into two classes to make a feature more manageable[9]. It is labeled a defect, but we categorize this as an enhancement. In the following sections, we make qualitative observations about the different categories of reports.

Table 2. Categorizing Reports in Groovy

Jira Label	Defect	Enhancement	Refactor
- Bug	23	7	0
- Improvement	2	21	4
- New Feature	0	1	0
- Task	0	1	1

Table 3. Categorizing Reports in CherryPy

Trac Label	Defect	Enhancement	Refactor
- Defect	29	13	0
- Enhancement	0	5	1
- Task	1	0	1

3.1. Enhancements

As was noted earlier, some bug reports and tickets refer to requested enhancements to support new user requirements, accommodate changing requirements, or in some cases, mitigate previously unforeseen circumstances. These sorts of reports end up in the bug database because developers use it to communicate with each other. In the Groovy project, a large number of reports fit into this category, suggesting that the culture of the team is to rely heavily on the issue tracker for project management (instead of communicating over email or managing separate requirements documents).

Sometimes users and developers disagree about whether a problem represents a defect or an enhancement. For example, in one bug report from the Eclipse project a user complains that Eclipse does not correctly support environment variable configuration[10]. Developers initially refuse to support this feature, but the reporter feels strongly and responds:

> "This is a very unfortunate decision. I ask that you reconsider. Inability to support env variable configuration in a launch configurations will continue

5. http://eclipse.org
6. http://groovy.codehaus.org
7. http://www.cherrypy.org
8. http://logilab.org/pylint
9. http://www.cherrypy.org/ticket/600
10. Bug reports are at https://bugs.eclipse.org/bugs/

> to cause (my company) SIGNIFICANT difficulty re: sharing launch configurations. We are still on the fence re: Eclipse usage...this is not going to help the Eclipse case."

Developers eventually make changes to support the user's request. Even though the user regards this as a critical problem, we do not classify this as a bug fix for our purposes, because the changes made do not correct an earlier introduced bug.

Other examples of enhancements we reviewed were requests for better error messages, requests for API or structural changes to improve performance, and requests to remove temporary code or "hacks". During the review of Eclipse plugins, we observed that commits associated with enhancements appeared more likely to contain added or removed methods and classes than commits associated with defects. 33% of enhancement commits contained added or removed methods and classes compared to 21% of defect commits. We observed similar proportions in the Groovy project. This could be a clue for future studies seeking to automatically categorize these commits.

3.2. Refactorings

Many refactoring changes involved removing dependence on deprecated features or old libraries. In Eclipse several bug reports called for eliminating unnecessary dependencies between certain components to encourage more loose coupling. Other refactoring changes were made to improve performance. One interesting case in the Groovy project is issue #1915[11] which called for replacing inefficient Java number constructors with invocations of `valueOf()`. This is an existing bug pattern in FindBugs and this change led to the removal of about a dozen warnings.

3.3. Defects

Among reports classified as defects, most were the direct result of an error or oversight by the developer that previously modified the offending code. For example, bug report #29753 for the Eclipse Launching plugin states: "Eclipse keeps a lock on jar files even after a launch configuration is finished". One would reasonably expect that the program should release a resource after using it, so this is classified as a defect. Other examples of defects we reviewed included failing to properly update system configuration in response to user actions, incorrect API usage, memory leaks and out of memory errors.

One source of ambiguity is cases where developers miss important requirements that were not clearly spelled out in advance. If we think the requirement could have been determined by a careful and rigorous software practice, we may classify this as a defect. But if the requirement is only apparent with the benefit of hindsight, it may be viewed as a new requirement. Two illustrative cases from the CherryPy project are issue #588 where developers missed a requirement that is specified in a standard HTTP RFC, and issue #622 where a rare chain of events could lead to errors on the client. Both of these where classified as defects by users, but we classified the second case as an enhancement.

Some cases classified as defects were not directly induced by the previous modification of the same lines. One example from the Groovy project is issue #2606 where an API class is used in ways it was not designed to be used, causing an Exception in some cases. Developers decided to change the API class to prevent the Exception, rather than change the code that was using it incorrectly. Another case from Groovy is issue #2672 where a quirk in an external API was causing a Null Pointer Exception on some platforms. The developer found a reference to this problem in the external API's bug database and applied a hack that indirectly fixed the problem.

3.4. Problems Found by Static Analysis

In our review of Groovy and CherryPy, we ran static analysis tools over multiple sequential revisions and noted the warnings that were removed. Only about 1 in 10 revisions for Groovy and 1 in 5 revisions for CherryPy had removed warnings. Most warnings were removed because the supporting context was changed or deleted. (Supporting context refers to containing methods, classes and packages used to identify warnings; when these are changed it causes warnings to go away and reappear as new warnings). There were a handful of revisions that appeared to directly fix

(a) org/eclipse/ui/texteditor/ConfigurationElementSorter.java, rev 1.5, line 144

```
try {
    manifestElements = ManifestElement.parseHeader(...);
} catch (BundleException e){
    continue;
}

int i=0;
while (i < manifestElements.length && !toTest.isEmpty()) {
    ...
```

(b) org/eclipse/ui/texteditor/ConfigurationElementSorter.java, rev 1.6, line 151

```
try {
    manifestElements = ManifestElement.parseHeader(...);
} catch (BundleException e){
    continue;
}
if (manifestElements == null)
    continue;

int i=0;
while (i < manifestElements.length && !toTest.isEmpty()) {
    ...
```

Figure 1. Code Snapshot showing fixed NPE problem that may be found with Static Analysis

11. http://jira.codehaus.org/browse/GROOVY-1915

warnings but they were low priority or performance issues like the inefficient number constructor issues described in Section 3.2.

During our review, we also looked for changes fixing problems that are similar to the patterns found by static analysis tools. Only a few commits fixed these kinds of patterns (about 5% of the Eclipse plugin changes). We were particularly interested in reports that contained stack traces for Null Pointer Exceptions because many static analysis patterns are devoted to finding these sorts of problems. An example is shown in Figure 1 (which corresponds to Eclipse bug report #70002). The developer inserts a check for null which indicates that the earlier assignment to `manifestElements` may return null. We can study these patterns to see if existing static analysis techniques could have made this determination earlier.

4. Related Work

Several past projects introduce and refine an approach to finding fix-inducing commits that is based on creating a link between the bug report database and the code repository using commit messages [9], [10], [5], [2]. Part of the challenge is to trace back from the fix commit to the fix inducing commit while accounting for changes in file structure including line number and method name changes. Kim et al [5] and Williams et al [10] also manually review the source code of fix commits to decide if they were true fixes, but do not review the bug reports to distinguish between those that refer to defects and those that are enhancements.

Other past projects have tried to develop automatic techniques to distinguish between enhancements and defects in bug reports. Mockus and Votta [7] analyze the textual descriptions in bug reports and use this to classify the changes as adaptive (adding new features), corrective, or perfective (restructuring the code). They also consider how long a bug report is active as well as the number of lines of code changed and use these metrics to validate the consistency of their classification across different projects. Antoniol et al [1] also successfully distinguish corrective maintenance issues from other types of issues using several machine learning techniques. Our study focuses on a narrower class of bug-fixes because we are interested in corrective maintenance on code that was previously modified in ways that induced the later fix. But it is likely that automatic methods may also be able to distinguish between these directly induced bug-fix changes (and associated bug reports) and other indirectly induced fixes and enhancements.

5. Conclusions

Our review of several software projects examines the observation that many SCM changes linked to reports in issue tracking systems are not bug fix changes. This observation impacts efforts to determine which earlier changes may have introduced bugs.

References

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 304–318, New York, NY, USA, 2008. ACM.

[2] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 19–26, New York, NY, USA, 2007. ACM.

[3] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, USA, 2007. ACM.

[4] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[5] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[6] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani, and R. Venkatapathy. Righting software. *Software, IEEE*, 21(3):92–100, May-June 2004.

[7] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[8] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... (short paper). In *Proceedings of the 5th International Symposium on Empirical Software Engineering. Volume II: Short Papers and Posters*, pages 18–20, September 2006.

[9] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[10] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36, New York, NY, USA, 2008. ACM.