The Google FindBugs Fixit

Nathaniel Ayewah
Dept. of Computer Science
Univ. of Maryland
College Park, MD
ayewah@cs.umd.edu

William Pugh
Dept. of Computer Science
Univ. of Maryland
College Park, MD
pugh@cs.umd.edu

ABSTRACT

In May 2009, Google conducted a company wide FindBugs "fixit". Hundreds of engineers reviewed thousands of FindBugs warnings, and fixed or filed reports against many of them. In this paper, we discuss the lessons learned from this exercise, and analyze the resulting dataset, which contains data about how warnings in each bug pattern were classified. Significantly, we observed that even though most issues were flagged for fixing, few appeared to be causing any serious problems in production. This suggests that most interesting software quality problems were eventually found and fixed without FindBugs, but FindBugs could have found these problems early, when they are cheap to remediate. We compared this observation to bug trends observed in code snapshots from student projects.

The full dataset from the Google fixit, with confidential details encrypted, will be published along with this paper.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.2.4 [**Software/Program Verification**]: Reliability

General Terms

Experimentation, Reliability, Security

Keywords

FindBugs, static analysis, bugs, software defects, bug patterns, false positives, Java, software quality

1. INTRODUCTION

Static analysis tools scan software looking for issues that might cause defective behavior. They can quickly find problems anywhere in code without needing to execute it. Their search is exhaustive for some classes of problems and they build on the wisdom of experts to identify problems developers may not be aware of.

ISSTA'10, July 12–16, 2010, Trento, Italy.

Despite this promise, static analysis tools can be challenging to use in practice. Sometimes warnings are found in code that is owned by someone else and the analyst may not understand the context of the warning. Unlike unit testing, some mistakes found by static analysis do not cause incorrect behavior. Even when a warning identifies a real defect, it may not represent a quality dimension an organization is interested in. And sometimes assumptions made by the analysis are incorrect, leading to false alarms; this causes much skepticism among developers and managers.

Many modern static analysis tools have made significant improvements to minimize the number of spurious warnings, including using heuristics, improved analysis techniques, annotations and even symbolic checking or other forms of dynamic analysis. FindBugs, a popular open source static analysis tool for Java, uses finely tuned analyzers called *Bug Detectors* to search for simple bug patterns [7]. These bug detectors contain numerous heuristics to filter out or deprioritize warnings that may be inaccurate, or that may not represent serious problems in practice.

Even with these improvements, we have observed that static analysis tools may receive limited use, depending on how warnings are integrated into the software development process. At Google, we conducted a large scale engineering review, involving hundreds of engineers and thousands of warnings. The goal was to bring potential defects to the attention of engineers, discover which bug patterns were most important to them, and assess the value of FindBugs and other tools.

The chief lesson from this exercise was that engineers viewed FindBugs warnings as worth fixing. Over 77% of the reviews identified the warnings as real defects and recommended fixing the underlying problems. In addition, the engineers' perceptions of the importance of defects generally matched the rankings in FindBugs, and many of the important issues were flagged for fixing. However, and interestingly, none of the serious bugs appeared to be associated with any serious incorrect behaviors in Google's production systems. We found serious bugs in code that had not yet been pushed to production, and in code that was not executed in production systems. And when the buggy code was executed in production, it was often in situations that did not result in serious misbehavior, though we sometimes observed subtle effects such as performance degradation.

We attribute these observations to the success of Google's testing and monitoring practices at identifying misbehavior in productions systems, not to a failure of static analysis. As code is written, it contains a set of coding mistakes,

some of which can be detected by static analysis. Mistakes that cause noticeable problems in unit tests, system tests, or production are generally quickly resolved. Thus, when static analysis is applied to code that has been in production, the only mistakes that remain are those that have not yet caused problems that attract attention. Of the residual issues found by static analysis, some might cause serious problems under some circumstance not yet experienced, but we've found that many of the residual issues are in code that isn't executed in production (and is unlikely to ever be) or is never executed in a way that causes the defect to manifest itself as incorrect behavior. We discuss some of these scenarios in Section 4.1, and conclude that the primary value of static analysis for correctness issues is it can be used early, before testing, and that doing so will identify mistakes more quickly and cheaply than testing would. In addition, static analysis is good at finding certain subtle problems related to security, performance, and concurrency that are not easily found by testing and production monitoring and often do escape into productions systems.

One way to study potentially serious bugs, that are found by static analysis and resolved before they get into production, is to go back through the code history and identify warnings that are removed at significant rates. We study this using code from student projects and from the Google code repository. A quantitative analysis of a large code base may indicate which bug patterns are consistently fixed, even when the developers are not using static analysis. However, we worry about the possibility that some bug removals may be the result of code changes and refactoring that is unrelated to the bug itself. For example, a source file may be deprecated and deleted, causing all contained warnings to disappear. To account for this "code churn", we introduce and experiment with *Noise Detectors*, described in Section 3.3, which insert spurious warnings that are associated with common code structures, not with bugs.

We provide some background on FindBugs and on the process at Google in Section 2; then we describe the methodologies used for the various studies in Section 3, and our results and findings in Section 4.

2. BACKGROUND

2.1 FindBugs

FindBugs is an open source static analysis tool for Java, that relies primarily on simple intraprocedural analysis to find known patterns of defective code. It does not aim to be sound (i.e. find all instances of a class of defect) but rather tries to filter out warnings that may be incorrect or low impact. FindBugs warnings are grouped into over 380 *Bug Patterns* which in turn are grouped into *Categories* such as Correctness, Bad Practice, and Security.

Much of the focus during the development of FindBugs has been on the *Correctness* category, which looks for possible logic errors in the program that can lead to null pointer dereferences, bad casts, infinite recursive loops and other problems. Much tuning goes into these detectors to ensure that users will want to fix at least 50% of the warnings they receive.

Each warning is assigned a priority (high, medium, or low) based on the severity of the associated problem. The priority allows users to compare two issues of the same bug pattern, but cannot be used to compare issues across different bug patterns. To facilitate this latter comparison, FindBugs recently started ranking warnings on a scale from 1 to 20, where 1 is assigned to the "scariest" issues. (For this study, we only consider issues ranked 1 to 12.) This bug rank is subjective and based on our experience reviewing warnings in practice over the last few years. In addition to the severity and impact of the issue, the bug rank factors in the likelihood that the underlying mistake may be quickly found when the code is executed. For example, an Infinite Recursive Loop occurs when a method unconditionally calls itself. We find that in practice, this bug pattern is either found quickly (because the program crashes with a meaningful stack trace), or it occurs in dead code. So we give it a reduced bug rank.

FindBugs can be run in many modes including from the command line, as a stand-alone application (which can also be deployed over the web using Java Web Start), or as a plugin for several popular IDEs and continuous build systems. The stand-alone application includes features to allow users to classify and comment on each issue and file new bugs in a central bug database.

2.2 The Google FindBugs Fixit

Google has been using FindBugs for several years [1] as part of a system called BugBot. BugBot incorporated results from several static analysis tools, including FindBugs, a commercial C/C++ tool, and an in-house Java static analysis tool. Through this system, FindBugs warnings were generally available a day or two after being committed to the version control repository, but developers often had to actively seek them out in order to review them. There were facilities that allowed an engineer to review all of the issues in a particular codebase or product, but the web interface was slow and difficult to use. The results from the commercial C/C++ tool were considered to be of very low quality, and too many of the FindBugs warnings in BugBot were low priority issues. Very few engineers at Google were making active use of the BugBot infrastructure, and in the Fall of 2008 the engineers who had been assigned to BugBot were all assigned to other tasks.

Despite these disappointing outcomes, we still believed FindBugs could provide value to the development process. We decided to coordinate with some engineers and managers to pursue a relaunch of FindBugs, with the following goals:

- Perform a broad review of which issues Google engineers thought were worth reviewing, and keep a persistent record of the classifications of individual issues. We used the techniques implemented in FindBugs and described in [14] to track issues across different builds of the software so that we could identify issues that were new and track reviews of previously seen issues.

- Deploy a new infrastructure that would allow for very efficient review of issues matching specified search criteria. Engineers could search for issues within a particular project, issues that were introduced recently, issues that have a high bug rank, and other reviews of a particular issue.

- Allow FindBugs to be run in continuous builds in a way that could be checked against records of which issues were new and which had already been examined and marked as unimportant. This would allow projects

to choose to have their continuous builds fail when a new, high priority and unreviewed FindBugs issue was introduced into their code base.

- Integrate with Google's internal bug tracking and source code version control system, so that from FindBugs developers could easily file bugs, see the status of bugs that had already been filed against issues, and see the version history of a file.
- Collect copious data from the use of FindBugs so that we could evaluate how it was being used.

On May 13-14, Google held a global fixit for FindBugs (for Java code) and ValGrind (for C/C++ code). Google has a tradition of company-wide engineering fixits [11], during which engineers focus on some specific problem or technique for improving its systems. A fixit might focus on improving web accessibility, on internal testing, on removing TODO's from internal software, etc. The primary focus of the FindBugs fixit was to have engineers use the new infrastructure, evaluate some of the issues found, and decide which issues, if any, needed fixing.

Most of the infrastructure developed for the Google FindBugs fixit was contributed to the open source FindBugs effort. Significant parts of it are specific to Google's internal system (such as integration with Google's internal bug tracking tool), but work is underway to provide these capabilities in a general framework that can be used by other companies and by open source efforts.

3. METHODOLOGY

3.1 Planning the Fixit

The Google fixit was primarily an engineering effort rather than a controlled research study. Engineers from dozens of offices across Google contributed to this effort. Developers were free to choose to review any of the issues, and were given no guidance on how to classify warnings. And while the primary focus of the fixit was over a two day period, a number of engineers had early access to the system, and usage continues, at a lower rate, since the fixit. Nevertheless, this effort provided a rich dataset of user opinions, as well as information on which issues were fixed. The results reported in this paper cover all the data collected through the end of June 2009.

During the fixit, users ran FindBugs from a web interface which launched a Java Web Start instance that contained all the warnings and was connected to a central database. Users could classify each issue using one of the classifications in Table 1, and could also enter comments. Reviews were stored in the database each time the user selected a classification. Users could also easily create an entry in Google's bug tracking system; many fields were populated automatically to facilitate this task.

The FindBugs infrastructure is designed to encourage communal reviews – each user reviewing an issue can see reviews on that issue from other users. However, during the two day fixit, the interface was modified slightly such that a user initially could not see any other reviews of an issue, or whether a bug report had been filed. Once the user entered a review for a particular issue, this information was provided. This setup allows us to assume that reviewers were mostly acting independently when classifying issues.

Engineers were not required to complete a certain number of reviews, but incentives, such as t-shirts for top reviewers, were provided to encourage more reviews. Incentives were also used to encourage users to provide detailed comments exploring the impact of the bug in practice.

3.2 Analyzing the Data

Prior to analyzing the data from the fixit, we anonymized certain confidential details, such as file names, reviewer identities, and any evaluations provided by engineers. Anonymization was done using one-way hashing functions so that it is still possible to group issues from the same file or package, or to identify all reviews by the same engineer.

We also captured the change histories of the files containing warnings, and information about which engineers owned each file. This information allows us to compare the reviews from file owners with those from non-owners. Within Google, any change to a source file requires a code review from someone who is an owner for the file. In general, all developers on a project are owners for all source files that comprise that project.

In the end, this study produced a large dataset with many variables. Most of our analysis focused on looking for correlations between variables, especially with the user classification. In some cases, we can only imprecisely infer the action we are trying to measure. For example, to determine if an issue has been fixed we can confirm that the issue is no longer flagged by the latest FindBugs runs, or we can search the bug tracking system for a corresponding report that is marked as fixed. The former approach would contain false positives, while the latter would contain false negatives.

We describe some of these challenges and considerations in this section.

3.2.1 Comparing Bug Rank with User Opinions

One of our goals is to compare the classifications users provide for an issue with the bug rank of the issue. Are the scariest issues more likely to receive a *Must Fix* classification? We approach this problem by clustering reviews into groups, with all issues in each group having the same bug rank. We can then compute the percentage of reviews in each group that have a particular classification, and correlate these percentages with the bug rank of the group. We use Spearman's rank-order coefficient because the bug rank is an ordinal variable. This method converts values into ranks within the variables before computing the correlation coefficient [5].

We experimented with several approaches to grouping reviews for this comparison:

Group By Issue: In this clustering, we can put all reviews of a particular issue in one group. This provides the finest level of grouping for this method, but can be very noisy since some issues will only receive one or two reviews. We can mitigate this a little, by only considering those issues with more than a threshold of reviews. Grouping at this level is interesting because it separates out each independent issue, and allows us to identify issues that buck the expected trend.

Group By Bug Pattern: This clustering groups all reviews of the same bug pattern and bug rank. Some bug patterns produce issues in different bug ranks, depending on the variety and inferred severity of the is-

sue. Again, grouping at this level allows us to identify bug patterns that have unexpectedly strong or weak user classifications.

Group By Bug Rank: This coarse clustering creates 12 groups, one for each bug rank. This will give us the high level trends describing how bug rank correlates with user classifications.

3.2.2 Determining the Fix Rate

Ultimately researchers and managers at Google would like to see issues get fixed, and understand which groups of issues are more likely to be fixed. This information can influence how warnings are filtered or presented to developers. As we mentioned earlier, it is difficult to get a precise count of the issues that are fixed. We can count the issues that stop appearing in FindBugs runs, but this leads to an overcount since some warnings will be removed by code churn. In Section 3.3, we describe an experimental approach that uses Noise bug patterns to try to separate significant removal rates from code churn. Of course the noise detectors were not used during the fixit, and this technique only applies to our analysis of the Google codebase.

The other approach for computing fix rate is to look for fixes in the bug tracking system. This only applies to data from the fixit. Unfortunately, not all issues fixed during the fixit were tracked in the bug tracking system; developers were not required to use it, and may have quickly fixed simple issues on their own.

In addition to considering the overall fix rate, and the fix rate for individual bug patterns, we are interested in examining different subgroups of issues that we suspect are likely to be fixed at higher rates. Specifically, we group issues in the following ways and consider the fix rates in each group:

By Category: Do issues in the Correctness category have a higher fix rate than other issues?

By Bug Rank: Do the scariest issues have a higher fix rate than other issues?

By Age: Do newer issues have a higher fix rate than older issues?

The last grouping reflects the fact that older issues are more likely to be in code that has been battle-tested. Any significant issues in this code are likely to be removed, and the issues left should largely have little impact on program behavior. Of course, there is no bright line separating old issues from new issues; we simply consider any issues introduced in the six weeks before the fixit as being new.

3.2.3 Checking for Consensus

We would also like to investigate if there is consensus between independent reviews of the same issue. Obviously the classifications made by users are subjective, but if users tend to give similar classifications to each issue, then we have more confidence in their decisions. In past lab studies, we have observed that independent reviewers generally are consistent about how they review issues [2, 3].

Unlike some of our earlier lab studies we do not control who reviews each issue. Some issues have only one reviewer, but one issue has 25, and users choose which issue they want to review. Still the large number of reviews allows us to make some general observations about how often users agree with each other.

Table 1: Grouping and Ordering User Classifications

Classification	*Ord1*	*Ord2*	*Ord3*	*Ord4*
Must Fix	1	2	1	1
Should Fix	2	3	1	2
I Will Fix	3	1	1	
Needs Study	4	4		
Mostly Harmless	5	5	2	3
Not a Bug	6	6	3	4
Bad Analysis	7	7	3	
Obsolete code	8	8		

Another confounding factor is that some of the classifications are very close in meaning and each reviewer may use different criteria to choose between them. For example *Must Fix* and *Should Fix* are close in meaning, and reviewers may have different opinions about which issues are *Mostly Harmless* and which are *Not a Bug*. Other classifications such as *Obsolete code* and *Needs study* are orthogonal to the primary classifications and do not necessarily signal disagreement. (Fortunately there are few of these classifications.) Our method for studying consensus accounts for these problems by grouping the classifications in different ways, using the schemes shown in Table 1. For example, in the *Ord3* ordering, we group *Must Fix*, *Should Fix* and *I Will Fix* classifications into one class, *Mostly Harmless* into another, and *Not a Bug* and *Bad Analysis* into a third; reviews with other classifications are left out of the analysis. (The scheme in Table 1 also represents different ways to order the classifications based on how serious the reviewer may view the problem, an idea we discuss in Section 4.3.3.)

3.2.4 Measuring Review Time

The review time is an important measure when trying to compute the cost of using static analysis. In previous studies, we have observed a relatively low average review time between 1 to 2 minutes for each issue [2]. A large study like this one gives us another opportunity to characterize how much time users spend reviewing issues. Nailing down representative review times is difficult because review times can vary widely for different issues and our users are not starting a stopwatch immediately before each review and stopping it immediately after.

In past studies, we have estimated review time as the time between each evaluation. In this study, this is complicated by the fact that users are not in a controlled environment and may not use the period between each evaluation exclusively for reviewing warnings. They may engage in other work activities, take a phone call, go out for lunch or even go home for the day returning the next day to continue evaluating warnings. A histogram showing the frequencies of review times shows many issues have low review times under 30 seconds, and some issues have very long review times. Closer inspection indicates that some users may have reviewed several issues together, giving their classifications all at once. We chose to filter out review times that were longer than 1 hour. This still left us with about 92% of the review times for analysis.

Another complication is that each time a user selects a classification in the drop down button or enters a comment, a timestamp is sent to our server. So a user can change their classification multiple times during one review, either because they accidentally clicked on the wrong review, or because they genuinely changed their mind. In the data there were 2001 classifications that were duplicates of existing reviews (i.e. the same reviewer and the same issue) usually within a few seconds of each other. To deal with this problem, we filter out many of the duplicate reviews for each issue and person, keeping only the last review, and any preceding reviews that have a *different* classification and occur more than 5 seconds before the review that immediately follows.

3.3 Analyzing Google's Repository

In addition to user reviews from the fixit, we collected and analyzed snapshots of Google's code repository. This data allows us to compare some of the trends extracted from the subjective reviews in the fixit, to more objective measures of which warnings were actually removed, and which ones tend to persist. These measures have been used as a proxy of the relative importance of bug patterns [10, 9].

To conduct this analysis, we detected each warning in each snapshot, and recorded its bug pattern, and the first and last snapshot in which it was observed. As we mentioned earlier, we do not actually know why issues are no longer reported, though we can detect the cases where an issue disappears because its containing source file is deleted. An issue may be removed because it caused a real problem, because someone used a static analysis tool that reported a warning, because a global cleanup of a style violation was performed, or because a change completely unrelated to the issue caused it to be removed or transformed so that it is no longer reported as the same issue. For example, if a method is renamed or moved to another class, any issues in that method will be reported as being removed, and new instances of those issues will be reported in the newly named method. The snapshots used in this analysis were taken between the shutdown of the BugBot project and the FindBugs fixit. Thus, we suspect that the number of issues removed because the warning was seen in FindBugs is small.

To provide a control for this study, we recently introduced new "noise bug detectors" into FindBugs that report issues based on non-defect information such as the md5 hash of the name and signature of a method containing a method call and the name and signature of the invoked method. There are 4 different such detectors, based on sequences of operations, field references, method references, and dereferences of potentially null values. These are designed to depend on roughly the same amount of surrounding context as other detectors. Our hope is that the chance of a change unrelated to a defect causing an issue to disappear will be roughly the same for both noise detectors and more relevant bug detectors. Thus, we can evaluate a bug pattern by comparing its fix rate to both the fix rate over all issues and the "fix" rate for the noise bug patterns.

3.4 Collecting Snapshots from Student Data

In order to more closely observe warnings being introduced and removed during development, we need a finer granularity of snapshots. The most extreme case would be observing developers as they program, capturing each key stroke. A project that comes close to this is the Marmoset project, at the University of Maryland [15]. Over several years, this project has captured snapshots of student development, in addition to providing an avenue for them to submit and test their projects. Specifically, this project has captured snapshots from students learning to program in Java, with snapshots taken every time the student saves a source file, and stored in a source repository. The project also marks those snapshots that were submitted to the central server, indicating when students received some feedback about their performance. Students exhibit different behaviors from professionals, but this dataset can still reveal or confirm expected trends about how warnings are added and removed during development.

3.5 Threats to Validity

The goal of the fixit was to bring issues to the attention of engineers, not to conduct a controlled study. Hence there are internal threats to its validity, including the fact that reviewers were free to choose which issues to review. Some reviewers sought out potential bugs their own code, and were likely more motivated to fix those issues. These threats are unavoidable when conducting a review this big with real engineers, and some the biases are offset by the large size of the dataset. In addition, the fixit and the analysis of Google's code repository may not generalize to other projects or organizations.

The analysis of student snapshots is limited by the fact that we can only approximately infer students' activities; we do not know when they run tests or see exceptions. In addition, the granularity of snapshots may be very different for different students, because we can only analyze compiling snapshots, and some students save often (including incomplete code fragments with syntax errors), while others make substantial code changes between saves. Ultimately, the student projects provide an opportunity to illustrate the introduction and removal of warnings, which we would be unable to do with confidential commercial data.

4. RESULTS

Table 2 overviews some high level numbers from this review. More than 700 engineers ran FindBugs from dozens of offices, and 282 of them reviewed almost 4,000 issues. There were over 10,000 reviews, and most issues (58%) received more than 1 review. Many of the scariest issues received more than 10 reviews each. Engineers submitted changes that made more than 1,000 of the issues go away. Engineers filed more than 1,700 bug reports, and 640 of these had fixed status by the time we stopped collecting data on June 25, 2009. Many of the unfixed bug reports were never assigned to an appropriate individual, which turned out to be a difficult challenge and a key step in getting defect reports attended to.

The choice of which issue to review was left up to the user, so it is interesting to see which issues they chose to review (Figure 1). Reviewers overwhelmingly focused on issues in the Correctness category, reviewing 71% of them compared to just 17% of issues from other categories, which matches our expectations that these are the issues most interesting to users. We identified 288 reviews in which the engineer was identified in the changelist as one of the owners of the file containing the issue; most users were reviewing code they

Table 2: Overall summary

Issues overall	9473
Issues reviewed	3954
Total reviews	10479
Issues with exactly 1 review	1680
Median reviews per issue	2
Total reviewers	282
Bug reports filed	1746
Reviews of issues with bug reports	6050
Bug reports with FIXED status	640

Bug Ranks	Reviews	Must Fix	Should Fix	MH
Scariest 1-4	2482	36.1%	44.3%	4.2%
Scary 5-8	3968	20.0%	48.8%	9.5%
Troubling 9-12	4029	11.3%	60.6%	11.6%
Categories				
CORRECTNESS	8100	23.5%	48.9%	9.1%
BAD_PRACTICE	807	10.2%	79.9%	3.5%
MT_CORRECTNESS	710	10.8%	53.8%	11.1%
EXPERIMENTAL	580	2.6%	71.4%	8.3%
SECURITY	231	30.3%	31.6%	24.2%
STYLE	51	0.0%	11.8%	0.0%

*MH = Mostly Harmless

Figure 1: Recommendations Grouped by Bug Rank and Category

did not own. We talk more about the differences between code reviewed by owners and non-owners in Section 4.4.3.

Figure 1 also shows the percent of reviews that received *Must Fix* and *Should Fix* classifications. A casual glance at the results suggests that scarier issues were more likely to receive a *Must Fix* designation, while lower ranked issues were more likely to receive a *Should Fix* designation. Meanwhile, Correctness and Security issues were viewed as the most serious. We explore these trends in more detail in Section 4.2.

4.1 Evaluating FindBugs

The fixit brought many issues to the attention of developers and managers, and many problems were fixed. In addition, over 77% of reviews were *Must Fix* and *Should Fix* classifications, and 87% of reviewed issues received at least one fix recommendation. But we were surprised that few of the issues actually caused noticeable problems in production. One defect that was identified was committed to the code base on the first day of the fixit, and picked up in the overnight FindBugs analysis. That same night, the defect identified by FindBugs caused a number of internal map reduce runs to fail and an automatic rollback of the change.

This is an example of what we call the "survivor effect", and it illustrates why real defects found in production code often have low impact. When code flies off a developer's finger tips, it contains many bugs, some that matter, and some that do not. The problems that matter usually come up during testing or deployment, causing the software to crash or behave incorrectly. Hence, they are mostly removed as the software matures. The ones that do not matter remain, and

August 3, 12:55pm: adds buggy code

```
    public String getVerticesNames() {
        ...
+       vertices.substring(0, vertices.length()-2);
+       return vertices;
    }
```

12:55 to 1:37 pm (42 minutes): adds code to other methods. Project is failing two local tests. Takes a BREAK for 22 hours.

August 4, 11:59am: attempts to fix. Local tests still failing.

```
-       vertices.substring(0, vertices.length()-2);
+       vertices.substring(0, vertices.lastIndexOf(","));
```

12:01pm: fixes the bug. Local tests now passing.

```
-       vertices.substring(0, vertices.lastIndexOf(","));
+       vertices = vertices.substring(0, vertices.lastIndexOf(","));
```

Figure 2: Bug: Ignoring the Return Value of String.substring()

```
    private LinkedList<Edge<E>> edges;
    public int getAdjEdgeCost(Vertex<E> endVertex) {
        if(edges.contains(endVertex)){
            throw new IllegalArgumentException("Edge already in graph");
        }
        ...
    }
```

Figure 3: Bug: Unrelated Types in Generic Container

the developer is usually oblivious to their existence. Static analysis is good at finding both kinds of problems, and is cheap to run. Furthermore static analysis can find these problems at any point in the development cycle, while testing and deployment can only find them later on, when they are more expensive to fix.

All these observations lead us to conclude that the real value of static analysis for correctness issues is its ability to find problems early and cheaply, rather than in finding subtle but serious problems that cannot be found by other quality assurance methods. Also, static analysis tools generally point to the cause of defective behavior, as opposed to testing which only discloses their symptoms. Of course, early in the process, the developer may not be able to distinguish between problems that matter and those that do not, and so the developer needs to fix all, an additional cost. Still, it often only takes a minute or two to fix problems early on, compared to the hours or days it takes to debug and remediate problems later on.

These observations do not apply to security bugs, which generally do not cause test failure or deployment problems. Static analysis can detect some of these, and often finds serious exploitable problems that have been missed throughout the development process.

4.1.1 *Observations from Student Development*

We would like to see the survivor effect in action; one way to do this is to look at fine grained snapshots of development. To do this, we turn to the data collected from student development by Marmoset. In this dataset we observe that 60% of all warnings introduced during development are fixed before the student's final submission. That number goes up to 90% if we only consider the scariest issues (bug ranks 1 to

Group By	Must Fix	Should Fix	Mostly Harmless
Bug Rank	-0.93	0.79	0.83
Bug Pattern	-0.34	0.1	0.26
Issue	-0.24	0.06	0.08

Figure 4: Correlating Bug Ranks with Reviewer Classifications

4). In some cases, students may have fixed the code in response to a FindBugs warning. Specifically, when students sent a submission to the central server, they got feedback on any FindBugs warnings found, and some students may have responded to this. But many of the cases we examined represented scenarios where the student had to figure out the problem, perhaps because a local unit test was failing before they even made a submission. In the final revision, most of the problems left over had no effect on the project's correctness, especially for students who successfully completed the assignment.

Sometimes a student would notice the problem soon after introducing it, and fix it immediately. About 20% of issues were fixed after 1 snapshot. Other times, the problem would persist for several snapshots before being fixed. A manual investigation of some of these cases reveals that in many cases, the student is advancing the development of their project, oblivious to the problem because they have not tried to run it yet. At some point, they may run local tests that fail; this leads to a period of debugging and the student needs to make a context switch to edit the older code.

An example of this sequence of events is shown in Figure 2. Here the student ignores the return value of `String.substring()`. The student continues development for about an hour, then takes a break and returns the next day without noticing the bug. At some point, it is likely that the student ran local unit tests, and would have noticed that two of them were failing. The process of debugging these failures would have revealed that a critical string contained the wrong value. After iterating through some fix attempts, the student recognizes the problem, almost 24 hours after FindBugs could have flagged it.

An example of one of the warnings left in the final revision for a student that did not affect program correctness is shown in Figure 3. Here the student checks if a container of `Edge<E>` elements contains a `Vertex<E>`. This check will always return false, but it appears to be purely defensive since the student throws a runtime exception when the condition is true.

4.2 Comparing Reviews with Bug Rank

A large fixit like this gives us an excellent opportunity to check if the opinions of engineers match the bug ranks established by FindBugs' developers. Of course, each organization has quality dimensions that it cares about above others, and the bug rank should be tailored accordingly. Here, we compare the engineers' reviews to the default ranking in FindBugs.

Figure 4 presents correlations between the bug rank and the percent of reviews that received a particular classification when issues are grouped by bug rank, by bug pattern and by issue (as described in Section 3.2.1). For example,

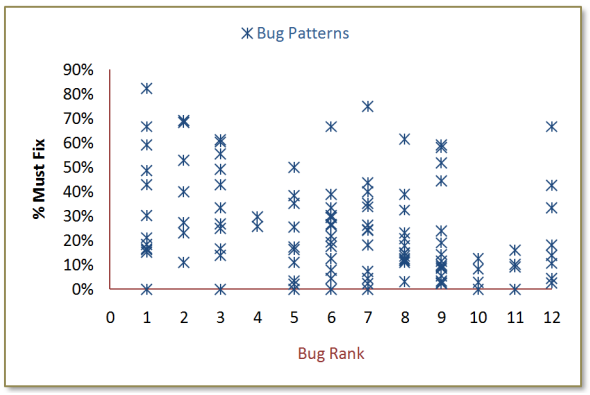

Figure 5: Must Fix Classifications By Rank

we measure a strong negative correlation (-0.93) when issues are grouped by bug rank and we compare the bug rank and the percent of issues in each group that received a *Must Fix* designation. The results show that when we cluster issues coarsely (by bug rank), we observe strong and significant (`p<0.01`) correlations with different classifications. Specifically, reviews associated with scarier issues are more likely to contain *Must Fix* classifications, while review for less scary issues are more likely to contain *Should Fix* or *Mostly Harmless* classifications.

The correlations observed when grouping by bug pattern are also interesting. While the correlations are weak, it is instructive to examine the patterns that have severe bug ranks but low *Must Fix* rates, and vice versa. In the scatter diagram in Figure 5, each point represents the percent of reviews for a particular bug pattern that were *Must Fix*. As this diagram shows, there were many among the scariest bug patterns that users were unlikely to classify as Must Fix.

When we look closely at this trend, we find it useful to make a distinction between *Loud* and *Silent* bug patterns. Loud bug patterns manifest as an exception or a program crash, and are often easy to detect without static analysis if they are feasible. So these defects, when found in production software, generally occur in infeasible situations or dead code; FindBugs often assigns a less severe bug rank to them. Silent bug patterns include those mistakes that cause the program to subtly run incorrectly. Many times, these subtle errors do not matter, but sometimes they do, and we think they should be reviewed. So FindBugs often gives this patterns a severe bug rank.

In many cases in our review, engineers were more inclined to give a Must Fix classification to loud issues than to silent issues. For example, one of the loud bug patterns is the infinite recursive loop: a method that, when invoked, always invokes itself recursively until the stack is exhausted. Sun's JDK has had more than a dozen such issues over its history, and Google's codebase has had more than 80 of them. Obviously this bug pattern is usually detected immediately if the method is ever called, and there are no known instances of this defect causing problems in production; either the defect is quickly removed or it occurs in dead code. So FindBugs assigns this bug pattern a less severe bug rank of 9. On the flip side, a classic silent pattern occurs when the type of an argument of a generic container's method is unrelated to the container's generic parameter. For example, a program

may check to see if a `Collection<String>` contains a *StringBuffer*. Such a check will always return false, and this error usually indicates a typo has occurred. This bug pattern has the bug rank of 1. In this review 52% of reviews classified infinite recursive loops as Must Fix, compared to 30% for the incompatible generic container argument pattern.

A related observation is the many instances where the subtle bug patterns had a large number of *Should Fix* classifications, together with few *Must Fix* classifications. This may indicate that the reviewer recognized the problem as a bug but did not think it was severe. For many of these bug patterns, around 20% of the reviews were Must Fix, while 60% were Should Fix.

There is largely no correlation when we group by issue, which is not surprising. Individual issues may display different characteristics from the bug pattern as a whole.

4.3 Which warnings are fixed?

We found evidence that certain subgroups of issues were more likely to be fixed. Specifically, issues that were new, issues in the Correctness category, or issues that were high priority all had higher fix rates than other issues. We observed this trend both in the fixit and in our subsequent analysis of the Google repository. We describe the evidence in these two datasets in Sections 4.3.1 and 4.3.2. In addition, we observed that issues receiving the strongest reviews from engineers were quickly fixed. In particular, issues that were marked as "I Will Fix" were quickly fixed by the reviewer, as described in Section 4.3.3.

4.3.1 Fix Rates from the Fixit

In Table 3, we compute the percent of issues that are fixed for all issues, and for different sub-groups of issues. In this case, we regard issues that no longer appear in the nightly FindBugs runs as being fixed. As described in Section 3.2.2, this approach over-counts the number of fixed issues, but since our primary goal is to compare the fix rates of different sub groups, this over counting is not a factor.

In Table 3, each row represents a different subgroup, derived by grouping issues by bug rank, by age, and/or by category. Specifically, in the category column, we either consider only Correctness issues (C) or all categories (blank). Similarly the rank column uses the marker "1-4" to indicate that we are only considering the scariest issues (and blank for all bug ranks). For this analysis, we treat issues introduced in the six weeks prior to the fixit (and any issues after the fixit) as new issues. The choice of six weeks is arbitrary but the results still hold even if the range is adjusted slightly. The other columns in Table 3 starting from the leftmost column are the fix rate, the number of issues in the subgroup that remain at the end of our study and the number of issues that have been removed (fixed). The last row represents the overall fix rate.

The results show that all subgroups have fix rates higher than the overall fix rate, though only the first four subgroups have statistically significantly higher values at the `p<0.01` level[1]. This indicates that Correctness issues, the scariest issues, and/or new issues are more likely to be fixed. The older Correctness issues do not have a much higher rate, likely because most issues were in this subgroup.

[1]To measure statistical significance, we used a chi-square test comparing the fix rate for each subgroup to the overall fix rate.

%	remain	fixed	rank	new	category
65.0	295	548		+	
64.5	252	457		+	C
59.8	227	338	1-4		
58.5	225	317	1-4		C
57.9	90	124	1-4	+	
56.7	88	115	1-4	+	C
53.0	1435	1617			C
52.7	1870	2084			

Table 3: Last Seen Fix Rate for Issue Subgroups

Another way to determine if an issue has been fixed is to look for fixes in the bug tracking system. We did not observe any significant trends using this approach, likely because at the end of our study, many of the issues filed had been assigned but not yet fixed. The fix rates for each subgroup were much lower than the fix rates in Table 3 (ranging from 34% to 39%), reflecting the fact that this approach undercounts the number of fixed issues.

4.3.2 Fix Rates in Code Repository

Table 4 shows the results from analyzing 118 snapshots of the Google codebase over a 9 month period. (To protect Google's intellectual property, we cannot publish numbers on the size of the analyzed code base, but we can report the number of warnings found.) For each bug pattern and category, we looked at how many issues were remove and how many persisted. This dataset was rather noisy and contained inconsistencies, but the size of the dataset offsets some of the noise. The snapshots were not all analyzed with the same version of FindBugs, and the code analyzed wasn't completely consistent. We made an effort to build and analyze the entire Java codebase at Google each day (initially, we only made snapshots every week). For various reasons, different projects and components might get excluded from the build for a particular day. In several cases, we made changes/improvements to FindBugs to improve the relevance/accuracy of the warnings (e.g., recognizing that a particular kind of warning was being reported in automatically generated code and was harmless, and changing the detection algorithm to not report the warning in that case).

We applied several steps to "clean" the data. We didn't consider issues that went away because we stopped seeing the class that contained the issue. Also, if more than one third of the reported issues for a bug pattern disappeared between snapshots (and there were more than 20 such issues), we attribute their disappearance to either a change in the analysis, or a systematic change to the code, and do not consider those issues. We also didn't consider issues that first appeared in the last 18 snapshots (since there wasn't really time to observe whether they would be removed). The time period did include the Google fixit in May 2009.

Overall 32% of the issues considered were removed. We don't know if this is the "natural" average removal rate, since it is biased by the fact that some detectors report far more issues than other detectors. Thus, we considered any removal rates above 37% to be higher than expected, and removal rates lower than 27% to be lower than expected. Based on those assumptions, we use a chi-square test to decide whether the removal rate for each bug pattern was significantly above 37% or below 27%. We use a negative chi value for those issues with a removal rate below 27%. In

chi	%	const	fix	max	kind
1887	65	1903	3659	321	*Correctness*
369	70	243	572	126	RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE
224	88	23	179	25	VA_FORMAT_STRING_EXTRA_ARGUMENTS_PASSED
187	74	86	245	57	RC_REF_COMPARISON
128	57	338	450	106	UUF_UNUSED_FIELD
123	78	38	137	20	EC_UNRELATED_TYPES
102	93	5	72	19	BC_IMPOSSIBLE_CAST
102	77	34	117	16	UR_UNINIT_READ
102	54	365	443	48	NP_NULL_ON_SOME_PATH
100	78	30	110	41	UMAC_UNCALLABLE_METHOD_OF_ANONYMOUS_CLASS
95	76	34	112	10	GC_UNRELATED_TYPES
87	62	123	206	22	UWF_UNWRITTEN_FIELD
28	41	2793	1968	485	NOISE_NULL_DEREFERENCE
0	37	5311	3127	293	NOISE_OPERATION
0	36	17715	10225	1192	*all noise warnings*
0	35	5391	2905	258	NOISE_METHOD_CALL
0	34	4220	2225	212	NOISE_FIELD_REFERENCE
0	32	69162	33415	1698	*all*
0	28	49544	19531	1305	*all non-correctness, non-noise warnings*
-195	18	3493	767	87	DM_NUMBER_CTOR
-202	7	904	70	11	UPM_UNCALLED_PRIVATE_METHOD
-209	13	1888	301	74	RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE

Table 4: Fix rate for bug patterns in Google code base

Table 4, we report the results which had chi value above 70 (or below -70), all of which are significant at the `p < 0.05` level, as well as the noise bug patterns and the groups of issues by category. The other columns in order are the percentage of issues that were removed, the number of issues that remained in the final snapshot, the number of issues that appeared in some version but not in the final snapshot ("fixed" issues), the maximum number of issues that disappeared between any two successive snapshots, and the name of the pattern.

Note that we are modeling these issues as independent variables, but often they are not. In some cases, a particular mistake (such as left shifting an `int` value by a constant amount greater than 31) will manifest itself multiple times in a class or method, and the issues will either all be fixed together or not at all. Sometimes, a single change to the code will resolve a number of warnings that are associated with the changed code. Furthermore, sometimes there will be a specific effort to resolve a particular kind of issue. There are many variations on this problem, and we try to capture some of this by reporting the maximum number of issues that disappeared between any two successive snapshots. When a substantial fraction of the total number of issues in a bug pattern disappear like this, it is reasonable to believe that they were removed as part of a single effort or due to a change in the FindBugs analysis engine. As noted before, we omit any cases where more than one third of the issues were removed between one pair of successive iterations.

Some of the removed issues (such as unused or unread fields), may reflect the refinement of incompletely implemented classes rather than fixing of defects. A number of the bug patterns with significant removals (impossible casts, comparison of unrelated types) are serious coding mistakes, so it is reasonable to postulate that they were removed because they were causing problems.

The most significant removal rate was for the bug pattern that occurs when a value is (redundantly) compared to null even though it has already been dereferenced. By contrast, a similar bug pattern (comparing a value to null even though it is known to be non-null due to a previous comparison) is the most likely to persist in the code. This suggests that this second bug pattern was not causing many problems and the redundant comparisons in this case were mostly defensive.

Interestingly, noise null dereference warnings had a removal rate that was significantly higher than the overall removal rate. Noise null dereference warnings are only generated in cases where the value being dereferenced is not guaranteed to be nonnull. Perhaps there are some bugs at these dereference sites, and it may be valuable for developers to review all recently created locations where a dereferenced value is not guaranteed to be nonnull.

4.3.3 Comparing Fix Rate to User Reviews

We would like to check if the issues that received many Must Fix and Should Fix classifications were more likely to be fixed. One approach is to order the classifications according to their severity and compare this to the fix rate of each classification. There is no absolute notion of ordering the classifications, so we experimented with several, shown in Table 1.

We observed strong and significant (`p<0.01`) correlations, shown in Figure 6, between our various orderings and the percentage of issues in each classification that were fixed. In other words, issues with the most severe classifications were more likely to have been fixed. In this figure, we are using both approaches described in Section 3.2.2 to determine which issues have been fixed.

In particular, issues that received *I Will Fix* classifications were quickly fixed. Since each issue received multiple classifications, we use the classification that the plurality of reviewers gave to each issue (called the consensus classification in Section 4.4). The results show that 88% of the reported issues marked *I Will Fix* have been fixed. Even when we consider those issues marked *I Will Fix* at least once (i.e. not necessarily the plurality of reviewers) we observe that over 70% have been fixed.

	Fix Rate	
	By Last Seen	Fixed in Bug DB
vs Ord1	-0.90	-0.83
vs Ord2	-0.98	-0.93

Figure 6: User Classifications versus Fix Rate

	All Reviews		Scariest Issues	
	CR	CR > 0.8	CR	CR > 0.8
Ord1	0.65	31%	0.64	30%
Ord3	0.87	73%	0.92	86%
Ord4	0.87	75%	0.93	87%

Figure 7: Consensus Rates for All and Scariest Issues

4.4 Other results

4.4.1 Consensus Classifications

One reason to have multiple reviews of a single issue is to determine if there is a consensus among the different reviewers about the importance of an issue. The issue of consistency is related to the question of whether an organization should have multiple reviewers for each issue, or just allow individuals to make decisions, especially about filtering out or suppressing issues. In surveys we have conducted, most respondents have indicated that their organizations do not have requirements on how many reviewers should look at an issue before it can be addressed (fixed or suppressed) [1].

Since some classifications are close in meaning, we experiment with grouping the classification in various ways, as discussed in Section 3.2.3.

Once the reviews are grouped based on their classifications, we count the number of reviews in each group for each issue. We used two methods to aggregate these counts and get a sense of the overall consensus. One is to count the number of reviews in the largest group for each issue (which we term the *Consensus Group*), aggregate this count over all issues, and divide this final number by the total number of reviews in the analysis. We call this the *Consensus Rate* (or the rate at which reviews end up in the consensus group). A second method is to compute the consensus rate for each issue (i.e. reviews in largest group divided by total number of reviews), and count the number of issues that have a consensus rate above a desired threshold. In Figure 7 we show these two measures, using a threshold of 0.8 for the second measure and using some of the classification schemes from Table 1. For example, when using the *Ord3* scheme described above, we observe a consensus rate of 0.87 for all reviews, and 73% of all issues have a consensus rate greater than 0.8. The consensus rate increases significantly when we group similar classifications as is done in *Ord3* and *Ord4*. We use this to infer that users generally agree, but the subjective nature of the review means they do not always give exactly the same classification.

4.4.2 Review Times

The review time is an important measure when trying to compute the cost of using static analysis. We computed a mean review time of 117 seconds which matches our previous observations. We also grouped the review times by classifications and observed that the *Obsolete Code* classification had the lowest review time at 64 seconds. Closer inspection confirms that some users quickly dispatched issues that occurred in files that were obsolete. Removing these reviews from consideration does not significantly impact the review time however.

4.4.3 Reviews from Different User Groups

The fixit dataset includes anonymized information about which user conducted each review and which users are listed as owners of different files. Using this information we can infer which users performed the most reviews (the super users) and we can track how users reviewed issues in files that they own.

The top reviewer examined 882 issues, and 18 out of the 282 users reviewed more than a hundred issues. We classified these users as super users and compared the classifications they gave with those of other users. Similarly, we compared the classifications of owners with that of non owners, focusing just on the issues that were reviewed by at least one owner. We observed the super users were significantly more likely to give *Must Fix* classifications and significantly less likely to say *I Will Fix*. On the other hand owners were much more likely to say *I Will Fix* or *Obsolete code* than non-owners, and much less likely to give *Must Fix* classifications. This suggests that owners were taking responsibility for fixing serious issues in their code. It also suggests that most super users were not owners and vice versa. Only seven of the super users owned any of the files they reviewed.

5. RELATED WORK

Other researchers have described their efforts to integrate static analysis into commercial processes, and the feedback they received from developers. Ours is the first to aggregate opinions from such a large group of engineers and report on which problems were actually fixed. Researchers at eBay experimented with enforcement-based customization policies, through which bug patterns are filtered and reprioritized, and developers are required to fix all resulting high priority warnings [8]. Practitioners from Coverity, a commercial static analysis vendor, observe one outcome of reviews: sometimes reviewers misunderstand a bug and mislabel it as a false positive, despite the best attempts of Coverity's team to convince them otherwise. This has led them to turn off some detectors that are easily misunderstood, so that developers do not develop a negative impression of the tool [4]. We observed this phenomenon, especially with the subtle bug patterns that engineers tended to undervalue.

Other researchers have analyzed the code history to identify top bug patterns. Early research at Google tried to predict which warnings will be fixed based on factors (such as file size) that are associated with warnings fixed in the past [13]. The researchers relied on information from bug reports to identify fixed issues, so this limited the size of their training dataset. Kim and Ernst also reprioritize warnings, using the removal of warnings as a measure of their importance, and increasing their confidence by emphasizing warnings removed during bug fix commits [9, 10]. By contrast,

we experiment with using noise detectors to validate that high removal rates of some bug patterns are not due to code churn. Our noise detectors were partially inspired by other papers which try to correlate the density of static analysis warnings with various measures of software quality [12, 13, 6]. The challenge these researchers face is that the warning counts are impacted by code churn, and noise detectors provide a way to detect the proportion of warning removals that are essentially random.

6. CONCLUSIONS

We successfully completed a large review of FindBugs warnings in Google's code base. The primary goal of the review was to bring problems to the attention of responsible parties, but we were also able to collect large amounts of data which we investigated in this paper. We observed that most reviews recommended fixing the underlying issue, but few issues caused serious problems in practice. Those issues that might have been problematic were caught during development, testing and deployment. The value of static analysis is that it could have found these problems more cheaply, and we illustrated this with anecdotal examples from student code.

We also observed that the importance placed on warnings by developers matched the bug ranks in FindBugs, but some bug patterns deviated from this norm. Specifically, users tended to overvalue some bug patterns that manifest as exceptions or program crashes, but are rarely feasible in practice, and undervalue more subtle bugs that are often harmless, but should be reviewed because they can cause serious problems that are hard to detect. We also observed that new, correctness and high priority issues are the ones most likely to be fixed, matching our expectations coming into this study. Users were also more likely to fix the issues that were classified as Must Fix, Should Fix, or I Will Fix.

Overall, the fixit was declared a success, and some managers were impressed by the high percentage of the reviews that gave a fix recommendation. Ongoing efforts are integrating FindBugs warnings into a code review tool so that issues are addressed earlier in the development process. Some developers already run the analysis regularly through their IDEs. We have also started a FindBugs Community Review, through which developers of various open source projects are reviewing and fixing warnings in their respective projects, using the infrastructure developed at Google.

Results from the fixit (including additional data collected subsequently) are available at:
`http://findbugs.sourceforge.net/publications.html`.

7. REFERENCES

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, 2008.

[2] N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA, 2008. ACM.

[3] N. Ayewah and W. Pugh. Using checklists to review static analysis warnings. In *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pages 11–15, New York, NY, USA, 2009. ACM.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[5] S. Boslaugh and D. P. A. Watters. *Statistics in a nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2008.

[6] R. P. Buse and W. R. Weimer. A metric for software readability. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130, New York, NY, USA, 2008. ACM.

[7] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM.

[8] C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 963–970, Montreal, Quebec, Canada, 2007. ACM.

[9] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 27, Washington, DC, USA, 2007. IEEE Computer Society.

[10] S. Kim and M. D. Ernst. Which warnings should i fix first? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54, New York, NY, USA, 2007. ACM.

[11] B. Mediratta and J. Bick. The google way: Give engineers room. *The New York Times*, Oct 2007.

[12] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 580–586, New York, NY, USA, 2005. ACM.

[13] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2008. ACM.

[14] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, New York, NY, USA, 2006. ACM.

[15] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 13–17, New York, NY, USA, 2006. ACM.