ABSTRACT

Title of dissertation: STATIC ANALYSIS IN PRACTICE

Nathaniel Ayewah, Doctor of Philosophy, 2010

Dissertation directed by: Professor William Pugh
Department of Computer Science

Static analysis tools search software looking for defects that may cause an application to deviate from its intended behavior. These include defects that compute incorrect values, cause runtime exceptions or crashes, expose applications to security vulnerabilities, or lead to performance degradation. In an ideal world, the analysis would precisely identify all possible defects. In reality, it is not always possible to infer the intent of a software component or code fragment, and static analysis tools sometimes output spurious warnings or miss important bugs. As a result, tool makers and researchers focus on developing heuristics and techniques to improve speed and accuracy. But, in practice, speed and accuracy are not sufficient to maximize the value received by software makers using static analysis. Software engineering teams need to make static analysis an effective part of their regular process.

In this dissertation, I examine the ways static analysis is used in practice by commercial and open source users. I observe that effectiveness is hampered, not only by false warnings, but also by true defects that do not affect software behavior in practice. Indeed, mature production systems are often littered with true

defects that do not prevent them from functioning, mostly correctly. To understand why this occurs, observe that developers inadvertently create both important and unimportant defects when they write software, but most quality assurance activities are directed at finding the important ones. By the time the system is mature, there may still be a few consequential defects that can be found by static analysis, but they are drowned out by the many true but low impact defects that were never fixed. An exception to this rule is certain classes of subtle security, performance, or concurrency defects that are hard to detect without static analysis.

Software teams can use static analysis to find defects very early in the process, when they are cheapest to fix, and in so doing increase the effectiveness of later quality assurance activities. But this effort comes with costs that must be managed to ensure static analysis is worthwhile. The cost effectiveness of static analysis also depends on the nature of the defect being sought, the nature of the application, the infrastructure supporting tools, and the policies governing its use. Through this research, I interact with real users through surveys, interviews, lab studies, and community-wide reviews, to discover their perspectives and experiences, and to understand the costs and challenges incurred when adopting static analysis tools. I also analyze the defects found in real systems and make observations about which ones are fixed, why some seemingly serious defects persist, and what considerations static analysis tools and software teams should make to increase effectiveness. Ultimately, my interaction with real users confirms that static analysis is well received and useful in practice, but the right environment is needed to maximize its return on investment.

STATIC ANALYSIS IN PRACTICE

by

Nathaniel Ayewah

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:
Professor William Pugh, Chair/Advisor
Professor Michael Hicks
Professor Jeffrey S. Foster
Professor Adam Porter
Professor Siva Viswanathan

To my parents,
Emmanuel and Grace
Ayewah

Acknowledgments

I would like to thank my advisor, Bill Pugh, for providing the impetus and resources to conduct this difficult research. Bill was a constant source of inspiration and insight as I searched for explanations for the phenomena we observed. I am grateful for this experience, which has taught me to be persistent and to seek to maximize the practical impact of my research. I have also had the support of many faculty members at the University of Maryland, both on and off my committee. Thanks to Mike Hicks, Jeff Foster, and the rest of PLUM (Programming Languages at the University of Maryland) group for providing the grounding I needed in programming languages, and constantly questioning my ideas and providing suggestions.

Of course, this research would not be possible without the willingness of the thousands of unnamed engineers who participated in some form by filling out surveys, conducting reviews or participating in lab studies. I want to extend my thanks to the professionals at various organizations who facilitated my studies or provided other technical or material support. Special thanks goes to Jason Yang and Dave Sielaff at Microsoft for providing mentorship and direction, and to Sunny, Mei and the rest of the Analysis Technologies team for providing technical advice and support. I look forward to doing great work with you all in the future. I also want to thank Dave Morgenthaler, John Penix and other researchers and engineers at Google who have allowed me to learn from their experiences. I have also received helpful advice and insights from Andy Chou of Coverity, Brian Chess of Fortify Software, Gary McGraw of Cigital and Jeff Williams of Aspect Security. My thanks to you all.

Finally, I want to thank my friends and family who have provided the spiritual and emotional support I have needed to make it this far. Many thanks to all the cool cats on the New Leaf band and all the other members of New Leaf Church who have been a constant source of encouragement, not to mention free meals. A special thanks is due to "the guys"—Jimmy, Matt, Alex, Brian, and Feno—who put up with my really late night study habits. And, most of all, to my family, Daniel, Beryl, Sarah, Martha, Christina (who at 2 months has only contributed occasional gleeful moments) and my parents, Emmanuel and Grace. Thank you all for believing in me, and filling me with faith in myself, in my community, and in the power of grace.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Why do software programmers make mistakes? Sometimes, mistakes occur because the problem is complex and the programmer forgets important constraints or dependencies. Sometimes mistakes occur because of a deficiency in the programmer's understanding of the language. But often, mistakes are just silly errors, comparable to typos a writer might make when composing an essay. And while many silly errors cause the compiler to fail, or the program to crash quickly, some errors can escape notice for some time, while causing the program to behave incorrectly.

Consider the code fragment in Figure 1.1, which was inserted into Glassfish[1] on March 2, 2010. Observe that the method does not advance past line 185 unless the local variable `firstLevelEntries` is null (because of the return statement). This means that the condition on line 187, which dereferences `firstLevelEntries`, will always throw a Null-Pointer Exception (NPE) if it is executed. This cannot be the intent of the programmer, so we can infer that this method must contain a mistake.

This process of inferring the presence of this mistake can be done automatically using several systematic analysis techniques. One approach is to scan forward through the method, keeping track of whether a value may be null or non-null. If a value known to be null is dereferenced, we can quickly conclude that a mistake is

[1] `http://glassfish.dev.java.net/`

Source: GlassFish | org.glassfish.osgijavaeebase.OSGiBundleArchive

```
183     private Collection<String> getSubDiretcories(String path) {
184         final Enumeration firstLevelEntries = b.getEntryPaths(path);
185         if (firstLevelEntries != null) return Collections.EMPTY_LIST;
186         Collection<String> firstLevelDirs = new ArrayList<String>();
187         while (firstLevelEntries.hasMoreElements()) {
188             String firstLevelEntry = (String) firstLevelEntries.nextElement();
189             if (firstLevelEntry.endsWith("/") firstLevelDirs.add(firstLevelEntry);
190         }
191         return firstLevelDirs;
192     }
```

Figure 1.1: Null-Pointer Dereference in GlassFish

present.

This systematic analysis can be done by a software program, called a *static analysis* tool. The word "static" refers to the fact that the tool examines programs without executing them.

In practice, inferring the presence of a mistake is not sufficient; we need to understand what the problem is, and decide if it matters. To do this, we need a deeper understanding of the semantics of the program. From the method signature, we can tell that the purpose of the method is to return a list of string paths, representing the subdirectories of the input path[2]. We also observe that the program fragment extracts an enumeration of entries from the input path and either iterates through this enumeration, or returns an empty list. With this information, we can quickly infer that the developer accidently used the wrong operator on line 185 (inequality ! = instead of equality ==); the empty list should be returned when

[2]despite the spelling error in the method name

`firstLevelEntries` is null, and the iteration should occur when it is not null, and not vice versa.

Observe that even though the analysis flags a potential null dereference on line 187, the real mistake is using the wrong comparison operator on line 185. It would be more user friendly for the analysis to say, "You used the wrong operator," instead of leaving the user to figure this out. But this would require the analysis to understand the purpose of this method, which is difficult to do without requiring the user to provide lots of descriptive metadata. Indeed it is possible to construct an identical code fragment with very different semantics. For example, it could be that the comparison is actually correct, and the mistake is a missing assignment of a non-null value to the target variable *after* the comparison but *before* it is dereferenced. So in general, static analysis can scan the code looking for *violated properties* (in this case, dereferencing a value on the branch in which it is null), but cannot necessarily identify the root mistake.

Observe also that this mistake does not always result in a Null-Pointer Exception. If `firstLevelEntries` is not null, the program does not crash, but returns an incorrect value. Hence it is possible for this mistake to go undetected for some time, and even after the incorrect behavior is noticed, a debugging effort may be needed to trace the problem to this line. In this example, the mistake was fixed 29 days later when the developer received an alert from a static analysis tool called *FindBugs* [62, 64].

This example illustrates the potential of static analysis to quickly find mistakes. And static analysis is not just limited to occasional typos; it can also flag

bad practices, confusing logic, unsafe code, or programs that exhibit poor performance properties.

This example also illustrates some of the limitations of static analysis. Static analysis algorithms do not usually understand the intended functionality and use cases of a program, and hence cannot necessarily determine if a mistake is important or feasible. More generally, analyzing any software to determine all its correctness properties is an undecidable problem. In practice, static analysis algorithms make simplifying assumptions, and focus on classes of problems that can be detected tractably. But even with these simplifications, many static analysis implementations run into speed or accuracy constraints that affect their usefulness. In particular, it is possible for an analysis to report some issues that are not actually mistakes, known as *false positives*. Conversely, an analysis may miss some mistakes that are real problems; these are called *false negatives*.

To deal with these challenges, some researchers and commercial vendors continue to develop increasingly sophisticated static analysis techniques that can run faster, and be more precise. Other researchers focus on tweaking existing analysis techniques based on problems observed in practice. Here, the goal is to minimize the number of false positives so that users perceive each warning generated as valuable.

Despite these improvements, some users are still skeptical of the value of static analysis and question whether it improves software quality and developer productivity in practice. Part of the problem is that other quality assurance activities—such as unit testing and code review—can find a greater scope of problems. In addition, using static analysis incurs some significant costs in practice. For example, when a

static analysis is first run on an established software project, it often finds numerous issues in legacy code that has not been edited for a while. Users generally conclude that these issues are unlikely to be important since that part of the software has been functioning correctly for some time, but they need to take steps to deal with these problems anyway, lest they drown out more important issues that result from more recent changes. In addition, users may find that simply changing the code in response to a static analysis warning may lead to regressions, and hence need to spend time considering the implications of every change before making it. Apart from these challenges, users may find that they need to spend much time engaged in manual repetitive tasks to deal with static analysis warnings, such as recording issues in a bug tracking system, assigning issues to the correct person, or suppressing issues in obsolete code.

To deal with these problems, organizations need to adopt effective strategies that enable them to address issues found by static analysis early, and to maximize their return on investment. In this dissertation, I present some of the experiences of real organizations, and describe insights generated from interacting with users and studying software artifacts. For example, I describe the reasons some are slow to adopt tools, explain why some seemingly serious defects can persist for a long time in a codebase, and identify the contexts in which static analysis is worthwhile.

To conduct this research, I rely on surveys, interviews, lab studies, reviews of static analysis warnings by professionals, and a number of studies which examine software artifacts directly looking for clues. Some detailed results, datasets and relevant code from the studies in this dissertation are archived online at:

http://findbugs.cs.umd.edu/inpractice/.

In the next section, I provide some basic definitions of terms I will use throughout this dissertation. In Section 1.2, I explain why static analysis and other quality assurance activities are necessary by considering the impacts of defects in practice. I go on to describe some of the key promises and challenges associated with using static analysis in Section 1.3. Finally, I present my main thesis and summarize my findings in Sections 1.4 and 1.5 respectively.

1.1 Some Definitions

In the literature, various definitions are used to describe software defects. Many researchers do not explicitly define defects, but imply that they lead to undesirable program behavior. Others use a more general definition that includes any kind of flaw. One definition in the literature is: "A defect is any unintended characteristic that impairs the utility or worth of an item, or any kind of shortcoming, imperfection, or deficiency" [43].

In our research, we emphasize the fact that not all mistakes are equally bad, and the context of a problem influences whether it is important or not. We use the following definitions:

> A *defect* is an implementation fragment or design feature that, when revealed to the software team immediately after it is created, along with an explanation of why it might be a mistake, the team would generally choose to fix.

A *bug* is a defect that causes undesirable program behavior. This could include incorrect results, degraded performance, or vulnerability to security threats.

These definitions make the notion of a defect more subjective, but also place the emphasis on users and their priorities. Some users (and teams) will want to address style issues like incorrect indentation or missing comments, while others will not consider these to be defects. Also, the emphasis on showing the problem to the software team soon after the defective feature is created assumes that software teams are more willing to fix problems during the early parts of the software process, and less willing to change the code after it has been deployed.

In addition to these definitions for software problems, we also need some definitions that are specific to static analysis tools:

A *warning* is a message from a static analysis tool, highlighting one or more potential defects in the software. Warnings are also sometimes called *alerts*.

A *bug pattern* is an idiom that represents a class of defects that are similar. Bug patterns are also sometimes called *rules*.

1.2 Software Defects In Practice

Software defects and failures are expensive, both for users who experience losses, and for software developers who spend resources mitigating problems. A 2002 report from NIST estimated that the annual economic cost to the US of inadequate software quality control ranged from $22 to $60 billion [132]. A study from

Carnegie Mellon University found that disclosures of security flaws hurt the stock price of the software company involved [85]. The negative effects of defective software are not limited to economic indicators. Increasingly software systems are used in safety critical systems, where defects can result in injury or loss of life. A famous example is the Therac-25 disaster, where a software error in a medical device led to patients receiving fatal overdoses of radiation [86]. Software defects also have political consequences. USA Today reports that a software system to manage the 2010 US Census was behind schedule and riddled with defects. As a result, the census process needed to be modified and "risks ballooning costs, delays and inaccuracies" [108].

Software systems are becoming more pervasive, and customers are more exposed to them. Devices like automobiles and refrigerators that previously did not contain software, now use software to monitor components and even connect to the Internet [75]. Other devices like mobile phones have seen their software become more complex and feature rich. If left unchecked, the problem of software defects will only grow and impact more users.

Organizations seeking to get a handle on this problem need to have good software development practices, effective quality assurance activities, and rigorous testing. The 2002 NIST report mentioned earlier calls for use of software quality metrics and testing infrastructure to identify problems early [132]. These techniques are effective for ensuring an application meets its requirements, but can be expensive to implement, and do not exhaustively exercise the program. Some safety-critical systems, such as aircraft flight control, benefit from using formal verification

methods to ensure that the underlying implementation precisely matches a given specification [105]. But this approach cannot overcome flaws in the specification, does not scale to general large applications, and it requires more technical expertise than average developers have.

Compared to some of the quality assurance techniques briefly discussed so far, static analysis appears to be relatively inexpensive, easy to use, and it can exhaustively search the code for problems. On the other hand, it is limited in the scope of problems it can detect, and sometimes identifies defects that do not matter. Still, static analysis may help to reduce the cost of finding software defects, and hence have a real impact on the experiences of customers of an increasing number of products.

1.3 Static Analysis: Promise and Challenges

The primary benefits of static analysis tools are that they find problems in software without executing it, and they can search software exhaustively, eliminating some classes of problems. But this exhaustiveness can also lead to some challenges. We refer to an analysis that can identify all instances of a particular problem as a *sound* analysis. The problem with sound analysis is that it often yields instances that are not real defects, due to constraints on the precision of the analysis. I discuss some of the challenges associated with sound analysis in Section 1.3.1. Even if the analysis identifies a real programming mistake, its warning may not be of interest to developers because the problem may be infeasible, unlikely, or have only minor

consequences. I discuss some of these scenarios in Section 1.3.2.

Other benefits of static analysis are that many tools are relatively fast, and can build on the wisdom of experts to identify problems developers may not be aware of. Many tools can be extended to find new classes of defects and some tools analyze binaries, which is useful when source code is unavailable.

Despite these benefits, we have observed several challenges that arise when users interact with tools in practice, and that limit their widespread adoption. Some of the warnings produced by static analysis are difficult for users to understand, especially when the variables and values involved occur in several different methods. In addition, sometimes warnings occur in code that has not been touched for a long time, or that is owned and maintained by someone other than the analyst. Other times the warning may not represent a quality dimension an organization is interested in. For example, internationalization warnings may not be relevant for applications that only expect to run in one locale or encoding. When deploying a static analysis tool in large projects with many developers, the analysts may be overwhelmed by the large number of initial warnings found the first time a tool is run, and may postpone addressing them to focus on more pressing needs. Furthermore, large organizations often have to figure out how to integrate warnings from different tools into one consistent interface.

1.3.1 Sound Analysis and False Positives

To illustrate the challenges of a sound analysis, consider the Java code fragment in Figure 1.2 (which is based on a blog post by Mark Dixon [37]). In constructing a test case, the developer places a JUnit assertion inside a thread. This is problematic, because exceptions thrown in an auxiliary thread are not propagated to the main thread. If the assertion fails, JUnit will not record the assertion error and the test will NOT fail as it should.

Suppose I set out to write a static analysis to flag every case where an assertion is made in a separate thread. In the example in Figure 1.2, the analysis correctly detects an assertion made in a separate thread from the main test thread. But consider the slightly modified example in Figure 1.3. Here, the developer constructs a *Runnable*, which is usually executed in a separate tread. But in this case, the *Runnable* is executed within the main thread (with the call to `r.run()`). Hence the assertion failure in this example will be detected by JUnit, and the code is defect free.

The challenge in these examples is that if I want my analysis to be sound, then I must flag any case where a threading construct such as *Thread* or *Runnable* is used, unless I can demonstrate that the code will be run in the main thread. In general, this is not always practical, and my analysis may issue a false warning for code fragments like Figure 1.3.

In practice, most modern commercial static analysis tools use various heuristics to reduce the number of false warnings, with the consequence that some classes of

```
1  public void test() {
2      Thread t = new Thread() {
3          @Override public void run() {
4              assertTrue(false);
5          }
6      };
7      t.start();
8  }
```

Figure 1.2: Assertion failure in separate thread is not seen by JUnit

```
1  public void test() {
2      Runnable r = new Runnable() {
3          public void run() {
4              assertTrue(false);
5          }
6      };
7      r.run();
8  }
```

Figure 1.3: Assertion failure is seen by JUnit because it occurs in main thread

defects are found using unsound analysis. For these classes, static analysis tools focus on finding as many useful defects as possible, rather than on eliminating all possible defects. As some researchers from Coverity[3], a commercial static analysis vendor, recently stated: "Unsoundness let us focus on handling the easiest cases first, scaling up as it proved useful" [23]. Even though these tools cannot guarantee the absence of most classes of problems, users still receive the benefit that all defects matching the tool's heuristics will be flagged immediately, no matter where they are in the code.

[3]http://www.coverity.com/

1.3.2 Infeasible, Unlikely or Low Impact Defects

Even if a static analysis tool is able to avoid false positives, not all problems flagged are of interest to developers. The root problem stems from a mismatch between what static code analyzers do and what developers ultimately care about. Static code analyzers look for silly mistakes in code, confusing code, or violations of good practice or desired safety properties. Developers are ultimately looking for incorrect behavior, and there are many instances where the problems identified by static analysis do not cause incorrect behavior.

Figure 1.4 illustrates this with an example in which the programmer checks the value in a byte array `b`. The programmer assumes the value checked is an unsigned value (from 0 to 255) and wants distinct behavior for values in or out of the range [32,128]. In fact, the values in a byte array are *signed* (from -128 to 127) and hence the check `b[offset] > 128` is always false and may be flagged by a static code analyzer as a nonsensical operation. However all the values in an unsigned byte that are greater than 128 are *negative* numbers in a signed byte, and these are caught by the first check `b[offset] < 32`. Indeed, assuming the programmer's

Source: Sun JDBC API | **sun.jdbc.odbc.JdbcOdbcObject**

```
85      if ((b[offset] < 32) || (b[offset] > 128)) {
86          asciiLine += ".";
87      }
88      else {
89          asciiLine += new String (b, offset, 1);
90      }
```

Figure 1.4: A coding mistake that does not cause incorrect behavior

goal is to separate basic printable ASCII characters from unprintable and extended ASCII codes, this code behaves correctly.

The decision to fix problems like this depends on many factors including how old the code is, what stage in the development process it is found, and the culture of the organization. Many organizations want to at least review most problems, incurring the cost of doing so rather than risk potential future high cost and embarrassment. Many others, in the face of time-to-market pressures, prefer to only see problems that might lead to incorrect behavior.

Even if a defect provably causes incorrect behavior, it may be unlikely in practice. For example, in Figure 1.5, static analysis can identify the repeated conditional test on line 165. The second comparison of `offx` to null is completely redundant, and is unlikely to have been inserted intentionally. Of course, it is still up to the human reader to decide if this aberration is associated with incorrect behavior, or is just a silly but harmless mistake. In this case, there is a strongly correlated variable `offy` that is used everywhere `offx` is used. So we might guess that the second comparison on line 165 should be `offy != null`.

The impact of this defect is that if one of these variable is null, and the other is not, the wrong branch will be taken, resulting in incorrect behavior. But if both variables are always null, or non-null at the same time, this defect will have no effect on the behavior of the program. Since both variables are initialized by reading correlated attributes from an XML document (on lines 81-82), it seems very unlikely that the negative scenario will ever arise in practice. Still, examples like this illustrate the potential of static analysis to find obscure scenarios that are unlikely

Source: Open Laszlo | org.openlaszlo.compiler.ResourceCompiler

```
81      String offx = element.getAttributeValue("offsetx");
82      String offy = element.getAttributeValue("offsety");
161     ...
162     if (!sources.isEmpty()) {
163         if (tagName.equals("preloadresource")) {
164             mEnv.getResourceGenerator().importPreloadResource(sources,
                    name, file);
165         } else if ((offx == null) && (offx == null)) {
166             mEnv.getResourceGenerator().importResource(sources, name, file);
167         } else {
168             mEnv.getResourceGenerator().importResource(sources, name, file,
169                     new Offset2D(offx, offy));
170         }
171         ...
```

Figure 1.5: Repeated Conditional

to come up during testing.

Even when a defect is feasible and causes incorrect behavior, its consequence may be minor in practice. For example, in Figure 1.6, static analysis can quickly detect that `listeners` is dereferenced on the branch in which it is guaranteed to be null (on line 268). As with Figure 1.1, the defect is not the dereference, but an incorrect comparison operator on line 267.

When we find defects like this in production code, we have to ask why no one has detected it yet. Has this problem not caused an exception, leaving a stack trace that quickly leads to a simple fix? It is easy to be drawn to the Null-Pointer exception that is thrown if `listeners` is null, but the semantics of the class suggest that `listeners` is probably never null in practice. (It is initialized in

Source: Sun JDK 1.6.0, b105 | sun.awt.X11.XMSelection

```
265     public synchronized void removeSelectionListener(
266             XMSelectionListener listener) {
267         if (listeners == null) {
268             listeners.remove(listener);
269         }
270     }
```

Figure 1.6: Null-Pointer Dereference

`addSelectionListener()` which is probably always called before `removeSelectionListener()`.) So the real impact of this defect is that the method parameter, `listener`, is never removed from the collection, and continues to receive events from this class. While this behavior violates the requirements of the class, it does not necessarily have bad consequences. Perhaps `listener` ignores the events, or its actions in response to events are inconsequential. Perhaps this class is only used in scenarios where listeners are added, but never removed. It is even possible that the class is only used in scenarios where removing listeners is the wrong thing to do, and the only reason why the program works is because of this defect!

Prudent organizations will want to find and resolve all the warnings highlighted in the last three examples. But when there are thousands of these low impact issues—a distinct possibility since static analysis may be used to explore every nook and cranny of the software—organizations need to weigh the cost of addressing all of them with the value that is gained. In general, static analysis has questionable value if it does not present users with issues that they want to fix, even if they are true defects. To illustrate this, consider the compiler warnings that are generated

every time a program is compiled. In many contexts, thousands of warnings are generated, and engineers have developed the habit of completely ignoring them. Some organizations set aside a dedicated period, usually after a major release, to go through and clean up some of these warnings. But many times the warnings are just ignored and taken as a fact of life. This experience suggests that if static analysis tools do not discriminate more between defects, and exclude issues that users are unwilling to fix, their overall value will decrease. In this dissertation, I explore this and other observations by studying the practice of real users, and exploring the artifacts left over from software development activities.

1.4 Thesis and Contributions

My research is built around a number of small studies which lead me to new insights, or confirm existing ideas about the way static analysis tools are used. I summarize some of these insights and ideas in the next two sections. My main ideas can be summarized with the following thesis statement:

> Static analysis is useful, and can find interesting software defects. But in practice, some found defects are not important, because they do not cause the software to misbehave. Furthermore, some classes of important defects are regularly caught by other, more expensive quality assurance activities before the affected code gets to production. Hence it is not sufficient to simply use static analysis tools — organizations need to adopt effective strategies to automatically identify important warnings early,

in order to maximize their return on investment. And static analysis tools need to provide features to support these strategies, helping organizations create a plan, inform and educate stakeholders, and measure progress and metrics.

I support this thesis statement by conducting studies to discover the opportunities and pitfalls associated with static analysis in practice, including some of the first studies to report on the experiences of real users of a modern static analysis tool and compare the opinions of hundreds of professionals reviewing defects in a commercial organization. The primary contributions of this research are as follows:

- I describe the experiences, motivations and challenges of real users seeking to adopt static analysis, and present the results of community-wide reviews of real defects.

- I provide insights on why seemingly serious defects may persist in a code base for a long time without causing problems, only to be later found by static analysis.

- I make observations about which defects are fixed in practice by projects that use static analysis and those that do not, and tackle the question of how to account for natural changes in the code, called *code churn*, that make the data noisy.

- I review real defects and some associated bug reports, and make observations that challenge the way static analysis tools approach the problem of flagging

potential null-pointer dereferences.

- I identify some best practices associated with the successful and cost-effective adoption of static analysis.

- I propose and conduct preliminary research on a framework to make it simpler to extend static analysis tools to find project-specific or API-specific defects, by providing examples or *mockups* of the defects.

1.4.1 Studying Static Analysis In Practice

We usually evaluate static analysis tools in terms of what defects they find, efficiency and accuracy. Tool vendors and researchers turn to benchmarks to demonstrate that their tools meet accuracy, performance and soundness constraints [59, 91, 34]. Traditionally, these practitioners assert the effectiveness of their tools by emphasizing the few false positives output by the tool. But, as I mentioned earlier, even warnings that are true positives may not be important. Furthermore, the designation of a warning as a false positive is often subjective.

If the ultimate goal of this endeavor is to encourage developers to adopt tools and use them effectively, then we need to better understand how they impact users and software processes. Controlled user studies can help us measure usability and performance characteristics of tools, and get feedback from users about their interaction with tools. This feedback can lead to improvements to the tool's interface. For example, in one study, researchers observed that users had some difficulty understanding warnings from a static analysis tool for deadlock and race detection,

because the warning trace crossed numerous method boundaries [77]. They were able to improve user outcomes by enhancing the user interface to superimpose multiple methods into a concise format, and by using checklists.

Controlled user studies are helpful for exploring the direct interaction between users and tools, but are limited when the goal is to consider all the factors that influence successful adoption of static analysis tools. Ultimately, to comprehend how tools impact users and software processes, we need to understand how they are used in practice. This is the focus of my research. As part of my research, I have sought to understand the overall value of static analysis tools. What proportion of analysis warnings actually signal incorrect behavior in practice and can these be found by other quality assurance methods at comparable cost? I have also studied the defects that occur in practice, and the choices developers make about which ones to fix. In other words, which warnings matter? I have also investigated the practical considerations, tradeoffs, costs and challenges that organizations deal with when they choose to use static analysis tools. I can use this research to question why tools are not used more often, and validate or invalidate the assumptions made by tool vendors and other researchers. Ultimately, I wish to identify best practices that increase successful adoption of static analysis tools.

In some ways this research is similar to efforts to improve spell checkers in word processors or spreadsheets by studying the habits of users. Unlike spell checkers though, static analysis tools demand more human investment and infrastructure to identify and remediate significant warnings.

My research practice alternates between direct interactions with users which

yield mostly qualitative and anecdotal information, and substantial studies of code artifacts and bug reports which yield mostly quantitative data. Analyzing large software artifacts reveals significant trends that may generalize, but tells us little about why these trends are observed. We fill this gap by directly interacting with users and organizations. Conversely, lab studies and user interviews help us generate hypotheses which we can then investigate quantitatively.

1.4.2 Research Limitations and Challenges

The primary difficulty when studying tools in practice is getting access to real users, software, and defects. The engineers and organizations we recruit to study have other priorities, and only limited willingness to assist us in our research. Many commercial users were reluctant to participate, in part to protect the proprietary nature of their code, and in part to avoid publishing information about the number of defects they shipped in previous releases. As a result, while we had ready access to many open source projects, we had limited access to commercial code bases, which are needed because of their potentially different characteristics. We also had limited access to static analysis tools other than FindBugs. Early on, many vendors were very protective, not wanting their tool to end up in the hands of researchers who may criticize its performance. As vendors have started appreciating the importance of understanding the performance of tools in practice, more have become willing to participate in open research.

Another challenge is that many trends associated with static analysis warnings

in code bases are noisy. Warnings come and go as the developers update the code, and their presence or absence may not be significant. In addition some projects may have more or less of a particular kind of warning based on the type of problem being solved, or the habits of the developers, and this may not be correlated with the underlying quality of the code. In general, it is hard to extract significant trends when analyzing the history of warnings in a code base.

Some of these challenges led us to focus more on extracting anecdotal or qualitative insights, not just constructing scientific experiments. These insights can inform the way we build tools, prioritize warnings, and integrate static analysis into software processes.

Most of my studies have been conducted using FindBugs [62, 64], an open source static analysis tool for Java from the University of Maryland, which has been downloaded more than a million times and is used by companies such as Google, EBay, Amazon, Sun, and Oracle. I have surveyed about a thousand FindBugs users, visited organizations that use FindBugs, interviewed several dozen developers, and conducted lab studies with students. I have also manually inspected hundreds of warnings in various code bases, developed techniques to automatically mine software repositories and bug reports, and made technical contributions to FindBugs' analysis. Given FindBugs' strong focus on defects associated with code quality, I have rounded out my research by working with static analysis tools that have a stronger focus on code security, including tools from Fortify Software [128] and Coverity [66].

1.5 Summary and Discussion

Through my research, I have observed that static analysis does find important defects, and users respond positively when asked about the value of the warnings they receive. Users indicate that static analysis finds subtle defects that are otherwise hard to detect, and educates them on correct programming practices. In reviews of warnings output by FindBugs, users recommend fixing most of the warnings. At the same time, some warnings are not considered defects by users, some defects have a low impact in practice, and many of the important defects are also captured by good quality assurance practices. Users have also found that they need to make a nontrivial investment in static analysis to deploy warnings to developers early without impeding their productivity, baseline or triage warnings in old code, integrate the results of multiple tools into a common interface, and filter out unwanted bug patterns. With these benefits and pitfalls in mind, organizations have started experimenting with different policies and infrastructure requirements to identify the scenarios in which static analysis is cost effective.

I present some background on static analysis tools and techniques in Chapter 2. I also provide some background on how researchers study software artifacts to extract insights about the software development activity.

In Chapter 3, I discuss a number of studies that probe the opinions and perspectives of static analysis users. One of my earliest studies was a survey of FindBugs users, followed by phone interviews with some participants. Through these studies, I observed that many users had not yet established formal processes for using

FindBugs. Instead, many use it in an ad hoc way, manually running the analysis whenever they remember to do so. But in practice, users need to run static analysis tools automatically to get the most value out of them. Some users accomplished this by including FindBugs in a continuous build system, by displaying warnings on a web page, by sending out nightly emails to developers with new warnings, by displaying warnings as part of the code review system, or by displaying warnings in the IDE. These approaches all exhibited varying levels of effectiveness. Some users also reported integrating FindBugs into a bug tracking system to make it easier to report warnings.

During the phone interviews, many users indicated that they wanted to adopt more formal policies and integrate static analysis tools into their software process, but they cited several barriers preventing them from doing so. One was the large number of initial warnings displayed the first time the tool is run on an established code base. These need to be "baselined" so users can focus on the (relatively few) warnings in recently written code. Another challenge was the need to integrate the results of multiple static analysis tools into one consistent interface for consumption by developers. Often a custom integration solution was needed for each organization. Another challenge was to customize the tool, filtering out unwanted bug patterns, and creating detectors for project-specific bug patterns.

The survey results also indicated that different users emphasized different categories of bug patterns. This suggests that the relevance and importance of a bug pattern depends on the user's context. For example, an organization running many applications on a production server may not care too much about null-pointer excep-

tions, because these can simply be logged and the application restarted. A desktop application, on the other hand, can be severely impacted by null pointer exceptions. Users need to understand their context, and develop threat models that inform the importance of bug patterns, especially for security warnings. Of course, despite this dependence on the user's context, some bug patterns, such as SQL injection vulnerabilities, are always bad.

In addition to surveys and interviews, I have conducted some studies that enable me to observe users evaluating warnings directly. In some small lab studies, I made basic observations about user interaction including how long it took to review each issue, how consistently independent reviewers evaluated the same issue, and what factors concerning the interface may have influenced their review. In these studies, users consistently identified certain bug patterns as severe, and others as low impact, and were not influenced by factors such as displayed priority, presentation order, or even the insertion of bogus warnings. These studies were a precursor to a larger industrial-based study in which hundreds of engineers reviewed thousands of FindBugs warnings in a commercial code base. Through this study, developers confirmed that they preferred to fix most of the issues they reviewed, and their perspectives matched the bug rankings in FindBugs.

These user studies are important because they enable me to interact with real users and learn from their challenges and experiences. But user opinions are subjective, and it is necessary to also study more objective measures of the impact of static analysis. In Chapters 4 to 6, I describe some studies that involve manual review and automatic analysis of various artifacts of the software development process, in-

cluding code repositories and issue tracking databases. It is through these studies that I observed that not all true defects are important. Some seemingly fatal defects end up being low impact because they occur in dead code, or are masked by surrounding code. I discuss some of these scenarios in more detail in Section 4.1. This means that it is not sufficient for tools to minimize false positives. Tools find many stupid mistakes, but not all are important problems, and users are looking for the "intersection of stupid and important." It is not easy for tools to determine which warnings are important, but we can rely on some heuristics to improve outcomes.

One interesting bug pattern, illustrated in Figure 1.7, is the infinite recursive loop. In this illustration, the method calls itself recursively unconditionally. A defect like this is surely always serious because it instantly results in a *StackOverflowException*. But whenever we find this defect in production, it is usually in dead code. The infinite recursive loop is an example of what we call a "loud" bug pattern. They are very obvious, clearly incorrect, and usually result in exceptions or crashes when executed. However in practice, when found in code that has been in production for some time, they are usually not serious. This is because, if they were causing problems, they would probably have been noticed. By contrast, some more "silent" bug patterns can cause subtle software misbehavior that is difficult to debug. I discuss loud and silent defects in Section 4.2.

These two observations—low impact defects, and the distinctions between loud and silent bug patterns—help explain why some defects can persist for a long time without causing any noticeable problems in the software. But a more general explanation comes from a phenomenon we have observed, which we call *The Survivor*

```
private final boolean isEnabled;
public boolean isEnabled() {
    return this.isEnabled();
}
```

Figure 1.7: An Infinite Recursive Loop

Effect. When software is written, developers make many mistakes including some that matter, and some that do not. Over time both classes of defects will be reduced as the developers test, review, and deploy the software. But most quality assurance activities are directed more at the defects that matter. So if static analysis is run on deployed software, it is likely to find many of the defects that do not matter. There may still be some important defects left that were not caught by other quality assurance activities, but these are often drowned out by the long list of defects presented at this stage. Furthermore, it is expensive to fix defects found at this stage. In studies of detailed snapshots of student code, described in Chapter 5, I observe many instances of students expending time and effort to solve a problem that is found by static analysis. They often eventually fix the problem, but not before using up energy that would have been saved if they had the static analysis warning. The benefit of static analysis is that it can find problems early when they are cheap to fix. I discuss the survivor effect in more detail in Section 4.3.

In my research, I also occasionally focus on null pointer defects, because many static analysis tools focus on catching potential null pointer dereferences. One observation derived from reviewing some of these warnings is that not every potential null pointer dereference is a defect. There are many cases where the developer ex-

pects the value to always be non-null. Checking for null in all these places is too cumbersome and makes the code hard to read. Hence if the dereferenced value is null at runtime, this is likely because of a defect elsewhere in the code. This observation leads to another insight: many null pointer exceptions are not due to mishandling potentially null values, but due to separate logic errors which manifest as null pointer exceptions. I confirmed this by reviewing several dozen bug reports (from an open-source project) containing null pointer exceptions. This observation explains some of the limitations for static analysis tools finding potential null pointer dereferences.

The pervasiveness of null pointer exceptions makes some wonder if it is a mistake to allow null values in the first place [61]. But in memory safe languages like Java, sometimes one would prefer a null pointer exception, because it causes the program to fail fast during development if there is a bug. I discuss this and other considerations associated with null pointer errors in Chapter 6.

Ultimately for static analysis in practice, finding defects is not enough. Users need a warning management infrastructure to prioritize which issues to address, suppress unwanted issues, audit issues collaboratively with others on the team, and analyze the history of warnings. The next generation of FindBugs includes features to encourage collaborative auditing of warnings (using a cloud database to store reviews). Users will also be able to hookup FindBugs to external bug tracking systems and source browsers. FindBugs already provides features to allow users to analyze the history of warnings, though our survey indicated that few users were even aware of these features, and even fewer used them. I discuss the factors associated

with cost effective usage of static analysis in Chapter 7.

Looking forward, there are some challenges that need to be addressed to more fully exploit the potential of static analysis. Once challenge is making tools easier to extend, so users can quickly identify custom project-specific or API-specific bug patterns. Currently, in order to extend tools, users need to understand the various internal analyses, or learn a tool-specific specification language. In Chapter 8, I explore an approach to enable users to specify new bug detectors by providing examples (and counter-examples) of the defects they want to find. This approach does not require the user to learn any new languages or understand the analysis engine. The user's specifications are in the same language, and use the same API-calls as the rest of the program, so the user can continue to take advantage of features of their IDE, such as refactoring. This discussion will serve as the basis for future research.

Another challenge is the perception of static analysis by developers. Some developers retain a negative perception of static analysis, due to the tendency of early tools to emit many spurious warnings, and despite the advances of more modern tools that minimize false positives. With the emergence of agile programming, we have observed that many developers are considerably more willing invest time in writing and maintaining unit tests than they are in reviewing old static analysis warnings. While testing can potentially find more kinds of problems than static analysis, users need to be encouraged to spend appropriate amounts of time on static analysis to retain its benefits.

One persistent challenge to the advancement of static analysis is the lack of

cooperation and openness between competing vendors. Many tool vendors do not want others to know *what* defects they find, because they are concerned others will quickly develop the technology to find the same defects. But to advance static analysis, practitioners need to develop new technologies to find more difficult bug patterns. Openness and cooperation will help this goal.

Chapter 2

Background

The goal of this chapter is to briefly introduce important terms and concepts. This section assumes a basic understanding of common programming language idioms and software development processes.

2.1 Defects found by Static Analysis

The best way to understand what static analysis tools do and why they are limited is to jump into some examples. Here are 2 examples that illustrate why static analysis cannot always infer the intent of the programmer:

```
if( argument != null || argument.length() != 0 ) {
    ...
}
```

Figure 2.1: Will throw a NullPointerException if `argument` is null

```
PrintWriter log = null;
if (anyLogging) log = new PrintWriter(...);
if (detailedLogging) log.println("Log started");
```

Figure 2.2: May throw a NullPointerException

Figures 2.1 and 2.2 illustrate some scenarios where a Java NullPointerException may be thrown. In the if-statement, `argument` is only dereferenced when it

is null. Of course, it is possible that `argument` is never null and so the exception never occurs. Still we can assume without fully understanding the semantics and context that this code likely does not represent the programmers intent and should be changed. In Figure 2.2 (taken from [63]) on the other hand, it is not clear if there is a problem. A NullPointerException will be thrown if `log` is not initialized before it is dereferenced, which happens if `anyLogging` is false and `detailedLogging` is true. It may not be possible to decide if this can happen at compile time, and a tool reporting this as defective code may be issuing a false alarm.

This leads us to an important consideration when designing or evaluating tools – is the underlying analysis is sound or complete? A *sound* analysis finds every defect in the targeted class, and sometimes (often) includes false warnings. A *complete* analysis ensures every warning found is a real error (i.e., does not return any false alarms), but it may not find every problem in the targeted class. A sound and complete analysis cannot exist for non-trivial programs and defect classes, and these two properties are often traded off for each other [83]. Most analyses aim to be sound, but many modern static analysis tools are neither sound nor complete in general. Instead they aim to find as many problems as possible, while using heuristics to minimize the number of false warnings. One such tool is FindBugs, which will flag Figure 2.1 but not Figure 2.2 [63].

Sometimes programmers are asked to assist tools by providing more semantic information using annotations. In Figure 2.3(a), a C function which performs some operations on a character buffer assumes that both the buffer and its length are

passed in by the programmer[1]. If the length is incorrectly specified, this could make the application vulnerable to a buffer overflow attack. This serious security flaw plagues many C and C++ applications because of functions in the standard library that can introduce this vulnerability if used incorrectly. One solution, advanced by Microsoft in their static analysis tool PREfast [83], is to annotate the function with enough semantic information so a tool can check the relationship between the parameters. Figure 2.3(b) illustrates these annotations using the Microsoft's Standard Annotation Language (SAL). The annotation `__out_ecount(cchBuf)` on `buf` indicates that parameter `cchBuf` should be the length of `buf`. PREFast scans the code to see if this constraint is met whenever `FillString` is invoked, and outputs a warning if it is not.

2.2 Tools, Interfaces and Interaction Methods

We should point out that "static analysis" can refer to a wide range of program analysis activities, from static type checking to bug finding to program verification [33]. Static type checking is probably familiar to most programmers because it is built into many languages and enforced by compilers that prevent programmers from using typed data values in incompatible ways. In our research, we generally focus on bug finding tools (which look for program behavior that potentially deviates from the programmers intent) and property checking tools (which try to exhaustively verify that a program has a desired property).

[1]Example from http://blogs.msdn.com/michael_howard/archive/2006/05/19/602077.aspx

(a) function without annotation

```
void FillString( TCHAR* buf, size_t cchBuf, char ch) {
    for (size_t i = 0; i < cchBuf; i++) {
        buf[i] = ch;
    }
}
```

(b) annotated function

```
void FillString( __out_ecount(cchBuf) TCHAR* buf, size_t cchBuf, char
    ch) {
    for (size_t i = 0; i < cchBuf; i++) {
        buf[i] = ch;
    }
}
```

Figure 2.3: Using annotations to inform a static code analyzer

We have already mentioned two bug finding tools: FindBugs [62, 64] which is open source, and PREfast [83]. FindBugs searches for potentially erroneous Java code idioms called *bug patterns*. It includes dozens of *bug detectors* that scan the byte code and output warnings or alerts. Bug patterns are organized into categories and warnings are assigned priorities (e.g., High, Normal, Low) based on the confidence of the analysis. Other popular open source bug finding tools are PMD [4] and Jlint [3], and some commercial tools are Grammatech CodeSonar [53], Coverity Prevent [66], Klocwork [81], Fortify Source Code Analyzer (SCA) [128] and Ounce (now part of IBM) [65].

Some tools like Fortify SCA and Ounce have a particular focus: finding security problems. The output from these tools is intended to support security audits and

code reviews. Unlike other bug finding tools, these tools do not generally minimize false positives because they aim to identify all vulnerabilities that may be exploitable [33].

Static analysis tools provide a diverse array of interfaces for interaction with users. Perhaps the most basic mode is as a command line tool that analyzes software when manually invoked. Even more convenient is when tools provide plugins to popular build systems like ANT or Maven so that they can be invoked as part of the build process with minimal configuration effort. Tools invoked using one of these modes may output results to the screen but more often save the results in some data format (e.g., XML, HTML or plain text) so that they can be processed or viewed by third party applications. These results may be sent automatically to developers using email, or by posting on a website or bug database. An emerging paradigm is to present results on a dashboard associated with a continuous integration server such as Hudson [76] or Cruise Control [8], which developers already use to regularly build and test the code base.

A different paradigm is to present warnings using a standalone graphical user interface (GUI) or as part of a custom view in an integrated development environment (IDE). These modes facilitate doing advanced tasks like filtering out entire classes of warnings corresponding to unimportant defects, or assigning issues to specific developers or to be ignored. One argument for using the IDE mode is that warnings can be presented to the developer as soon as the problematic code is typed by using subtle but conspicuous markers in the editing space. The counter-argument is that this mode may affect the responsiveness of the IDE (for complicated anal-

yses) or irritate developers by highlighting issues prematurely (before they finish typing all they want to express semantically).

One expectation is that the mode of interaction affects the way developers use tools and the facilities they employ. For example, if a developer interacts through an IDE plugin that makes it easy to filter out entire classes of warnings, that developer may be more likely to tune the analysis tool to meet their needs than a developer that has to manually edit XML configuration files. Similarly developers that view warnings on a continuous integration dashboard that they check regularly may be more likely to respond to these warnings (by fixing or suppressing) than developers that have to remember to manually run the static analysis tool every once in a while.

2.3 Mining Software Artifacts

Modern software development processes rely on a number of tools to facilitate collaboration between all members of a team. These tools include software repositories that maintain every version of every file created during the process, and bug databases that store all communications associated with a defect's remediation from the moment the defect is reported to the moment a fix is verified. Other tools are build servers and continuous integration dashboards for tracking the success of compilation and unit testing for each snapshot of the code repository, code coverage and other metric reporting tools, forums and wikis for discussion and documentation, and calendars for scheduling.

Each of these tools captures a part of the story of the software development

effort from conception to deployment. I use the phrase "software mining" to refer to any effort to search or examine the information captured in these tools for the purpose of discovering trends, making prescriptions or diagnosing problems.

A concerted effort and community has formed around mining software repositories in particular, because these resources often contain the most detailed, automatically acquired information about the development process. Interesting trends can be identified by visualizing the repository, cross-matching the different data types collected (such as authors and timestamps), or comparing information in repositories with data collected from other tools such as bug tracking databases. I include some of these interesting trends in my discussion on mining software repositories in Chapter 5.

Chapter 3

User Perspectives and Experiences

In casual interactions with users of FindBugs, I often receive positive feedback about the bugs FindBugs has found, or the programming principles the user has learned from using it. On the other hand, it is not clear that these users use FindBugs as regularly as they write unit tests, for example. Throughout this project, I have sought to interact more formally and directly with users to learn about their experiences and opinions. How effectively are they using FindBugs, and what limitations are holding them back from fully adopting it?

In this chapter, I describe three sets of studies that shed some light on user perspectives and experiences. Early in this research project I surveyed about 1,000 FindBugs users and observed that most did not have formal processes for using static analysis. I also interviewed over a dozen of the participants by phone and learned about some of the challenges they had integrating FindBugs into their software process. These studies are discussed in Section 3.1.

In some follow-up studies, described in Section 3.2, I observed students in a controlled lab setting as they reviewed some preselected warnings from two static analysis tools. I captured basic information about how long each review took, and how consistently independent reviewers evaluated particular issues. These studies were a precursor to a large study in which hundreds of engineers reviewed thousands

of warnings in an industrial code base. With such a large number of reviews, we were able to quantify the opinions of the users and aggregate the results by bug pattern, age and severity to find trends. This study is discussed in Section 3.3.

In general, these studies suggest that *static analysis tools should be run automatically, otherwise users are unlikely to run them regularly.* Popular approaches for doing this include integrating warnings into code reviews, continuous or nightly builds, or the Integrated Development Environment (IDE). Each of these approaches has its limitations. And even with infrastructure in place to automatically present warnings to developers, organizations still have to decide how to deal with the large number of initial warnings, how to integrate results from multiple tools into a common interface, and how to customize tools to filter out irrelevant bug patterns and add project-specific ones.

Another lesson is that the relevance of a warning often depends on the nature of the application. Desktop applications have different priorities from server-based applications, for example. And entertainment applications such as video games have different tolerances from safety critical software such as an airplane's flight control system. Of course, some warnings are always bad, and all warnings are cheaper to fix when shown to the developer earlier in the development process.

3.1 User Survey and Interviews

From November 2007 to November 2009, I conducted an online survey to learn about the experiences of FindBugs' users, and collected 1045 responses. This was

a wide ranging and exploratory survey to generate feedback from users about how and how much FindBugs was used, how it was integrated into the software development process, which bug pattern categories were important, and which features of FindBugs were used. I also conducted about 18 informal phone interviews with consenting survey respondents in the US and Europe to better understand their context and to get more detailed information about their experiences, challenges and suggestions. Ultimately, the survey and interviews helped to provide some insight on the value that static analysis tools may bring to the software development process, and what obstacles prevent their adoption. Some results from these studies have been reported in previously published work [13, 12].

3.1.1 Methodology

The survey was prepared and delivered using Survey Monkey, a popular web-based provider of survey solutions used by many companies and researchers[1]. I targeted FindBugs users by advertising on the FindBugs web site and through its mailing lists.

There was no preset limit on the number of participants, nor was there a preselected invitation list. This means I cannot measure a response rate (which might indicate how much the survey can be generalized), and there may be some self-selection bias [24, 120]. In other words, some users with strong opinions may be more likely to provide feedback than other users. But the goal was to get as many responses as possible, particularly for the qualitative questions. And since

[1] http://surveymonkey.com

there were over 1,000 responses, some of the quantitative trends observed are likely significant. To encourage user participation, we offered prizes in the form of T-shirts and coffee mugs from the FindBugs store to randomly selected respondents. User responses were handled confidentially so that individual users cannot be identified in any reports. This research involving human participants was approved by the Institutional Review Board at the University of Maryland.

The main body of the survey includes 27 multiple choice or simple numeric questions and 2 essay questions. This organization makes it easier to quantify and analyze most of the responses, and reduces the burden on participants. Many of the multiple choice questions include a choice labeled "Other (please specify)" to allow users to provide more qualitative information. The survey begins by asking for basic demographic information such as the level of the user's education and experience, the nature of the user's code, and the type of organization the user is affiliated with. Highlights from these questions are discussed in Section 3.1.2. The final multiple choice question presents the users with a number of statements, and asks them to indicate how strongly they agree or disagree with each using a Likert scale [87]. The essay questions ask users to specify any customizations they have made to FindBugs, and for additional feedback about what they like or dislike about FindBugs, and how it affects their software development process.

At the end of the survey, participants indicated if they would be willing to be contacted for further interviews. I contacted some of the consenting participants and arranged phone interviews. Each interview lasted about 30 minutes, and was a free-flowing conversation about how users discovered FindBugs, how their orga-

nization handled quality assurance, and what their priorities were. The interviews were recorded with the permission of participants, and later partially transcribed to preserve the main points for future review and analysis.

As part of the effort to construct the survey and prepare for the interviews, I conducted some pilot surveys and interviews with users in the Northern Virginia Java Users Group[2] and the Fraunhofer Center at the University of Maryland[3]. These pilots helped to refine some of the questions in the final survey, and indicated that the study results would be difficult to analyze if not reduced to simpler multiple choice questions.

The following sections highlight some of my main observations.

3.1.2 Survey Demographics

Table 3.1 shows the basic demographic statistics from the survey. The results indicate that many participants were experienced industry professionals. Users had an average of 10 years of professional experience working on software projects, and an average of 3 years of experience using automatic fault detection tools like FindBugs. The top primary roles users identified were Software Developer, Software Architect, Project Manager and Consultant/Specialist. Many users also indicated that they had secondary and tertiary roles as Quality Assurance/Testing and Build Engineers. Only a handful of responses were from researchers or students. This skew towards industry participation supports my goal of understanding how FindBugs is used by

[2]http://novajug.org/

[3]http://fc-md.umd.edu/

Table 3.1: Survey Demographic Statistics

Experience		Organization Size	
Average Professional Experience	10 years	1 to 50 employees	28%
Average Experience with Fault Detection Tools	3 years	50 to 200 employees	16%
Average Experience with FindBugs	2 years	200 to 1,000 employees	17%
Primary Role or Job Function		1,000 to 10,000 employees	20%
Software Developer	56%	10,000 or more employees	19%
Software Architect	21%	Project Age	
Project Manager	7%	Less than 6 months	26%
Consultant/Specialist	6%	6 months to a year	22%
Affiliate Organization Sector		1-2 years	27%
Technology and Communications	49%	2-5 years	42%
Finance and Insurance	11%	Over 5 years	25%
Services	9%		
Education	7%		

Percentages are of the respondents who answered the question.
For project age, users may select more than one option.

professionals.

It is also interesting to note the industries that participants represent. Unsurprisingly, most participants worked in technology and communications companies, but there was also a strong showing from the finance sector, where critical software flaws can lead to high profile failures or security breaches.

The responses were distributed uniformly among organizations of different sizes, with around half of the responses from small to mid-size organizations, and half from large organizations with more than 500 employees.

The sizes and ages of code bases subjected to static analysis varied widely. 10% of respondents reported running FindBugs on code bases larger than 1 million lines of code and the plurality of respondents (42%) reported that their code bases were 2 to 5 years old.

3.1.3 Is FindBugs Useful?

Many users expressed strong positive sentiments about FindBugs, both through the opinion-based multiple-choice questions and the open-ended essay questions. Table 3.2 summarizes a question in which users were asked to agree or disagree with several statements. Most users agreed or strongly agreed that their investment in FindBugs was worthwhile, that it had found serious problems, and that warnings were easy to understand and fix. The statement that FindBugs has found serious problems that users fixed is particularly interesting, because it suggests that users perceive that they are getting real value out of it. Whether this translates into consistent and regular usage is another matter, which I discuss in Section 3.1.4.

The results in Table 3.2 also show that only half of the users agree or strongly agree that FindBugs was speeding up the quality assurance process, but another 30% were indifferent to this question. Static analysis may speed up the quality assurance process by identifying problems sooner or enabling faster code reviews, but it may also slow it down by giving developers additional work.

We also asked users if the presence of static analysis affected other parts of their quality assurance process. Only a few users (18%) felt that project managers rely too much on the number of bugs reported when measuring code quality, and even fewer users (8%) said that FindBugs reduced the number of unit tests they write. However a large number of users were indifferent on both questions (52% and 39% respectively) so these results may not be representative of survey respondents.

I also received qualitative feedback that strongly indicates that users found

Table 3.2: Percentage of users who Agree or Strongly Agree with statements about FindBugs

Our investment (in time) in FindBugs has been worthwhile	90%
FindBugs has found serious problems in projects I or my team have worked on	81%
FindBugs warnings and bug descriptions are easy to understand	75%
FindBugs warnings have generally been easy to fix	67%
FindBugs speeds up our quality assurance process	50%
Project managers rely too much on "number of bugs" reported by tools like FindBugs when measuring code quality.	18%
FindBugs has had the (unintended) effect of reducing the number of unit tests we write	8%

FindBugs valuable. For example, one user commented:

> We have a project where FindBugs found some serious problems in highly critical safety related software, issues that might have caused it to run less efficiently or wrong. FindBugs saved our collective (butts). It's as simple as that.

One recurring theme was the educational value users said they received from FindBugs. It taught them things about Java they did not know previously. One manager I interviewed insisted that junior developers use FindBugs regularly, just so they can learn good practices and understand some nuances about the language.

Not all comments were positive. Some users complained about the performance of FindBugs in the IDE, or about how difficult it is to extend FindBugs with new plugins.

During interviews, we tried to find out how users heard about FindBugs and what motivated them to start using it. Many users heard about FindBugs through presentations at the annual Java One conference[4] or through online videos, podcasts and articles.

The motivations for using FindBugs varied widely. Some users reported that even brief trials running the tool taught them new things about Java and its potential pitfalls. Some of these respondents used FindBugs to ensure compliance with coding standards and to educate new developers. Many users highlighted the fact that FindBugs found real correctness problems, not just style issues, as reason to choose FindBugs over other tools like CheckStyle or JLint.

One user was an outside consultant called in at the end of each major phase to do quality control. He ran FindBugs as a first step to look for clusters of problems and sniff out problematic trends. He expressed an unusual concern that if developers use FindBugs regularly, they would "tune" the code base to the tool, perhaps removing low priority warnings that could identify potentially defective modules. On the other hand, if the problems found and fixed improve software quality, then the organization and clients are well served.

Another user worked on an Agile software development team [60] that emphasized test-driven development and ran static analysis tools only at the end of each iteration. His perspective was that static analysis was useful for finding potential future problems that did not immediately manifest in tests. For example, FindBugs flags the practice of calling a non-final method from a constructor. This is a

[4]http://java.sun.com/javaone/

Table 3.3: Lack of formal policies for using FindBugs

Our developers only occasionally run FindBugs manually	60% of users
No policy on how soon each FindBugs issue must be human reviewed	81%
Running FindBugs is NOT required by our process, or by management	76%
FindBugs warnings are NOT inserted into a separate bug tracking database	83%
No policy on how to handle warnings designated "Not A Bug"	55%

potential problem only if the class is subclassed.

Few users reported doing any initial cost-benefit analysis to measure the return on investment in FindBugs. In some cases, adoption was pushed by one champion who was convinced it would bring value. This champion would lead the effort to integrate FindBugs into existing processes, or do the initial work to filter out unwanted bug patterns. One user recently joined his company and found that a previous effort to use FindBugs had failed as developers stopped running it daily. So he pushed to automatically run it as part of the continuous build, taking some of the onus off developers.

3.1.4 Users Lack Formal Processes

One of the most revealing observations from our survey was that most respondents did not seem to have any formal policies for using FindBugs and other static analysis tools (Table 3.3). Their organizations just expected developers to run tools once in a while, and they had not really considered questions like: "who decides if a warnings should be fixed?" or "how should we filter out false alarms?".

Some teams did identify the need for a way to suppress warnings that are

Table 3.4: Handling issues designated "Not A Bug"

Filter out using FindBugs filters	25% of users
Suppress using @SuppressWarnings	17%
Close in a bug tracker or database	5%
No policy	55%

not bugs or that are low impact issues (Table 3.4). FindBugs' filter files[5] were the most common method, followed by source level suppression using annotations (such as @SuppressWarnings). In the interviews, one user explained that source level suppression using annotations was attractive because the suppression information is readily available to future code reviewers. Other users had the practice of fixing all issues identified by FindBugs to make the issues go away. Some did this because they did not want new issues to be drowned out, and others did this because they felt it made the code cleaner. As one user put it: "the effort to reformulate source code to avoid FindBugs warnings is time well spent."

Another survey question focused on who decides that an issue should be fixed (Table 3.5). In many cases, the person who writes the code is responsible for reviewing the warning, deciding if it is relevant, and resolving the issue. Other approaches include having peer reviews or team reviews. Warnings found in older code can be hard to fix and require approval from management. One question all this raises is whether two different individuals will interpret warnings the same way (and hence make similar review judgments). If reviewers often reach different conclusions, then

[5]FindBugs' filter files are XML files that contain references to specific bug patterns, packages, classes, methods, or fields, which can be included or excluded from analysis results.

Table 3.5: How do you or your project team decide when a warning is "Not A Bug"?

The reviewer makes this decision independently	38% of users
The reviewer makes this decision for trivial cases, but nontrivial cases go to a team or to management	17%
At least two reviewers must agree	6%
The issue must be reviewed by a team or management	5%
No policy	31%

organizations may need to be more careful about how they choose reviewers. I have explored this question with some lab studies and a warning review (described in Sections 3.2 and 3.3 respectively) which indicate that independent reviewers are usually consistent with each other. This suggests organizations can save money by not requiring multiple independent reviews for most issues.

During interviews I tried to understand why adoption of FindBugs was unsuccessful in some cases. One reason given is that developers were discouraged by the large number of warnings presented the first time the tool is run on existing code. Developers are often resistant to changing code that has been in production for a while and may perceive the warnings in low regard because they refer to problems that have not manifested during execution. Organizations in this situation needed to either filter out warnings older than a certain time period, or make a concerted effort to remediate certain bug patterns all at once.

Another problem that sometimes came up was that users did not know how to write custom bug detectors and found the prospect of learning to do so daunting. Some users expressed that they did not know what problems FindBugs catches

(the list of bug detectors is quite long), and hence they were worried that writing a custom detector might be reinventing the wheel. FindBugs does not generally use sound analyses so the absence of a warning does not imply the absence of the problem.

Finally, we interviewed at least one developer who expressed that FindBugs was imposed on his team by management and a separate security team. This imposition came about because an earlier developer error had caused a major security breach in one of the organization's web applications (the organization was a State Department of Health). One of the responses was to make developers review all warnings regularly and document the ones they did not fix. The security team saw this requirement as a first layer of defense, but the developer expressed some reservation because he felt that few of the warnings were serious bugs.

3.1.5 Issues Users Care About

FindBugs classifies warnings for each bug pattern into high, medium, or low priority groups depending on the severity of the issue and the confidence of the analysis. Part of the goal is to reduce the number of false positives among issues that receive a high priority label. Our survey indicates that most users review at least the high priority warnings in all categories (Table 3.6). This is the expected outcome, since high priority warnings are intended to be the sorts of problems any user would want to fix. A surprising number of users also review lower priority warnings (though the review categories vary from user to user). This indicates

Table 3.6: Proportion of users that review at least high priority warnings for each category

Bad Practice	96% of users	Malicious Code Vulnerability	86%
Performance	96%	Dodgy	86%
Correctness	95%	Internationalization	57%
Multithreaded Correctness	93%		

that while high priority warnings are relevant to most users, lower priority warnings may or may not be relevant depending on the user's context. One could even imagine providing preconfigured settings for different contexts that emphasize or deemphasize certain bug patterns based on whether the subject application is a web application or desktop application.

During interviews some users, particularly those building web applications, were more interested in security related issues (such as SQL injections) but at least one user indicated that input validation was their primary defense and this made many security warnings obsolete. FindBugs does not have many detectors dedicated to security issues, so most users were relying on FindBugs to find issues related to correctness such as dereferences of potentially null variables. Users also reported looking for synchronization issues and race conditions, problems that can manifest in multithreaded environments.

3.1.6 Summary

The survey and interviews provide the first clues that users are getting some value out of FindBugs, including some educational value because it informs them

of good programming practices. But this study also indicates that some users are struggling to integrate FindBugs into their regular software development process. These users confirm that just using static analysis in an ad hoc way does not seem to be sufficient because developers may forget to run it regularly, or new warnings may be drowned out by stale issues that have not been resolved or suppressed. These users also confirm that they are actively trying to make the initial investment to establish infrastructure that will enable them to baseline old issues, file some new issues automatically in their issue tracking systems, or aggregate the results of multiple static analysis tools into one interface. The observations from this survey inform some of the questions I ask throughout my research. I will return to some of the responses from the survey at relevant points in future chapters.

3.2 Lab Based Controlled Studies

The surveys and interviews indicated that users perceive that FindBugs' warnings are generally valuable and worthwhile reviewing. But all warnings are not equal, and it would be interesting to observe user perceptions of different specific warnings. To facilitate this discovery, I conducted a number of studies that bring users into direct interaction with FindBugs and other static analysis tools. I describe some lab studies involving a few participants in this section, and a larger study involving hundreds of professionals in Section 3.3.

The lab studies enabled us to observe students interacting with static analysis tools in a controlled environment. These studies were partly done in preparation for

larger studies conducted with hundreds of engineers in their working environments, and hence had numerous goals. One goal was to see how long it would take to review each warning and decide if it is worth fixing, if it is low impact, or if it is not a problem. This enables us to make a rough estimate of the cost of using static analysis.

Another goal is to determine if multiple independent reviewers agree about the significance of each warning. Sometimes when researchers and tool vendors talk about warnings, we assume that the rightness or wrongness of the warning is clear. But separate users may place a different value on each warning, even disagreeing about whether the problem is plausible. Differences may also come about because users make errors in judgement. These studies enable us to improve our intuition about user consistency.

The process of observing users as they interact with tools may also reveal insights about the practice and rigor of reviewers. What resources do they rely on to understand a warning and decide if it is a problem? And what sorts of mistakes do reviewers make in deciding that a warning is not a bug, or that it is a bug? What if we insert a fake warning? Will users be careful enough to detect this deception?

In a similar vein, we want to investigate what biases may influence a reviewer, including the priority label assigned by the static analysis tool, and the order in which warnings are presented. Some users may trust that the analysis is correct, and not feel the need to manually verify its assertions. Other users may be skeptical, and refuse to accept that there is a bug unless they can prove it to themselves.

Of course, student reviewers in a lab do not precisely represent the decisions

made in the real world, where professionals worry about the cost of fixing warnings. One way to improve this limitation is to ask reviewers to make some of the considerations engineers make in practice, including whether the alleged problem is likely to occur often or only rarely, and whether the problem is in deployed client software (and hence a software patch will need to be released), or just in local code under development. We attempt this in one of the lab studies by providing users with a checklist for them to go over as they review each warning.

The first lab study focuses on capturing the review time, and consistency of independent reviewers. This study is described in Section 3.2.1, and reported in more detail in [13]. The second study introduces a checklist, and focuses on observing how different factors (including order, priority, and bug pattern) correlate with reviewer evaluations. In this study, we randomized the order and priority of the warnings, and even inserted some fake warnings. This study is described in Section 3.2.2, and reported in more detail in [15].

Ultimately, the goal in both these studies is to make qualitative observations about the users' interactions with tools.

3.2.1 Study 1: Review Times and Consistency

This study involved two static analysis tools: FindBugs and Fortify Source Code Analyzer. Fortify SCA is a commercial static analysis tool that specializes in finding potentially exploitable security vulnerabilities in source code. We recruited 12 students (10 graduate and 2 undergraduate) from the University of Maryland's

Computer Science Department, using email, fliers and word of mouth. We did not require our users to have any prior experience using static analysis tools, and none of them had used the tools they were reviewing. Users had between 1 and 10 years of programming with Java (the average was 6 years) and between 0 and 5 years of using the Eclipse IDE (the average was 3 years). The first six users reviewed warnings from FindBugs while the next six reviewed Fortify SCA warnings.

Both tools were run on DSpace[6] (version 1.4.2), an open source web based application for accessing and managing text, audio, video and other resources generated during research and teaching. DSpace was one of the benchmarks in the 2008 Static Analysis Tool Exposition[7] organized by the National Institute of Standards and Technology (NIST). Both FindBugs and Fortify SCA participated in the exposition.

Participants were asked to review 23 FindBugs warnings (including correctness, bad practice and multi-threaded correctness warnings) or 21 Fortify SCA warnings (including warnings on HTTP Response Splitting, SQL Injections and Race conditions). Users were asked to rate warnings on a 3 level scale using labels native to the tools. For FindBugs the levels were "Must Fix," "Low Impact," and "Not a Bug." For Fortify SCA the levels were "Exploitable," "Suspicious," and "Not an Issue."

The studies were conducted using the Eclipse IDE and corresponding plugins for both tools. To facilitate analysis we logged some user actions (such as selecting

[6]http://dspace.org/

[7]http://samate.nist.gov/index.php/SATE

a view or rating a warning) using a customized version of the HackyStat[8] Eclipse plugin [74], which transparently collects data about user activities and sends it to a central repository.

The experiment was divided into four parts: a tutorial, a practice session in which participants reviewed four warnings, a timed main session and a background survey. During the practice session, participants were asked to "think out loud" as they performed the review to provide qualitative information about what decisions they were making and why. During the main session participants reviewed the assigned issues starting with the highest priority warnings, mimicking the way tools usually present the warnings to users.

During the tutorial, participants viewed a web page which described the tools with illustrations and outlined the tasks the user was expected to perform. In particular, the tutorial showed users how to navigate through warnings, designate a rating to each one, and add comments. The tutorial for Fortify SCA was longer because it included more detailed information about HTTP Response Splitting, SQL Injection and Race Conditions. These descriptions and examples were adapted from the information provided by Fortify SCA. In addition the Fortify SCA tutorial included a checklist of steps for users to follow. I designed the checklist based on the documentation provided by Fortify. The checklist was intended to reduce the complexity of some of the tasks and to ensure participants consider all relevant factors before choosing a designation. An example of a checklist for SQL Injections is shown in Figure 3.1. Participants were encouraged to ask any questions they had

[8]http://hackystat.org

Use the following Checklist to determine if a segment of code is an SQL injection

1. Does data enter the program from an untrusted source? If NO, then not an SQL injection
2. Is the data used to construct an SQL query? If NO, then not an SQL injection
3. Is the data validated between its entry and where the constructed SQL statement is executed? If YES, then GOTO 3.a., otherwise GOTO 4
   (a) Is the data validated using blacklisting (removing or escaping potentially malicious characters)? If YES, then code is still vulnerable to SQL injection because blacklisting is not as effective
   (b) Is the data validated using white listing (only allow certain predetermined inputs)? If YES, then not an SQL injection
4. Do you see any other security mechanism to prevent SQL injection? If YES, then use your best judgment to determine if the security mechanism is effective

Figure 3.1: SQL Injection Checklist

to the experimenter.

3.2.1.1 Results and Observations

Table 3.7 shows the review times for FindBugs and Fortify SCA. The times for each tool are sorted from shortest to longest. Users spent an average of 98 seconds reviewing each FindBugs warning. This average drops to about 87 when the last (outlier) is excluded. Users spent about 120 seconds for each Fortify SCA issue. It was interesting to note that the review times were not very long for either tool in this simple study, and in particular that Fortify SCA reviews were not much longer than FindBugs despite its increased complexity.

In Table 3.8 we measure how much reviewers agree with each other in designating a warning to a level on the 3-level scale for their tool. For example, all

Table 3.7: Review Times for FindBugs and Fortify SCA

FindBugs	Fortify SCA
73	88
76	90
90	103
97	108
98	143
151	189
Average: 98	120

Table 3.8: Level of Agreement among six reviewers

Tool	Total Warnings	6 agree	5+ agree	4+ agree
FindBugs	23	7	12	21
Fortify SCA	21	3	6	11

6 reviewers made the same decision for 7 of the FindBugs warnings, but only 3 of the Fortify SCA warnings were unanimous. The results indicate a greater level of agreement among FindBugs users which may reflect that the warnings are simpler to understand.

One interesting exception occurred during the practice session and is illustrated in Figure 3.2. Here, a switch statement is missing breaks and each case falls through to the next one. FindBugs flags this as a bug but 4 users concluded that the programmer intended the fall through to initialize all variables. The other 2 users reviewed this as Must Fix, and may have not noticed the programmers possible intent. Of course, the programmer in this example should probably insert comments indicating that breaks were omitted intentionally if in fact this is the case. In

```
public DCDate(String fromDC) {
    ...
    switch (fromDC.length()) {
    case 20:
        // Full date and time
        hours = Integer.parseInt(fromDC.substring(11, 13));
        minutes = Integer.parseInt(fromDC.substring(14, 16));
        seconds = Integer.parseInt(fromDC.substring(17, 19));
    case 10:
        // Just full date
        day = Integer.parseInt(fromDC.substring(8, 10));
    case 7:
        // Just year and month
        month = Integer.parseInt(fromDC.substring(5, 7));
    case 4:
        // Just the year
        year = Integer.parseInt(fromDC.substring(0, 4));
    }
    ...
}
```

Figure 3.2: Switch statement with no breaks. Some users concluded this was not a bug while others declared this a Must Fix

another case, FindBugs flagged a possible null pointer dereference that would only occur if an earlier exception was thrown. The programmer provided a comment that the exception "should never happen," but 4 users still concluded that the warning was a "Must Fix" while the other 2 reviewed this as Not an Issue.

With Fortify SCA warnings, we noticed that some of the disagreement may have resulted from reviewers getting confused as they went through the trace. In one case half the users rated a HTTP Response Splitting warning as Exploitable while the other half rated it as Not an Issue. The comments indicate that some of

those who thought it was not an issue concluded that the offending variable was sanitized using the `URLEncoder.encode` method, but in fact a different variable was sanitized.

A brief survey administered after each study captured more feedback about the user's experience. In the survey 5 of the 6 FindBugs users indicated that they generally understood the warnings or that they were familiar with the problems from previous experience, while 4 of 6 users indicated that it was not difficult to decide if a warning represented a bug. But users were split over whether it was easy to distinguish between "Must Fix" and "Low Impact" bugs. Some of these users complained that they were not familiar with the code and could not investigate too deeply, so it was hard to decide the real impact of the warning.

In the Fortify SCA survey, 5 of 6 users indicated that they understood the warnings, but most still thought it was difficult to decide if a warnings was a bug. In addition 5 of 6 users found it hard to distinguish between "Exploitable" and "Suspicious" issues. Some users said they were conservative, rating as Exploitable any issue for which a reasonable chance of failure existed.

Both FindBugs and Fortify SCA provide a clickable trace for each warning that contains links to relevant parts of the code. FindBugs trace links to affected fields and classes and the line where the warning occurs. Fortify SCA's traces contained a call hierarchy tracing the cause of each issue from the source (e.g., where a taint enters the program) to the target where the vulnerability is exposed. Fortify SCA's traces were much longer than those in FindBugs and users relied more on the traces in Fortify SCA to understand the warnings. One observation is that users often

looked beyond the trace, referring to the type hierarchy or just doing a text search to find out more about variables and types. Some users indicated that even after going through the trace to confirm the warnings, they did not know whether to trust the tools. Such users would spend some time trying to find a clue that might suggest that the tool was wrong. One Fortify SCA user indicated that they were not confident enough to rate any issues as Not an Issue (there was also one FindBugs reviewer that did not rate any warnings as Not a Bug).

3.2.2 Study 2: Factors Influencing Review

This study focused on how some factors may influence a reviewer's judgment about the severity of a warning and the reviewer's willingness to fix it. We looked for correlations between these factors and reviewer responses to a checklist. Some factors we could consider are:

- Displayed Priority: Is the reviewer more likely to take a warning more seriously if the tool assigns a higher priority label or color to it?
- Order: Is a reviewer more skeptical about the initial warnings or the warnings near the end of the review?
- Bug Pattern: Are certain types of warnings inherently more interesting to reviewers?
- Context: Is the reviewer examining modules that have already been deployed or modules that are still under development?

- Impact: The impact of an unfixed issue could range from warning messages sent to a log file, to serious logical errors.

- Perception (of the tool): Do reviewers assume that a tool's analysis is correct or do they try to verify assertions made by the tool? For example, if a tool declares that a variable is null at a critical point in the code, do users inspect the code to verify this or assume that the tool is correct?

If factors other than the bug pattern of an issue are influencing the reviews of an issue, then organizations may need to consider these factors when adopting policies to govern the use of static analysis. In this study, we considered the influence of displayed priority, order of presentation, bug pattern and context. The only factor we controlled directly is the bug pattern. The priority labels next to each warning, and the order in which the warnings were presented was randomized. We focused on FindBugs warnings associated with possible null pointer exceptions because these tend to be unambiguous, but we were still able to consider a wide range of bug patterns (see Table 3.9).

To review each warning, participants completed a checklist (shown in Table 3.10). The first checklist question tests the reviewer's understanding of the FindBugs warning. Our past research has indicated that most FindBugs warnings should be easy to understand, but some users may feel they need more information (through unit tests). This question also gives users the opportunity to quickly identify those warnings they think are bogus, and hence avoid answering the other checklist questions. Users who do not understand the warning also skip the remaining checklist

Table 3.9: Bug Patterns used in controlled study

Bug Pattern	# of Issues
NP (Always Null): A NullPointerException is always thrown when the referenced line is executed	3 (1 Fake)
NP (Null On Some Path): There is a path through the code that, if executed, is guaranteed to throw a NullPointerException	2 (1 Fake)
NP (Null Parameter Dereference): A method call passes null to an unconditionally dereferenced parameter	2
NP (Unwritten Field): An uninitialized field is read	1
PZLA: Prefer to return a zero-length array instead of null	1
RCN (Redundant Check for Null): Check of a value that is known to be non-null. May indicate a logic error.	4 (1 Fake)

questions.

The four checklist questions that follow give reviewers multiple scales for measuring the severity of the warning and the level of their response. Reviewers indicate severity in terms of how often the issue occurs and how the issue affects code behavior. Reviewers indicate the level of their response by indicating whether they would fix the bug (and in what contexts) and whether they would filter this bug pattern out of future reviews.

The checklist responses all range from strong responses (e.g., substantial deviation) to weak responses (e.g., minor deviation) to negative responses (e.g., no deviation). This design is useful when we analyze the results because it allows us to compare the different checklist questions (by considering only strongest responses, for example).

Table 3.10: Checklist Questions for each Issue

ISSUE UNDERSTANDING: Which of the following statements best describes your understanding of the problem?

- I have enough information to understand this problem
- I need to write a test case to better understand this issue
- I think this is a bogus issue which cannot occur and does not affect code behavior
- I do NOT understand this issue

ISSUE OCCURRENCE: Under what circumstances can the behavior described by this issue occur?

- Under normal, intended use
- Only in situations that do not appear to be among intended use cases
- I do NOT think it can occur at all

CODE BEHAVIOR: What is the apparent impact of the issue on the behavior of the code?

- It behaves in a way clearly at substantial odds with the intended behavior
- It does NOT behave as intended, but difference does NOT appear to be substantial
- No apparent difference in behavior

FIX DECISION: What do you recommend? (Select all that apply)

- Definitely change the code to fix the problem
- Change the code only if risk of impacting deployed code is not high
- Change the documentation to make code clearer
- No changes necessary, code is OK

FILTERING DECISION: Would you want a static analysis tool to show you issues like this?

- Yes, definitely, even in old code
- OK, particularly in new code, or if there aren't a lot of them
- I'd rather not bother looking at such issues

Finally we introduced some bogus warnings to see if reviewers would catch these or trust the tool's analysis. Figures 3.3 and 3.4 illustrate some of the bogus warnings that we inserted. In Figure 3.3, FindBugs incorrectly asserts that `argument` will always be null when it is dereferenced in the if-statement. But careful

inspection of the closed-circuit disjunction should convince the reader that `argument` is dereferenced ONLY when it is NOT null. Still some reviewers may assume FindBugs got its analysis correct, especially those who have reviewed a similar warning in which the dereferenced variable was null due to programmer error.

```
637     private void handleUidl( String argument ) {
638         //Return all messages unique ids
639         if( argument == null || argument.length() == 0 ) {
640             ...
642         }
643     }
```

FindBugs: "Null pointer dereference of `argument` on line 639"

Figure 3.3: Bogus Warning – FindBugs incorrectly asserts that the dereference of `argument` in the if-statement will throw a NullPointerException

Figure 3.4 is a more subtle and ambiguous bogus warning. FindBugs asserts that the null-check on line 133 is redundant because `listenAddress` is known to be non-null (because it was dereferenced on line 117). The FindBugs analysis misses the assignment to `listenAddress` on line 132 which may return null (FindBugs does not do interprocedural analysis so it usually cannot assert that this return value is non null). But the FindBugs warning does not expressly state where `listenAddress` is known to be non-null, so the reviewer may assume that FindBugs is doing an interprocedural analysis to determine the nullness of the value on line 132. The question is does the reviewer trust this interprocedural analysis (and hence remove the redundant null-check), or does the reviewer conclude that the analysis may be wrong. (Of course, the reviewer is free to drill down into the method call to try and

```
116     InetAddress listenAddress = ...
117     if (listenAddress.isSiteLocalAddress())
118         isLocalRun = true;
119     try {

131     } catch (IOException e) {
132         listenAddress = getLocalHostAddress();
133         if( listenAddress != null ) {
134             address = listenAddress.getHostAddress();
135         }
136     }
```

FindBugs: "Redundant nullcheck of `listenAddress` on line 133, which is known to be non-null"

Figure 3.4: Bogus Warning – FindBugs incorrectly asserts that `listenAddress` is known to be non-null because it was dereferenced on line 117

make this determination.)

As with the last study, we recruited 12 students (11 graduate students and 1 undergraduate), 2 of which had experience with FindBugs. All participants saw the same warnings and other variables were randomized. The application under review was Java Email Server[9] (Version 1.6.1) a SMTP and POP3 email server. The application was modified to insert more warnings including some fake warnings. All the inserted code was derived from real warnings seen in other applications.

Also as in the last study, participants were given a tutorial, followed by a practice review, an untimed main session and a brief survey.

[9] http://ericdaugherty.com/java/mailserver/

Table 3.11: Issue Understanding vs Bug Patterns

	NP	PZLA	RCN	Fake
Understand Real Bug	63	12	30	25
Understand with Test Case	4	0	2	2
Bogus Warning	5	0	3	9
Don't Understand	0	0	1	0

NP = Potential Null Pointer Dereference

PZLA = Prefer zero-length array

RCN = Redundant Check for Null

3.2.2.1 General Results

Most of our analysis focus on the four review questions, not on the issue understanding question. As Table 3.11 shows, most users understood most of the issues. In the case where a user rated an issue as a bogus warning, the negative response is automatically entered for the four review questions. One observation is that the 3 fake warnings received a bogus warning only 9 out of 36 times. But as our discussion in Section 3.2.2.3 will reveal, reviewers did not appear to be fooled by the fake warnings based on their responses to the other checklist questions. The low count on the issue understanding question likely indicates that users misunderstood the question and assumed it only referred to how much they understood the static analysis warning.

3.2.2.2 Consistency of reviews

There are two types of consistency we are interested in. One is the consistency across reviewers for each checklist question; in other words, how much do reviewers

agree with each other when they review an issue. The second is the consistency across checklist questions; in other words, do reviewers tend to give a strong response on one question but a weak or negative response on another question for the same issue.

To consider the consistency across reviewers, we count the number of times they agree for each question and issue in Table 3.12. (For example, the table shows that 8 reviewers agreed on the issue occurrence decision question for the first issue.) In 30 of the 52 cases, 8 or more reviewers agreed, and all issues had at least one question in which 8 or more reviewers agreed. But only three issues had 8 or more agreements for all questions (issues 2, 7 and 8). This highlights the idea that the consistency across reviewers for a particular issue depends on what question they are trying to answer. For example, all users agree that issue 10 (a redundant check for null) does not cause deviation from intended behavior, but are split on whether to fix or filter this issue.

Table 3.12 also supports our investigation into the consistency across checklist questions by shading each cell to indicate which answer reviewers are agreeing on. The dark shade represents agreement at the strongest level, the light shade represents agreement at the middle level, and the absence of shading represents agreement at the weakest level. The results show agreement at the strongest level for most questions among the real issues and agreement at the weakest level for the fake issues (issues 2, 4 and 11, at the bottom of the table). The exception is with the redundant check for null issues (10 - 13) where users rate the issues as normally occurring, but give weak responses to the other questions.

Table 3.12: Level of Agreement for Each Issue

Issue #	Occurs	Behavior	Fix	Filter
1	8	7	8	8
3	6	9	11	10
5	8	10	8	7
6	5	7	7	8
7	8	8	8	8
8	10	12	10	11
9	10	5	7	7
10	7	12	6	6
12	5	8	6	6
13	8	7	8	6
2	10	11	10	9
4	8	8	7	4
11	6	12	7	7

Another way we measure consistency across checklist questions is to count the number of reviews in which all four questions got exactly the same level of review. Out of 156 reviews, 82 (or 53%) presented exactly the same level of response for all four questions, and 123 (or 79%) had all questions at the same level or off by one.

Another measure we use is Pearson's correlation coefficient for each pair of questions for all reviews (see Table 3.13). To enable this, we encode the checklist responses numerically from 3 to 1 with 3 representing strongest responses. While this is not a perfect measure, the high positive correlations do suggest that most reviews were consistent across questions.

Table 3.13: Correlation Coefficients for Checklist Responses

	Occurs	Behavior	Fix	Filter
Occurs	1	.69	.75	.75
Behavior		1	.80	.80
Fix			1	.75
Filter				1

3.2.2.3 Factors Affecting Reviews

In this section we consider how different bug patterns, displayed priorities and presentation orders affect the reviews of issues. One way to do this is to consider the number of strong responses for each factor level relative to the number of strong responses across all factor levels.

Consider Table 3.14, where the columns represent different bug pattern groups and the rows represent the number of strong responses for each checklist question. The last column (labeled *Agg*) aggregates the number of strong responses across all bug patterns for each question. For example, 83 out of 156 reviews (53%) gave strong responses to the Issue Occurrence question. We go on to calculate the proportion of reviews giving strong responses for each bug pattern group. For example, 44 out of 72 reviews (61%) for the NP bug pattern group gave strong responses to the Issue Occurrence question. To test the significance of this relative to the overall strong response rate of 53%, we do a chi test comparing the number of strong responses (44) and remaining responses (28) to the expected value for these two quantities (based on the overall rate). In Table 3.14, we indicate the result of the chi test with a blue-shaded (+) or a red-shaded (-) if the ratio for the bug pattern is significantly

Table 3.14: Strongest Checklist Reviews vs Bug Pattern Groups

	NP	PZLA	RCN	Fake	*Agg*
# of Reviews	72	12	36	36	156
Normally Occurs	44	10	20	9	83
chi-test ($p < 0.05$)	N	(+)	N	(-)	53%
Substantial Deviation	53	3	9	4	69
chi-test ($p < 0.05$)	(+)	N	(-)	(-)	44%
Always Fix	52	3	10	5	70
chi-test ($p < 0.05$)	(+)	N	(-)	(-)	45%
Always Show	52	5	10	8	75
chi-test ($p < 0.05$)	(+)	N	(-)	(-)	48%

greater than or less than the ratio for all bug patterns respectively, or with N if there is no significant difference. The chi test is limited because the size of some factors is quite small, but this allows us to visualize some general trends. We are also assuming the factor levels are independent of each other.

The results in Table 3.14 indicate an effect due to the bug pattern group: NP issues were more likely to receive strong responses while RCN and Fake issues were less likely to receive strong responses. We can further break down the NP bug group into distinct bug patterns. Table 3.15 shows that the Always Null and Read of Unwritten Field patterns had many strong responses, while the Null on Some Path and Dereference of Null parameter had fewer strong responses. The low count of strong responses for Fake issues indicates that users were not fooled by the fake issues and did not assume the analysis was correct.

Table 3.16 shows another analysis, this time with the priority label displayed next to each issue. The results indicate that while the rate of strong responses was

Table 3.15: Strongest Checklist Reviews vs NP Patterns

	Always Null	Null on Some Path	Param Deref	Unwritten Field
# of Reviews	24	12	24	12
Normally Occurs	13	8	13	10
chi-test ($p < 0.05$)	N	N	N	(+)
Subst. Deviation	16	10	15	12
chi-test ($p < 0.05$)	(+)	(+)	N	(+)
Always Fix	19	8	15	10
chi-test ($p < 0.05$)	(+)	N	N	(+)
Always Show	18	7	16	11
chi-test ($p < 0.05$)	(+)	N	N	(+)

Table 3.16: Strongest Checklist Reviews vs Displayed Priority

	High	Normal	Low
# of Reviews	58	46	52
Normally Occurs	28	27	28
chi-test ($p < 0.05$)	N	N	N
Substantial Deviation	28	23	18
chi-test ($p < 0.05$)	N	N	N
Always Fix	29	22	19
chi-test ($p < 0.05$)	N	N	N
Always Show	29	24	22
chi-test ($p < 0.05$)	N	N	N

slightly higher for issues with a high priority label, the difference does not appear to be significant. Similarly, when we construct a table in which columns represent all strong responses for a particular index in the presentation order, we observe no significant difference due to ordering.

3.2.2.4 Comparison with Expert Participants

We compared the results from the student participants to results from more experienced participants. These experienced reviewers are real FindBugs users, recruited from a FindBugs-interest mailing list, and asked to perform the study remotely using a Java Web Start interface to access the warnings and submit their reviews. In other words, the more experienced users were not in a controlled environment, but their opinions are solicited to compare with the student users.

The Java Web Start interface had some limitations. Users could not drill down into the source code to get more details and some users had trouble starting the interface. This expert review served as a test drive of the automated review system that we used in our larger study, described soon in Section 3.3.

The patterns observed among experts was generally similar to that of regular participants. In other words, NP warnings were rated more strongly, and users were not fooled by fake warnings. (The experts' study did not randomize the order or displayed priority.) Reviews were also consistent among participants.

Experts gave strong responses at a slightly lower rate than regular participants, though this was not statistically significant except for the Issue Occurrence question.

On the other hand, experts selected the "Always Show" filtering decision at a slightly higher rate, though also not statistically significant.

3.2.2.5 Qualitative Feedback from Reviewers

Most participants indicated in the post experiment survey that the warnings were generally easy to understand and were not new to users. This is not surprising since the warnings selected, and FindBugs warnings in general, tend to refer to simple errors. Users were more split about whether it was easy to decide if an issue was a bug or if it should be fixed. Half the users said it was easy to decide both of these properties, while the rest disagreed or were indifferent. One user commented that for some warnings (like the redundant null check warnings) it was difficult to decide whether to fix or not, because he was concerned about possible side effects in other parts of the code. Half the users also indicated that the displayed priority did not influence their review, though some users complained that the color coding for priority labels (red, orange and yellow) made it hard to distinguish between them.

3.2.2.6 Threats to Validity

Lab studies like this always have an external validity problem; it is unclear how much the results generalize. This is particularly pronounced in studies of static analysis warnings because, as our results indicate, the choice of bug patterns affects the responses received. We also believe that in practice some of the checklist responses would be impacted by factors not considered in this study including the

policies of the reviewer's organization.

3.2.3 Summary

Users consistently identified certain bug patterns as severe, and others as low impact. The difference of opinions between users was not great, though some users erred in some of their reviews. So though some users may disagree about the severity of a problem, this disagreement was not pronounced in our study. We will have an opportunity to compare this outcome with the results of a larger study in the next section.

Users reviewed warnings fairly quickly, which indicates that it would not be too costly to ask engineers in a real company to review FindBugs warnings in their own code. Indeed, engineers may review warnings faster because they are more familiar with the code. These results may not generalize to other static analysis tools and contexts.

Most users did not appear to make any mistakes in their understanding of the warnings. When mistakes occurred, it may have been because the error path went through many procedures, or the user was not rigorous enough. This problem of rigor may not occur if the user is reviewing code they wrote, but it is certainly possible for static analysis to mislead a third party reviewer, especially if some unusual aspect of the code is inserted intentionally and no comment is provided.

3.3 FindBugs Community Reviews

Now we shift gears and try to draw significant quantitative trends from having hundreds of reviewers evaluate thousands of warnings. Ideally, we would like to observe users as they review warnings, and control for various factors. But to conduct a large review, we have to rely on professionals whose main priority is ensuring that their code is high quality, not conducting a research study. Furthermore, many potential users may have various confidentiality concerns. In particular, software companies generally do not want information about their bugs being published. Hence we have to propose activities that support the needs and goals of a large organization, while also allowing us to extracted some (limited) information that can inform our research.

Apart from giving us significant trends, these reviews also provide an opportunity to demonstrate the value of static analysis to developers and managers. Some professionals are skeptical about static analysis tools, and believe that most warnings are not worth fixing. But this perception may not hold if they spend time reviewing warnings in code they have written and find interesting problems.

3.3.1 The Google FindBugs Fixit

FindBugs has been used in some capacity at Google for several years. Early incarnations of the process at Google integrated FindBugs warnings into a tool called BugBot, along with warnings from other tools. This system would periodically analyze the code base, and display warnings on a web interface that was available

to all developers. As Google sought to improve its process, the static analysis champions adopted a service model for a brief period. During this period, warnings were centrally triaged and important defects were forwarded to the appropriate developers. This approach allowed the static analysis team to get a sense of which warnings were important to developers, and to reprioritize warnings accordingly. By the end of this process, they had created an internal ranking for warnings. But this approach did not scale as Google's code base grew. So it was abandoned and an effort to push warnings into the code review process was started. We discuss the process Google has employed in more detail when we discuss best practices in Section 7.4.

By Fall of 2008 the full vision had not yet been realized, and had run into some roadblocks. There were still thousands of unreviewed warnings in the code base, and it was clear that FindBugs was receiving limited use. In addition, there were many skeptics about the value of static analysis, and many of the warnings were perceived to be low priority. The system fell into disuse, and the engineers supporting the system were reassigned to other tasks.

So what was the problem with the experience up to that point? Perhaps the warnings from FindBugs were not of high enough value or relevance to users. Or perhaps more infrastructure support was needed to make static analysis worthwhile. In particular, warnings needed to be more readily available to developers, so they could review them without significant additional effort beyond their normal workflow to run the static analysis or organize the warnings. The presence of many low priority issues also indicates that the ranking process for warnings needed to be

refined. Finally, the system needed better integration with existing bug reporting system and source code repositories, so that users could easily investigate warnings and assign them to other developers.

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 [129] 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 FindBugs with Google's internal bug tracking and source code version control system, so that 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. Google has a tradition of company-wide engineering fixits [100], 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 some of these capabilities have been extended into a general framework that can be used by other companies and by open source efforts.

3.3.2 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

Table 3.17: User Classifications

Must Fix	Should Fix	I Will Fix
Needs Study	Mostly Harmless	Not a Bug
Bad Analysis	Obsolete code	

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 chapter 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 3.17, 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 was intended to ensure 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.

Prior to analyzing the data from the fixit, we anonymized certain confidential details, such as file names, reviewer identities, and any comments 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.

This study enabled us to compare the reviews provided by users for each issue with the severity suggested by FindBugs. As I mentioned earlier, FindBugs assigns a priority (high, medium, low) to each warning 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, we 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, and we refer to issues ranked 1-4 as being in the *Scariest* group, while issues ranked 5-8 are in the *Scary* group, and issues ranked 9-12 are in the *Troubling* group. 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.

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.

3.3.3 General Results

The fixit brought many issues to the attention of developers and managers, and many problems were fixed. Table 3.18 overviews some high level numbers from

Table 3.18: 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

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. 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 3.5). Reviewers overwhelmingly focused on issues in the Correctness category, with 71% of reviewed issues in this category compared to just 17% for 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 did not own. We

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 3.5: Recommendations Grouped by Bug Rank and Category

talk more about the differences between code reviewed by owners and non-owners in Section 3.3.8.

Figure 3.5 also shows the percentage of reviews that received *Must Fix* and *Should Fix* classifications. Over 77% of reviews were *Must Fix* and *Should Fix* classifications, and 87% of reviewed issues received at least one fix recommendation. 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 the next section.

3.3.4 Comparing Reviews with Bug Rank

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,

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 3.6: Correlating Bug Ranks with Reviewer Classifications

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 [27].

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, de-

pending on the variety and inferred severity of the issue. 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.

Figure 3.6 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. For example, 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.

When we group reviews by bug pattern, we observe similar correlations, but they are very weak and not statistically significant. This indicates that there must be some bug patterns that deviate from the expected trend. To explore this deeper, consider the scatter diagram in Figure 3.7. In this diagram, each marker represents a bug pattern, its position on the x-axis represents the bug rank assigned to that bug pattern, and its position on the y-axis represents the percentage of reviews in

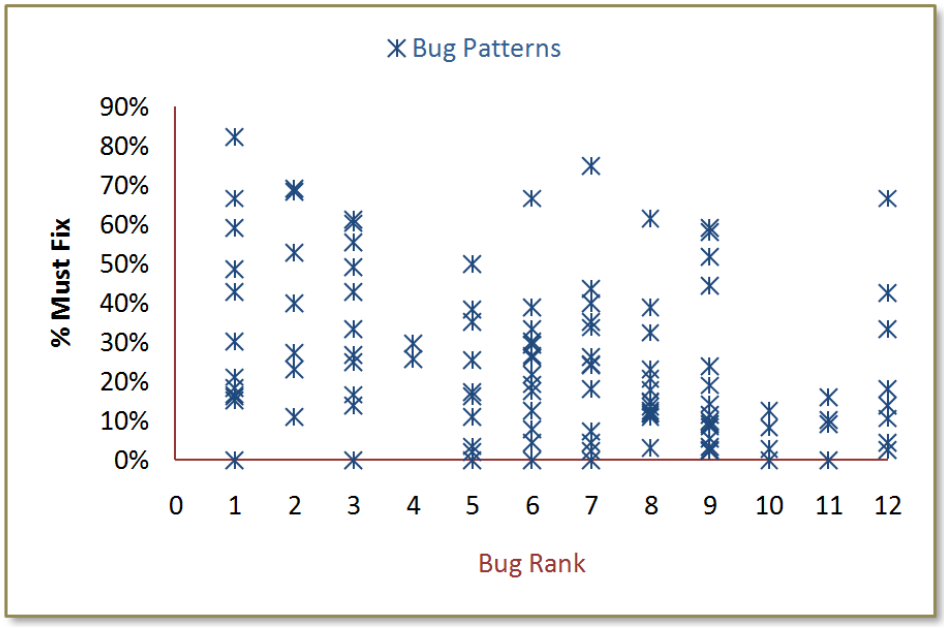

Figure 3.7: Must Fix Classifications By Rank

that bug pattern that were *Must Fix*. The diagram visualizes a weak negative trend, with many deviations from the norm. Specifically there are some bug patterns that have severe bug ranks but low *Must Fix* rates, and vice versa.

Looking more closely at the individual bug patterns, this trend may be partially explained by a distinction between two types of bug patterns, which we call *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 generally 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

Table 3.19: Reviews for Two Silent and Two Loud Bug Patterns

Rank	Bug Pattern	% Must Fix	% Should Fix
1	No relationship between generic parameter and method argument	30%	48%
1	Suspicious reference comparison	15%	68%
5	Null pointer dereference	35%	33%
9	An apparent infinite recursive loop	52%	34%

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. Consider the examples in Table 3.19; the first two bug patterns are severely ranked silent patterns, while the last two are lower ranked loud patterns. 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.

As Table 3.19 shows, 52% of reviews classified infinite recursive loops as Must Fix, compared to 30% for the incompatible generic container argument pattern. Similarly, a suspicious reference comparison (which uses the == operator instead of `equals()`) has a low *must-fix* rate of 15%. But notice that both silent bug patterns have much higher *should-fix* rates. This suggests that reviewers were more alarmed by the loud warnings, giving them severe reviews, but were content to give the silent issues less severe reviews. I talk more about the distinction between loud and silent warnings in the next chapter, in Section 4.2, focusing on how they persist in the code repository.

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.

3.3.5 Fix Rates from the Fixit

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 5.2, we describe an experimental approach that uses Noise bug patterns to try to separate significant removal rates from code churn. 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.

In Table 3.20, we compute the percentage 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 (i.e., issues that were "last seen"

before the end of our study). As we mentioned earlier, 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 3.20, 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 3.20 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[10]. 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.

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,

[10]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 3.20: Last Seen Fix Rate for Issue Subgroups

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 3.20 (ranging from 34% to 39%), reflecting the fact that this approach undercounts the number of fixed issues.

3.3.5.1 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 3.21.

We observed strong and significant (`p<0.01`) correlations, shown in Figure 3.8, between some of our 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

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

Figure 3.8: User Classifications versus Fix Rate

earlier 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 3.3.6). 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.

3.3.6 Consensus Classifications

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 the lab studies discussed earlier, we observed that independent reviewers generally are consistent about how they review issues. 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 our earlier surveys, 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).

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.

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 3.21. 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.

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 re-

Table 3.21: 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		

views 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 3.9 we show these two measures, using a threshold of 0.8 for the second measure and using some of the classification schemes from Table 3.21. 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.

	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 3.9: Consensus Rates for All and Scariest Issues

3.3.7 Review Times

The review time is an important measure when trying to compute the cost of using static analysis. In previously discussed lab studies, we observed a relatively low average review time between 1 to 2 minutes for each issue (Section 3.2.1). 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. In the end, 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.

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.

3.3.8 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.

3.3.9 Summary of Lessons from the Fixit

Overall, the fixit was declared a success, and some managers were impressed by the high percentage of the reviews that gave a fix recommendation. Researchers at Google have started improving the supporting infrastructure, including an effort to integrate FindBugs warnings into the code review process, and some developers

already run the analysis regularly through their IDEs.

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 section. We observed that most reviews recommended fixing the underlying issue. 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 (*loud bug patterns*), but are rarely feasible in practice, and undervalue more subtle bug patterns (*silent bug patterns*) that are often harmless, but should be reviewed because they can cause serious problems that are hard to detect. In the end, we chose to NOT modify FindBugs' rankings in response to these observations. This is because FindBugs' rankings aim to emphasize those issues that should be *reviewed* first, not necessarily those issues that should be fixed. Since we observe that many loud bug patterns are readily fixed if they matter, we use the rankings to encourage users to review the subtle issues first.

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*. Finally, our analysis indicates that there was consistency among independent reviewers, and that most reviews were completed fairly quickly, validating our earlier findings from lab studies.

One surprising outcome was that we did not find any problems that were actively wreaking havoc in production systems, or the proverbial "million dollar

bug". We think this has to do with the robust monitoring systems at Google, and this seems to reinforce our belief that many critical defects are eventually found by other quality assurance activities, though perhaps at greater expense than if static analysis is used. I discuss this observation in more detail in the next chapter, in Section 4.3.

3.4 Summary and Related Work

The user interactions in this chapter indicate that static analysis is well received by many users. Users have stated that static analysis has found useful problems, and is easy to use. And when asked to review thousands of issues flagged by FindBugs, professional engineers recommended fixing most of them. Still, it is clear that users face challenges bringing static analysis into their day-to-day activities.

Most research on static code analyzers focuses on creating new analysis methods, or refining existing ones. As tools proliferate and mature, some researchers are starting to turn their focus to the interaction between tools and developers or processes. Layman et al. [84] observe developers directly as they complete programming tasks and introduce faults that cause warnings from a static code analyzer. They try to determine which factors cause a programmer to interrupt their activity to fix the fault. They conclude that users are more likely to address warnings if they are relevant to their current primary task.

Khoo et al. [77] focus on the task of triaging warnings output by a tool to decide which ones should be fixed. They observe that many static code analyzers

output lists of program statements (called paths) that may represent the flow of data or control through program functions, or exception stack traces among other things. These paths are often difficult to navigate and comprehend, so they provide a code visualization that concisely displays inlined functions for each element in the path, allowing users to expand or collapse these functions. They also observe that inexperienced users do not always ask the right question or know what to look for when triaging warnings. To solve this problem, Khoo et al. propose providing checklists for each warning that direct the user to look for specific properties in code to decide if the warning is valid. They found that the visualization and checklist combined to make reviewers more efficient without affecting their accuracy.

Other research focuses on helping teams establish the right processes for using static analysis [58, 81, 52, 32, 69]. I will discuss these reports in more detail in Chapter 7.

Chapter 4

Understanding Why Defects Persist

The user studies described in the previous chapter enabled us to interact with and observe users, so that we could use their experiences and sentiments to judge the value of static analysis. But many of the outcomes of these studies are subjective, and may not generalize to all cases. Fortunately, we do not have to rely only on this research method. Software developers produce many artifacts as they engineer each application, and we can use these artifacts to make inferences about the impact of static analysis, or defects found by static analysis. The benefits of this approach are that we can analyze large quantities of data, and we are not as reliant on the engineers that own the code.

In the next chapter, I will describe some studies in which manual and automatic methods are used to mine software artifacts for significant trends. In this chapter, I present some anecdotal observations from manually reviewing hundreds of warnings in several code bases.

One of the striking observations is that when we analyze production software, we often find interesting defects that have been around for a long time. Some look so obvious that we wonder why they have not been detected, and if they are causing any problems. An example is the defect in Figure 4.1 which flags a comparison operation that will always be false. The invocation `simpleType.getName()` returns

Source: Eclipse SDK 3.5 | org.eclipse.jdt...debug.eval.ast.engine.ASTInstructionCompiler

```
3831        SimpleType simpleType = (SimpleType) type;
3832        if ("java.lang.String".equals(simpleType.getName())){
3833            return Instruction.T_String;
3834        }
3835        return Instruction.T_Object;
```

FindBugs: "Call to equals() comparing different types on line 3832"

Figure 4.1: Long-Lasting Defect in Eclipse

a `Name` object, not a string. The effect is that line 3833 is dead code, and this method returns the type id for an object instead of a string. This defect was first seen when analyzing version 2.0 of the Eclipse SDK, and is still around in version 3.5, 8 years later. Code coverage analysis of this code fragment indicates the all lines except line 3833 are executed as a result of unit tests, and yet all tests pass[1].

We investigate why defects persist as part of a large inquiry into whether static analysis finds problems that matter in practice. In other words, are defects found by static analysis valuable? It is not uncommon to encounter projects where a number of bugs remain unresolved for some time. They are neither fixed, nor suppressed. On the other hand, many users report that static analysis tools like FindBugs have "saved" them from embarrassment, by finding potential problems.

In general, manual and anecdotal observation indicates that static analysis does find consequential mistakes. However, most consequential correctness warnings

[1]We eventually contacted the primary developer for this code fragment, who confirmed that this defect does not impact program behavior because problems caused by this issue are mitigated elsewhere. Still, a fix has been identified and is planned.

found by static analysis can and will be detected by other good quality assurance practices such as testing and code review. This explains why warnings left in production code often do not matter. The value of static analysis is that it can find these bugs early in development, when they are cheapest to fix. An exception is certain subtle defects, such as the security, concurrency, and performance defects discussed in Chapter 7, which often escape other quality assurance methods, and are best found using static analysis.

This observation represents a paradigm shift for some developers who wait until the end of the development process before running static analysis, hoping to find bugs missed by their quality assurance. Most of those bugs won't matter, and it is more expensive to fix bugs at this stage. Running static analysis early is the key to finding bugs that matter, and fixing them more cheaply. Running static analysis early may also prevent costly efforts down the road, such as lengthy debugging sessions or sending out patches.

I start this chapter by returning to the observation that static analysis is good at finding stupid mistakes made by developers, but not all mistakes are important. In Section 4.1, I present some software defects that are caused by stupid mistakes, but that have little or no impact on code behavior. The goal of users is to find the intersection of stupid mistakes and important ones. Hence many of these true but low impact defects will persist, even they are flagged by static analysis tools.

Another distinction useful for understanding why some defects persist, is the distinction between "loud" warnings, and "silent" warnings, which I introduced in Chapter 3. Loud warnings are caused by defects that result in program crashes or

exceptions, and are almost always detected, if they matter. Loud warnings that persist for a long time are often in dead code. Silent warnings are associated with defects that do not directly cause the program to crash, but may put the program in an incorrect state, or point to suspicious or confusing code. It may be harder to predict whether these will be low impact or important. Some silent bug patterns are often low impact, but sometimes have a real impact on correctness. I discuss the persistence of loud and silent warnings in Section 4.2.

I conclude the chapter in Section 4.3 by discussing "The Survivor Effect": the phenomenon that important defects are often found by other quality assurance methods, leaving mostly less important ones in production code. I present some examples of the survivor effect observed in practice, discuss some exceptions, and argue that static analysis is best used early in the software development process.

4.1 True But Low Impact Defects

We already discussed the risk of false positives back in Section 1.3.1. Static analysis cannot completely understand the semantics of programs, and hence may sometimes make incorrect assumptions, leading to false warnings. Early static analysis tools used naive analysis, and were riddled with false positives, which were disruptive to developer productivity. Modern tools use sophisticated heuristics and more rigorous analysis to minimize the number of false warnings. One tool vendor boasts that fewer than 15% of its warnings are false positives [66].

One problem with this focus on false positives is that it seems to imply that

the remaining defects are true defects that developers should fix. But when we review warnings in practice, we are surprised to find a number of true defects that have little or no impact on program behavior. Many of these low impact defects are associated with some mistake or bad practice, and the case could be made that they should be fixed. But correcting software takes up time and resources, and the fix could have unexpected side effects that lead to more serious problems, especially if the modified lines are quite old.

In this section, we go over the reasons why true low impact defects occur (with examples from real software), and discuss the scenarios when these defects should be fixed.

4.1.1 Deliberate Defects

Sometimes the defects found by static analysis were inserted intentionally by developers. Figure 4.2 shows two examples where the developers intend to throw *Run Time Exceptions*, but instead of explicitly creating and throwing the exceptions, they insert faulty code. This seems like bad practice, especially if the default messages generated by the system are not informative. But the program behaves exactly as intended, and the system messages are at least sufficient to find the line with the error, so this approach may be seen as a useful shorthand by some developers.

In some cases, a static analysis tool may want to avoid bothering developers with warnings on intentional defects (by, for example, parsing nearby comments and suppressing the warning if the comment is "throw error"). But in many cases,

Source: Sun JDK 6 | com.sun.jndi.dns.DnsName

```
345     if (n instanceof CompositeName) {
346         n = (DnsName) n; // force ClassCastException
347     }
```

Source: Sun JDK 6 | com.sun.java.util.jar.pack.Attribute

```
1042    if (layout.charAt(i++) != '[')
1043        layout.charAt(-i); // throw error
```

Figure 4.2: Two intentional errors

deliberate defects are attempts by developers to violate some rule or convention established by the organization. If the developers feel these violations are justified, then they should be forced to use an explicit suppression mechanism to hide warnings (especially source level suppression), so that future developers understand why this violation was necessary. This also enables managers and researchers to find and investigate these violations by occasionally turning off all suppressions.

4.1.2 Masked Defects

Sometimes the code surrounding a defect prevents it from having any effect on program behavior, effectively masking the defect. We already discussed one example back in Section 1.3.2 (in Figure 1.4) where a developer accidentally assumes a value is an unsigned byte value (from 0 to 255), when in fact it is a signed value (from -128 to 127). Despite this mistaken assumption, the developer's conditional check still accepts values in the correct range [32,128] for basic printable ASCII characters.

Developers may not feel compelled to fix masked defects, but they should still

inspect the warnings closely, because masked defects often imply a misunderstanding of relevant invariants, and there might be other logical defects or questionable code nearby.

4.1.3 Infeasible Statement, Branch, or Situation

Sometimes a defect occurs only in a situation that a developer believes is infeasible, but the static analysis is unable to verify this by examining the code. In Figure 4.3, FindBugs complains about a possible null pointer dereference on line 171, which will occur if certain checked exceptions are thrown on line 167, leaving the variable set to null. But the developer is convinced that the exceptions will never be thrown, and indicates so in the comments.

One way to "fix" this, or at least prevent a static analysis warning, is to insert a failing assertion in place of the comments. This is good practice in general because if the developer's beliefs are incorrect, the assertions will generate an error, usually during development time, so the problem is quickly found and fixed. But this fix is not compelling if the code is already in production, where assertions are usually turned off.

4.1.4 Code that is Already Doomed

Sometimes, a defect occurs in a situation where the computation is already doomed, and the resulting runtime exception is not a significantly worse outcome than any other behavior that might result from fixing the defect.

Source: Sun JDK 6 | com.sun.corba.se.impl.dynamicany.DynAnyComplexImpl

```
165     String expectedMemberName = null;
166     try {
167         expectedMemberName = expectedTypeCode.member_name(i);
168     } catch (BadKind badKind) { // impossible
169     } catch (Bounds bounds) { // impossible
170     }
171     if ( ! (expectedMemberName.equals(memberName) ... )) {
```

FindBugs: "Possible null pointer dereference of `expectedMemberName` on line 171"

Figure 4.3: Infeasible situation

Source: Sun JDK 6 | com.sun.org.apache.xml.internal.security.encryption.XMLCipher

```
2224    if (null == element) {
2225        //complain
2226    }
2227    String algorithm = element.getAttributeNS(...);
```

FindBugs: "Possible null pointer dereference of `element` on line 2227"

Figure 4.4: Doomed situations: vacuous complaint

Figures 4.4 – 4.6 shows three examples of doomed situations. In Figure 4.4 the comment indicates the intention of the developer to complain about a null parameter, but no action is taken and thus a null pointer exception will occur. Perhaps null is never provided as an argument to this method. But even if it is, it seems likely that the appropriate remedy for this warning would be to throw a null pointer exception when the parameter is null. Since the existing code already gives this behavior, changing the code is probably unwarranted (although documenting the fact that the parameter must be non-null would be useful).

Source: XSLT2Runner (abridged) | org.apache...output.TransletOutputHandlerFactory

```
116     SerializationHandler result = null;
121     if (_method == null)
123         result = new ToUnknownStream();
125     else if (_method.equalsIgnoreCase("xml"))
128         result = new ToXMLStream();
131     else if (_method.equalsIgnoreCase("html"))
134         result = new ToHTMLStream();
137     else if (_method.equalsIgnoreCase("text"))
140         result = new ToTextStream();
149     result.setEncoding(_encoding);
```

FindBugs: "Possible null pointer dereference of `result` on line 149"

Figure 4.5: Doomed situations: missing else clause

Figure 4.5 shows what is effectively a switch statement, constructed using `if .. else` statements. This pattern is relatively common, even to the detail of not having an else clause for the final `if` statement. Thus, if the final `if` statement fails, `result` will be null and a null pointer exception will occur. While this code is highly questionable, the appropriate fix would likely be to throw an IllegalArgumentException if none of the `if` guards match, and the impact of a null pointer exception is unlikely to be significantly different than that of throwing an IllegalArgumentException.

Figure 4.6 shows an example where the program has detected an erroneous situation, and is in the process of creating an exception to throw. However, due to a programming error, a null pointer exception will occur when `node` is dereferenced. While the code is clearly mistaken, the impact of the mistake is minimal.

Source: Apache Xerces (abridged) | com.sun.org.apache.xerces.internal.util

```
78      Node node = null;
79      switch(place.getNodeType()) {
80          case Node.CDATA_SECTION_NODE: {
81              node = ...
82              break;
83          }
84          case Node.COMMENT_NODE:
85              ...
86          default: {
87              throw new IllegalArgumentException("...("
88              + node.getNodeName()+')');
89          }
```

FindBugs: "Possible null pointer dereference of `node`"

Figure 4.6: Doomed situations: defect in exception handling

4.1.5 Testing code

In testing code, developers will often do things that seem nonsensical, such as checking that invoking `equals(null)` returns false. In this case, the test is checking that the `equals` method can handle a null argument. We can't ignore nonsensical code in testing code, since it may reflect a coding mistake that results in the test not testing what was intended.

4.1.6 Logging or other unimportant case

We have also seen a number of cases of a bug that would only impact logging output, or assertions. While accurate logging messages are important, bugs in logging code might be deemed to be of lower importance. Figure 4.7 shows code in

Source: Sun JDK 6 | com.sun.org.apache.xml.internal.resolver.Catalog

```
818     String userdir = System.getProperty("user.dir");
819     userdir.replace('\\', '/');
820     catalogManager.debug.message(1, "Malformed URL on cwd", userdir);
```

FindBugs: "Method ignores return value of String.replace() on line 819"

Figure 4.7: Logging defect

which the call to `replace` is performed incorrectly. The `replace` method cannot modify the String it is invoked on - Java Strings are immutable. Rather, it returns a new String that is the result of the modification. Since the return result is ignored here, the call to replace has no effect and the userdir may contain back slashes rather than the intended forward slashes.

4.1.7 When should such defects be fixed?

Should a defect that doesn't cause the program to significantly misbehave be fixed? Defects found by static analysis early in the software development process are cheaper to fix than those found later on. Since it may not be possible to know their long term impact at this early stage, users should endeavor to fix all of them. But if the software system is mature, then additional considerations come into play. The main arguments against fixing such defects is that they require engineering resources that could be better applied elsewhere, and that there is a chance that the attempt to fix the defect will introduce another, more serious bug that *does* significantly impact the behavior of the application. The primary argument for fixing such defects is that it makes the code easier to understand and maintain, and

less likely to break in the face of future modifications or uses.

When sophisticated analysis finds an interprocedural error path involving aliasing and multiple conditions, understanding the defect and how and where to remedy it can take significantly more engineering time, and it can be more difficult to have confidence that the remedy resolves the issue without introducing new problems. Warnings from less sophisticated static analysis may be easier to understand and fix, but care still needs to be taken to understand the context of the defect, instead of blindly applying a fix in response to a message from the static analysis. And even simple defects suggest holes in test coverage; additional unit tests should be created to supplement defect fixes.

4.2 Loud and Silent Warnings

In measuring the significance of warnings found, I find it useful to make a distinction between loud and silent warnings. Loud warnings are associated with exceptions and program crashes, while silent warnings do not generally stop the program, but may leave it in an incorrect state. As we discussed in the last chapter, developers reviewing static analysis warnings are more alarmed by loud warnings, but these warnings usually occur in dead code if they are not found immediately. Silent warnings, on the other hand, may be connected with serious but subtle or rare misbehavior.

A classic case of a loud warning is an infinite recursive loop, such as the one in Figure 4.8. The method `widgetDefaultSelected()` unconditionally calls

Source: Eclipse SDK 3.5 | org.eclipse.ui.internal.dialogs.CustomizePerspectiveDialog

```
1047    bindingLink.addSelectionListener(new SelectionListener() {
1048        public void widgetDefaultSelected(SelectionEvent e) {
1049            widgetDefaultSelected(e);
1050        }
1051        public void widgetSelected(SelectionEvent e) {
1052            PreferenceDialog dialog = ...
```

FindBugs: "There is an apparent infinite recursive loop on line 1049"

Figure 4.8: Infinite Recursive Loop in Eclipse

itself, and hence will always throw a *Stack Overflow Exception* if invoked. Defects like this are usually quickly and easily found, if they matter. So when we find them in production code, we can generally assume that this is dead code. In this case, we observe that the developer implementing the `SelectionListener` interface only really wants to implement `widgetSelected()`, but is required by the API to also implement `widgetDefaultSelected()`. The developer does so with a naive implementation that simply calls the other method... or at least intends to. The methods have similar names, and the developer probably selected the wrong one from a code assist list. Still, the defective method is not intended for use; it was implemented primarily to satisfy the API.

Some loud warnings only manifest in scenarios that may not be feasible. Developers faced with such warnings have to choose whether to apply a fix immediately, or defer the fix until the problem manifests itself. This latter option is only available if the application can tolerate the failure by, for example, restarting itself, or falling back on some redundancy. In this case, the failure will usually include a stack trace

Source: Eclipse SDK 3.5 | org.eclipse.core.runtime.Plugin

```
544     public void setDebugging(boolean value) {
545         if (bundle == null)
546             this.debug = value;
547         String key = bundle.getSymbolicName() + "/debug";
548         ...
549     }
```

FindBugs: "Possible null pointer dereference of `bundle` on line 547"

Figure 4.9: Possible null pointer dereference in Eclipse

that can be used to track down the source of the problem.

One example of loud warnings that may not be feasible, are the possible null pointer dereference warnings flagged by FindBugs, such as the one in Figure 4.9. In this example, the variable `bundle` is compared to null, and then later unconditionally dereferenced, leading to the warning. The only scenario in which an exception is thrown is if `bundle` is null, and it is not clear if this is possible. In Java, null pointer dereferences produce informative stack traces, and developers may wait to see if they are feasible, especially if an immediate fix is unclear. In the case of Figure 4.9, it turns out that the null dereference warning is associated with an error in the control flow logic of the method. The developers fixed the problem by adding a return statement to the if-statement that compares `bundle` to null. Null pointer dereferences are explored in more detail in Chapter 6.

A classic example of a silent defect is ignoring the return value of a string operation, as is the case in Figure 4.10. This mistake has been in the code base since version 1.0, and has gone unrepaired for almost nine years. This defect may

Source: Eclipse SDK 3.5 | org.eclipse.pde.internal.core.schema.SchemaEnumeration

```
26      public void write(String indent, PrintWriter writer) {
27          String description = getDescription();
28          if (description != null)
29              description.trim();
30          if (description != null && description.length() > 0) {
31              ...
32              writer.println(indent3 + "<documentation>");
33              writer.println(indent3 + description);
34              writer.println(indent3 + "</documentation>");
35              ...
```

FindBugs: "Method ignores return value of String.trim() on line 30"

Figure 4.10: Ignored Return Value in Eclipse

result in a slightly incorrect string, a relatively low impact problem, but without static analysis, it would be difficult to detect.

Another silent defect occurs when a developer compares incompatible types. In Java, the `equals()` method used to compare two objects receives any object as an argument. If the two objects are of unrelated types, this comparison is not expected to throw an exception, but rather to return false. It seems unlikely that a developer would want to use a condition that is always false as part of any control logic. So this must be a mistake. Sometimes this defect causes the program to seriously misbehave, and the problem is eventually detected following a sometimes lengthy debugging session. Other times, it leads to subtle and silent changes in program behavior that are hard to detect. We saw one example from Eclipse at the beginning of this chapter (in Figure 4.1) that has gone unrepaired for many years. Another example from the Apache Lucene project is illustrated in Figure 4.11. In

Source: Apache Lucene | org.apache.lucene.demo.html.HTMLParser

```
41      StringBuffer title = new StringBuffer();
116     ...
117     void addText(String text) throws IOException {
126         ...
127         if (!titleComplete && !title.equals("")) { // finished title
128             synchronized(this) {
129                 titleComplete = true; // tell waiting threads
130                 notifyAll();
131             }
132         }
133     }
```

FindBugs: "Call to equals() comparing different types on line 127"

Figure 4.11: Comparing a StringBuffer to a String is always false

this case, the developer intends to add a condition to check that the *StringBuffer* `title` is not empty on line 127. Instead, the developer uses a faulty comparison with a *String* which leads to the execution of the synchronized block on line 128, whether the string buffer is empty or not. This code fragment occurred in some demo code which was distributed with Lucene, and may not have been rigorously tested. On the one hand, demo or sample code is often not as important as the rest of the software. On the other hand, sample code like this is often copied into many projects, and corrections do not propagate to all those projects. Indeed a Google search indicates that this class has been copied dozens of times, and some of those copies contain this defect, while others contain a fix that was later applied (replacing the bad comparison with `!(title.length() == 0)`).

Sometimes the distinction between a loud and a silent warning is subtle, but

Source: Eclipse 3.5 | org.eclipse.team.ui.synchronize.AbstractSynchronizeParticipant

```
211     public void setInitializationData(...) {
218         ...
219         fName = config.getAttribute("name");
220         if (config == null) {
221             fName = "Unknown";
222         }
225         String strIcon = config.getAttribute("icon");
226         if (strIcon != null) {
227             fImageDescriptor = ...
228         }
229         ...
```

FindBugs: "Nullcheck of value previously dereferenced on line 220"

Figure 4.12: Redundant comparison to null where value is previously dereferenced

the two still exhibit different traits. Consider two variants of a *Redundant Comparison to Null (RCN)* rule in Figures 4.12 and 4.13. In the first RCN example in Figure 4.12, the variable `config` is compared to null right after it is dereferenced. This comparison is useless, or *redundant*, because if `config` is ever null, an exception will be thrown on line 219. This loud warning is like the *Possible Null Pointer Dereference* discussed earlier (in Figure 4.9) because it is detected quickly if `config` is null in practice, or it is typically a harmless[2] defensive (but useless) check if `config` is never null.

In the second RCN example in Figure 4.13, the variable `baseName` is compared to null on line 342, even though it is assigned a value that is guaranteed to be non-

[2]Actually this example may not be harmless even if config is never null. Can you spot why? I will return to this example at the end of this section.

Source: fawkeZ Project (jCoderZ.org) | org.jcoderz.phoenix.cmpgen2.CmpGenerator

```
331     private void generateCmpBean (CreateTableStatement stmt) {
340         ...
341         String baseName = stmt.getBeanName() + "Entity";
342         if (baseName == null)
343         {
344             baseName = sqlNameToJavaName(stmt.getTableName());
345         }
346         ...
```

FindBugs: "Redundant nullcheck of `baseName`, known to be non-null, on line 342"

Figure 4.13: Redundant comparison to null where value guaranteed to be non-null

null on the previous line. (Even if the expression `stmt.getBeanName()` returns null, its subsequent concatenation with the string constant would yield the non-null string "`nullEntity`"). So the comparison on line 342 is redundant. Since `baseName` can never be null, no exception is thrown, and this is a silent warning. Like other silent warnings, this comparison could be a harmless defensive check, but it could also contain a serious subtle error that occurs rarely and is hard to detect without static analysis. If we look closely at this example, we observe that the developer intends to construct a base name using either the bean name or, if that fails, the table name. But if `stmt.getBeanName()` returns null, then `baseName` is set to "`nullEntity`", which is probably not correct. It appears that the developer really intended to check if `stmt.getBeanName()` returned null *before* concatenating it to the string constant. Of course, if `stmt.getBeanName()` is never null, then this mistake is mostly harmless.

So in summary, loud warnings often manifest themselves and are detected in practice if they are serious and feasible. The loud warnings that persist are often in dead code, or are infeasible. Silent warnings are often low impact, but may be associated with serious program misbehavior. If this misbehavior is rare, they can be hard to detect using other quality assurance methods, and can wreak havoc long after the code is written. Even if the misbehavior is not rare, silent defects can be hard to debug. Static analysis can provide real value by finding these subtle defects, though reviewers are often more alarmed by the loud warnings.

Of course, these observations are not hard fast rules, and there are exceptions. Indeed, looking back to the first (loud) RCN example in Figure 4.12, we observe that the code may have a subtle defect even if `config` is not null. The purpose of the method is to set some initialization data, and on line 226, `strIcon` is compared to null right after it is initialized using the `getAttribute()` method. Since `fName` is initialized in the same way, it seems reasonable to assume that the programmer intended to also compare it to null, but used the wrong variable on line 220. So now we see that the real defect is that if `fName` is initialized to null on line 219, it is not updated to "`Unknown`" on line 221 as it should be.

4.3 The Survivor Effect

We have seen that some true defects end up having a low impact in practice, and some defects can silently lurk undetected and pose a danger to the application. When we review production code, we often find more of the former, and less of

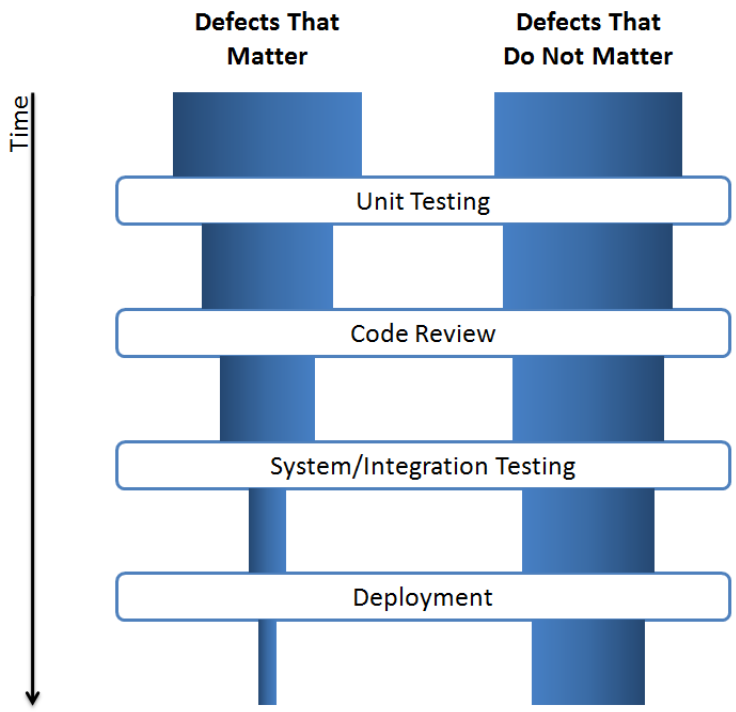

Figure 4.14: The Survivor Effect: Comparing defects that matter with defects that do not matter

the latter. It may be that developers just produce more low impact defects than important ones. But another reason is that the popular quality assurance methods, such as unit testing and code review, are more geared towards finding defects that negatively impact application behavior. Hence these important defects are more likely to be caught before the code goes into production, leaving many of the lower impact ones behind.

Figure 4.14 illustrates this *"Survivor Effect"*. The good news is that static analysis can find both defects that matter, and defects that do not. But the number of defects that matter drops more quickly as the development cycle progresses. So if static analysis is used later in the software process, a greater proportion of defects

will seem unimportant, and the potentially important ones will be drowned out. Furthermore, all defects are potentially more expensive to fix after the software has been released. If static analysis is used earlier, it may not be clear which defects are important and which ones are not. But the key point is that *all the defects are usually cheaper to remedy at this early stage.* In addition, static analysis can detect defects earlier in the process than unit testing and code review; it can detect defects as soon as the code is written. Hence the best value for static analysis occurs when it is used early. In addition, avoiding the scenario where developers have to wade through many unimportant defects late in the development cycle to find the few important ones may lead to a more positive perception of static analysis.

During the surveys and interviews described earlier, some respondents indicated that they would run FindBugs only at the end of important intervals. Essentially the purpose was to clean out any problems that may have been missed during the interval. One participant responded: "We run FindBugs before each release of a Release Candidate". Another said his team would "Run FindBugs on a project milestone basis". Users with this paradigm may not be retaining the most value out of their use of static analysis. Part of the problem is the perceived cost of weeding through low impact warnings. Users also reported FindBugs was too slow in the IDE, and this discouraged them from using it at this, the earliest and most valuable time. Still, the survivor effect suggests that some of these costs are covered by the effectiveness of static analysis at finding and resolving problems more cheaply.

One user, who relies on the Agile software development process [42], suggested running static analysis at the end of each two week iteration. The two-week interval

is short enough that it is still relatively cheap to fix any problems found. The agile process encourages test-driven development, and advocates for this note that the tests find a greater range of problems. This user acknowledges that most problems are found by testing, but notes that static analysis is still valuable for finding "potential" problems, or defects that may crop up in the future. For example, FindBugs flagged some cases where the user was calling a non-final method from a constructor. This can be a problem if the class is sub-classed, and the called method is overridden leading to potentially unexpected behavior.

Another indication of the survivor effect came during the Google FindBugs Fixit, described in Section 3.3. Over 77% of the reviews contained a fix recommendation, 87% of reviewed issues received at least one fix recommendation, and many issues were fixed. However, none of the serious bugs appeared to be associated with any serious incorrect behaviors in Google's production systems. Some serious defects were found in code that had not yet been pushed to production, and in code that was not executed in production systems. The defects that were executed in production seemed to not result in serious misbehavior, or produced only subtle effects, such as performance degradation.

One interesting example occurred on the first day of the Fixit, when a defected was committed into the code base, 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. Hence the defect never made it into production. This was an example of how Google's testing and monitoring practices are effective at preventing misbehavior in productions systems.

Of course, automatic rollbacks and nightly build failures are often a more expensive way to address problems. For one thing, the development schedule may be delayed by nightly build failures. In this case, FindBugs could have prevented the loss of development cycles that resulted from the overnight failure.

Figure 4.14 also illustrates that not all defects that matter are caught, so static analysis can find important problems, even when it is used late. Indeed, some classes of problems are best found using static analysis, and will tend to persist if static analysis is not used. These includes problems that are not directly linked to incorrect behavior, such as security, performance and concurrency defects. Applications that potentially have a high exposure or sensitivity to these classes of defects will benefit greatly from using static analysis.

In general, users need to tradeoff the cost of static analysis with the benefits of using it, especially using it early. It is difficult to do an absolute cost-benefit analysis, but there are scenarios where it is very likely that static analysis is cost effective. These scenarios depend on the type of defects the application is sensitive to, and the nature of the application. Organizations can also make static analysis cost effective by adopting certain best practices, and building the right infrastructure to deploy warnings to developers. I discuss cost effective static analysis in more detail in Chapter 7.

Chapter 5

Mining Software Repositories for Defects

Software repositories store a wealth of history about static analysis, including which warnings are introduced/removed and when, and which components have the most warnings. We can access this history by analyzing older versions of the software and comparing the warnings in each version. We have to deal with some challenges, including keeping track of a warning from one version to another, and deciding if a warning removal represents an attempt to fix it. Despite these challenges, mining the software repository is attractive because it allows us to observe software development without bothering developers.

We use manual and automatic approaches to search through this history looking for general trends, or examples of interesting defects found by static analysis, and explore why these defects persist. The data is quite noisy and these explorations provide mostly qualitative insights. But given the amount of data involved, we can also look for statistically significant trends.

In this chapter, I discuss some studies that look into the software repository in an effort to validate some of the observations in the last chapter, namely the presence of true but low impact defects (which was discussed in Section 4.1), the distinctions between loud and silent defects (from Section 4.2), and the survivor effect (from Section 4.3). In the first set of studies, discussed in Section 5.1, we

manually reviewed the warning history of several large projects. Specifically, we examined some defects that have been fixed to evaluate how impactful they were and how complicated the fix was. In follow up studies described in Section 5.2, we automatically analyzed some software repository snapshots to identify the removal rates of different classes of defects. Here we are using "defect removal" as a proxy for "defect fix", which is limited because not all defect removals are caused by intentional fix efforts. Still, this study provides a means for reprioritizing warnings and enables us to understand the characteristics that are associated with high removal rates.

These two studies are limited by the fact that they only capture warnings that are checked into the code repository. I discuss some approaches for getting more fine-grained snapshots of the development process in Section 5.3. We used one of these approaches to capture regular snapshots of student development at the University of Maryland. Studying these snapshots enables us to observe the survivor effect, among other trends.

5.1 Manual Reviews of Large Software Systems

Over the last few years, and throughout our research study, we have undertaken several manual reviews of warnings in various projects. The manual reviews are useful for a number of reasons beyond just providing interesting anecdotes about the value of static analysis. The reviews enable us to characterize the severity of warnings in different bug patterns, and explore the distinctions between different

classes of warnings. For example, we can compare loud and silent warnings. We are not just reviewing defects that persist in the code base; we also consider defects that have been fixed in earlier revisions and try to characterize them to see how complicated the fix is.

Through these reviews, we have observed evidence that many FindBugs warnings can be fixed with relatively simple fixes, but we have also observed many instances of low impact defects for the reasons described in Section 4.1. In the next two sections, I discuss our reviews of warnings in a code base that was not developed with FindBugs (the Java JDK) and a code base that periodically ran FindBugs (GlassFish). In code bases that run FindBugs, we looked for evidence of fixes that may have been induced by specific warnings.

5.1.1 Review of Sun's JDK 1.6.0

We analyzed builds b12 through b105 of JDK 1.6.0 (89 builds) so that we could review a subset of the warnings generated. One of the subsets we reviewed was warnings that were removed at some point in the build history. Specifically, we looked at each high/medium priority correctness warning that was present in one build and not reported in the next version, but the class containing the warning was still present. To simplify the analysis, we only examined defects from files that were distributed with the JDK. In the end, we reviewed 53 defect removals; the results are shown in Table 5.1. Interestingly, 37 of the fixes were small changes that seemed to directly target the warning. We have no way of knowing how much effort went

Table 5.1: Classification of Warnings Removed During JDK 1.6.0's Development

Small change that appears to target the warning	37	70%
Change that only a partial remedy to the underlying problem	5	9%
Substantial code change or refactoring that has a broad scope	11	21%

into detecting these problems with simple fixes, but it is likely that if static analysis had been applied earlier, this effort would not have been expended. Five of the changes had the effect of lowering the priority of the warning (according to FindBugs' heuristics) because they reduced the likelihood of the defects causing software misbehavior, but they did not eliminate the defects. The developers responsible for these changes might have made different choices if they were aware of the static analysis warnings, potentially preventing additional effort down the road to fix the root defect.

In our next study, we considered the warnings that remained in the last build, and tried to determine how impactful they were. We reviewed 379 high/medium priority warnings, as shown in Table 5.2. 10% of the warnings looked serious, and it was clear that the method containing the warning would behave in a way that was substantially at odds with its intended function. Another 46% of warnings were associated with some deviation from intended behavior. Of course, it is possible that many of these have no real impact in practice, because the method is never called, or the defect is mitigated by some distant code fragment or process. And we reviewed a sizable number of warnings (42%) that appeared likely to be true but low impact defects.

Table 5.2: Classification of Warnings Remaining in JDK 1.6.0 build 105

Likely infeasible or cause little or no deviation from intended behavior	160	42%
Likely to cause some deviation from intended behavior	176	46%
Likely to cause substantial deviation from intended behavior	38	10%
Bad analysis by FindBugs	5	1%

So these reviews indicate to us that static analysis can find significant defects in a well-used production system, but also validate our expectations that many defects are low impact in practice. We broke down these numbers by bug pattern in an earlier publication [17]. It is interesting to note that some bug patterns almost always appeared to be low impact (such as a potential null pointer dereference on an exception path), while others almost always appeared to be serious (such as integer shift by an amount in an illegal range). This understanding has partly informed the way we rank warnings, and can inform the way organizations choose which warnings to filter out.

5.1.1.1 A Note on Warning Density

We measured the warning density in each of the builds analyzed, to validate our expectations about the typical density for a large production system. We calculate density as the number of warnings per 1,000 lines non-commenting source statements[1]. FindBugs' heuristics are tuned to produce a relatively sparse density

[1]We can compute the number of non-commenting source statements accurately using the line number tables associated with each method in a class file. Statements that span multiple lines

Table 5.3: FindBugs Warning Densities in JDK 1.6.0 build 105

Build	b12	b51	b105
# Warnings	370	449	407
warnings/KLocNCSS	0.46	0.45	0.42

of warnings, particularly high/medium priority correctness warnings. For example, Table 5.3 shows some densities for early, mid-way, and late builds. This sparse density (approximately 1 warning every 2,000 lines) reflects the desire to not inundate developers with too many warnings.

5.1.2 Review of Glassfish v2

Glassfish is an open-source, Java EE Application server, used by many enterprises[2]. Members of the Glassfish project have shown substantial interest in FindBugs, and have been running FindBugs against their nightly builds for several years. They have experimented with several approaches to alerting developers about warnings, including posting warnings on a web page, emailing results to developers, and including warnings in a continuous build.

At the time we conducted our review, Glassfish had been using FindBugs for about a year. We analyzed Glassfish v2, builds 09-b33, and looked for warnings that were present in one version and not reported in the next build. We restricted our analysis to high/medium priority correctness warnings, ignored defects that disap-

count as one line. Using this measure usually results in a value that is about 25-33% of the total number of lines in the file.

[2]http://glassfish.dev.java.net/

Table 5.4: Classification of Warnings Removed During Glassfish's Development

Substantial code change or refactoring that has a broad scope	8	14%
Small change that appears to target the warning	50	86%
Mention FindBugs in the commit message	17	29%

peared because the file containing them was removed, and only considered files in the Glassfish source distribution. There were a total of 58 bug defect disappearances, as is illustrated in Table 5.4. A significant number of the fixes only required small edits, and 17 cases included a commit message that made it clear that the fix was in response to FindBugs. This large number of small fixes does raise the question of whether users are doing due diligence to make sure the code is correct. It is tempting for users to simply fix the defect flagged by FindBugs, without considering its wider implications. Still in this review, it appears most fixes were straight-forward with few side effects.

Despite the usage of FindBugs, the defect density in Glassfish v2 was equivalent to the density in JDK 1.6.0. Specifically, the defect density for high/medium priority correctness warnings in build 33 files included in the source distribution was still 0.44 defects / KLocNCSS (which corresponds to 334 warnings).

5.2 Fix Rate and Code Churn

In addition to user reviews from the fixit discussed in Section 3.3, 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 [79, 78].

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 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.

Table 5.5 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. An effort was made to build and analyze the entire Java codebase at Google each day. 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).

Before analyzing this history, we applied several steps to "clean" the data. We didn't consider issues that went away because the class that contained the issue was deleted or became unavailable. Also, if more than one third of the reported issues for a bug pattern disappeared between snapshots (and there were more than

Table 5.5: Fix rate for bug patterns in Google code base[3]

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

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

[3]Detailed descriptions of each bug pattern are available online at http://findbugs.cs.umd.edu/inpractice/

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 Table 5.5, 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 (*const*), the number of issues that appeared in some version but not in the final snapshot (*fix*), the maximum number of issues that disappeared between any two successive snapshots (*max*), and the name of the pattern or group of warnings (*kind*).

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.

5.3 Finer-Grained Snapshots

The studies described so far are limited because they only capture defects that make it into the software repository. But in practice, many defects are found and fixed (or suppressed) before a developer checks any code into the repository, especially if the developer is alerted by static analysis tools in the IDE or build system. We need to include these *transient* defects in our studies to more accurately understand which bug patterns developers choose to fix, and to determine if developers are wasting energy debugging problems that can be identified more quickly by static analysis. Transient warnings may refer some of the more important defects that occur during development, since users often do some quality assurance activities before committing any code into a repository, and certainly before issuing a release.

The primary challenge when capturing these transient defects is keeping the data capture lightweight so that it does not interfere with development activities. Some static analysis frameworks store all warnings in a central database for all users in an organization, and hence already capture some of this information. Other frameworks only record warnings locally, and hence need to be instrumented to save this local information in a persistent location.

Another challenge is inferring the state of warnings. Specifically, we would like to know when a defect gets fixed, causing the warning to disappear, or when a warning is suppressed using source-level suppression or some other mechanism. We may need to modify static analysis tools to get access to information about which warnings are suppressed.

In this section, I describe two approaches that attempt to collect more fine-grained snapshots of development activities, to more accurately measure the comings and goings of static analysis warnings. In Section 5.3.1, I describe and analyze fine-grained snapshots which have been captured from student development activities at the University of Maryland. To capture these snapshots, a copy of the student's workspace was saved to a CVS repository every time the student saved a file. In Section 5.3.2, I describe the ATMetrics system—which I worked on at Microsoft—to instrument a heterogenous pool of static analysis tools through the IDE. ATMetrics is going to be deployed to thousands of developers at Microsoft [18].

5.3.1 The Marmoset Project

The Marmoset project was started at the University of Maryland to enable students to submit programming assignments to a central server and get instant feedback about their performance [130]. A research component of this project also captured snapshots from students learning to program in Java, persisting their source files to a CVS repository everytime they saved a change. 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.

We can use this dataset to investigate which defect classes are introduced during development, and which ones are eventually fixed. In addition some defects may be removed quickly, while others may persist, affecting the behavior of the program until they are fixed. In addition to quantitative trends, we can also inspect

the code looking for anecdotes that reveal some aspect of the user's interaction with defects. In particular, we may be able to identify cases where students are "spinning their wheels" trying to make the program work, when the core defect is something that can be identified by static analysis. If a study like this is conducted in a commercial environment, these anecdotes could be used to estimate the cost of NOT using static analysis, i.e., the effort wasted by developers trying to find defects that can be detected almost instantaneously by static analysis.

5.3.1.1 Methodology

To conduct this study, we selected a few projects from two semesters, and attempted to compile every snapshot available[4]. Of course, not every snapshot compiles, because students sometimes save incomplete source files. Indeed, one of the limitations of this approach is that we will get a different granularity of snapshots from each student (see discussion on threats to validity in Section 5.3.1.5). We then analyzed the compilable snapshots using FindBugs, and used FindBugs' historical analysis features to aggregate the results for each student, allowing us to measure the number of compilable snapshots between when a warning is introduced and when it is removed.

One immediate observation from our analysis is that some warnings were already present in the initial project code templates that were provided to students at the start of each project. These mostly lower ranked warnings were duplicated for each student, and hence accounted for about 25% of all warnings seen. We ex-

[4]Snapshots were available for students who formally provided consent.

Table 5.6: Overview of Analyzed Marmoset Data

	Semester 1	*Semester 2*
Number of Students	118	38
Number of Projects	2	3
Number of Warnings[5]	3,894	2,398
Warnings Fixed	2,750 (71%)	1,759 (73%)
Correctness Warnings	832 (21%)	722 (30%)
Rank 1-4 Warnings	187 (5%)	171 (7%)

clude these warnings from our analysis, and focus on warnings introduced during development.

5.3.1.2 Overview of General Trends

Table 5.6 presents some high level facts from this study. When we exclude warnings present in the templates provided to students, we observed over 6,000 warnings, and about 72% of them were fixed before the students' final submissions.

Closer inspection of the warnings reveals that some of the most common bug patterns refer to low impact defects that are only visible because of the fine granularity of this study. Table 5.7 lists the top 10 bug patterns. Five of the top six (marked with an asterisk) are likely to be very transient, and occur frequently only because students often save incomplete code. For example, *Dead Local Store* defects occur when a value is saved in a local variable which is never used; this occurs often as the student is writing the method and is always eventually fixed. Unsurprisingly, these five bug patterns have high fix rates, above 89%. Other bug patterns may

[5]Excludes warnings present in the first revisions

Table 5.7: Top Bug Patterns

Rank	Bug Pattern	Total	Fixed
1	*DLS_DEAD_LOCAL_STORE	1,298	1,180 (91%)
2	*URF_UNREAD_FIELD	985	913 (93%)
3	SBSC_USE_STRINGBUFFER_CONCATENATION	611	90 (15%)
4	*UUF_UNUSED_FIELD	465	424 (91%)
5	*NP_UNWRITTEN_FIELD	355	317 (89%)
6	*UWF_UNWRITTEN_FIELD	240	235 (98%)
7	SIC_INNER_SHOULD_BE_STATIC	157	103 (66%)
8	⋄GC_UNRELATED_TYPES	157	137 (87%)
9	NP_NONNULL_RETURN_VIOLATION	128	63 (49%)
10	OS_OPEN_STREAM	126	39 (31%)
12	⋄RV_RETURN_VALUE_IGNORED	119	105 (88%)
16	⋄NP_ALWAYS_NULL	62	57 (92%)

have some transient instances like this, but these five bug patterns are the most dominant, and they skew our results. Hence we exclude them from the rest of our analysis. Doing so reduces the number of warnings to 2,949, and the number fixed to 1,440, or 49%.

The rest of the top bug patterns exhibit two general trends. Some bug patterns with high fix rates (marked with a ⋄) appear to be serious defects that are fixed often. The remaining bug patterns are low impact bug patterns that have lower fix rates. We manually inspected some of the bug patterns in these two groups to see if the high impact defects are being fixed because they are causing problems, and the low impact defects are being ignored because they have no effect. We discuss our observations in Section 5.3.1.4.

Table 5.8: Fix Rates for Different Subgroups

%	remain	fixed	category	rank	priority
86.0	50	306	C	1-4	
85.8	51	307		1-4	
81.5	47	207	C	1-4	H
81.3	48	208		1-4	H
78.4	207	752	C		
75.5	164	506	C		H
66.0	317	615			H
48.8	1,509	1,440			

5.3.1.3 Bug Patterns with High Fix Rates

Interestingly, once we exclude the highly transient defects described in the previous section, the most severe bug patterns tended to be the ones with the highest fix rates. Table 5.8 illustrates what happens when we group defects by bug rank, by priority, or by category. It shows that any combination of high priority, correctness, or Rank 1-4 (scariest) issues results in a fix rate that is statistically significantly higher than the overall fix rate. These are the bug patterns we are most interested in inspecting manually to see what sorts of problems they are causing for students. This high fix rate also partly confirms the survivor effect discussed earlier in Section 4.3, i.e., that the most serious defects are generally fixed, even if the user is not using static analysis. Of course, in some cases, it is possible that students may have run FindBugs and fixed the defects in response to an alert. But, particularly for the defects that persist for many snapshots, it appears that the student was often fixing the defect as they were trying to make the program work correctly.

5.3.1.4 Manually Inspecting Defects that Persist

Sometimes a student would notice the problem soon after introducing it, and fix it immediately. About 37% of resolved issues were fixed after just 1 snapshot (28% if we exclude the highly transient bug patterns flagged in Table 5.7). Other times, the problem would persist for several snapshots before being fixed. A manual investigation of some of these scenarios 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 5.1. 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.

The other part of the survivor effect is that defects that do not matter remain undetected and unresolved. We observed some cases of seemingly serious defects that remain in the last revision and do not prevent students from passing the assignment. An example is shown in Figure 5.2. Here the student checks if a container of `Edge<E>`

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

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.

5.3.1.5 Threats to Validity

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 pro-

```
1    private LinkedList<Edge<E>> edges;
2    public int getAdjEdgeCost(Vertex<E> endVertex) {
3        if(edges.contains(endVertex)){
4            throw new IllegalArgumentException("Edge already in graph");
5        }
6        ...
7    }
```

FindBugs: "`Vertex` is incompatible with expected argument type `Edge` on line 3"

Figure 5.2: Bug: Unrelated Types in Generic Container

vide an opportunity to illustrate the introduction and removal of warnings, which we would be unable to do with confidential commercial data.

Another threat to internal validity is that since students are working on the same programming assignments, they are likely to make the same kinds of mistakes. So some warnings may occur disproportionately often. Because this threat, we do not draw too much significance from the absolute number of defects that occur, but instead focus whether they are resolved, and make qualitative observations about whether they matter. Finally, some observations in this study may not generalize because student development and debugging skills are not equivalent to professional skills.

5.3.2 ATMetrics: Instrumenting Static Analysis on the Desktop

The Analysis Technologies team at Microsoft manages a number of static analysis tools and provides support to thousands of developers who use these tools. ATMetrics (Analysis Technologies Metrics) was an effort to better understand how

developers are interacting with these tools, and learn from their experiences, with the ultimate goals of improving the tools and associated processes, and demonstrating the value of using tools. In particular, ATMetrics focused on getting previously unavailable information: the developer's activities on the desktop. Microsoft developers likely fix or suppress many issues before the code is checked in because they are alerted by the build system, and because some warnings can prevent code integrations.

To capture this missing information, ATMetrics setup lightweight instrumentation on developer workstations to capture metrics on which warnings occur, which ones are fixed or suppressed, and other details about user interaction with static analysis. The primary challenges of putting together a system like this in an industrial context were correctly and robustly inferring the actions of developers with very little overhead, and supporting many heterogeneous static analysis tools. In addition, the IDE containing much of the instrumentation was still under development during this effort. In this section, I present an overview of the key questions driving the design of ATMetrics, and the implementation challenges I encountered. I do not include any results because they are not yet available.

5.3.2.1 Key Questions

The most basic question is: which warnings occur on the desktop, which ones are fixed and which ones are suppressed? If a warning occurs often but is generally suppressed or ignored, then this may indicate that the associated analysis needs to

be tweaked or the warning deprioritized.

Another question is: how do static analysis tools impact developers? We expect developers to modify their coding styles as static analysis tools alert them of best practices or code conventions they were previously unaware of. We can validate this expectation if we observe that certain warnings are introduced at lower frequencies as the developer becomes more experienced. Alternatively we may observe different patterns in different teams or at different parts of the software development cycle. All these trends can inform the policies we recommend to groups and the way we promote tools.

One possible impact of tools on developers comes from the presence of issues in legacy code. When a developer makes even small changes to a legacy file, they may be confronted with many old warnings. One way to deal with this is to *baseline* old warnings, i.e., to temporarily hide them from view. The rationale is that warnings in older code are less likely to be serious since this code has undergone extensive quality assurance testing. We expect that developers will fix more issues sooner if they are only shown new ones. Using our instrumentation, we can compare the fix rates of developers that hide old issues with those of developers that keep them visible.

The ultimate goal is to understand how usage of static analysis affects the quality of the final software product. This is in general a hard question to answer since there are many factors that may affect component quality. Hence our goal is simply to look for trends and correlations between static analysis usage patterns and existing business metrics. We can use internal metrics about components such as the

number of reported crashes or security flaws, and compare these metrics with data from our instrumentation including fix and suppress rates, new issue introduction rates, and whether baselining is used or low priority issues are filtered out. We can also compare our instrumentation data with the policies and practices we observe in different groups.

5.3.2.2 Implementation and Challenges

The ATMetrics implementation builds upon existing platforms and processes used in Microsoft. Specifically, the instrumentation was designed to be a lightweight add-on to a static analysis viewer, and the custom data points are transmitted and aggregated using Microsoft's Software Quality Metrics (SQM) [101], a platform for collecting remote data from thousands of volunteers, used in many Microsoft products.

One of the primary challenges in constructing this instrumentation was inferring whether issues are being fixed, suppressed or ignored. We are not parsing the source code or even monitoring every key stroke as this would be too much overhead. All we can see is when issues appear and disappear. Based on this, we have to classify issues into one of the three groups: *Fixed*, *Suppressed* or *Ignored*.

Any issues displayed in the viewer are moved to the *Ignored* group. The exception is baselined issues, which can appear in the viewer if the user changes appropriate filters.

If an issue that was previously displayed in the viewer disappears, there are

several inferences that we could make. An issue could disappear because it was fixed, because it was suppressed, because code churn put the issue out of reach of the analysis or even because the containing source file was not analyzed. We do not have enough information in the viewer to make all these inferences, so we need to refer to the static analysis tools to get more information. This is another example of information that is only available to us on the desktop.

To detect which issues have been suppressed, we query the static analysis tools for the full list of all issues generated before suppression is applied. Any issues in this full list that do not appear in the viewer can be inferred to be suppressed. This strategy is limited by the fact that we have many heterogeneous tools and is not implemented for all tools.

Any issue that does not appear in the viewer and is not suppressed is considered fixed, unless its containing source file was excluded from the most recent analysis run. A file could be excluded because it is not actively checked out, or because the developer chose to build only a subset of files. If we can determine that the containing source file was excluded, then we allow the issue to retain its existing classification.

More details about the implementation have been included in a technical report [18].

5.4 Summary and Related Work

Through my research mining code repositories, I have observed evidence that important defects are fixed early, while those that do not matter are left behind. This reinforces the notion that static analysis should be run early to be most useful. And some of those defects fixed early may have been a pain to resolve. In my study of marmoset data, I observed students expending effort to address problems that they could have found and fixed easily using FindBugs.

To handle the noisy nature of the data in the repository, I have experimented with using noise warnings to represent code churn. Noise warnings are designed to depend on some aspect of the surrounding context, and their presence does not signify a real defect. Hence their addition and removal should be unrelated to the actual effort to fix any defects, and should track closely with the natural changes in the code. So it is interesting to observe the classes of defects that are fixed significantly more often than noise defects, even if the project is not using static analysis. Equally interesting is that some defects are fixed significantly less often than noise. In addition, some subgroups of issues (high priority, or issues in the correctness category) are fixed significantly more often than other defects, validating the effort in FindBugs to classify these warnings accordingly.

Some researchers have started mining software repositories to capture information about the density of static analysis warnings, which can be used in subsequent studies and projections. Nagappan and Ball [106] studied the pre-release defect densities for Windows Server 2003 and found a strong correlation between the den-

sity of static analysis warnings in different components and the pre-release defect density identified by testing. Buse and Weimer [30] found a correlation between the presence of FindBugs warnings in components and a custom metric for readability.

Other researchers have focused on predicting important bug patterns based on which warnings were fixed in the past. Ruthruff et al. [125] used logistic regression models to predict which warnings will be fixed based on factors associated with warnings that were fixed – factors such as the age of the source file containing the warning, the number of warnings in this file, and the file churn. Kim and Ernst predicted which warnings should be ranked high based on how long they remained in the code base [78] and which warnings were fixed [79]. They did not assume developers saw the static analysis warnings but correlated the removal of a warning to its importance, and they increased their confidence in this removal by emphasizing warnings removed during bug fix commits. Our studies are similar to these except that we introduce noise detectors to help us identify what portion of the removal rate may be due to the natural code churn of software development.

Some of these studies depend on the identification of "fix commits," or changes in the repository that resolve a defect. Usually fix commits are identified by highlighting the commit messages that link to a report in a separate bug tracking system. These fix commits are then used to flag significant warning removals, or to track down and study the bad changes that precipitated the need for a fix in the first place, so called "fix-inducing" commits. But there are pitfalls to this approach for identifying fix commits. In related earlier work, I reviewed potential fix commits (and associated source code) from two Java projects and one Python project [14].

I observed that not all bug reports are associated with a defect; a sometimes substantial number are requests for enhancements or reminders of needed tasks. So associated source code changes may not be fix commits. Even when the bug reports are related to a defect, the associated source code changes may include other activities such as adding test cases, or refactoring code. And even when the source code changes are fixing defects, the fix may not be located near the actual defect [14].

The question of finding fix-inducing commits has been explored and refined by researchers focused on creating a link between the bug report database and the code repository using commit messages [127, 138, 80, 11]. Kim et al. [80] and Williams et al. [138] manually reviewed the source code of fix commits to decide if they were true fixes, but did not review the bug reports to distinguish between defects that refer to incorrect behavior and those that are enhancements. Antoniol et al [9] distinguished between enhancements and bugs in bug reports using automatic classification techniques which could be instructive to future research efforts to identify bug fix commits.

Other projects have explored instrumenting software development activities, though I am not aware of any other work that instruments static analysis tools in a commercial environment to capture developer interactions on the desktop. One popular framework for instrumenting software development activities is Hackystat [73, 72], a general purpose framework that enables software projects to define, collect and analyze a wide variety of metrics. The data collection system I used at Microsoft—SQM—was designed for robust lightweight collection from millions of customers, not just software teams. I chose to use SQM because it is supported

within Microsoft and widely used in many products. But I still had to make many of the same considerations and tradeoffs Hackystat users make including assuring data correctness, distinguishing files and projects, making the system configurable, and scaling to potentially millions of data points.

Chapter 6

Null Pointer Bugs in Practice

Much of the focus of static analysis tools in Java is on detecting potential null pointer dereferences (NPDs). Within FindBugs, significant effort has been devoted to tweak the numerous bug detectors that are associated with null pointer errors. And many publications from the research community focus on null pointers [63, 107, 16, 112].

This focus is not surprising, because null pointer exceptions (NPEs) seem to show up regularly in Java executions, and can be difficult to debug. This is in part because many API methods (including many in the Java language) return null to represent "no answer," rather than throw an exception or some other action. For example, `Map.get(K key)` returns null if the key is not associated with any value, `File.listFiles()` returns null if the target file does not represent a readable directory, and `Queue.poll()` returns null if the queue is empty. The presence of null in languages like Java is the source of some controversy, with one observer describing null as a billion dollar mistake [61], and others regarding null return values as dishonest [111] or evil [7].

So static analysis tools seek to make a positive contribution by finding potential NPDs, and throughout this thesis, I have presented numerous examples of successful finds. But there are pitfalls in practice; enough to encourage us to investigate the

ways NPDs occur in real software, and the types of problems found by static analysis.

One pitfall is that in memory-safe languages like Java, potential NPDs do not necessarily signal defective code. Sometimes potential NPDs occur only in scenarios where expected preconditions or invariants are violated, and the only reasonable action is to throw some kind of runtime exception. Of course, it may be more informative in some of these cases to throw an exception that enables debuggers to better understand the source of the problem, but always doing so would make the source code more complicated, and hard to read and maintain. It makes sense to throw an NPE when a null value is supplied to a parameter that is required to be non-null. But doing so by way of an explicitly constructed exception is only slightly more informative than doing so by dereferencing the null value.

Another pitfall is that NPEs are often the conspicuous side-effect of more subtle logic errors. Static analysis is good at directing developers to the site of a potential NPD, and even in some cases to the site of the source of the null value. But this may not be sufficient aid in cases where a more inconspicuous problem is at fault.

Finally, static analysis tools are limited in their ability to correctly infer the nullness of values, and sometimes flag cases that are impossible. Even when the analysis is correct, we observe some cases where the circumstances that would lead to a null value assignment are highly unlikely in practice.

In this chapter, I explore null pointer defects in practice, highlighting some of these pitfalls, and discussing the pros and cons of API design choices that lead to

the spread of NPDs[1]. In Section 6.1, I review reasons why a potential NPD may not signal defective code. In Section 6.2, I describe a study in which we inspected dozens of real NPEs reported in bug tracking systems, to understand what caused them and how they were fixed. This study indicated that many errors were not associated with mishandled null values, but rather were unrelated logic errors which manifested as null pointer exceptions. In Section 6.3, I present a comparison of different null pointer analysis tools, identifying how often they found cases that were impossible, or implausible, and highlighting some of the challenges of the analysis. Tools included in this review were XYLEM [107], Coverity Prevent [66], Fortify SCA [128], Eclipse TPTP [48], and FindBugs. Finally, in Section 6.4, I discuss API design considerations and the challenge of making static analysis effective for null pointer defects.

6.1 When is it a Defect?

Recall that we define a defect as a problem that developers would generally choose to fix. One would assume, then, that any feasible potential NPD must be a defect. But in managed languages like Java, a reasonable developer may choose to pass when faced with concern about NPDs, by assuming certain preconditions are met.

Consider the example in Figure 6.1, which attempts to delete a directory. The call to `listFiles()` on line 6 returns an array of all the files in the directory rep-

[1] pun intended

```
1   /**
2    * Deletes the directory at dirName and all its files.
3    * Fails if dirName has any subdirectories.
4    */
5   public static void deleteDir(File dir) {
6       File[] files = dir.listFiles();
7       for (int i = 0; i < files.length; i++) {
8           files[i].delete();
9       }
10      dir.delete();
11  }
```

Figure 6.1: A potential null pointer dereference if `dir` is not a directory

resented by `dir`. But this call can also return null, which it does if `dir` does not represent a directory. The value returned is unconditionally dereferenced on line 7, without any check to confirm that `dir` is a directory. Some researchers consider this a defect [112], but FindBugs does not. This is because there may be an unstated precondition to the `deleteDir()` method that `dir` should refer to a readable directory, and if this precondition is violated, then a runtime exception should be thrown. If this is the case, then the most significant defect is outside the `deleteDir()` method. Within the `deleteDir()` method, the only reasonable "fix" is to throw a more informative runtime exception, such as an *IllegalArgumentException*, which would make the problem slightly easier to debug.

If `deleteDir()` is a public utility method in a library, then throwing an NPE because of an invalid argument is confusing and undesirable behavior. In the next section, we document some cases where developers do resolve a potential NPD by

throwing a different runtime exception. If, on the other hand, `deleteDir()` is a private method used only within a specific application, then developers may choose to ensure that it is never called with an inappropriate argument, and fix any unexpected exceptions by tracking down the fault outside the `deleteDir()` method. Constructing explicit exceptions makes the code more verbose and hard to read, and some situations call for establishing and enforcing preconditions instead.

To clarify, I am not suggesting that an analysis which flags this case as a potential NPD is technically incorrect. Indeed, some developers may want to see warnings that point to problems like this. But the question of whether the flagged code should be modified is subjective, and depends in practice on the priorities and constraints on the organization. Remember that every code change raises the possibility of introducing a new error, and takes time away from other software quality activities.

In fact, there is an issue with the `deleteDir()` method in Figure 6.1 that is potentially more serious than the null pointer issue. The `delete()` method returns false if the deletion was not successful, and the `deleteDir()` method ignores this return value. As a result, the `deleteDir()` method might delete some but not all files in the directory, and not provide any warning, or signal that the deletion was incomplete.

Figure 6.2 illustrates another example. The `createTask()` method is used frequently in the Apache Ant project[2] to create a new instance of a specified task. As the documentation in the figure indicates, if the task name is not recognized, then

[2] `http://ant.apache.org/`

Source: Apache Ant | org.apache.tools.ant.Project

```
1172    /** ...
1173     * @return an instance of the specified task, or <code>null</code> if
1174     * the task name is not recognised. ...
1175     */
1176    public Task createTask(String taskType) throws BuildException {
1177        ...
1178    }
```

Figure 6.2: Method in Ant that sometimes returns null

a null value will be returned. If this return value is unconditionally dereferenced, a potential NPD could occur. A number of static analysis tools issue dozens of warnings for these potential NPDs [107] (also see Section 6.3). However, it seems unlikely that developers will often use an unrecognized task name, since usually only a few standard and some custom tasks are used within each project. Some standard task definitions may be unavailable if Ant is started with a corrupt properties file, but developers may not find this a compelling enough reason to insert explicit null checks everywhere `createTask()` is called.

One common thread through these examples is that they involve standard API calls (from Java and Ant) that sometimes return null, but usually return a non-null value. Developers may choose to deal with these APIs by checking for null every time they are used, or enforcing logical rules and policies to ensure they are never used in a way that returns null. If developers choose that latter approach, then static analysis warnings about potential NPDs involving these API calls are not useful. However, there may be some opportunity for developers to customize

the static analysis and use it to enforce the logical rules and policies. I talk more about custom bug detectors in Chapter 8.

6.2 Mining Bug Reports for Null Pointer Exceptions

One way to better understand how NPEs impact code quality in practice is to examine bug reports from real projects. We manually reviewed some bug reports and associated source code changes from the Apache Ant project to better understand why NPEs occur and how developers deal with them. This gives us an opportunity to assess how much the different static analysis techniques might help with the tasks of finding and resolving problems. This also gives us an opportunity to speculate on the helpfulness of tools that enable developers to track down the source of the null value. Specifically, we examine some of the bug reports highlighted in previous research to use static analysis to assist developers debugging NPEs [126].

During the review, we observed that many problems were not due to mishandling null values, but a logic error which manifests itself as an NPE. Usually the developer has to fix this root logic error, but in some cases it was more convenient to fix the dereference site by anticipating null and recovering.

A review like this is necessarily subjective, but we restrict ourselves to questions that can generally be answered objectively[3].

[3]Detailed results from this review are available at `http://findbugs.cs.umd.edu/inpractice/`

6.2.1 Procedure

We identified candidate bug reports by searching Ant's Bugzilla database[4] for Null Pointer Exceptions. We reviewed 50 reports including all the issues that were unresolved (two), and six issues that are referenced in [126], with the remaining issues selected randomly. During the review, we examined the bug report comments and the associated source code changes looking for trends and answers to specific questions we had.

We used a number of strategies to identify the relevant source code. Most bug reports included a stack trace which we used to identify the source line where the exception occurred. In addition, many source code changesets associated with bug reports included a bug report number in the commit message, following a convention used by the project's developers. We were able to find most of the relevant source changes by searching the code repository[5] for this number. Where these strategies did not work, we relied on the bug report comments to understand and find the problem. In some cases, it was helpful to search duplicate issues for a stack trace.

In our final results, we excluded 5 cases for which we could not find enough information to make our classifications with confidence, often because the developers were making many refactoring and design changes not necessarily related to the reported problem. We also excluded 4 reports that were not relevant because they did not contain NPEs, or in one case, the NPE was in the reporter's test case. In the end we reviewed and classified 41 issues.

[4] https://issues.apache.org/bugzilla/

[5] http://fisheye6.atlassian.com/browse/ant

6.2.2 Classification

We looked through a number of bug reports to decide what classification schemes might lead to the most objective results, while also providing useful information. In the end, we settled on the following primary classifications which do not capture all the trends we observed but allow us to be fairly objective during our review.

6.2.2.1 Dereference Site Classification

This classification helps us decide if the original developer made incorrect assumptions at the dereference site or if their assumptions were correct. We use this to indicate whether the problem was due to mishandling of null values at the point where the NPE occurred or due to a logic error elsewhere.

Local logic error: The code handles a potentially null value in inconsistent ways. For example, a variable may be unconditionally dereferenced, and then checked for null a few lines later without writing to it in between. In this case, the developer first assumed the value was not null, then assumed it might be null.

Should have checked for null: The code should have anticipated null and handled it. For example, we use this designation if the unconditionally dereferenced value was returned from a method that specifies that it can return null.

Unrecoverable null: Code correctly expects the value to be nonnull. For example, many methods explicitly indicate in the comments that a given parameter

should not be null. This designation represents the scenario where a null value reaching the dereference site probably indicates a problem somewhere else in the code.

In each review, particularly when deciding between *Should have checked for null* and *Unrecoverable null*, we may have to look at surrounding code for evidence before assigning the issue to one classification or the other.

6.2.2.2 Local Analysis Check

Is it obvious from local information the circumstance under which the value is null? Here we use two classifications:

Local check suffices: We can determine the source of the null value by just looking at the local method, or reading the specifications of methods called in the local method.

Nonlocal search needed: It is not obvious from the local information why the value is null.

This classification gives us some idea of how easy it is to understand the problem and how often an analysis technique would have to search deeply to find the null source.

6.2.2.3 Corrective Action

What was done to resolve the problem? This classification only applies to those issues that have been resolved and where we can identify the corrective source

code or changeset. Our classifications are:

Fix local logic error: Correct inconsistent handling of a value within the method containing the dereference site

Anticipate null, recover: Introduce a guard in the local method to check for null and handle it specially, or avoid dereferencing it.

Throw better exception: Detect the null value and throw a more appropriate exception. This can done in the local method containing the null dereference or in a preceding method. Like the previous classification, this fix anticipates null, but it does not recover. Rather it causes the program to fail more gracefully.

Prevent null occurrence: Change code logic so that null does not occur at the dereference site. This fix can either prevent the original null assignment from occurring or avoid calling the method containing the NPD in a way that leads to an exception.

Extensive refactor: In some cases, developers make many changes to refactor the code, or add or remove features as part of the fix. This could even involve removing the method which contained the null dereference.

6.2.2.4 Other Classifications

In addition to the primary classifications mentioned above, we also made some basic observations about each issue:

	Local	Should Check	Unrecoverable
Fix Local Logic	0	0	0
Anticipate Null	0	9	8*
Throw Exception	1	3	4
Prevent Null	1	3	9*
Refactor	0	0	3

*For two reports the corrective action was to both anticipate and recover from null, and to prevent null from happening. We split each of these cases, adding 0.5 to each row for each case.

Table 6.1: Corrective Action Classification

Dereferenced Value: From the perspective of the dereference site, the dereferenced value can be a parameter, a field, a value returned from a method, or a local variable

Where is the fix? The corrective action can be near the dereference site (i.e., in the same method), near the null source, near both or near neither.

6.2.3 Observations

In Table 6.1, I cross-tabulate each Dereference Site classification with each Corrective Action. I observe that 24 reviews received an *Unrecoverable null* classification (or 59% of the issues that received classifications). Only a third of these issue were resolved by anticipating null and recovering. For many of these issues, the appropriate solution was to change the way a value was initialized to prevent null from occurring in the first place. Usually this involved replacing null strings and null containers with empty strings and empty containers respectively.

By contrast, 15 reviews (37%) received a *Should have checked for null* classification and 12 of these were resolved by anticipating null (3 threw a better exception,

while 9 anticipated null and recovered). In many cases, the developer simply forgot to check the value returned from a Java API or other API call that is known to return null.

In the following sections, I present some examples that illustrate these observations, and provide more observations from the review.

6.2.3.1 Handling "Unrecoverable null" issues

For most of the *Unrecoverable null* issues, the corrective action was to prevent the null value from reaching the dereference site or to throw a different exception. These represent the cases where it did not make sense to check for null at the dereference site, and the developer had to instead dig out the root problem. Often unrecoverable null situations occur when a developer unconditionally dereferences a parameter expecting it to be non null. There may also be cases where a developer unconditionally dereferences the return value of a method like `Map.get(key)` because the developer expects (or needs) the key to be in the map at that point in the code.

For example, in issue 17396[6] the reporter observes an NPE when redirecting inputs from a file to an Ant task. The NPE occurred when the `setNewProperty()` method passes its parameters to a Java `Hashtable` (which does not accept null values). Developers were not expecting null values to be passed to any of the parameters because they were reading from a stream. Developers concluded that a `BufferedReader` in the input handler was reading more from the input than nec-

[6]Available at https://issues.apache.org/bugzilla/show_bug.cgi?id=17396.

essary and replaced it with a `DataInputStream` (see changeset 274185[7]). In this case, simply checking for null near the dereference site would not have fixed the root problem.

Another example is issue 5980 in which the reporter observes that the process for executing shell commands is broken on Windows XP. After digging into the problem, the reporter notices the following code fragment is missing a reference to Windows XP:

```
if ( osname.indexOf("nt") >= 0 || osname.indexOf("2000") >= 0) {
    ...
}
```

This inadequate code fragment occurs in a number of places, and is updated (in changeset 271003) to include a reference to XP:

```
if ( osname.indexOf("nt") >= 0 || osname.indexOf("2000") >= 0 ||
    osname.indexOf("xp") >= 0) {
    ...
}
```

Table 6.2 provides more descriptions of the cases we observed in which an unrecoverable null issue was resolved by preventing the null assignment. In many cases, the solution was to change the way the value was initialized. Issue 38056 is a good example of a null string initialization that was replaced with an empty string initialization.

Table 6.2 also describes some of the cases where an unrecoverable null was fixed by throwing a different exception, or refactoring the code. In all the cases where a

[7]Available at http://fisheye6.atlassian.com/changelog/ant/?cs=274185.

Table 6.2: Some "Unrecoverable null" issues fixed by Preventing Null, Throwing Exception, or Refactoring

Issue #	Change-set	Corrective Action	Comments
5980	271003	prevent-null-occurrence	Ant process for executing shell commands is broken on Windows XP, because the code that initializes the "shell launcher" fails to account for XP.
9069	272715	prevent-null-occurrence	The command line for an Execute task was accidentally left uninitialized prior to executing the task.
9138	272826	prevent-null-occurrence	The cleanup() method in AntClassLoader sets some fields to null. Unfortunately, the instance of AntClassLoader is reused after cleanup() is called. Fixed by removing the null assignments from cleanup()
17396	274185	prevent-null-occurrence	A method which expects non-null parameters is receiving null from an input stream, causing the exception. Fixed by changing the type of the input stream from BufferedReader to DataInputStream.
38056	359329	prevent-null-occurrence	A string parameter is allowed to be null, though it is dereferenced later in a method that expects it to be non-null. Fixed by replacing null initializations with empty string.
11833	273253	throw-better-exception	NPE thrown if websphere.home property not set. Fixed by throwing BuildException with message instructing user to set websphere.home property.
25826	275854	throw-better-exception	NPE thrown if DestDir attribute not set. Fixed by throwing BuildException with message instructing user to set DestDir property.
2442	269834	extensive-refactor	Did not anticipate that two tests could be run concurrently. Fix makes many changes to make code more thread safe, including eliminating a field that was causing race conditions.
15994	273546	extensive-refactor	A Buffered reader was returning null, which was passed into a method that unconditionally dereferenced. The fixer deleted both the source and dereference site, opting to use a completely different and more robust method to fix the problem.

different exception is thrown, the developers recognized that the root problem was that some external prerequisite had not been satisfied. For example, an environment variable or property may not have been set, resulting in an unwanted null value. Since the developer cannot advance without this prerequisite, the code is updated to throw a more appropriate exception, with a message to users on what they need to do to remedy the problem. The extensive refactoring cases represent situations where the root problem is more complicated than the apparent NPE. In some sense, the developers may be grateful for the NPEs which exposed underlying design flaws.

6.2.3.2 Anticipating null to resolve unrecoverable null issues

The number of cases where an *Unrecoverable null* issue was fixed by anticipating null was surprisingly high. This seems like a bad practice because a null value in these cases usually indicates a problem elsewhere in the code. Upon closer inspection, we observed that this practice was probably harmless, or the developers accounted for the possibility of a problem elsewhere in the code.

Bug report 5637 is an interesting example of this. The reporter observed an NPE in `XMLJunitResultFormatter` whenever he threw anything from a Test Setup wrapper. The problem was in the `endTest()` method where a dereferenced variable, `currentTest`, could be null. This surprised developers because one would expect that `currentTest` would always be non null when a test ends. After some investigation, developers determined that an exception in the Test Setup wrapper could lead to `endTest()` being invoked even though `startTest()` had not been

called. Since the Test Setup wrapper is a JUnit extension and not a part of Ant, they could not investigate or resolve the problem there. They decided to fix the code by calling `startTest()` if `currentTest` is null (changeset 270512). This did not solve the root problem (calling `endTest()` without calling `startTest()`) but it appeared to be adequate, and one of the commenters added:

> "I think your patch is OK, however it would be nice to add a little comment in the code to explain why it is necessary."

Hence the solution explains this fix, including a comment that has persisted for eight years through the current version of ANT:

```
// Fix for bug #5637 – if a TestSetup is used and
// throws an exception during setUp then startTest
// would never have been called
if (currentTest == null) {
    startTest(test);
    currentTest = (Element) testElements.get(test);
}
```

Table 6.3 lists some of the other cases we observed that were classified like this. Many of these represent unusual corner cases, and perhaps simply anticipating null was a convenient solution.

6.2.3.3 Anticipating null and Preventing null

As Table 6.1 indicates, there were two cases where the corrective action was to both anticipate null, and to prevent null from happening. We interpret this as the developers programming defensively in case their efforts to prevent null from happening

Table 6.3: Some "Unrecoverable null" issues fixed by Anticipating and Guarding for Null

Issue #	Change-set	Corrective Action	Comments
5637	270512	anticipate-null-recover	NPE occurs in endTest() because the currentTest field could be null. This unusual behavior occurs if the test setup fails, and startTest() is not called. Fixed by calling startTest() from inside the endTest() method.
6871	271748	anticipate-null-recover	A change to a super class causes a field that was previously assumed to be non-null to be null. Fixed by adding a guard to check for null.
24344	275602	anticipate-null-recover	Two API methods return null when called from a forked task. Fix by only dereferencing return value if not null.
24440	275615	anticipate-null-recover	A parameter is unexpectedly null with certain inputs to an Ant Task. Fixed by returning from the method when the parameter is null.
31840	276965	prevent-null-occurrence anticipate-null-recover	The root problem is that a string attribute "antlib" is never set, so the fix is to set this attribute. However as a defensive measure at the dereference site, an empty string is used if value is null.
44009	704496	prevent-null-occurrence anticipate-null-recover	A Vector field is not initialized by some users. Fixed by using an empty Vector to initialize, but also guard against null at the dereference sites.

were unsuccessful. These cases are included in the descriptions in Table 6.3. One example of this is bug report 44009 where an NPE occurred in the `MimeMailer.send()` method because the `headers` field could be unexpectedly null. Developers expected all users of this class to initialize the `headers` field but this was not happening in all cases. They resolved this by fixing the classes that failed to initialize `headers`, but then as a precaution they added a guard to the `MimeMailer.send()` to anticipate null and recover.

This fix illustrates a principle which I will discuss more in Section 6.4.1: it is usually better to use an empty container to indicate the exceptional or uninitialized case, than to use null. In this example, `headers` is a Java *Vector* that was initialized to null by default to indicate "no value." The fix changes this default initialization to an empty vector, which leads to more robust code.

6.2.3.4 Handling "Should check for null" issues

Table 6.4 describes some of the issues that were classified as *Should have checked for null*. In many cases, the developer simply forgot to check the value returned from an API call that is known to return null. Hence the solution is usually to insert a guard and skip the offending code.

For example, issues 23320 and 26222 refer to calls to the `getClassLoader()` method in the Java `Class` type. This method returns null for any classes loaded by the "bootstrap class loader," which is the loader for the initial set of classes. Usually this set includes just core classes like `String`, but in various IDEs it may include

some Ant classes. Developers check the return value of this method in some other places in the Ant code, but in these two cases, they neglected to do so.

In some cases, the method returning null is another Ant method which indicates in its specification that it can return null. Sometimes due to inadequate documentation, the called method does not explicitly say it can return null, but it clearly does so in the first few lines. For example, in issue 40847, the reporter observes an NPE at the following statement:

```
StringTokenizer tok = new StringTokenizer(classpath.getValue()," ");
```

The problem is that `classpath.getValue()` can return null; indeed its first three lines are:

```
if (values.size() == 0) {
    return null;
}
```

The developers fix this by adding a guard to avoid the string tokenization if null is returned (in changeset 474481). Here part of the blame probably belongs to the `getValue()` method, which does not explicitly specify that it can return null. And it may be better for this method to return an empty string but developers cannot change this behavior because of backwards compatibility[8].

Table 6.4 also lists two interesting cases where developers choose to prevent the null occurrence, instead of guarding for null. In issue 43292, the null value

[8]One interesting note is that the original reporter found this problem with Parasoft's Jtest BugDetective, which uses interprocedural static analysis to "explore execution paths" [54].

Table 6.4: Issues classified as "Should have checked for null"

Issue #	Change-set	Corrective Action	Comments
10360	273214	anticipate-null-recover	Dereferences the return value of a method that returns null through a condition in its first 2 lines. Fixed by continuing to next loop iteration when returned value is null.
14232	273483	anticipate-null-recover	Method passes the value returned from System.getProperty() into a File object, neglecting that the value returned could be null if the property is not defined. Fixed by guarding for null, and returning immediately.
23320	275280	anticipate-null-recover	SplashTask.class.getClassLoader() returned null when called from JBuilder9, and was unconditionally dereferenced. Fixed by guarding for null, and using alternative method to get class resources.
26222	275899	anticipate-null-recover	Locator.class.getClassLoader() returned null when called from Eclipse, and was unconditionally dereferenced. Fixed by guarding for null, and using alternative method to get class resources.
34878	278239	anticipate-null-recover	Dereferences the return value of a method that indicates that it returns null in its specification. Fixed by adding null-check to a subsequent guard.
40847	474481	anticipate-null-recover	Dereferences the return value of a method that returns null for certain inputs. Fixed by adding null-check to a surrounding guard.
38622	377166	throw-better-exception	Dereferences the return value of Project.getReference(), which indicates in its specification that it can return null. Fixed by throwing a BuildException to inform user to call another method first.
42179	531575	throw-better-exception	Either a file or dir attribute are required for a fileset, but if both are missing, then FileSet.getDir() returns null. Fixed by throwing BuildException informing user that file or dir attribute needs to be set.
43292	572302	prevent-null-occurrence	The return value from the API method FileUtils.readFully() is dereferenced even though it can be null. Problem Fixed by changing the API method so that it returns an empty string instead.
43659	572363	prevent-null-occurrence	The fix to Issue 43292 is problematic due to backwards compatibility. Developers revert this fix, and choose instead to introduce another API method, FileUtils.safeReadFully() to replace the old method in the future.

originates in an API method, `FileUtils.readFully()`, which is known to return null, and other calls to it check for null. Initially, the developer chooses to fix the API method, so that it returns an empty string instead of null. The developer comments:

> "I would rather change FileUtils.readFully() rather that have all the clients do the null check stepdance... I fixed FileUtils.readFully to do the right thing."

Later, while addressing issue 43659, the developer has a change of heart, citing backwards compatibility concerns, and chooses instead to introduce an alternative API method, `FileUtils.safeReadFully()`. This new method simply wraps around the old one, and returns an empty string instead of null.

6.2.3.5 Local logic errors

There were only two cases classified as a local logic error. This may explain why an intraprocedural analysis tool like FindBugs (i.e., one that does not track values across procedure boundaries) finds so few warnings. This may also be a product of the survivor effect, which I described earlier in Section 4.3, and which suggests that many of these local errors will be weeded out by the quality assurance process before the code is released. Static analysis would be more valuable during development, where it can find and fix such issues more quickly and cheaply.

In one issue (bug report 3394), an NPE is thrown in a "Depend" task if a cache is not specified. Careful inspection of the method reveals that when the `cache` field

```
String value = null;
int posEq = name.indexOf("=");
if (posEq > 0) {
    value = name.substring(posEq + 1); ...
} else if (i < args.length - 1) {
    value = args[++i];
}
```

Figure 6.3: Snapshot of code that processes Ant arguments

is null, the dereferenced local variable is never initialized.

The other issue was more subtle, but still within the scope of a static analysis. In issue 22065, the reporter observes that an NPE is thrown if the user specifies the command line argument "-Debug" instead of "-debug". The problem is that while "-debug" is a recognized argument, "-Debug" is treated as a "-D" property with name "ebug" and no value. A snapshot of the code that processes the value is in Figure 6.3.

The if-statement accounts for the possibility that there might be an "=" between a name and a value, but not for the possibility that there is no value! When `value` is dereferenced later on, a static analysis can determine that there is a path in which a null value is guaranteed to be dereferenced. Of course, since the "-Debug" argument is incorrect anyway, the fix was to throw a more descriptive `BuildException` when no value is specified. This issue was detected by FindBugs.

6.2.3.6 Finding the source of the null value

Static analysis algorithms can potentially assist users debugging null pointer exceptions by identifying the source of the null value that was later dereferenced citesinha-issta-2009. However, it is not clear how useful this assistance is because in many cases, the source of the null value is easy to find, or the NPE is the result of an unrelated logic error, and knowing the source of the null value is not helpful.

In this review, only 2 of the *Should have checked for null* issues and 5 of the *Unrecoverable null* issues had fixes near the source of the null value. *Unrecoverable null* issues tended to have fixes at locations unrelated to both the null source and dereference, so information on the source of the null value may not be as helpful. *Should have checked for null* issues tended to have fixes near the dereference site, so just having a stack trace might be sufficient, though developers might need to search around to determine if, for example, there are other places where they should be checking for this null value.

Sinha et al [126] identify 6 Ant issues for which their analysis is able to find a definite or possible source for the null value. In our review of those 6 issues, we found one issue (bug 34878) for which identifying the source of the null value was difficult. The attribute value for a DOM element was null, and tracing back where the attribute was defined to have a null value was tricky. Since it doesn't really make sense for DOM element attribute values to be null, it might have been better to catch the problem at the point where the attribute was set to have a null value.

In the other 5 cases, the null value was returned from a method that returned

null under some circumstance. In 3 of the 5, the null was dereferenced in the same method that invoked the null returning method, and in 2 of the 5 methods, the null value was passed as a parameter that should never be null, and subsequently dereferenced. In the 5 different cases, there were varying degrees of clarity as to why the methods returned null. Three of the methods were well documented and clearly explained the circumstances under which they return null. One (bug report 34878) had minimal Javadoc that mentioned that the method could return null but didn't explain the circumstances under which null was returned. The other (bug 34878) didn't explain the null return in the method documentation, but the method was very short, and the circumstances under which it returned null were obvious from examining the code.

6.2.3.7 Other Observations

For our Local Analysis Check question we observed that 30 of the 41 issues (73%) required a nonlocal search to identify the source of the null value. All the *Unrecoverable null* issues required a nonlocal search, while all the *Local logic errors* only required a local search. A local search suffices for 9 of the 15 *Should have checked for null* issues. These results suggests that many of the reviewed issues might have been hard to resolve with just a stack trace. On the other hand, most bug reports contained much additional information about the context in which the problem occurred and this simplifies the task of resolving them.

In both *Local logic errors*, the dereferenced value was a local variable. In most

Should have checked for null issues, the dereferenced value was returned from a method, while the most popular type of dereferenced value for *Unrecoverable null* issues was a parameter, followed by a field.

We also observed that many *Unrecoverable null* issues had fixes that were neither near the dereference of the null value nor its source, while many *Should have checked for null* issues had fixes near the dereference site.

6.3 Null Pointer Dereferences found by Static Analysis

We reviewed potential NPD warnings from several static analysis tools which were used to analyze Ant 1.6.5. During each review, we decided if the issue was impossible, or implausible. Warnings were classified as impossible if surrounding code logic made it impossible for the highlighted value to be null as reported by the tool. Warnings were classified as implausible if a NPD could theoretically occur, but it seemed unlikely that a null value would be generated in practice. These included several instances where the return value of `createTask()` was unconditionally dereferenced, as discussed earlier (Section 6.1, Figure 6.2). All other issues were classified as plausible. Of course, some of this review is subjective, and classifying an issue as plausible does not mean that it is — it just means we were unable to determine if the null value was unlikely.

The tools included in this study were Coverity Prevent 4.5.0 [66], Eclipse TPTP 3.5.0 [48], FindBugs 1.3.8, Fortify 360 SCA 2.1.0 [128] and XYLEM (November, 2009) [107]. For consistency, we only report issues where a value thought to

Table 6.5: Null dereferences reported in Ant 1.6.5

Tool	Total	Plausible	Implausible	Impossible
Coverity	46	17	15	14
Eclipse	31	11	1	20
FindBugs	11	11	0	0
Fortify	44	14	1	29
XYLEM	57	35	15	7

be null is later dereferenced. Some tools also report other kinds of null pointer issues, such as redundantly comparing a value to null *after* it has been dereferenced. Fortify SCA reports the combined results of FindBugs and its own analysis engine. On the recommendation of Andy Chou of Coverity, we enabled an undocumented and unsupported *effects analysis* feature in Coverity Prevent. Without this feature, significantly more results, all impossible, were reported for Coverity Prevent.

FindBugs relies on an *intra*-procedural analysis and simple heuristics to find potential NPDs [63]. In particular, FindBugs seeks to minimize false positives, and chooses to turn the dial towards reporting warnings for which there is some confidence, over reporting all possible issues. Other static analysis tools adopt different philosophies and use more sophisticated analysis, including some *inter*-procedural analysis. So it is interesting to compare the outcomes of FindBugs with other tools.

Table 6.5 shows the results of this study. FindBugs reports the fewest warnings of all the tools, but all of them were classified as plausible. XYLEM uses a sophisticated interprocedural analysis and reports the most warnings, including twice as many plausible warnings as any other tool. But it and the other tools

Table 6.6: Review of XYLEM warnings in Ant 1.6.5

#	review	why
15	plausible	clear coding mistakes
7	plausible	plausible error condition not handled
13	plausible	seems plausible, but not clear what situation would cause it to arise
15	implausible	calls to Project.createTask() with well known String constant
4	impossible	coupled variables
1	impossible	call context guarantees nonnull return value
2	impossible	value previously dereferenced and thus can't be null

reported a significant number of warnings that seemed implausible, or impossible. None of the plausible issues in Table 6.5 are known to have caused any field failures. (One reported by XYLEM in Ant 1.5.0 is known to have caused a field failure: Bug 10360).

Table 6.6 breaks down the reviews of XYLEM warnings in more detail, including reasons why warnings were classified as they were. All the implausible warnings were calls to `createTask()` with a well known string constant, making it unlikely that a null value would be returned if the properties file is not corrupted. Again, some developers see these are potential defects, and choose to modify the code to include checks. But Ant developers seem to have made the choice to enforce the precondition that `createTask()` only be called with valid parameters. One heuristic used by some static analysis tools is to consider how often the return value is unconditionally dereferenced, and how often it is compared to null. This heuristic

coupled with statistical analysis could inform the static analysis tool of the intent of developers.

Many of the issues classified as impossible involved Java API methods that sometimes return null, but that we could show did not return null because of the surrounding program logic. Figure 6.4 presents some examples from warnings reported by Coverity Prevent. The first two examples illustrate scenarios where the calling context guarantees the flagged value will not be null. In part (a), `getParentFile()` returns null if the File does not contain any instances of the `File.separator` character, but the line above it guarantees that it does. In part (b), `zf.getInputStream(ze)` can return null if there is no matching entry in the `ZipFile`. This value is then passed into `extractFile()`, which unconditionally dereferences it. But since the `ZipEntry` (assigned on line 120) is an element of the same `ZipFile`, we are guaranteed a non-null input stream.

The next two examples illustrate coupled variables, which make a NPD impossible. In part (c), we are guaranteed that `loader.getResource()` returns a non-null value on line 729, because an earlier call to `loader.getResourceAsStream()` on line 726 returned a non-null value. Similarly, in part (d), we are guaranteed that `getParentFile()` returns a non-null value on line 382, because an earlier call to `getParent()` on line 381 returned a non-null value. Static analysis can be more effective if it understands these couplings, but this property is not always obvious for methods that are not part of large standard APIs.

(a) Warning that `getParentFile()` might return null (JonasDeploymentTool.java)

```
587     File f = new File(outputdir + File.separator + key);
588     f.getParentFile().mkdirs();
```

(b) Warning that `zf.getInputStream(ze)` might return null (Expand.java)

```
117     zf = new ZipFile(srcF, encoding);
118     Enumeration e = zf.getEntries();
119     while (e.hasMoreElements()) {
120         ZipEntry ze = (ZipEntry) e.nextElement();
121         extractFile(..., zf.getInputStream(ze), ...);
122     }
```

(c) Warning that `loader.getResource()` might return null (XMLCatalog.java)

```
726     InputStream is = loader.getResourceAsStream(location);
727     if (is != null) {
728         source = new InputSource(is);
729         URL entryURL = loader.getResource(location);
730         String sysid = entryURL.toExternalForm();
731         ...
```

(d) Warning that getParentFile() might return null (JJTree.java)

```
381     while (root.getParent() != null) {
382         root = root.getParentFile();
383     }
384     ...
337     if ((root.length() > 1) ...
```

Figure 6.4: Impossible dereferences reported by Coverity Prevent

6.4 API Design and Null

6.4.1 API Choices

Many API designers run into the challenge of trying to decide what value to return from an API method when no appropriate response is available. Designers would ideally like to return a special value that can be handled naturally, but often have to resort to returning null, or throwing an exception, and both of these options have serious drawbacks. Part of the problem is that in many languages, return values can only have one type; so, for example, if the return type is an integer, one cannot return a string with the value "No Answer Available."

Sometimes, a designer can choose to explicitly limit the range of allowed return values (e.g., all positive numbers), so that a value outside this range can be used to signal an exceptional case (e.g., returning -1). But this strategy does not work if the range of allowed return values includes all values in the domain of the return type. A special case of this strategy—one that API designers often forget—is when the return type is a string, an array or collection. In some languages, like Java, designers have the choice of returning an empty string, array or collection, to indicate that no answer is available. The advantage of this approach (over returning null or throwing an exception) is that most callers do not have to do any special handling or checks; their code can ignore the problem of exceptional cases altogether. For example, if the API method returns an array of integers, and the caller wants to compute their sum, then the caller can simply write a loop that iterates over all values in the array. In the exceptional case, the array length is zero, and the loop is never

executed. Of course, sometimes the empty string, array or collection is one of the possible non-exceptional return values, and API designers need some other solution to indicate "no answer" or other exceptional cases.

We have already discussed some of the tradeoffs between throwing an exception and returning null. Choosing to return null often makes the calling code simpler, especially when the caller is confident the exceptional case will never happen. For example, `Map.get(K key)` returns null if the key is not associated with any value. Often `Map.get(K key)` is called in contexts where the developer knows the key is in the map—the developer may verify the key's presence using `Map.containsKey()`, or may be iterating through the map's keys when `get()` is called. If developers in this situation had to handle an exception, it would require additional boilerplate code, and make the resulting software harder to read. In Section 6.4.2, we do a more detailed case study of the uses of `Map.get()` and conclude that the decision to return null, rather than throw an exception, is a wise one.

On the other hand, sometimes null is an *unexceptional* return value, and the designer needs to find another solution to indicate that no answer is available. For example, in the case of `Map.get(K key)`, a null return value could mean that the key is not in the map, or it could mean that the key is mapped to null. So, a null return value is not sufficient to indicate the exceptional case; developers need to use `Map.containsKey()`.

Other API methods throw exceptions, instead of returning null, to indicate the no-answer case. Exceptions allow the designer to differentiate between multiple no-answer cases, and provide information to the caller that makes the program easier

to debug. In particular, null return values can be stored (unchecked) into a field or database, only to be discovered unexpectedly at some later point, when it is much harder to understand the source of the error. By contrast, every exception includes a stack trace that enables the debugger to find the source of the error, and the name and message associated with the exception provides more context.

The downside of exceptions is that developers need to add verbose boilerplate code to handle special cases, and this can make the software hard to read and maintain. Designers can avoid requiring boilerplate code by using runtime exceptions[9], but this can lead to unexpected program crashes if the caller forgets to handle the exceptional case. This is comparable to the scenario where the caller fails to check a null return value before dereferencing it.

In summary, API designers should seek to use a special return value (such as an empty string or array) to indicate the no-answer case, wherever possible. Where this is not possible, designers may need to make tradeoff decisions between null return values and thrown exceptions. One useful rule to help with this decision is provided by Effective Java: "a well-designed API must not force its clients to use exceptions for ordinary control flow" [25, Item 57]. Furthermore, when an API method is used frequently, in situations where the caller does not expect an exceptional value, then requiring the caller to always handle the exceptional case would annoy them. On the other hand, if the designer wishes to differentiate between multiple exceptional

[9]Runtime exceptions are automatically propagated up the stack, and do not need to be handled in anyway by the caller. Other exceptions (called checked exceptions) must be handled by using a *try-catch* block, or explicitly passing the exception up the stack.

Table 6.7: Invocations of Map.get

Software	invocations	null checked	unconditional dereferences
JDK 1.7.0	2516	1040	325
JBoss 5.1.0	3095	1680	105
Glassfish v3	1225	1672	90

cases, so that callers can handle each appropriately, then thrown exceptions are usually the best approach.

Another option available to API designers is to couple the API method with a guarding method, as is done with `Map.get()` and `Map.containsKey()`. This way, developers can avoid calling the API method in the exceptional case. Another benefit of this approach is that static analysis tools can be extended enforce the rule that the API method must be guarded by the partner method, as described in Chapter 8. But this approach makes the API method call non-atomic, and is more expensive since it requires two calls.

6.4.2 Case Study: Uses of Map.get()

We reflect further on the design choices made in `Map.get()` by considering the ways it is used in practice. `Map.get()` is an example of a frequently used API method that is often invoked in situations where the caller does not expect a null return value. Thus, it seems reasonable to study how the many invocations of `Map.get()` handle the possibility of a null return value.

Specifically, we would like to evaluate the quality of the results of a static analysis tool that generates a warning every time a `Map.get()` return value is un-

conditionally dereferenced. In reviewing these warnings, we are sometimes able to decide if the program logic surrounding the dereference makes a NPD impossible. We also discover heuristics that can make such a static analysis tool more effective. Ultimately, we conclude that the API design choice of using null to indicate an exceptional case seems wise, and a static analysis tool that flags potential NPDs associated with `Map.get()` may be undesirable.

We used FindBugs to examine the invocations of `Map.get()` in several software projects, and counted the number of times its return value is compared to null, or unconditionally dereferenced. Our results, reported in Table 6.7, suggest that about half of invocations are null checked, and about 8% are unconditionally dereferenced.

We manually reviewed the 325 places in the JDK where the return value of `Map.get()` was unconditionally dereferenced. We observed three common idioms in the surrounding program logic that guaranteed the presence of the key in the map:

- The code contained a loop over the keys in the map, and for each key was calling Map.get.
- The code contained an earlier call to Map.containsKey
- The code contained an earlier call to Map.get with the same key.

Of course, the presence of the key does not guarantee that the value will be non-null—the key could be mapped to null. But the unconditional dereference does seem to imply that the developer expects the value to be non-null, and is only concerned about guarding against the no-answer case. In addition, these idioms are useless if another thread might remove keys from the map.

Table 6.8: Idioms used to ensure key present for `Map.get()` call

Surrounding Program Logic Idiom	Count
Iterating through `KeySet`	91
Guarded by call to `containsKey()`	55
Previous check of `Map.get()` != null with the same key	46
No obvious common idiom; NPD might be feasible	133

Table 6.8 presents the number of instances classified into each idiom. A static analysis tool could be enhanced to recognize these idioms to reduce false positives. But over a third of the cases reviewed contained no obvious idiom, and many of these could be false positives since they have been in the code for a long time. And there are many other methods like Map.get() which have common cases in which the return value is never null. (Some cases were presented in Section 6.3 and Figure 6.4.) Constructing static analysis for all these idioms, with few false positives, may not be feasible.

Even though flagging unconditional dereferences of `Map.get()`'s return value will yield many false positives, it is worth noting that many of these unconditional dereferences were associated with questionable or inefficient code. For example, Figure 6.5 part (a) contains three calls to `Map.get()` and one to `Map.containsKey()` (and two of the calls are in a loop). The developer unconditionally dereferences the calls to `Map.get()`, and is confident that the key is in the map because they are all guarded by the call to `Map.containsKey()` on line 295. But this arrangement is inefficient, and could be improved by making one call to Map.get() and comparing its return value to null, as is done in part (b).

Source: Sun JDK 5 | com.sun.codemodel.internal.JFormatter

(a) Inefficient repeated calls to `Map.get()`

```
295     if(collectedReferences.containsKey(id)) {
296         if( !collectedReferences.get(id).getClasses().isEmpty() ) {
297             for( JClass type : collectedReferences.get(id).getClasses() ) {
298                 if (type.outer()!=null) {
299                     collectedReferences.get(id).setId(false);
300                     return this;
301                 }
302             }
303         }
304         collectedReferences.get(id).setId(true);
305     }
```

(b) Single call to `Map.get()`

```
ReferenceList refs = collectedReferences.get(id);
if ( refs != null ) {
    for( JClass type : refs.getClasses() )
        if ( type.outer() != null ) {
            refs.setId(false);
            return this;
        }
    refs.setId(true);
}
```

Figure 6.5: "unchecked" dereferences of Map.get()

6.4.3 Sometimes, an NPE is Better

Finally, despite all this focus on preventing NPEs, we should note that sometimes, a developer would wish to have a NPE if the alternative is a subtle but cataclysmic defect. In particular, many times a potential NPD is associated with a subtle defect which manifests if the NPD does not occur. Even though NPEs are annoying, they do alert developers of a problem and provide a stacktrace for debugging. By contrast, subtle defects may silently lead to undesirable behavior, such as memory leaks or performance degradation, that is hard to debug.

Consider the following code fragment:

```
if (out == null) out.close();
```

If `out` is null, a NPE will be thrown. However, the real worry is what happens when `out` is non-null; no exception will be logged or reported and *the resource will not be closed*, potentially leading to a variety of serious problems. Variations on this mistake have shown up in a number of software projects, including in Ant 1.6.5 (MAudit.java, line 303).

Another example is shown in Figure 6.6. FindBugs complains about the call to `insertDocument()` on line 249 because its second argument is not allowed to be null (i.e., it is unconditionally dereferenced), but `doc` is guaranteed to be non-null in the branch that calls this method.

A more sophisticated static analyzer might detect that at the one place where `replaceDocument()` is called, the second argument is always non-null, and thus the potential NPD is infeasible. However this observation reveals a potentially

Source: Sun JDK 5 | ...apache.xalan....xsltc.dom.DocumentCache

```
246     private synchronized void replaceDocument(String uri, CachedDocument
            doc) {
247         CachedDocument old = (CachedDocument)_references.get(uri);
248         if (doc == null)
249             insertDocument(uri, doc);
250         else
251             _references.put(uri, doc);
252     }
```

FindBugs: "Method call passes null for non-null parameter on line 249"

Figure 6.6: Mistake in Xalan DocumentCache

more serious problem: if `doc` is never null, then `insertDocument()` is never called. Instead every execution of this method will accesses the `_references` map directly on line 251. This is significant because `insertDocument()` contains logic to cap the size of `_references`, and hence the current implementation could cause it to grow without bound. It turns out that the mistake is that the developer used the wrong variable in the comparison on line 248; it should have been: `if (old == null)`.

So it is useful for static analysis to raise a potential NPD in cases like this where some other subtle bug may lurk. Fortunately in this case, it turns out that the variable `old` is also likely never null because the singular caller of this method ensures that `_references` already contains an entry for `uri`. So the fact that the developer is null-testing the wrong value will have no impact.

Some of these observations, and others we have made in this chapter, seem to fly in the face of recent calls for a move towards "failure-oblivious computing" [124], which seeks to ensure programs never fail. But in fact, one does not need

to prevent exceptions to make a program robust. Exceptions can be caught and logged, allowing the program to continue. Or redundancy can be built into the system, so that failing programs are quickly restarted. This is particularly relevant for server-based applications.

6.5 Summary and Related Work

Through these reviews of potential NPD defects in practice, I have observed that static analysis without careful heuristics may flag many potential NPDs that developers do not want to fix. Often these potential NPDs involve standard API calls that sometimes return null, but usually return a non-null value. If developers choose to enforce logical rules and policies to ensure these API methods are never used in a way that returns null, then there are pitfalls for a static analysis that tries to flag potential NPDs. FindBugs aims to reduce false alarms, and does not flag many of the potential NPDs discussed in this chapter, but some other commercial vendors currently adopt a more aggressive approach. Ultimately, control should be passed to the user to match the aggressiveness of the analysis with the needs of their application.

Meanwhile, API designers can help the cause by making informed decisions about how to specify exceptional return values. Using checked exceptions eliminates most of the concern surrounding potential NPDs, but makes the calling code more verbose, difficult to maintain, and frustrating to write. API designers should endeavor to use empty strings, arrays and containers where possible, or provide

coupling methods which can be used to guard the null-returning API methods.

Through this research, I have also made some qualitative observations about the occurrence of NPEs in practice. NPEs are often manifestations of separate problems and logic errors. It may be that a property has not be set, or a component is incorrectly initialized, or a prerequisite has not been met. And receiving an NPE soon may be preferable to having the incorrect values stored in data structures, only to cause errors further down the pipeline. During my review of NPEs in Ant, I observed that in many cases, the preferred solution was to prevent the null value from reaching the dereference site (by correcting initialization code, for example), but sometimes developers chose to anticipate null values and insert guards to address them.

An open question is whether errors involving NPEs are easy to diagnose and resolve. Many of the cases I reviewed were relatively straight forward, and often the original bug report contained enough contextual information to identify the cause of the problem. In addition, many of the errors followed common patterns such as forgetting to initialize a property, or dereferencing the value returned from an API call that is known to return null. But at the same time, many bugs involved multiple methods, and users may still sometimes benefit from having tools that enable them to track null values to their source.

Several researchers have described analysis techniques for detecting potential NPDs [64, 63, 107, 71, 136], and many commercial analysis tools provide some detectors for this purpose, including the tools evaluated earlier in Section 6.3. A number of researchers have explored techniques to assist developers debugging NPEs. Sinha

et al. combine information from the stack-trace with a static backward data-flow analysis to find the null value assignment [126]. Bond et al. use a dynamic approach to keep track of null values, so that if they are dereferenced, the runtime system can provide information to help developers pinpoint the source of null [26].

Other researchers, focused on creating reliable systems, have explored techniques for preventing NPEs altogether. Dobolyi and Weimer present a system that transforms Java code at compile time by inserting null checks and error-handling code around all potential null dereference sites [38]. They rely on various policies to decide what object to insert in place of null, and allow the program to continue running with limited overhead.

Chapter 7

Cost Effective Static Analysis

So far we have seen that static analysis can find important defects, and users have provided positive feedback about the value of the warnings they receive. At the same time, some warnings are not considered defects by users, some defects have a low impact in practice, and many of the important defects are also captured by good quality assurance practices. Users appreciate static analysis for its educational value, and for finding subtle defects that are otherwise hard to detect. But users have also found that they need to make a nontrivial investment in static analysis to deploy warnings to developers early without impeding their productivity, baseline or triage warnings in old code, integrate the results of multiple tools into a common interface, and filter out unwanted bug patterns. With these benefits and pitfalls in mind, organizations ultimately need to know if using static analysis is cost effective.

It is difficult to measure in absolute terms the cost benefit of static analysis, because many factors affect its utility and the way it is used. I have found it more helpful to ask: *"When is static analysis cost effective?"* This question directs organizations to deploy static analysis into the scenarios and contexts where it is most cost effective first, before expanding usage into other scenarios. For example, some types of defects—including certain security, concurrency and performance defects—are cost effective to find using static analysis, because they are difficult to detect

using other methods. In addition, some applications are much more sensitive to defects than others. For example, a defect in a flight control system on an airplane is far more likely to have calamitous consequences than a defect in a productivity support tool for programmers. Cost effectiveness is also influenced by the infrastructure used to deploy static analysis within an organization, and the practices and policies governing developer activities.

In this chapter, I discuss in more detail how these four factors—the defect's type, the defect's context, the static analysis infrastructure, and best practices—are related to cost effectiveness.

7.1 Cost Effective Defects

Some subtle defects are best found using static analysis, because they are hard to detect using other methods. These include certain security, concurrency and performance bug patterns. Organizations should evaluate the exposure of their applications to these classes of defects when deciding if static analysis is worthwhile. For example, if the application is web-based and accessible to the general public, then it is usually cost effective to review security-related warnings. If the application is running in an embedded environment or is sensitive to timing issues, then performance-related warnings should be reviewed. Similarly, applications running in distributed environments can suffer from pernicious concurrency bugs that occur rarely or are hard to replicate in test environments, and would benefit from having tools point out some of these problems.

These classes of defects may not exhibit the survivor effect we discussed in Section 4.3. In other words, both defects that matter and those that do not may end up in production code, and persist for a long time, because they are missed by other quality assurance methods. The defects may even be causing serious problems in production, and the software team may not be aware of this, or may not be able to debug the problem.

Static analysis is not a panacea for these classes of defects, and much research needs to be done to improve the state of the art for all of them. But organizations who are exposed to any of these classes of defects are usually grateful to have any kind of assistance minimizing their number.

7.1.1 Secure Programming with Static Analysis[1]

Security vulnerabilities include coding mistakes that enable a malicious user to use an application in ways not intended by its developers. Security defects are only a problem if an attacker finds them and exploits them for gain. Hence, a security bug can persist for years without problems, only to be later exploited, with devastating consequences. This is why security defects do not generally exhibit the survivor effect we discussed earlier—any defect flagged by a static analysis tool, including those in old production code, could potentially be very serious.

One of the reasons why security defects are hard to find is that the popular methods for quality assurance—code review, and especially software testing—focus on making sure an application has all required functionality. But security defects

[1]Title taken from book by Chess and West [33]

```
1    void badfunc(char* input) {
2        char buffer[1024];
3        strcpy(buffer, input);
4        ...
5    }
```

Figure 7.1: Buffer overflow vulnerability if `input` is arbitrarily set by user

are not always violations of the requirements, but are sometimes unintended "functionality." As one technology executive puts it: "Reliable software does what it is supposed to do. Secure software does what it is supposed to do, and nothing else" [33]. Static analysis is a valuable aid because it can search for code patterns that are known to be associated with this unintended functionality.

The classic example of a security defect is the buffer overflow vulnerability, illustrated in Figure 7.1. Well informed programmers know that `strcpy` is considered unsafe; this example contains a buffer overflow vulnerability if the value being copied (`input`) can be set by an attacker to any arbitrary value. In this case, since `buffer` is defined on the stack, an attacker could provide a string that overwrites the contents of the stack, including the return address of the calling function. This means that after the function completes, program control will return to whatever address the attacker wishes (e.g., an address inside `buffer`) and start executing the exploit. Buffer overflow exploits are also possible if `buffer` is allocated on the heap.

Static analysis can detect defects like the one in Figure 7.1, and others that result from using "untrusted" input in unsafe ways. Even in memory safe languages like Java, incorrect handling of input vectors provided by a malicious user can lead to security exploits, including SQL injections, and cross-site scripting among other

problems. For example, SQL injection vulnerabilities occur when untrusted input is used (without validation) to construct SQL commands, enabling an attacker to execute commands on the database, and possibly gain access to unauthorized data. Cross-site scripting (XSS) vulnerabilities occur when untrusted data is displayed on a webpage, again without proper validation, enabling an attacker to send compromising scripts to other users.

All these vulnerabilities give malicious users the power to harm an organization in various ways, both loud and subtle. An attacker might find a vulnerability using brute force methods to throw all kinds of inputs at a system to see what happens. An organization might use the same techniques—called penetration testing—to proactively find these vulnerabilities before the attacker does. But the odds are stacked against the organizations because, whereas an attacker need only find one vulnerability to get to work, organizations need to find all of them. Static analysis is appealing because it is exhaustive. In our interviews, one security consultant informed us that his team does not mind weeding through false positives output by a static analysis tool, because some may be associated with potentially exploitable defects.

Securing software is one of the major factors driving the adoption of static analysis tools. Recall the anecdote from Chapter 3 about the user in our interviews who admitted that a past security attack had led his organization—a state department of health—to turn to static analysis. Specifically, static analysis was made mandatory by an external security team as a first layer of defense, causing some frustration among developers, who felt forced to address minor issues to sat-

isfy upper management. Still, the developers found static analysis to be a useful enhancement to their code reviews.

Security concerns have gained prominence over the last decade as more applications are exposed to the network, and many developers are still not educated about writing secure code. As Paul Kurtz, a security expert, said in a recent interview, "The talent coming out of schools right now doesn't have the security knowledge it needs," [141]. Organizations trying to get a handle on the problem are adopting new security processes and frameworks like the Building Security In Maturity Model (BSIMM) [97, 99], Software Security Assurance (SSA) from Fortify Software [44], the Security Development Lifecycle (SDL) from Microsoft [104], and others (discussed in more detail in Section 7.4). Tool vendors are trying to take advantage of this opportunity to automate some of these processes [141]. Static analysis is a big part of this automation, because it helps to educate developers, making them aware of code patterns and practices that are likely to be insecure, so that they can improve their practices, and avoid problems in the future.

7.1.2 Concurrency Defects

Like security defects, concurrency defects can be hard to detect using testing, because the underlying problems often occur only rarely. Researchers have developed a variety of tools and strategies to support testing concurrency. Several frameworks enable users to run large tests thousands of times in an attempt to nondeterministically generate as many interleavings of threads as possible, and

hopefully find any rare interleavings that are defective [6, 39]. Other frameworks enable users to exercise specific interleavings, but use timers to coordinate between the threads, which introduces an unnecessary timing dependency [90, 89, 137, 57].

In an earlier work, I developed MultithreadedTC, a Java framework which enables users to construct deterministic and repeatable unit tests for small concurrent abstractions [119]. This framework uses a clock to coordinate the activities of multiple threads, even in the presence of blocking and timing issues. The clock advances to the next "tick" when all threads are blocked, and test designers can delay operations within a thread until the clock has reached a desired tick. A framework like this is helpful for ensuring that a concurrent abstraction meets its requirements, but does not generally find rare defects that result from unlikely interleavings.

Concurrency bugs are also difficult for static analysis, because some defects result from the unusual and unlikely interplay between different parts of the code, running in different threads and processes. But active research is advancing the state of the art, and producing new static analysis that can detect potential deadlocks [5] and data races [116], mismatched API calls (e.g., lock without unlock), and bad uses of concurrency APIs. And static analysis can search the code exhaustively, finding problems that would be otherwise hard to find.

Static analysis may be especially good at catching bad practices that are likely to affect multi-threaded correctness. FindBugs provides bug detectors to find incorrect or dubious uses of thread-related calls like `Thread.start()`, `Thread.sleep()`, and the `synchronize` keyword. It also flags some instances where static fields are used in ways that are not thread-safe, or when API types that are unsafe for multi-

threaded use (like `Calendar` or `DateFormat`) are used in multi-threaded situations.

The new Java concurrency library (`java.util.concurrent`) introduced in Java 5 is supposed to encourage users to relegate many concurrency management tasks to standard constructs and utilities, and focus on the business logic [55]. But one side-effect is that many users mix up the new APIs with the existing Java constructs, in ways that can be incorrect. For example, we have observed users calling the `wait()` monitor on a `java.util.concurrent.locks.Condition` object, instead of using one of the `await()` methods defined by the `Condition` interface. Some users have also attempted to synchronize on instances of classes in the `java.util.concurrent` package, like `ConcurrentHashMap`. These classes use a different (and incompatible) concurrency control mechanism from other classes, and should not be used with the `synchronize` keyword. FindBugs can detect these infractions.

Also like security warnings, concurrency warnings can be very educational, teaching users the correct way to use API constructs. One interesting defect that results from misunderstanding an API (or neglecting to follow it) is ignoring the return value of the `putIfAbsent()` method in a `ConcurrentHashMap`. `putIfAbsent()` is designed to ensure only one value is associated with a key. So if the key is already in the map, then the value passed to `putIfAbsent()` may not match the value in the map. If the user continues to use the value passed into the map thinking the put operation was successful, then they might be using the incorrect value. Issues like this may be widely misunderstood within an organization, and static analysis can help standardize the practices of different development teams.

7.1.3 Performance Defects

Some performance defects are the side effect of confusing control logic, and as such are difficult to detect through testing or static analysis. Static analysis can helpfully flag confusing or dubious code, which often has little or no impact on program correctness, but may be associated with a performance defect, however minor.

Static analysis can also inform developers of inefficient APIs, the careless use of language features, and inefficient memory use. For example, FindBugs can detect when an `Integer` object is created using its constructor (`new Integer(int)`) instead of the more efficient static factory method (`Integer.valueOf(int)`) which enables caching. FindBugs can also detect when a Java primitive is boxed into its corresponding object, only to be immediately unboxed back to a primitive—this can occur when developers do not understand where boxing and unboxing is occurring.

FindBugs can also highlight obscure language features that lead the unsuspecting developer to write inefficient code. For example, a developer may initialize a static final field with a huge String constant, not realizing that this field will be inlined (copied into the classfile) for any class that references it. Or the developer may use `java.net.URL` instances in a `HashMap` or some other collection, not realizing that `URL.equals()` and `URL.hashCode()` are blocking operations, and can be very inefficient because they connect to the internet to perform domain name resolution.

If performance is an important constraint for the application, then static analysis is a cost effective way to find these problems, because many warnings can be

reviewed and fixed quickly. In many cases, the static analysis may suggest an alternative API that provides identical behavior but increased efficiency. In fact, some situations may call for using static analysis to automatically fix the code, when it can be proven that the modifications do not change the correctness properties of the program. Many automatic performance optimizations have already been built into compilers. Some static analysis tools support automatic fixes [113], and many IDEs use lightweight static analysis to provide automatic fixes at the user's direction [2, 70, 109].

As we have seen, many performance defects are associated with recognizable code patterns that are known to be inefficient, and there are likely to be many patterns that are project-specific or API-specific, and hence not encoded into standard static analysis tools. Organizations have an opportunity to increase the value they get from static analysis by extending tools with custom bug detectors to find these inefficient patterns (more discussion on this in Chapter 8).

7.1.4 Other Subtle Defect Classes

There are other defect classes—including some bad practice and correctness bug patterns—that affect program behavior, but in subtle ways, and hence may not be detected without static analysis. For example, FindBugs detects instances where users repeatedly create new Random objects, using them only once each time. This can lead to low quality random numbers. FindBugs also detects various instances where users compare unrelated types, or query a generic container with an argument

whose type does not match the generic parameter. These checks always return false, allowing the program to continue, but potentially with subtle bugs. Incorrect results may be buried in tables and databases, and escape detection without static analysis.

Another interesting class of problems is internationalization defects, which occur when software does locale dependent operations (like string transformations) without taking into consideration the locale of the user. These defects only affect those applications that expect to run in multiple locales and process international characters. Static analysis is particularly effective at tracking down internationalization defects, and some tools, such as Globalyzer [67], are focused on just this problem.

Other low priority defects—including violations of naming conventions, confusing method names, incorrect capitalization, etc.—are easy to detect with static analysis. Organizations which place a high value on long-term software maintainability should use static analysis to enforce coding standards like these across the organization, and enhance the value of code reviews.

In summary, organizations need to know which types of defects they care about. Some of these defects are best found with static analysis—in some cases, it does not make sense to expend resources to find them any other way. For these defects, it is generally cost effective to use static analysis, though other factors like process and infrastructure (discussed in later sections) can make static analysis even more effective.

7.2 Applications and Contexts

In addition to the type of defect, the nature of the application can also affect the cost effectiveness of reviewing defects. Some applications are mission or safety critical, and very sensitive to any kind of defect. These include airplane control systems, certain medical devices, software on the space shuttle, and so on. Other applications handle sensitive information that needs to be kept private, or support critical infrastructure that needs to be robust against failures. These include the NASDAQ stock exchange, software controllers for the power grid and railway systems, and various government and bank databases. The above applications rely on a variety of substantial, redundant and expensive quality assurance activities including rigorous testing, redundant code reviews, detailed annotations, formal specifications and formal verification. Static analysis can be used to eliminate defects as early as possible, thereby reducing the amount of work needed later on by the more expensive quality assurance activities.

On the other extreme, some applications are insensitive to all but a few correctness defects. These include prototypes that are only intended as a proof of concept, or quick scripts written to perform tasks on the local machine. In these applications, any correctness defects that matter are usually noticed quickly, because the developer is also the user. Of course, short scripts written quickly often grow over time, and become maintenance nightmares. Hence these developers would be well-served by lightweight static analysis tools that do not interfere with their need for speed, especially tools that integrate seamlessly into their workspace or IDE. Some static

Table 7.1: Responses to survey question on use of FindBugs Filters

Use FindBugs Filters	337	32%
Use another process to filter warnings	37	4%
No filtering	458	44%
No response, or other	213	20%

analysis tools have been developed for Python [88], Perl [133], and Ruby [50, 51].

Of course, as we discussed in the previous section, there is often a strong link between the nature of the application, and the type of defects that matter, or do not matter. Static analysis tool builders need to be more aware of this, and provide preset configurations (or a setup wizard) that are based on the application context. Otherwise users have to take the time to filter out bug patterns one at a time. Our experience with FindBugs, as indicated by the survey results in Table 7.1, is that most users do not do any filtering, but run the tool as is out of the box.

Finally, in addition to the standard bug patterns, there are project-specific, or library-specific, or API-specific bug patterns that are only relevant to a small set of applications. Organizations can make static analysis more cost effective by extending tools to find these bug patterns, but they are unlikely to do so unless it is easy to extend tools, as we will see in Chapter 8.

7.3 Developing Effective Infrastructure

Many static analysis tools are initially built with a strong focus on the analysis engine, and little focus on the user interface. Tool creators push to find more warn-

ings, reduce false positives, and improve performance. The user interface is a simple GUI, or a simple plugin for an IDE, or a batch process invoked from the command line, or a web application that allows users to upload files for analysis. But if tool creators want to maximize the return on investment for users, they must provide features that allow tools to fit seamlessly into software development processes, and enable users to easily review warnings and act on them. In addition, many users and organizations find that they have to setup some custom infrastructure to support the nuances of their particular process, or integrate the warnings from multiple tools into a consistent interface. To bring this about, organizations may need to have a *static analysis champion* who is enthusiastic about the tools and promotes their consistent usage.

One of the key challenges is enabling users to run tools automatically. As we observed during our surveys, many users who do not run tools automatically, do not use them regularly or consistently. We have observed many different approaches to this problem, including running the analysis as part of continuous or nightly builds, running the analysis before code check-in or branch merges, or running the analysis in the background of a build and displaying alerts in an IDE or through popup notifications.

Other features enable developers to interact with and manage warnings. When static analysis tools are first run on old code, they often produce thousands of warnings, far more than developers care to handle. Left unchecked, these old warnings can actually drown out newer more relevant warnings. Developers need facilities to establish a baseline, so that old warnings are hidden, and only warnings that occur

after the baseline are visible. Beyond this, developers also need facilities to review or comment on warnings, suppress specific warnings, and to filter out entire classes of warnings that are not considered relevant. Relatedly, tools need to be able to keep track of warnings as the software changes from version to version. This is a non trivial task as the line numbers, and other contextual information can change over time, and a tool needs to remember if a warning has been suppressed or commented on. I will return to the broader challenge of consistency in a moment, in Section 7.3.2.

7.3.1 Advanced Features

Beyond the important features that enable automation and warning management, advanced users also seek to support collaboration between multiple developers, integrate static analysis with other software management tools, and study the historical trends associated with warnings in each project.

Collaborative features are useful when multiple users are responsible for each warning. One user could provide a review indicating that an issue should be fixed, but action on the issue needs to be taken by another user. Collaborative features enable multiple users to share reviews, or filter warnings that have not been reviewed by anyone, and ensure that when one user suppresses a warning, this information is passed on to everyone.

Organizations may prefer to build on existing software management tools such as issues tracking systems and source repositories to enable collaboration. Since or-

ganizations use a wide variety of software management tools, and often customize them for internal use, this integration may need to be developed internally introducing a high initial cost. But once seamless integration between the static analysis interface and these tools is established, users can benefit from static analysis with limited additional cost to use it.

Once organizations have used a tool for a while, it is important to go back and analyze the history of their usage to determine which bug patterns are consistently fixed, and which ones are consistently suppressed. Organizations can use this information to reprioritize defect classes, or modify their filters. Historical data can be captured by analyzing every release or revision of a project, as we do in Chapter 5. Organizations may perform even more fine-grained analysis by instrumenting the developer's desktop to capture information about fixes and suppressions that never make it into a persistent repository. This enables organizations to make inferences about developer habits and determine, for example, if developers are learning new coding best practices from static analysis [18].

7.3.2 The Challenge of Consistency

One challenge for static analysis tools is producing consistent results, meaning that unless there is a good reason for the results to change, the same issues should be reported from run to run, and there should be a clear correspondence between individual issues reported in different runs. A number of factors conspire to make consistency difficult:

- As the context surrounding the warning changes, it is challenging to maintain consistency across different versions of the software artifact.

- As the analysis engine is tweaked and improved, static analysis needs to maintain consistency across different versions of the tool.

- Some static analysis tools perform *effort-limited* analysis to improve performance. This means that each bug detector does not necessarily search exhaustively. Instead they only search inter-procedural paths up to a certain depth, or until a timer expires. This makes it challenging to maintain consistency across different runs of the tool on the same version of the artifact and analyzer. For example, trivial changes in memory layout or timing can change the order in which hash table entries are enumerated, causing inconsistency in what the analysis does.

Since many organizations choose to baseline (or hide) older issues, it is important for tools to clearly identify new issues. Inconsistency could cause some older issues to be marked as new issues. Furthermore, when a user suppresses an issue using any method other than source level suppression, it is important that the analysis does not change the way it identifies warnings. Otherwise previously suppressed issues may resurface and have to be redundantly addressed in the future. Similarly, when users review issues communally, and provide comments, the analysis needs to consistently keep track of the link between this information and the warning, otherwise reviews are lost.

A number of static analysis tools including FindBugs, Fortify SCA, and Coverity Prevent maintain consistency by use variants of a method that computes a hash value for each warning. The hash value depends on some of the context surrounding the warning, but is intended to be invariant and robust to some changes, like line numbers [129].

7.3.3 Enhancements to FindBugs

Early versions of FindBugs supported a command-line mode, and a stand-alone GUI. Over time, plugins have been built to integrate FindBugs into various IDEs, into continuous build servers, and into the Maven and Ant build processes. These enhancements enable users to run FindBugs automatically. FindBugs also has features to enable users to keep track of warnings from version to version of their software, to filter out unwanted bug patterns, and to suppress individual warnings. It supports source level suppression (using annotations), and also provides a filter file format that can be used to suppress individual warnings or groups of warnings.

Advanced users are able to extend FindBugs with new bug detectors, and to do a historical analysis of the fix and suppress trends in their projects. New bug detectors are written using pure Java extensions of the appropriate classes, and extenders have access to many facts from the FindBugs analysis, including information about the type hierarchy, the types of values (from the dataflow analysis), and the sequence of statements (from the control flow analysis). We discuss the process of extending FindBugs in more detail in Section 8.3.1.

To perform a historical analysis, FindBugs provides some batch scripts that operate on its XML database format, and compute information about how many warnings are added or removed after each analysis, which additions/removals are caused by a change to a source file, and which ones are caused by the creation/deletion of files, and how many warnings are active in the latest version. Users can also compute code churn information (similar to that in Figure 5.5), breaking down the fixes for each bug pattern, and comparing this to the overall fix rate.

Additional enhancements have been made to FindBugs, motivated by results from our research, and particularly by the engineering fixits described in Section 3.3. The key enhancement is the introduction of a cloud infrastructure to enable FindBugs to store warnings in a remote database, not just in a local XML database. This enables multiple reviewers to collaboratively access the same warnings, and share reviews. We have also tweaked the way FindBugs ranks warnings, to reflect the fact that many loud warnings are not that important in practice, and subtle warnings may be more pertinent to reviewers (see Section 4.2). The new FindBugs GUI also provides facilities to connect users to various issue tracking systems, so that warnings can be filed as bug reports and assigned to appropriate individuals to be fixed.

7.4 Best Practices and Policies for Cost Effective Static Analysis

The practices and policies organizations put in place when they adopt static analysis affect the return on investment they get from the tools, and whether they

will stick with the tools in the long term. In our surveys (in Chapter 3), we observed that many FindBugs users had not yet implemented formal processes for using static analysis. So understanding what practices work, and how they help, is an important need.

Many of the infrastructure features discussed in the previous section naturally facilitate good practices, even without any specific policies. For example, facilities that run tools automatically and regularly alert developers increase the chance that important issues will be noticed and addressed. In addition, features that enable developers to consistently track warnings and to suppress or baseline some warnings ensure that old warnings do not drown out more pertinent recent (and cheap to fix) warnings. But even with these facilities, effective practices and policies are still needed. For one thing, developers may not feel any external pressure to deal with static analysis warnings, the way they feel when a customer reports a problem [58]. In addition, the absence of clear policies can lead to inefficiencies, such as having multiple developers redundantly review a warning, or failing to bring warnings to the attention of the right person.

There is no one-size-fits-all solution for all organizations, but rather teams employ different policies depending, in part, on some of the context discussed earlier, in Section 7.2. Some teams adopt a zero-tolerance approach that seeks to eliminate all warnings, or blocks code check-in if there are unresolved problems. Others feel this is too heavy-handed, especially if the focus is on getting new features to market in time, and instead focus on controlling defect density. Whatever the context, the consensus is that having well thought out policies is better than an ad hoc approach.

In addition, with all the additional pressures on developers to meet deadlines, or to utilize other quality assurance activities, static analysis can easily be left by the wayside. Hence, it is helpful to have a champion, who encourages tool usage and highlights the successes and return on investment from using static analysis.

In this section, I discuss various best practices currently promoted in the research community, and review the experiences of a number of organizations that have integrated static analysis in various ways, sometimes through trial and error.

7.4.1 A Focus on Security

Much of the research and thinking on practices and policies for using static analysis have come from sources that were focused on security. A number of organizations have made security-focused modifications to general software development processes (including the Waterfall model and Agile development), and most of these modified processes include a significant role for static analysis. These security-focused models include Microsoft's Security Development Lifecycle (SDL) [104], OWASP's Comprehensive Lightweight Application Security Process (CLASP) [110], Gary McGraw's Touchpoints [98], Fortify's Software Security Assurance (SSA) [44], and the Building Security In Maturity Model (BSIMM) [99, 97] created by a group of experts.

Microsoft's SDL includes requirements about how input/output data should be handled, how memory should be managed and other constraints. Within some of these requirements, SDL requires that static analysis be used to detect some of

the vulnerabilities, including cross-site scripting (XSS), memory overflows, banned APIs and other problems [29]. Specifically, SDL recommends several tools produced by Microsoft, including PREfast [83] (also known as the /analyze option in Visual Studio) and the Code Analysis Tool for .NET (CAT.NET) [92] to detect XSS vulnerabilities in managed code projects.

CLASP is a more lightweight process than SDL, with fewer requirements. It identifies some best practices, including the recommendation that teams should "Integrate security analysis into (the) source management process" [117]. CLASP recommends using both static and dynamic analysis to conduct this security analysis, and advocates doing the static analysis automatically by integrating it into the check-in or build processes. In addition, CLASP calls for "using efficient but less accurate technology to avoid most problems early, and deeper analysis on occasional builds to identify more complex problems" [117].

Similarly the other processes cited above all have some role for static analysis. They recognize that it should be used early, and acknowledge that users will have some challenges integrating static analysis into their software development life cycle (SDLC), especially if it is not run automatically.

One helpful resource for developers is a detailed checklist of possible security vulnerabilities, which helps them know what to look for. A report from IBM includes such a detailed checklist, which includes vulnerabilities associated with security-related functions, incorrect input/output validation and encoding, improper error handling or logging, insecure components and coding errors [22]. Static analysis can find many of the problems in such checklists and, in the process, educate developers

about what they look like.

Once concern is how to transition from a process that does not currently focus on security to one that does. A report from Ounce Labs (now IBM) identifies ways to phase in security slowly, and avoid sudden disruptive changes [52]. The report recommends that teams gradually add security to the SDLC by focusing on key projects, identifying a champion, developing coding standards, continuously informing all key stakeholders of progress made, and developing indicative metrics. Another perspective on successful adoption, this time from authors affiliated with Fortify Software, has similar recommendations: start small, address the most severe issues first, appoint a champion, develop metrics and standards/guidelines [32].

The models and processes discussed so far have a horizontal focus, and can be applied to software development in any industry. But some vertical industries also have security standards that motivate the use of static analysis. One key industry is the Payment Cards Industry (PCI), which has a Security Standards Council that issues security requirements and standards for handling private data of payment cards, such as credit and debit cards [115]. The Privacy Rights Clearinghouse reports that over 356 million data records have been exposed as a result of security breaches [118], and application security has been identified as one of the key culprits. Ounce Labs has produced a report detailing how software teams can use static analysis to meet PCI compliance requirements [82]. One of the requirements under vulnerability management is to "develop and maintain secure systems and applications." Specifically, applications should be reviewed for common vulnerabilities. Ounce Labs recommends that to support compliance, a static analysis should

look for coding errors as well as design flaws. Coding vulnerabilities include buffer overflows, race conditions, poor input validation, and other common defect classes. Design flaws include poor access control, weak cryptography, and incorrect error handling and logging. The report concludes that static analysis is the "foundation of a range of potential options available to organizations to monitor the security and compliance state of their applications", and emphasizes that static analysis can find problems earlier than other options, when they are cheapest to fix [82].

7.4.2 Best Practices Identified by Vendors

A number of popular tool vendors have put out white papers that identify best practices, culled from their experiences helping users adopt static analysis [58, 81]. Many of these recommendations focus on integrating security into the SDLC [52, 32], and these were discussed in the previous section. Most of these papers also highlight infrastructure features of their respective tools that simplify the management of the recommended practices and processes. Many of these vendors have a "professional services" group which consists of engineers and consultants who help customers maximize the return on their investment. Specifically, these consultants visit with customers, assist with installation and integration, educate developers, and even build custom features to support specialized customer needs.

Coverity promotes its static analysis offering—*Coverity Prevent*—as a resource for objectively evaluating code, and encourages its customers to create a "Defect Resolution Process" to inspect, prioritize and resolve both old and new warnings

[58]. While the specific implementation of this process is different for each user, there are five general steps required for each implementation. Not surprisingly, Coverity prevent has features to facilitate all the steps below:

1. **Determine Goals and Metrics:** Teams should identify goals for static analysis that are measurable and that align with their broader software development goals. This is where the *context* of the application, discussed earlier, is relevant. Goals can range from resolving all warnings, to establishing thresholds for warnings density (i.e., number of warnings per line), total unresolved warnings, or total uninspected warnings.

2. **Develop a Project Plan:** Based on the goals identified, and on the fact that each of Coverity's warnings take an average of 5 minutes to review and an average of 30 minutes to repair, teams should create a project plan that sets aside some time per week for each developer to address legacy issues, as well as new issues.

3. **Assign Ownership:** Establish a (preferably automatic) process for assigning ownership of each potential defect. For example, ownership may be based on the component in which the warning is found, or the last person to edit the line of code containing the defect. In some cases, automatically assigned ownership will need to be adjusted to a more appropriate owner. Ownership helps create accountability, and teams should measure progress in terms of the number of defects resolved so far.

4. **Notify Owners:** Owners should be notified (via email) daily for new issues, with regular monthly reminders for outstanding issues.

5. **Integrate with SDLC:** Teams should integrate their issue tracking systems with static analysis, so that ownership, notification and other parts of the process can be handled in a way that is familiar to all team members.

Another report, this time from Klocwork, focuses on making static analysis part of an effective peer code review strategy [81]. Klocwork Insight aims to enable collaboration with an interface that supports asynchronous reviews, highlights changes in the code, and displays static analysis results. In the ideal case, static analysis warnings are reviewed and fixed by the original developer, before any peer code review is conducted. But in practice, some issues may be unclear to the developer, or may have unforeseen effects on another developer's code. In these cases, Klocwork's tight integration of static analysis in the peer review process ensures these issues can still be handled seamlessly [81].

7.4.3 Experiences at Google

I have already discussed some of Google's experiences using FindBugs to analyze its Java code base (in Section 3.3.1). Early experiences with FindBugs were mixed. Even though the analysis was run automatically, and warnings were displayed on an internal web interface, the tool received limited actual usage from engineers. Part of the problem was that the analysis and presentation of warnings was outside the normal workflow of developers, and users were not under any pres-

sure to review the warnings. Another limitation was that the system did not capture information about warnings that were fixed, or otherwise removed. Developers were turned off by stale warnings, and questioned the value of the analysis.

The static analysis champions within Google decided to adopt a "service model" through which warnings were centrally reviewed and significant defects were filed in Google's regular bug tracking system. The team reviewed thousands of warnings and filed over 1000 bug reports in a six month period. This effort also enabled them to reprioritize warnings based on the feedback received from developers. They established an internal ranking for bug patterns based on the fix rates and false positive rates they observed.

The service model approach was successful, but did not scale as the size of Google's Java codebase grew. In addition, there was still some skepticism about the overall value of FindBugs. To address these concerns, we organized an engineering fixit, discussed earlier in Section 3.3, in which hundreds of engineers spent 1 or 2 days reviewing thousands of warnings and providing feedback. Over 77% of the reviews contained a fix recommendation, and the feedback was very positive. However, we observed through this process that many defects were mitigated in some way by Google's redundant systems and monitoring practices, or were found in code that had not yet been pushed into production. This reinforced the idea that we need to push FindBugs warnings as early as possible for them to be valuable. While some developers run FindBugs as a plugin in their IDEs, the ultimate goal is to automatically run static analysis in the background and integrate warnings into Google's internal code review system [12]. This approach will allow developers to

discuss warnings, and provide some accountability about fixing them.

These experiences have indicated that one of the challenges for static analysis is demonstrating to developers that it is valuable for them to review warnings which, while definitely mistakes, may not cause software misbehavior.

7.4.4 Experiences at Microsoft

A number of reports have discussed some of the experiences at Microsoft deciding how to integrate static analysis over the last decade [83, 18]. I have also had the opportunity to interview a number of senior engineers and previously reported on their experiences and perspectives [18].

Over the last decade, Microsoft has conducted several research projects, which have produced a wide variety of experimental static analysis tools, each with a different focus. In recent years, the focus has shifted to pushing these tools into the regular software development process of the largest product groups at Microsoft, involving thousands of developers working on tens of millions of lines of code against strict deadlines.

Some of the tools are inter-procedural and rely on heavyweight global static analysis, and hence are too time consuming to be used by every developer. These tools, including PREfix [31] and Global Esp [35], are run periodically centrally, and the defects identified are filed automatically into the defect database of the product.

On the other hand, intra-procedural tools, such as PREfast [83] based plugins, are lightweight and more suitable to be run on the developer's desktop while the

code is being constructed. A wide range of PREfast plugins have been developed for tackling critical problem areas such as security, concurrency, performance, internationalization issues, and device driver issues. These tools typically analyze one function at a time based on function contracts and field invariants specified using Microsoft's source-code annotation language (SAL) [103].

Many of these lightweight tools are enabled by default on the desktop machines of every programmer in the organization, using the Microsoft Auto Code Review (OACR) build infrastructure [102]. OACR integrates static tools into a common and automated build environment which runs the checkers in the background. Developers are notified with a pop up message about the warnings. Warnings are grouped into warning numbers and warning numbers are classified with severity levels. When developers review the warnings, they have the opportunity to fix the code or suppress the warnings.

Another level of quality control is through the "quality gates" that are applied when moving code from one branch to a higher branch (called reverse integration). A class of critical checks form the "minimum bar". Reverse integration is prohibited until all warnings from the minimum bar are fixed. This mechanism ensures that the most serious issues can be caught and fixed early in the development process. For big legacy code bases, adding a new check to the minimum bar may introduce a large number of warnings triggered by pre-existing bugs. When this occurs, a baselining mechanism is used to "mask" these warnings in order to avoid a sudden disruption to the development schedule. Typically these pre-existing bugs are fixed during a concerted cleanup effort at the early stage of a product cycle.

The process in place at Microsoft ensures that serious defects are brought to the attention of developers as soon as possible, and provides some accountability by adding messages to the nightly build, or preventing reverse integration. But the tools used also require developers to extensively annotate their code, a task requiring non trivial effort. I interviewed six senior developers who each have several years of experience using Microsoft's tools, to discover some of the history and challenges associated with using tools and annotations, and to learn their perspectives on static analysis.

All the interviewees felt that using static analysis was worthwhile, though most emphasized the relative importance of code review and testing, recognizing that each quality assurance activity can find different kinds of problems. Users appreciated the exhaustiveness of static analysis, and even reported changing their programming styles to avoid static analysis warnings, leading to more maintainable code. Even with these sentiments, one user still expressed the importance of reducing the "noise" or false positives in tools, saying that tools with less noise are taken more seriously.

Most interviewees reported that they usually resolved all the high priority issues (called errors), and one user working with a security team aimed to fix all potential defects, including low priority warnings. When working on new code, users usually fixed issues just before checking code into the source repository. But many users also worked on code owned by someone else; in this case, they would wait for issues to be flagged by the overnight build and focus on those issues, to minimize changes to someone else's code.

Some users pointed out that close to milestones, the emphasis is usually on

minimizing code changes, so only the most serious issues are fixed. Alternatively, during development cycles dedicated to cleaning up code (often after a major release), teams usually devote resources to wade through lower priority issues and warnings flagged in legacy code.

Obviously the type of warnings that interest users depend on the nature of the code they work on. Many interviewees worked with unmanaged C and C++ code that often included large legacy components. Hence they were most interested in problems related to potential buffer overflows. They also reported that many of the warnings pointed to missing annotations and unused variables.

Users perceived that most issues were worth fixing, though they mentioned that sometimes it was necessary to suppress issues or rewrite the code to make the warnings go away. One user mentioned that this would often happen when code conventions in legacy code did not match the expectations of the tools, and refactoring would be burdensome and potentially error prone. For example, different legacy components may have different conventions for dealing with error states including returning status codes or throwing exceptions.

In general, care was needed to effectively use tools on legacy code. One user reported that anytime a legacy routine was touched, the developer was expected to clean up any old warnings that may be present. But in general, users preferred to address issues in legacy code as part of a dedicated cleanup cycle. Some users credited an "auto-fix" feature in some tools (used to automatically correct some problems) as one property that made the cleanup process feasible. One user cautioned that assigning the task of cleaning up issues in legacy code to junior developers or con-

tractors can sometimes lead to regressions because they are not as familiar with the code. Outside the cleanup cycle, any new legacy issues (i.e., issues found by new or modified static analysis techniques) need to be added to a baseline so developers can focus on problems in new code.

7.5 Summary and Related Work

In this chapter, I have discussed the experiences, and subsequent recommendations, of some organizations and experts who have wrestled with the challenge of using static analysis cost effectively. The specific implementation of best practices varies for each user, depending on the type of defects they care about and the nature of their application. Some security-focused users review all the low impact issues, and prefer to receive as many false positives as possible so they can look for possible vulnerabilities in surrounding code. Some applications are sensitive to any kind of defect, while others are more sensitive to specific subclasses of subtle defects that are hard to detect without static analysis.

Sometimes organizations cannot always reach the ideals recommended in this chapter. For example, one would like to limit costs by having the responsible owner review a defect as early as possible. But sometimes it is not possible to determine who should have this role, or the problem may be too complex for one person to handle alone. Organizations should not allow challenges like this to deter their adoption of static analysis, or cause their processes to devolve into ad hoc use. Clear policies can help prevent inefficiencies, and provide accountability for developers who

do not otherwise feel any external pressure to deal with static analysis warnings.

Other researchers have described their efforts to integrate static analysis into commercial processes, and the feedback they received from developers. 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 [69].

Practitioners from Coverity review defects with customers to encourage them to see the benefits of using static analysis [23]. They observed one interesting outcome from their 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.

Chapter 8

Finding Bugs By Example

Static analysis can be seen as a sophisticated search for code fragments that are thought to be defective. This search is guided by patterns of code components that are deemed to be defective by experts. Many patterns are general, and can occur in any program written in a particular language, or even in programs in multiple languages. But some bug patterns are limited to only a few projects, driven by the idiosyncrasies of the projects, or the software libraries and APIs that are used. These bug patterns are not likely to be included in off-the-shelf static analysis tools, because they do not apply widely. If project teams do not extend tools to include these patterns, then they are only detecting a proportion of bugs that can be detected, and are not retaining the full value of static analysis.

Indeed, some of the value of static analysis is tied to the ability of developers to notice their mistakes, that may be repeated by others within a particular project, and formally capture the offensive code pattern. Otherwise, the effort and frustration that went into debugging the problem may be experienced by others on the team. However, whenever I ask developers if they can think of potential bug patterns, they often cannot. Part of the problem may be that respondents are looking for bug patterns that will apply widely, rather than issues that are only applicable to the four or five developers they work with. Indeed, if the effort to construct a

Source: Gallery (SourceForge.net) | HTTPClient.AuthorizationInfo
(a) fields with different casing requirements

```
110     /** the host (lowercase) */
111     private String host;
115     ...
116     /** the scheme. (e.g. "Basic")
117      * Note: don't lowercase because some buggy servers use a case-sensitive
118      * match */
119     private String scheme;
```

(b) constructor that follows casing scheme

```
193     public AuthorizationInfo(String host, int port, String scheme,
194                              String realm, String cookie)
195     {
196         this.scheme = scheme.trim();
197         this.host = host.trim().toLowerCase();
203         ...
204     }
```

Figure 8.1: Rule informally specified by comments indicates how field contents should be cased

bug detector is too difficult, then it does not make sense to invest in creating one, if it is only likely to be used once or twice. It may be better to send out an email to local team, leave a comment in a conspicuous part of the code, or look out for the problem during a code review.

Figure 8.1 illustrates this with some code fragments from the open source *Gallery* project on SourceForge.net[1]. Here the developer uses comments to indicate that the `scheme` field should not be lowercased, because this could lead to buggy behavior (a). Sure enough, every place fields are initialized, the lowercase method is

[1] http://sourceforge.net/projects/gallery/

applied to the field `host` but not the field `scheme` (b). However, if a new developer joins the team and attempts to update this class, he or she may not notice the comments, and may break the rules. A simple bug detector could be used to flag any inappropriate assignment of a lowercased string to `scheme`, but only if the effort to write such a detector can be justified. In this case, since both fields are private, and are likely only set from constructors, it appears unlikely that a problem could

(a) note on line 388 indicates formatting constraints

```
382     /**
383      * retrieves the field for a given header. The value is parsed as a
384      * date; if this fails it is parsed as a long representing the number
385      * of seconds since 12:00 AM, Jan 1st, 1970. If this also fails an
386      * IllegalArgumentException is thrown.
387      *
388      * <P>Note: When sending dates use Util.httpDate().
394      ...
395      */
396     public Date getHeaderAsDate(String hdr)
397                     throws IOException, IllegalArgumentException
398     {
399         String raw_date = getHeader(hdr);
400         if (raw_date == null) return null;
422         ...
423     }
```

(b) example of rule application: `Util.httpDate()` is used whenever a date field is set

```
627     public void setIfModifiedSince(long time)
628     {
629         super.setIfModifiedSince(time);
630         setRequestProperty("If-Modified-Since", Util.httpDate(new Date(time)));
631     }
```

Figure 8.2: Rule informally specified by comments indicates how the property associated with a parameter (`hdr`) should be formatted

arise in the future.

Another example from the same project is shown in Figure 8.2. In this case, the developer's comment is part of the documentation for a public API method, and it indicates that certain strings representing dates should be formatted using `Util.httpDate()` (a). In an interesting twist, the date strings we are referring to here are actually values associated with a key in a separate property map. This means that callers have to remember to apply this rule when adding values to the property map (b), *not when calling this method*. It would be easy for a new developer to break this rule, especially if they only update the property map, and never call this method, and hence never see this comment. Still, the cost effectiveness of a bug detector to flag potential violations depends on how difficult it is to write the detector, and how often a violation could occur.

So, in summary, organizations are not likely to extend tools to find local problems unless it is simple to do. Indeed in my survey, most users report that they have not extended FindBugs with new bug detectors, as shown in Table 8.1. When asked to comment on their thoughts on custom bug detectors, users indicate that the current process is too complicated. One user writes:

> "It's a killer to write custom detectors—no good documentation is available. Especially not when you come from .NET"

Another user, commenting on whether he has written any custom detectors writes:

> "Not yet, but I'm going to. When/if I find the time."

Table 8.1: Responses to survey question on use of custom bug detectors

Custom bug detectors, released to the public	31	3%
Custom bug detectors, NOT released to the public	39	4%
Custom bug detectors from a third party vendor	13	1%
No custom bug detectors	641	61%
Do not know how to make bug detectors	308	29%
Other, or no response	236	23%

*Respondents can select more than one response

So are existing techniques for extending static analysis easy to use? My review of some of the available methods suggests that they all impose a non-trivial technical burden on users. In particular, they usually require users to learn a new specification language and/or understand the mechanics of the analysis engine, such as control and data flow facts.

To simplify the process of writing a bug detector, let's consider what the developer knows. The developer is familiar with what the bad code fragment looks like. "I'll know it when I see it" could well be the user's refrain. The developer is also familiar with the programming language and the project as a whole. So perhaps we can take advantage of this knowledge, and enable developers to specify bug patterns without needing to know much else.

In this chapter, I explore an approach to specify new bug detectors by providing examples of the bug. I will call these examples "Mock Bugs." Using these mock bugs, an automatic process can then try to infer what the bug detector should look like. The mock bugs should be written in the target language, and the user should not need to learn many conventions or annotations. In addition, the mock

bugs should compile, and may use relevant project or API features. One advantage of these features is that the project team can apply existing software engineering techniques and infrastructure to develop and maintain mock bugs. For example, they can be developed in the IDE without needing special plugins, and refactoring techniques that update the API will also affect the bug detector. However, mock bugs are not intended to be executed.

One useful metaphor is that these bug detectors are like unit tests, except instead of executing them repeatedly, an inference engine pulls them in, and creates a bug detector that is then run on the entire code base, raising alerts when problems are found.

Through this research, I have observed that a user will usually have to provide multiple examples, as well as some counter-examples to reduce false positives and false negatives. All this is highly dependent on the decisions made by the automatic inference engine. One possible feature to enhance the user experience, is to make the development of mock bugs interactive. As the developer produces more examples and counter-examples, the inference engine presents the user with a snapshot of the kinds of problems found by the generated bug detector, so the user can identify potential false positives or false negatives and can add more examples accordingly.

Another consideration is how custom bug detectors should be deployed. Since the bug patterns have a narrow focus – within a single team, for example – the resulting bug detectors may not be appropriate for all projects in an organization, and need to be deployed only for local use. Furthermore, as these bug detectors evolve, different teams may place different requirements on them, and they need to

diverge to serve the various needs.

We discuss these and other considerations in this chapter. The discussion in this chapter will be highly conceptual, and should form the basis for future research. The ideas discussed have not been implemented yet. I start by investigating project-specific and API-specific bug patterns in the wild. These patterns are culled from bug repositories, and best practices surrounding specific APIs, and are discussed in Section 8.2. In Section 8.3, I go over some of the existing methods for writing a bug detector, and compare them with a specification that uses mock bugs. I also discuss related work in Section 8.4. But first, I go into more details about how a user may iteratively use examples to instruct a static analysis tool to find a custom bug.

8.1 Mock Bug Detectors

8.1.1 A Simple Example

Imagine a scenario where a Java developer forgets to assign the return value of the string `trim()` operation back to itself. Since Java strings are immutable, the operation does not modify the string, and is effectively useless. Let's say our developer notices the mistake, and now wants to write a mock bug detector to flag other instances of this mistake. What is the simplest code fragment the developer could write as an example of the bug? Perhaps the following:

```
void bug(String any) {
    any.trim();
}
```

Here we are using the convention that a method prefixed by the name "bug" contains a mock bug (we can call this a *Mock Method)*, and its `String` parameter can refer to any string, not just a string that was passed in as a parameter.

There are several ways an inference engine could interpret this mock bug detector. It could create a detector that flags any call to the `trim()` method, or it might notice that the return value in this case is not used and hence only flag invocations in which the return value is ignored. Let's assume the inference engine chooses the former interpretation; this means that even cases where the return value is correctly assigned back to the string would be flagged as a defect. Our developer will need to provide more examples to refine the mock bug detector. Here's the simplest example to exclude this false positive:

```
void notBug(String any) {
    any = any.trim();
}
```

Here we have introduced another convention: a method prefixed by the name "notBug" contains a *counter-example*, that should *not* be flagged as a bug.

In response to this counter example, the inference engine should refine its output bug detector to flag any call to `trim()`, *except* those that assign the return value back to the string variable. But this refinement may not be sufficient, because the bug detector would still incorrectly flag cases where the returned value is assigned to another string. What we really want to say is that if the return value is used in any way, then this is not a bug. Let's try introducing a new convention that if a value is returned from the mock method, then it is assumed to be used. So we can

```
@MockDetector
class UNUSED_TRIM_DETECTOR {
    void bug(String any) {
        any.trim();
    }
    void notBug(String any) {
        return any.trim();
    }
}
```

Figure 8.3: A mock detector to detect an unused value returned from `String.trim()`

provide another counter example:

```
void notBug2(String any) {
    return any.trim();
}
```

So now, the inference engine flags any call to `trim()`, *except* when the return value is used in any way. This second counter-example, makes the first counter-example redundant. The two mock methods are sufficient for us to specify our mock bug detector, which we do in Figure 8.3.

8.1.2 Benefits and Challenges

This example highlights some of the benefits and challenges of using this approach. One benefit is that the mock detectors are basically examples and counter-examples of the rule, and hence are simple to write. They are also pure compilable Java, and hence can be written in any IDE, deployed in jar files, and refactored

with the rest of the program. However they are never executed; the inference engine reads the examples and constructs the real bug detectors.

One limitation is that some conventions are needed to make the mock methods more expressive, and hence minimize the number of examples that need to be provided. As we design this approach, we have to keep in mind the tradeoff between the number of conventions the user needs to learn, and the expressiveness of each mock method. This tradeoff is heavily influenced by the choices the inference engine makes. There is always more than one way to interpret an example, and sometimes there is no obvious choice. More research in the form of experiments and user studies is needed to decide which choices are best.

Another challenge is that the developer should not have to understand the choices the inference engine will make. Instead, the developer should be able to rely on an iterative workflow in which they provide the simplest example or counter-example, run the detector to identify false positives, and false negatives, and then provide more examples. This actually mirrors the way experts write bug detectors. They often run the detectors against real code bases or sample test cases[2] to ensure that there are not too many false positives or negatives.

Note that we can provide a convenient interface to show the developer all the warnings found, so she can identify any false positives, but we have no way of showing the developer a list of false negatives. She has to figure this out on her own by checking to see if the source she was targeting is included in the list of defects

[2]A convenient repository of test cases for static analysis tools has been compiled by NIST's SAMATE Reference Dataset (SRD) project, and is online at http://samate.nist.gov/SRD/

found. We can aid this effort by keeping track of warnings that disappear as the mock detector is refined; some of these may be false negatives.

Simplifying this interactive user workflow is critical to ensuring that users are willing to use this approach for writing custom bug detectors. Much of our research so far has focused on writing dozens of examples for various bug patterns to determine if this workflow is reasonable.

8.1.3 Generalizing and Specializing

Going back to our working example in Figure 8.3, one limitation is that right now the mock detector is limited to only the `trim()` operation on a string. But really, we want a rule that applies to any method that returns a modified version of the string. We have several choices here:

1. We could require the user to explicitly create examples for all the methods affected. This would probably put too much burden on the user.

2. We could introduce a new convention that would correspond to any method. The obvious choice — `any.anyMethod()` — does not work in a pure Java solution because the *String* class does not have such a method. An alternative is use an annotation, such as: `anyMethod(any)`.

3. We could task the inference engine with figuring out when the developer is using additional examples to generalize or specialize. For example, if the user provides two examples, one for `trim()` and one for `substring()`, the inference engine could conclude that this bug pattern must apply to any operation on a

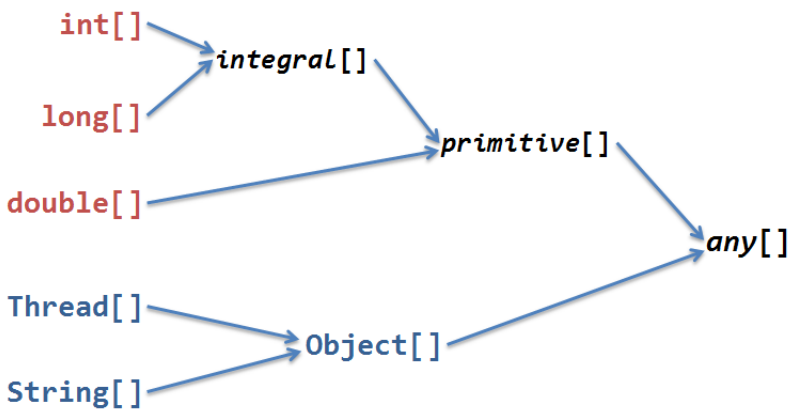

Figure 8.4: A sample lattice for deciding how to generalize or specialize different types

string that returns a value. If the user then provides a counter-example that violates this rule, the inference engine could revert to assuming the rule only applies to the specified methods.

The third option is very attractive, but making it effective would require careful construction of an internal lattice which the inference engine could use to decide how to generalize or specialize. Let's crystallize this with another example rule: "do not call the `hashCode()` method on a Java array". In Java, the `hashCode()` method on an array ignores the length and content of the array when computing a hash value. The recommended approach to construct a hash value for an array `a`, is to call `java.util.Arrays.hashCode(a)`. Our developer may start building a mock detector by writing the following:

```
void bug(int [] any) {
    any.hashCode();
}
```

This will flag any invocation of `hashCode()` on an `int` array. But this produces many false negatives, since we want this rule to apply to other kinds of arrays. Let's say our developer adds another mock method, this time using a `long` array:

```
void bug(long [] any) {
    any.hashCode();
}
```

We can generalize these two examples in several ways. One is to conclude that this rule applies to any array. Another is to decide that an `int[]` and a `long[]` generalize to any *primitive*-typed array, including `boolean[]` and `double[]`. Yet another way is to generalize an `int[]` and a `long[]` as an *integral*-typed array, thereby excluding floating point numbers. These decisions would be driven by a lattice, like the one in Figure 8.4. If we use this model, then our developer would need to add one more example to generalize the rule to any array:

```
void bug(Object [] any) {
    any.hashCode();
}
```

Of course, for this bug pattern, it may make more sense to apply another convention: that a generic type parameter refers to any type. Then we can parameterize our mock detector class with a generic type, `T`, and write just one mock method:

```
void bug(T [] any) {
    any.hashCode();
}
```

Used appropriately, this paradigm of generalizing and specializing the inference may be a powerful way to make mock bugs expressive without needing too many conventions. In my early designs for mock bugs, I found myself constantly adding new conventions until mock bug detectors were beginning to resemble an entirely new specification language. Handing more responsibility over to the inference engine enabled me to focus on writing simple examples for many bug patterns, though it may limit the scope of problems that can be detected using mock bug detectors.

8.1.4 Other Considerations

We need to make some additional considerations to make this approach useful. One is that a good bug detector should include a descriptive message about what the problem is and how it might be fixed. We would have to supply our developer with some convention for providing this message. Another consideration is that since the final bug detector depends on the choices made by the inference engine, we are limited in the ways we can update the engine. We would not want the bug detector to change unexpectedly, long after the developer has written the mock bug detector. Such a change could introduce new false positives or new false negatives. One way to deal with this is to use versioning, so that each mock bug detector is tied to a particular version of the inference engine.

Finally, despite our best efforts to make this as expressive and general as possible, we recognize that some complicated bug patterns can only be written by experts who understand the underlying analysis and/or use a custom specification

language. In the next section, we explore the kind of custom bug patterns that occur in projects, and that can be targeted by this approach. It turns out many of them are quite simple, and an example-based approach with few conventions might provide the right balance of expressiveness and simplicity.

8.2 API-Specific Bug Patterns

8.2.1 Searching for API-Specific Rules

Before establishing a framework for mock bug detectors, we need to characterize API-specific bug patterns. Regular bug detectors can be quite complicated, requiring the analysis to keep track of facts across procedure boundaries. By contrast, the API-specific rules we are targeting are quite simple, and consist of various constraints on how a developer should use an API. The scope of complexity of these bug patterns should drive our design choices, including how many and what conventions are included in the framework.

To help us characterize API-specific bug patterns, we chose to search for real examples in the wild. We searched a variety of resources to find examples, including existing API-specific rules in various static analysis tools, coding standards established by industry organizations, API documentation, and open source code.

Existing API-specific rules in static analysis tools usually target popular and widely used libraries such as the Java APIs (including J2EE and Java Beans), JUnit [1], Ant [45], Apache Commons [46], and popular logging APIs [47]. We are not necessarily expecting that mock bug detectors will be used to target these libraries,

since they are widely used and hence are already supported by many static analysis tools. However, we expect that the properties and considerations which pertain to the rules made for these libraries, will also apply to other more obscure APIs. In addition to the rules built into the static analysis tools, we also searched for third-party bug detectors which are often API-specific. In particular, many third-party bug detectors for FindBugs are collected in an open source project called *fb-contrib* [28].

Another source of custom bug patterns is coding standards from industry organizations like Open Web Application Security Project (OWASP)[3], MITRE[4], and the U.S. National Institute of Standards and Technology (NIST)[5]. These institutions provide best practices, anti-patterns, and reference datasets that focus on security defects.

OWASP develops and maintains the Enterprise Security API (ESAPI), an open source, multi-language, web application security framework that provides a standard set of APIs for developers to use for authentication, session management, access control, input validation and other security-related features. ESAPI comes with detailed secure coding guidelines that include a list of banned Java API methods, and rules about how URLs should be handled, and how data should be protected.

MITRE maintains a detailed dictionary of anti-patterns known as the Common Weakness Enumeration (CWE) [93, 94, 95]. The CWE is a numbered, hierarchical

[3]http://www.owasp.org

[4]http://www.mitre.org/

[5]http://www.nist.gov

index of software flaws that can lead to security vulnerabilities, and it includes some API-specific flaws included a category of weaknesses called "API Abuse" (CWE-227). Many of these weaknesses can be targeted by mock detectors.

NIST sponsors a project (with the U.S. Department of Homeland Security) called Software Assurance Metrics And Tool Evaluation (SAMATE)[6]. Among other activities, this project has produced the SAMATE Reference Dataset (SRD), which is a repository of "test cases" that contain security flaws. The test cases are culled from production code, student code, and synthetic examples. Many of the examples were contributed by commercial static analysis vendors, and they cover a wide range of languages and security flaws.

Another approach for finding API-specific bug patterns is to search API documentation and source code for keywords that may indicate an informally specified rule. Keywords include "required", "must", "follow", "buggy" etc. The examples presented earlier in this chapter in Figures 8.1 and 8.2 were found by using Google Code Search[7] with some of these keywords.

8.2.2 Characterizing API-Specific Rules

After reviewing dozens of candidates for API-specific rules, we sought to construct a classification that can be helpful when making design decisions for the mock framework. We found it instructive to characterize each rule based on the constraints it places on what the developer can code. Using this paradigm, we identified five

[6]http://samate.nist.gov/

[7]http://www.google.com/codesearch

popular constraints—occurrence constraints, ordering constraints, type constraints, value constraints, and usage constraints—that apply to many rules. Often a rule may be composed of more than one of these constraints. Additional, less frequently used constraints include naming constraints and cardinality constraints.

Occurrence constraints basically state that in a specified context, a given pattern must occur, or is prohibited. Given the metaphor of mock bugs, a user can specify the *must-occur* constraint by writing a *bug-method* that does not contain the desired pattern in the appropriate context, and writing a *notBug-method* that does contain it. The opposite arrangement would be used for the prohibited constraint. An example of this constraint was the earlier example of rules to prohibit calling `hashCode()` on an array, discussed in Section 8.1.3. Another example is rules that require overriding methods to call the corresponding super class method.

A more complicated example is found in the documentation of the `saveState()` method for the `StateHolder` interface in the JavaServer Faces Specification[8]:

> If the class that implements this interface has references to instances that implement `StateHolder` (such as a `UIComponent` with event handlers, validators, etc.) this method must call the `saveState(...)` method on all those instances as well. *This method must not save the state of children and facets.* That is done via the `StateManager`.

This specification requires that implementing methods *must* call the corresponding `saveState()` methods on any other state holders referenced by the class,

[8]http://java.sun.com/javaee/javaserverfaces/

except those designated as children or facets. Already with this example, we notice that a mock method will not be enough for the user to fully specify the context. Users will need several mock classes—they could be anonymous implementations of the `StateHolder` interface—to provide examples of `saveState()` methods that follow or violate the specification, so that the inference engine can construct an appropriate bug detector. We also notice that it is important for users to use the same context for each example, only changing the lines that relate to the pattern or anti-pattern. The inference engine can then focus on the differences when deciding where to raise alerts.

Many examples of the occurrence constraint are simple API bans. For example, some Java projects ban calls to `System.out.println()`, and the OWASP ESAPI project has a list of Java API methods that should never be used, but should be replaced with methods in its API [49]. Given the prevalence of this rule template, it may be beneficial to provide a shorthand or convention for specifying banned APIs.

Ordering constraints state the relationship between two or more patterns. This could include simple sequencing relationships (pattern-A must follow/precede pattern-B), or more complicated domination relationships (e.g., pattern-B must be guarded by pattern-A).

Type constraints place some requirement or restriction on the types used in a specified context. An example is the rule that loggers should be created with a type parameter that is restricted to the class that they belong to. Another example is rules that require a group of classes to implement the *Serializable* interface.

Value constraints place some requirement or restriction on the range or nature

(a) Rule: No hard-coded passwords

```
1    void bug() {
2        ESAPI.authenticator.createUser(username, "password", "password");
3    }
4    void notBug(String password) {
5        ESAPI.authenticator.createUser(username, password, password);
6    }
```

(b) Rule: logHTTPRequest() should ignore any parameters named "password"

```
1    void bug(Request any, Logger anyLogger) {
2        ESAPI.httpUtilities().logHTTPRequest(any, anyLogger, null);
3    }
4    void notBug(Request any, Logger anyLogger) {
5        String[] ignore = { "password" }
6        ESAPI.httpUtilities().logHTTPRequest(any, anyLogger, Arrays.asList(ignore));
7    }
```

Figure 8.5: ESAPI Rules that have value constraints

of a value. This include rules requiring numeric values to be within a specified range, or string values to match a specified regular expression. Philosophically, one would expect such rules to be enforced by the application, but in some specialized or legacy cases, it may be reasonable to use a mock bug detector. Figure 8.5 illustrates some examples from OWASP's ESAPI. The first rule specifies that the `createUser()` method should never be called with a string literal. The second rule specifies that the `logHTTPRequest()` should always be called with a filter to exclude the "password" field. This second example suggests the need for parameterized mock bug detectors, so that other users could use this rule even if their password field has a different name, by just providing the name of the password field.

These examples also further illustrate how the behavior of the mock bug detec-

tor will depend on the decisions of the inference engine. If we design the inference algorithm to interpret literal strings as "any literal string", then rule (b) would need to be extended with additional examples to clarify that we only mean the literal string "password". Alternatively, if we design the inference algorithm to only match the value of the literal string specified, then rule (a) would need to be extended to clarify that we mean any literal string. These choices in our design of the inference engine should be driven by empirical and user studies.

Usage constraints indicate if, and how a value is used. The `String.trim()` examples presented earlier in this chapter (in Figure 8.3) are an example of usage constraints. As we saw in those examples, there are many ways a value might be used, and often the usage constraint is that it be used in *any* way. Hence it seems reasonable to introduce some convention for any usage. Earlier we used a return-value convention, but this would not be sufficient if the usage constraints are to apply to multiple values. Other options include returning a collection of values that are used, or annotating the values.

Naming constraints simply limit the names that can be given to classes, methods, or variables, usually to match some preferred style. *Cardinality constraints* are needed for a few rules that limit the number of occurrences of certain instances or types. For example, PMD[9] has a rule that each class should only define one logger.

Finally, looking through the constraints we have identified, it seems clear that mock bugs are not the most effective way to express some of them. For example, the

[9]PMD is a popular static analysis tool for Java that has many lightweight rules, and is online at http://pmd.sourceforge.net/

banned APIs mentioned in the discussion of occurrence constraints would probably be best supported by an XML or form-based interface, through which a user can simple enter the fully qualified names of all banned methods. Naming constraints could be specified using annotations with regular expressions, or using XPath-like queries to target a group of classes, as is done in PMD (see discussion in Section 8.3.2). But despite the convenience of some of these approaches, mock bugs appear to be the right combination of simplicity and generality for a wide range of defects. In the next section, we compare mock bugs with other existing approaches for specifying custom bug detectors.

8.3 Writing a Bug Detector

Existing approaches for writing bug detectors generally aim to give users almost as much power as the experts that implement the core bug detectors. In fact, most static analysis tools feature core rules that are written using the same specification framework that users can use to write additional rules. This paradigm is handy for expert users who specialize in extending tools, but does not appeal to our target audience: regular developers writing simple rules. In particular, this paradigm requires the user to understand the underlying analysis and/or learn a new specification language. In this section, we discuss a representative set of sample rules from various tools. This is by no means an exhaustive list of all tools, but aims to cover the range of styles available.

8.3.1 FindBugs Bug Detectors

FindBugs bug detectors are implemented in Java using the visitor pattern [41] to walk through a bytecode classfile looking for matching patterns. A user can provide custom bug detectors by constructing a class that implements the `Detector` interface and adding the resulting jar file to a plugin directory. The user will also need to provide some XML configuration files so that FindBugs can find the new detector.

Figure 8.6 illustrates the basic scaffolding code needed by a simple FindBugs bug detector. Each detector is initialized using a `BugReporter` (line 3) which is later used to generate warnings (line 10). Users override the `visitClassContext()` method to search for the desired pattern. From this context, users can examine the classfile's metadata for its methods, fields and other characteristics, or access the dataflow analysis, or traverse the abstract syntax tree to get to deeper contexts. Since many detectors have similar setups, FindBugs provides a wide variety of base classes that can help simplify the user's task.

This plugin infrastructure has some advantages. Since the plugins are written in Java, users do not need to learn a new language, and many developers are familiar with the visitor pattern that serves as the basic metaphor for writing a plugin. In addition, the user has all the power that the experts have, since all bug detectors are written this way[10].

However this plugin infrastructure is difficult for casual users to learn. For

[10]Of course, some bug detectors inspire the experts to modify the internals of FindBugs, and custom detector developers may not be at liberty to make such modifications.

```
1  public class BasicDetector implements Detector {
2      BugReporter bugReporter;
3      public BasicDetector(BugReporter bugReporter) {
4          this.bugReporter = bugReporter;
5      }
6      /** Visit the ClassContext for a class which should be analyzed. */
7      @Override public void visitClassContext(ClassContext classContext) {
8          // use 'classContext' to access metadata and search for patterns
9          // if a match is found ...
10         bugReporter.reportBug(
11             new BugInstance("BUG_NAME", Priorities.HIGH_PRIORITY)
12                 .addClass(classContext.getJavaClass())
13                 .addString("<additional info>")
14         );
15     }
16     /** Invoked after all classes have been visited. Used by any detectors which
17      * accumulate information over all visited classes to generate results. */
18     @Override public void report() { }
19 }
```

Figure 8.6: A Basic Bug Detector for FindBugs

one thing, users are searching for patterns in the byte code, not in source code. Hence users may need to inspect the byte code to know what the reference pattern they are searching for actually looks like. In addition, users need to understand any analysis facts they intend to use, and provide lots of scaffolding code, some of which is shown in Figure 8.6. Indeed, FindBugs detectors can become quite verbose. Finally, users have to manually configure the detector, editing XML files to provide a name, description, and other metadata about the detector.

8.3.2 XPath Queries for PMD

Like FindBugs, PMD also allows users to write new rules using Java code and the visitor pattern to navigate over the abstract syntax tree (AST) representation of a source file. But PMD targets many rules that are simple enough to be specified by searching for the relevant node in the AST, and performing some simple checks. This is particularly true of rules that enforce a coding style, or naming convention. For example, one rule is that the package name should not contain uppercase letters. A Java implementation of this rule would simply walk down the AST until a package declaration is found, and generate a warning if it does not use the correct casing. Since the AST is a structured document, an analogy can be made with searching an XML document to find key nodes or attributes. XPath is a powerful language for searching XML documents [68], and it makes sense to try and apply it to the rules in PMD.

PMD allows rule writers to treat the AST as an XML document, and search for nodes using XPath. For example, the package-name casing rule above could be specified using the following XPath query:

```
//PackageDeclaration/Name[lower-case(@Image)!=@Image]
```

The only nodes which match this query are package declarations where the name has some uppercase letters. With this rule, such nodes will be flagged and a warning will be generated. The clear advantage of this approach is that it can be very concise for simple rules. However, XPath queries can get very complex for more difficult rules, making them hard to understand or maintain. In addition, rule

writers need to understand the AST structure, including metadata such as node and attribute names. The AST structure is not obvious from looking at the source code, because the parser may insert many redundant nodes. Recognizing this difficulty, the PMD team has provided a Rule Designer which enables rule writers to visualize the AST, and experiment with different XPath queries.

8.3.3 The Metal Language

Metal is a high-level, state-machine language used to write compiler extensions that can be used to check software for rule violations [40, 19]. It is a predecessor to *Coverity Extend*, Coverity's software development kit for writing custom bug detectors. It allows rule writers to combine patterns written in an extended version of the target language, with a state-machine whose transitions can be used to specify whether a rule is matched or not.

Figure 8.7 illustrates a state-machine (taken from [19]) that checks to see if the return value from a string method is used. In this case, the target language is Java and the pattern language is an extension of Java. The rule specified is that, since Java strings are immutable, it does not make sense to ignore the return value of a string operation. The example starts by declaring and initializing some meta-variables on lines 2-4: `str` will represent the return value and is used in multiple states; `tracking` is a hash map used to temporarily store values that have been returned, but not yet used. The state-machine contains two states defined on lines 5 and 10. Starting from the `start` state, whenever the pattern

```
1   sm stringchecker {
2       state decl { java.lang.String } str;
3       { public HashMap tracking = new HashMap(); }
4       init { tracking = new HashMap(); }
5   start:
6       { str = java.lang.String.anymethod(...) }
7       ==> str.tracked, {
8           tracking.put(str.getDefinition(), ...);
9       };
10  str.tracked:
11      { str } ==> { // matches any use
12          tracking.remove(str.getDefinition());
13      };
14      final { bugs.addAll(tracking.values()); }
15  }
```

Figure 8.7: A Metal-style rule for tracking unused values returned from String operations

`java.lang.String.anymethod(...)` is matched, the state-machine updates the `tracking` hash map, and transitions to the next state `str.tracked` (lines 6-8). In this state, any usage of the return value will result in the meta-information being removed from the `tracking` hash map (lines 11-12). As a final action, any values left in the `tracking` hash map are flagged as bugs on line 14, because they are never used.

The Metal approach is very elegant and powerful, and many core rule checkers have been written, primarily for C and C++. But this approach requires rule writers to learn two specifications: the state-machine syntax and semantics, and the pattern-matching language which is an extension of the target language.

8.3.4 Comparing to Mock Bugs

The frameworks presented in this section have many strengths and weaknesses, and generally have to tradeoff between power and ease of use. Mock bugs aim to emphasize ease of use, but may also be quite powerful because they rely on the rule writer being able to express the rule using a few examples. In addition, mock bugs aim to enable users to write rules without having to learn a custom specification language or understand the underlying analysis.

8.4 Summary and Related Work

A framework for specifying static analysis rules by providing examples and counter-examples may encourage more organizations and users to write more project-specific rules and increase the value they get out of static analysis. In this discussion of such a conceptual framework, my main design goals have been simplicity and ease of use, with few requirements on the rule writer to learn a new language, or understand the internals of the analysis engine. But there are limitations because examples may not capture all the intent a rule writer wants to express, or because some rules may require too many examples to be sufficiently unambiguous. Still, the examples I have explored indicate that this approach has some promise.

It remains to be seen whether mock bugs will be both convenient enough, and powerful enough to be useful. Practical implementations and subsequent user studies are needed to indicate if this paradigm puts too much burden on users to provide many examples, or if the lack of conventions makes this approach too constraining

for the rules users want to specify.

There is actually some earlier work on using code patterns, or "examples" to search for corresponding code patterns. Paul and Prakash [114] introduce a pattern description language for searching for code patterns, but use additional syntax to extend the target language to make their search more expressive. Their target audience is software maintainers who need to find code fragments that exhibit certain properties. Devanbu later comments on this work by pointing out that this kind of search can be performed by traversing abstract syntax trees (ASTs) [36], as is done by many modern static analysis tools. Indeed, traversing the AST may be the preferred mode for general search tasks, but for our goal of matching simple defective fragments, code patterns are more effective. The pattern description language presented by Paul and Prakash was later extended by Matsumura et al. to search for "bug code patterns" [96]. They provide new syntax for specifying the absence of a statement, and other constructs that enable them to search for implicit code rules, which I call project-specific rules.

A number of researchers have focused on the problem of finding and flagging project-specific rules [131, 139, 140, 135, 134]. Spinellis and Louridas provide a method for augmenting APIs so that static analysis can later verify that invocations of the API methods use them correctly [131]. They wrote some augmentations for some Java API classes, extended FindBugs to understand their augmentations, and searched large code bases finding hundreds of potential defects. Other researchers have attempted to automatically discover custom bug patterns in various projects by mining the history in their source repositories. Williams and Hollingsworth search

for specific patterns of function invocations, such as *"called after"* or *"conditionally called after"*, to come up with system-specific rules that can then be used to search for defects [139, 140]. Thummalapenta and Xie use data mining techniques such as association rules to find exception handling rules [135] and rules that involve alternative patterns (that may be substituted for each other in practice) [134].

Finally, there are numerous specialized languages for specifying bug patterns from research community [121], as well as from commercial enterprizes like Coverity [40, 56, 142, 10], Microsoft [21, 83, 20, 31], and IBM [123, 122].

Chapter 9

Conclusion

Through this research, using numerous studies, I have explored the nuances of static analysis in practice. On the one hand, static analysis is clearly useful for finding defects early, and vital for subtle subclasses of defects that are otherwise hard to detect. In addition, users laud its educational value, and its ability to exhaustively reach into the dark corners of the code and shed light on rare bugs. On the other hand, static analysis carries some costs that need to be understood and managed. Spurious warnings and low impact defects may eventually lead many to abandon tool adoption, unless users create effective strategies to run tools automatically, alert the right people early, deal with issues in legacy code, and other challenges.

Researchers and tool vendors should continue to focus on finding more bugs, more accurately, more quickly. But it is also clear from my research that other equally important factors determine whether static analysis tools are used successfully, and these factors can be understood by studying the practice of real users dealing with real defects. This kind of research exposes the varying expectations and contexts of different users. For example, some users want tools to output many warnings, including false positives, so they can investigate the surrounding code for security vulnerabilities, while many other users decline to fix true defects that do not impact software behavior in practice. This research also enables vendors to factor

in the practices of users when making tweaks to their heuristics, such as deciding how loud and silent warnings should be ranked, and whether a potential null-pointer dereference is a defect.

Our motivation to conduct this research comes from wondering why static analysis is not used more. If it can find serious defects, surely everyone will want to adopt it. Part of the limitation may be that many do not realize the benefits of static analysis. But a big part is the perception that the effort users spend on static analysis does not payoff. These users judge warnings in isolation, and often do not run the analysis until the code is moderately mature. These users need to understand that the value comes from running it early, and utilizing infrastructure that makes it easy to use.

In summary, static analysis is well received by the users we studied, and many have begun taking steps to introduce infrastructure and practices that will maximize their return on investment. Many tools have also started adding features to enable software teams to integrate static analysis with existing issue tracking databases and code repositories, and to collaboratively resolve issues. As tools continue to develop, they will become better at figuring out when a defect really matters, and alerting developers using convenient modes. Feature rich tools will make it easier for organizations to introduce static analysis into their process, integrate them with existing software tools and other static analysis tools, and extend them with project-specific analysis.

Bibliography

[1] Junit testing framework. http://www.junit.org, 2007.

[2] Eclipse. http://eclipse.org/, 2009.

[3] Jlint. http://jlint.sourceforge.net/, 2010.

[4] PMD. http://pmd.sourceforge.net/, 2010.

[5] R. Agarwal, L. Wang, and S. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. *Hardware and Software, Verification and Testing*, pages 191–207, 2006.

[6] Matt Albrecht. Using Multi-Threaded Tests. http://groboutils.sourceforge.net/testing-junit/using_mtt.html, September 2004.

[7] Marty Alchin. Returning none is evil. http://martyalchin.com/2007/nov/20/returning-none-is-evil/, November 2007.

[8] Alden Almagro, Paul Julius, and Jeffrey Fredrick. CruiseControl. http://cruisecontrol.sourceforge.net/, 2010.

[9] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël 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.

[10] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 143, Washington, DC, USA, 2002. IEEE Computer Society.

[11] Lerina Aversano, Luigi Cerulo, and Concettina 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.

[12] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, 2008.

[13] Nathaniel Ayewah and William 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.

[14] Nathaniel Ayewah and William Pugh. Learning from defect removals. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 179–182, Washington, DC, USA, 2009. IEEE Computer Society.

[15] Nathaniel Ayewah and William 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.

[16] Nathaniel Ayewah and William Pugh. Null dereference analysis in practice. In *PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, USA, 2010. ACM.

[17] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian 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.

[18] Nathaniel Ayewah, Yue Yang, and David Sielaff. Instrumenting Static Analysis Tools on the Desktop. Technical Report MSR-TR-2010-17, Microsoft Research, February 2010.

[19] Godmar Back and Dawson Engler. Mj - a system for constructing bug-finding analyses for java, 2004. Available online (10 pages) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.8707.

[20] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[21] Thomas Ball and Sriram K. Rajamani. Slic: A specification language for interface checking (of c). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[22] Ryan Berg. The Path to a Secure Application. Technical Report RAW14198-USEN-01, IBM Software, December 2009. Available online (16 pages) http://www.ouncelabs.com/resources/112-the_path_to_a_secure_application.

[23] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson 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.

[24] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, New York, NY, USA, 2008. ACM.

[25] Joshua Bloch. *Effective Java*. Addison Wesley, 2 edition, 2008.

[26] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. *SIGPLAN Not.*, 42(10):405–422, 2007.

[27] Sarah Boslaugh and Dr. Paul A. Watters. *Statistics in a nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2008.

[28] Dave Brosius. fb-contrib: A findbugs auxiliary detector plugin. http://fb-contrib.sourceforge.net, September 2005.

[29] Bryan Sullivan and Michael Howard. The Microsoft SDL and the CWE/SANS Top 25. http://blogs.msdn.com/b/sdl/archive/2009/01/27/sdl-and-the-cwe-sans-top-25.aspx, January 2009.

[30] Raymond P.L. Buse and Westley 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.

[31] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[32] Pravir Chandra, Brian Chess, and John Steven. Putting the tools to work: How to succeed with source code analysis. *IEEE Security and Privacy*, 4(3):80–83, 2006.

[33] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, 1 pap/cdr edition, July 2007.

[34] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. Begbunch: benchmarking for c bug detection tools. In *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pages 16–20, New York, NY, USA, 2009. ACM.

[35] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, New York, NY, USA, 2002. ACM.

[36] Prem Devanbu. On 'a framework for source code search using program patterns'. *IEEE Trans. Softw. Eng.*, 21(12):1009–1010, 1995.

[37] Mark Dixon. Static analysis: false positives and false negatives. http://www.enerjy.com/blog/?p=144, November 2007.

[38] Kinga Dobolyi and Westley Weimer. Changing java's semantics for handling null pointer exceptions. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 47–56, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[40] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[41] Erich Gamma (et al.). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[42] Kent Beck et al. Manifesto for Agile Software Development. http://www.agilemanifesto.org/, 2001.

[43] William A. Florac. Software Quality Measurement: A Framework for Counting Problems and Defects. Technical Report CMU/SEI-92-TR-22, Carnegie Mellon University, September 1992.

[44] Fortify Software. Introducing Software Security Assurance. http://www.fortify.com/company-partners/ssa.jsp, 2010.

[45] Apache Software Foundation. Apache Ant. http://ant.apache.org/, 2010.

[46] Apache Software Foundation. Apache Commons. http://commons.apache.org/, 2010.

[47] Apache Software Foundation. Apache log4j. http://logging.apache.org/log4j/, 2010.

[48] Eclipse Foundation. Eclipse Test and Performance Tools Platform Project. http://www.eclipse.org/tptp/, 2010.

[49] OWASP Foundation. ESAPI Secure Coding Guideline. http://www.owasp.org/index.php/ESAPI_Secure_Coding_Guideline, 2010.

[50] Michael Furr. Diamondback Ruby. http://www.cs.umd.edu/projects/PL/druby/, 2010.

[51] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300, New York, NY, USA, 2009. ACM.

[52] Anthony Gerkis and Jack Danahy. Software Security Governance in the Development Lifecycle. Technical Report Doc.20071001-1.0, Ounce Labs, 2007.

[53] GrammaTech. CodeSonar. http://www.grammatech.com/products/codesonar, 2010.

[54] Rick Grehan. Jtest treks to code-testing supremacy. *JavaWorld*, October 2006.

[55] JSR 166 Expert Group. JSR 166: Concurrency Utilities. http://jcp.org/en/jsr/detail?id=166, September 2004.

[56] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM.

[57] Per Brinch Hansen. Reproducible testing of monitor. *Softw., Pract. Exper.*, 8(6):721–729, 1978.

[58] Matthew Hayward. Effective Management of Static Analysis Vulnerabilities and Defects. White paper, Coverity, 2010. Available online (20 pages) http://www.coverity.com/library/pdf/Coverity-Effective-Management-of-Static-Analysis-Defects.pdf.

[59] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50, New York, NY, USA, 2008. ACM.

[60] J. Highsmith and A. Cockburn. Agile software development: the business of innovation. *Computer*, 34(9):120–127, 2001.

[61] Tony Hoare. Null references: The billion dollar mistake. http://qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake, March 2009.

[62] David Hovemeyer and William 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.

[63] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14, New York, NY, USA, 2007. ACM.

[64] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, New York, NY, USA, 2005. ACM Press.

[65] IBM. Ounce. http://www.ouncelabs.com/, 2010.

[66] Coverity Inc. Coverity Prevent. White paper, Coverity, 2008. Available online (4 pages) http://www.coverity.com/library/pdf/coverity_prevent.pdf.

[67] Lingoport Inc. Globalyzer Software. http://lingoport.com/globalyzer, 2010.

[68] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/, November 1999.

[69] Ciera Jaspan, I-Chin Chen, and Anoop 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.

[70] JetBrains. IntelliJ IDEA. http://www.jetbrains.com/idea/, 2010.

[71] Xiaoping Jia, Sushant Sawant, Jiangyu Zhou, and Sotiris Skevoulis. Applying static analysis for detecting null pointers in java programs. Technical report, DePaul University, 1999.

[72] Philip M. Johnson. Requirement and design trade-offs in hackystat: An in-process software engineering measurement and analysis system. In *ESEM '07 Proceedings*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[73] Philip M. Johnson, Hongbing Kou, Michael Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Improving software development management through software project telemetry. *IEEE Softw.*, 22(4):76–85, 2005.

[74] P.M. Johnson, Hongbing Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, Shenyan Zhen, and W.E.J. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 641–646, 2003.

[75] Leander Kahney. Your car: The next net appliance. http://www.wired.com/science/discoveries/news/2001/03/42104, March 2001.

[76] Kohsuke Kawaguchi. Hudson: Extensible continuous integration server. http://hudson-ci.org/, 2010.

[77] Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path projection for user-centered static analysis tools. In *PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 57–63, New York, NY, USA, 2008. ACM.

[78] Sunghun Kim and Michael 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.

[79] Sunghun Kim and Michael 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.

[80] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. 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.

[81] Klocwork. Modernizing the Peer Code Review Process. White paper, 2010. Available online (7 pages) http://www.klocwork.com/resources/white-paper/code-review.

[82] Ounce Labs. Meeting the PCI Application Security Requirements. White paper, 2010. Available online (10 pages) http://www.ouncelabs.com/writable//resources/file/pci_appsecurity_compliance.pdf.

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

[84] Lucas Layman, Laurie Williams, and Robert St. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 176–185, 2007.

[85] Robert Lemos. Study: Flaw disclosure hurts software maker's stock. *SecurityFocus*, June 2005. Available online (1 page) http://www.securityfocus.com/news/11197.

[86] Nancy G. Leveson and Clark Savage Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[87] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[88] Logilab.org. Pylint. http://www.logilab.org/857, 2010.

[89] Brad Long. *Testing Concurrent Java Components*. PhD thesis, The University of Queensland, July 2005.

[90] Brad Long, Dan Hoffman, and Paul Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6), 2003.

[91] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[92] Mark Curphey. Anti-XSS 3.0 Beta and CAT.NET Community Technology Preview now Live! http://blogs.msdn.com/b/cisg/archive/2008/12/15/anti-xss-3-0-beta-and-cat-net-community-technology-preview-now-live.aspx, December 2008.

[93] R.A. Martin, S.M. Christey, and J. Jarzombek. The case for common flaw enumeration. In *NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics*. SAMATE, NIST, 2005.

[94] Robert A. Martin and Sean Barnum. Common Weakness Enumeration (CWE) Status Update. *Ada Lett.*, XXVIII(1):88–91, 2008.

[95] Robert A. Martin and Sean Barnum. Creating the secure software testing target list. In *CSIIRW '08: Proceedings of the 4th annual workshop on Cyber security and information intelligence research*, pages 1–2, New York, NY, USA, 2008. ACM.

[96] Tomoko Matsumura, Akito Monden, and Ken-ichi Matsumoto. A method for detecting faulty code violating implicit coding rules. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 15–21, New York, NY, USA, 2002. ACM.

[97] Brian Chess Gary McGraw and Sammy Migues. The Building Security In Maturity Model. http://bsimm2.com/, 2010.

[98] Gary McGraw. *Software security: building security in*. Addison-Wesley, February 2006.

[99] Gary McGraw, Brian Chess, Sammy Migues, and Elizabeth Nichols. Software [in]security: Bsimm2. *informIT*, May 2010. Available online (1 page) http://www.informit.com/articles/article.aspx?p=1592389.

[100] Bharat Mediratta and Julie Bick. The google way: Give engineers room. *The New York Times*, Oct 2007.

[101] Microsoft. Software Quality Metrics, July 2005. http://www.microsoft.com/windowsvista/privacy/privacy_b1.mspx#EAFAC.

[102] Microsoft. Microsoft Auto Code Review (OACR). MSDN Library, October 2009. http://msdn.microsoft.com/en-us/library/dd445214.aspx.

[103] Microsoft. SAL Annotations. MSDN Library, July 2009. http://msdn.microsoft.com/en-us/library/ms235402.aspx.

[104] Microsoft Corp. Microsoft Security Development Lifecycle. http://www.microsoft.com/security/sdl/, 2010.

[105] L. E. Moser and P. M. Melliar-Smith. Formal verification of safety-critical systems. *Softw. Pract. Exper.*, 20(9):799–811, 1990.

[106] Nachiappan Nagappan and Thomas 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.

[107] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for Java. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society.

[108] Haya El Nasser and Paul Overberg. Census software plagued by defects. http://www.usatoday.com/news/nation/census/2010-02-17-Census-software_N.htm, March 2010.

[109] NetBeans. NetBeans Jackpot. http://wiki.netbeans.org/Jackpot, 2010.

[110] OWASP. *Comprehensive, Lightweight Application Security Process v1.2*. The Open Web Application Security Project, 2007. http://www.owasp.org.

[111] Andy Palmer. Returning null considered dishonest. http://andyp-tw.blogspot.com/2008/08/returning-null-considered-dishonest.html, August 2008.

[112] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, New York, NY, USA, 2008. ACM.

[113] Parasoft. Jtest Static Analysis. http://www.parasoft.com/jtest, 2010.

[114] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng.*, 20(6):463–475, 1994.

[115] PCI. PCI Security Standards Council. https://www.pcisecuritystandards.org/, 2010.

[116] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Context-sensitive correlation analysis for detecting races. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[117] Pravir Chandra. CLASP Best Practice. http://www.owasp.org/index.php/Category:CLASP_Best_Practice, June 2006.

[118] Privacy Rights Clearinghouse. Chronology of Data Breaches. http://www.privacyrights.org/ar/ChronDataBreaches.htm, June 2010.

[119] William Pugh and Nathaniel Ayewah. Unit testing concurrent software. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 513–516, New York, NY, USA, 2007. ACM.

[120] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting on-line surveys in software engineering. In *ISESE '03: Proceedings of the 2003 International Symposium on Empirical Software Engineering*, page 80, Washington, DC, USA, 2003. IEEE Computer Society.

[121] Daniel J. Quinlan, Richard W. Vuduc, and Ghassan Misherghi. Techniques for specifying bug patterns. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 27–35, New York, NY, USA, 2007. ACM.

[122] G. Ramalingam, Alex Warshavsky, John Field, Deepak Goyal, and Mooly Sagiv. Deriving specialized program analyses for certifying component-client conformance. *SIGPLAN Not.*, 37(5):83–94, 2002.

[123] Ganesan Ramalingam, Alex Warshavsky, John H. Field, and Mooly Sagiv. Deriving specialized heap analyses for verifying component-client conformance. Technical Report RC22145 (W0108-015), IBM Research, August 2001.

[124] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[125] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE '08: Proceedings of the 30th inter-*

national conference on Software engineering, pages 341–350, New York, NY, USA, 2008. ACM.

[126] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for Java runtime exceptions. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 153–164, New York, NY, USA, 2009. ACM.

[127] Jacek Śliwerski, Thomas Zimmermann, and Andreas 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.

[128] Fortify Software. Fortify SCA. http://www.fortify.com/, 2010.

[129] Jaime Spacco, David Hovemeyer, and William 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 Press.

[130] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson 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.

[131] Diomidis Spinellis and Panagiotis Louridas. A framework for the static verification of API calls. *Journal of Systems and Software*, 80(7):1156–1168, July 2007.

[132] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. `http://www.nist.gov/director/planning/upload/report02-3.pdf`, May 2002.

[133] Jeffrey Thalhammer. Perl::Critic. http://www.perlcritic.org/, 2010.

[134] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294, Washington, DC, USA, 2009. IEEE Computer Society.

[135] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 496–506, Washington, DC, USA, 2009. IEEE Computer Society.

[136] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2007. ACM.

[137] Luke Wildman, Brad Long, and Paul A. Strooper. Testing java interrupts and timed waits. In *APSEC*, pages 438–447, 2004.

[138] Chadd Williams and Jaime 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.

[139] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.

[140] Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering system specific rules from software repositories. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[141] Tim Wilson. Why Can't Johnny Develop Secure Software? *DarkReading.com Security*, June 2010.

[142] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. Meca: an extensible, expressive system and language for statically checking security properties. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 321–334, New York, NY, USA, 2003. ACM.